COREMEDIA

Studio Developer Manual

CoreMedia Content Cloud – v13

List of Figures

List of Tables

List of Examples

1. Introduction

This manual describes the configuration of and development with *CoreMedia Studio*. You will learn, for example, how to add your own Favorites, how to change or add labels, or how to customize forms.

- Chapter 2, *Overview* [13] gives a short overview of *CoreMedia Studio*.
- Chapter 3, *Deployment* [18] describes how to deploy *CoreMedia Studio* into different servlet containers.
- Chapter 4, *Quick Start* [24] describes how to set up a development workspace that is ready for *CoreMedia Studio* development.
- Chapter 5, *Concepts and Technology* [32] gives an overview of the concepts and technologies used by *CoreMedia Studio*. It is not a prerequisite for the following chapters, but will give you valuable insight into the underlying concepts.
- Chapter 8, *Using the Development Environment* [99] introduces the build tools and processes that are recommended for the development of *CoreMedia Studio*.
- Chapter 9, *Customizing CoreMedia Studio* [118] explains specific customizations of *CoreMedia Studio*.
- Chapter 10, *Rich Text Editing* [416] provides an overview of the richtext editing capabilities in *CoreMedia Studio* and how to adapt them to your needs.

> **CAUTION**
>
> Since version 1.3, the *CoreMedia Studio* API is marked *final*, meaning that changes and extensions to the API are guaranteed to be backwards compatible. Any changes to the API are however described in the release notes, and it is recommended to consult these when upgrading to a newer version, so that you can benefit from added functionality or more convenient or powerful ways to make use of certain features.

1.1 Audience

This manual is intended for developers who want to customize *CoreMedia Studio*. You should know the basics of *CoreMedia CMS*. Knowledge about the *Unified API* is particularly helpful. You should also have a solid understanding of Maven, TypeScript, JavaScript and Ext JS.

1.2 Typographic Conventions

CoreMedia uses different fonts and types in order to label different elements.
The following table lists typographic conventions for this documentation:

Element	Typographic format	Example
Source code Command line entries Parameter and values Class and method names Packages and modules	Courier new	`cm systeminfo start`
Menu names and entries	Bold, linked with |	Open the menu entry **Format|Normal**
Field names CoreMedia Components Applications	Italic	Enter in the field *Heading* The *CoreMedia Component* Use *Chef*
Entries	In quotation marks	Enter "On"
(Simultaneously) pressed keys	Bracketed in "<>", linked with "+"	Press the keys <Ctrl>+<A>
Emphasis	Italic	It is *not* saved
Buttons	Bold, with square brackets	Click on the **[OK]** button
Code lines in code examples which continue in the next line	\	`cm systeminfo \` `-u user`

Table 1.1. Typographic conventions

In addition, these symbols can mark single paragraphs:

Pictograph	Description
	Tip: This denotes a best practice or a recommendation.
	Warning: Please pay special attention to the text.
	Danger: The violation of these rules causes severe damage.

Table 1.2. Pictographs

1.3 CoreMedia Services

This section describes the CoreMedia services that support you in running a CoreMedia system successfully. You will find all the URLs that guide you to the right places. For most of the services you need a CoreMedia account. See Section 1.3.1, "Registration" [5] for details on how to register.

NOTE

CoreMedia User Orientation for CoreMedia Developers and Partners

Find the latest overview of all CoreMedia services and further references at:

http://documentation.coremedia.com/new-user-orientation

- Section 1.3.1, "Registration" [5] describes how to register for the usage of the services.
- Section 1.3.2, "CoreMedia Releases" [6] describes where to find the download of the software.
- Section 1.3.3, "Documentation" [7] describes the CoreMedia documentation. This includes an overview of the manuals and the URL where to find the documentation.
- Section 1.3.4, "CoreMedia Training" [10] describes CoreMedia training. This includes the training calendar,the curriculum and certification information.
- Section 1.3.5, "CoreMedia Support" [10] describes the CoreMedia support.

1.3.1 Registration

In order to use CoreMedia services you need to register. Please, start your initial registration via the CoreMedia website. Afterwards, contact the CoreMedia Support (see Section 1.3.5, "CoreMedia Support" [10]) by email to request further access depending on your customer, partner or freelancer status so that you can use the CoreMedia services.

1.3.2 CoreMedia Releases

Downloading and Upgrading the Blueprint Workspace

CoreMedia provides its software as a Maven based workspace. You can download the current workspace or older releases via the following URL:

https://releases.coremedia.com/cmcc-13

Refer to our Blueprint Github mirror repository for recommendations to upgrade the workspace either via Git or patch files.

> **NOTE**
>
> If you encounter a 404 error then you are probably not logged in at GitHub or do not have sufficient permissions yet. See Section 1.3.1, "Registration" [5] for details about the registration process. If the problems persist, try clearing your browser cache and cookies.

Maven artifacts

CoreMedia provides parts of its release artifacts via Maven under the following URL:

https://repository.coremedia.com

You have to add your CoreMedia credentials to your Maven settings file as described in section Section 3.1, "Prerequisites" in *Blueprint Developer Manual* .

npm packages

CoreMedia provides parts of its release artifacts as npm packages under the following URL:

https://repository.coremedia.com/nexus/repository/coremedia-npm/

The .npmrc is configured to be able to utilize the registry (see Section 3.1, "Prerequisites" in *Blueprint Developer Manual*).

License files

You need license files to run the CoreMedia system. Contact the support (see Section 1.3.5, "CoreMedia Support" [10]) to get your licences.

1.3.3 Documentation

CoreMedia provides extensive manuals, how–tos and Javadoc as PDF files and as online documentation at the following URL:

https://documentation.coremedia.com

The manuals have the following content and use cases:

Manual	Audience	Content
Blueprint Developer Manual	Developers, architects, administrators	This manual gives an overview over the structure and features of *CoreMedia Content Cloud*. It describes the content type model, the *Studio* extensions, folder and user rights concept and many more details. It also describes administrative tasks for the features.
		It also describes the concepts and usage of the project workspace in which you develop your CoreMedia extensions. You will find a description of the Maven structure, the virtualization concept, learn how to perform a release and many more.
Connector Manuals	Developers, administrators	This manuals gives an overview over the use cases of the eCommerce integration. It describes the deployment of the Commerce Connector and how to connect it with the CoreMedia and eCommerce system.
Content Application Developer Manual	Developers, architects	This manual describes concepts and development of the *Content Application Engine (CAE)*. You will learn how to write Freemarker templates that access the other CoreMedia modules and use the sophisticated caching mechanisms of the CAE.
Content Server Manual	Developers, architects, administrators	This manual describes the concepts and administration of the main CoreMedia component, the *Content Server*. You will learn about the content

Manual	Audience	Content
		type model which lies at the heart of a CoreMedia system, about user and rights management, database configuration, and more.
Deployment Manual	Developers, architects, administrators	This manual describes the concepts and usage of the CoreMedia deployment artifacts. That is the deployment archive and the Docker setup. You will also find an overview of the properties required to configure the deployed system.
Elastic Social Manual	Developers, architects, administrators	This manual describes the concepts and administration of the *Elastic Social* module and how you can integrate it into your websites.
Frontend Developer Manual	Frontend Developers	This manual describes the concepts and usage of the Frontend Workspace. You will learn about the structure of this workspace, the CoreMedia themes and bricks concept, the CoreMedia Freemarker facade API, how to develop your own themes and how to upload your themes to the CoreMedia system.
Headless Server Developer Manual	Frontend Developers, administrators	This manual describes the concepts and usage of the *Headless Server*. You will learn how to deploy the Headless Server and how to use its endpoints for your sites.
Multi-Site Manual	Developers, Multi-Site Administrators, Editors	This manual describes different otions to desgin your site hierarchy with several languages. It also gives guidance to avoid common pitfalls during your work with the multi-site feature.
Operations Basics Manual	Developers, administrators	This manual describes some overall concepts such as the communication between the components, how to set up secure connections, how to start application.
Search Manual	Developers, architects, administrators	This manual describes the configuration and customization of the *CoreMedia Search Engine* and the two feeder applications: the *Content Feeder* and the *CAE Feeder*.

Manual	Audience	Content
Studio Developer Manual	Developers, architects	This manual describes the concepts and extension of *CoreMedia Studio*. You will learn about the underlying concepts, how to use the development environment and how to customize *Studio* to your needs.
Studio User Manual	Editors	This manual describes the usage of *CoreMedia Studio* for editorial and administrative work. It also describes the usage of the *Native Personalization* and *Elastic Social* GUI that are integrated into *Studio*.
Studio Benutzerhandbuch	Editors	The Studio User Manual but in German.
Supported Environments	Developers, architects, administrators	This document lists the third-party environments with which you can use the CoreMedia system, Java versions or operation systems for example.
Unified API Developer Manual	Developers, architects	This manual describes the concepts and usage of the *CoreMedia Unified API*, which is the recommended API for most applications. This includes access to the content repository, the workflow repository and the user repository.
Utilized Open Source Software & 3rd Party Licenses	Developers, architects, administrators	This manual lists the third-party software used by CoreMedia and lists, when required, the licence texts.
Workflow Manual	Developers, architects, administrators	This manual describes the *Workflow Server*. This includes the administration of the server, the development of workflows using the XML language and the development of extensions.

Table 1.3. CoreMedia manuals

If you have comments or questions about CoreMedia's manuals, contact the Documentation department:

Email: documentation@coremedia.com

1.3.4 CoreMedia Training

CoreMedia's training department provides you with the training for your Core-Media projects either live online, in the CoreMedia training center or at your own location.

You will find information about the CoreMedia training program, the training schedule and the CoreMedia certification program at the following URL:

https://www.coremedia.com/training

Contact the training department at the following email address:

Email: training@coremedia.com

1.3.5 CoreMedia Support

CoreMedia's support is located in Hamburg and accepts your support requests between 9 am and 6 pm MET. If you have subscribed to 24/7 support, you can always reach the support using the phone number provided to you.

To submit a support ticket, track your submitted tickets or receive access to our forums visit the CoreMedia Online Support at:

https://support.coremedia.com/

Do not forget to request further access via email after your initial registration as described in Section 1.3.1, "Registration" [5]. The support email address is:

Email: support@coremedia.com

Create a support request

CoreMedia systems are distributed systems that have a rather complex structure. This includes, for example, databases, hardware, operating systems, drivers, virtual machines, class libraries and customized code in many different combinations. That's why CoreMedia needs detailed information about the environment for a support case. In order to track down your problem, provide the following information:

Support request

- Which CoreMedia component(s) did the problem occur with (include the release number)?
- Which database is in use (version, drivers)?
- Which operating system(s) is/are in use?
- Which Java environment is in use?

- Which customizations have been implemented?
- A full description of the problem (as detailed as possible)
- Can the error be reproduced? If yes, give a description please.
- How are the security settings (firewall)?

In addition, log files are the most valuable source of information.

To put it in a nutshell, CoreMedia needs: *Support checklist*

1. a person in charge (ideally, the CoreMedia system administrator)

2. extensive and sufficient system specifications

3. detailed error description

4. log files for the affected component(s)

5. if required, system files

An essential feature for the CoreMedia system administration is the output log   *Log files*
of Java processes and CoreMedia components. They're often the only source
of information for error tracking and solving. All protocolling services should run
at the highest log level that is possible in the system context. For a fast break-
down, you should be logging at debug level. See Section 4.7, "Logging" in *Opera-
tions Basics* for details.

Which Log File?

In most cases at least two CoreMedia components are involved in errors: the
Content Server log files together with the log file from the client. If you know
exactly what the problem is, solving the problem becomes much easier.

Where do I Find the Log Files?

By default, application containers only write logs to the console output but can
be accessed from the container runtime using the corresponding command-
line client.

For the *docker* command-line client, logs can be accessed using the **docker
logs** command. For a detailed instruction of how to use the command, see
docker logs. Make sure to enable the timestamps using the `--timestamps`
flag.

```
docker logs --timestamps <container>
```

For the *kubectl* command-line client in a Kubernetes environment you can use
the **kubectl logs** command to access the logs. For a detailed instruction of how
to use the command, see kubectl logs. Make sure to enable the timestamps using
the `--timestamps` flag.

```
kubectl logs --timestamps <pod>
```

1.4 Changelog

In this chapter you will find a table with all major changes made in this manual.

Section	Version	Description

Table 1.4. Changes

2. Overview

CoreMedia Studio is a web application that is in the center of your web activities. It gives you complete control over context's determinants and lets you easily create compelling and engaging content experiences. Technically, *CoreMedia Studio* is a single-page Ajax application, using a REST based network protocol for communication.

2.1 Architecture

Figure 2.1, "Architecture of CoreMedia Studio" [14] shows the architecture of *CoreMedia Studio*. The top-level layer comprises content editing applications such as the *CoreMedia Studio* core application and its plugins. *CoreMedia Blueprint* defines several plugins, showcasing *Studio*'s various extension points.

Editing applications are built on a layer of editing components that deal with CoreMedia content objects. Editing components are built on the UI Toolkit layer which provides generic components for building rich internet applications. On this layer, components can be implemented in TypeScript and then compiled to Ext JS. UI components separate layout, model and functionality according to the MVC paradigm. Models that are backed by server-side data are implemented as client-side beans that fetch the requested values via REST. UI components offer localization support. The lower level layers comprise the REST API of the *CoreMedia CMS*.
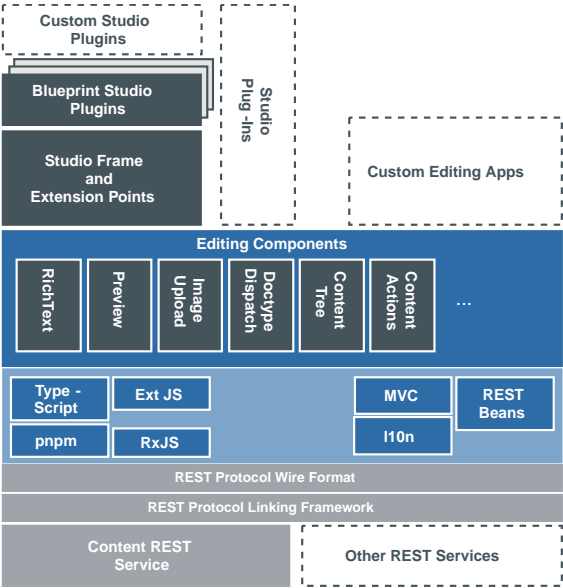
Figure 2.1. Architecture of CoreMedia Studio

As shown below, on the server side, *CoreMedia Studio* consists of two servers: One that serves static resources, one that implements the dynamic REST service. The static resources are those that define the client-side UI structure (HTML

and JavaScript) and the client-side layout (CSS and images). The dynamic re-
sources can be accessed via the Content REST Service. When you start *Core-
Media Studio* from your browser, it loads the static resources and initializes the
Ext JS UI component tree, Studio plugins and model beans. Using the *RxJS* library,
model beans issue requests to access the Content REST Service, which is the
interface to the CoreMedia backend systems and load data from the returned
JSON objects.
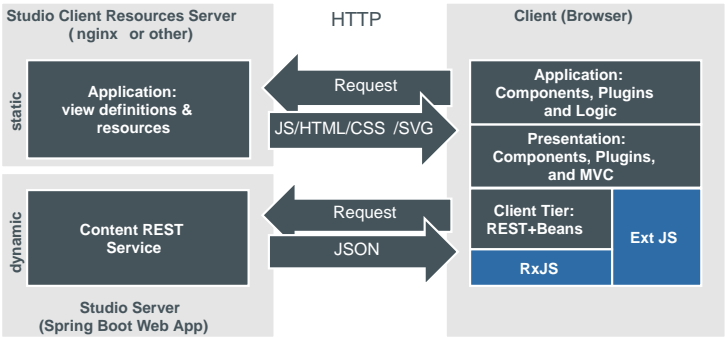
Figure 2.2. Runtime components

2.2 Technologies

This section gives you a brief overview of *CoreMedia Studio*'s underlying technologies. These are the TypeScript to Ext JS compiler and build tools named *Jangaroo*, the JavaScript UI framework Ext JS, and *CKEditor* for rich text editing.

Ext JS

Ext JS is a cross-browser rich internet application framework developed by Sencha Inc. It offers JavaScript UI widgets and client side MVC. To this end, Ext JS provides components, actions and data abstractions. Components can be customized by plugins. Component trees are described in JSON notation. Ext JS defines the JavaScript properties `xtype` and `ptype` to distinguish between components and plugins.

In short, Ext JS has the following features:

- clean object-oriented design,
- hierarchical component architecture (component tree),
- large UI library with mature widgets, especially mature business components (`Store` abstraction, `DataGrid`),
- built-in layout management,
- good drag and drop support with sophisticated visual feedback,
- declarative UI description language (JSON).

Ext JS also provides a rich set of utility functions to deal with components or plain JavaScript objects and functions. The complete Ext JS documentation can be found on http://www.sencha.com/learn/Learn_About_the_Ext_JavaScript_Library.

Jangaroo

CoreMedia's tools to support TypeScript as a source language for Ext JS development are released under the Jangaroo brand. While Sencha, the vendor of Ext JS, provides basic TypeScript typings for the configuration API of their components in order to use them from React and Angular, CoreMedia / Jangaroo support the full Ext JS API in TypeScript, generated from the official Sencha Ext JS documentation. TypeScript source code is compiled to Ext JS-compatible JavaScript. This approach is called Ext TS and described in detail in Section 5.2, "Ext TS: Developing Ext JS in TypeScript" [38].

To support the declarative development of complex components, Ext JS uses JSON-like *Config* objects. Ext TS enhances these Config objects with strong

static typing, using a utility type and function, consequently called `Config`. Using static typing leads to a superior developer experience in any IDE that supports TypeScript, like JetBrains' *IntelliJ IDEA Ultimate* or *WebStorm* and Microsoft's *Visual Studio Code*.

The *CoreMedia Studio* builds on Ext JS 7: https://www.sencha.com/products/extjs/#overview.

CKEditor

CKEditor is a browser based open source WYSIWYG text editor (ckeditor.com). Common editing features found on desktop editing applications like Microsoft Word and OpenOffice are brought to the web browser by using *CKEditor*.

For details regarding integration of *CKEditor* into *CoreMedia Studio* see Chapter 10, *Rich Text Editing* [416].

3. Deployment

This chapter describes how to deploy *CoreMedia Studio* to different servlet containers.

> **NOTE**
>
> Perform all configurations of *CoreMedia Studio* described in this chapter in the module `studio-webapp` of *CoreMedia Blueprint* workspace before building or later on during deployment of *Studio*.

3.1 Connecting to the Repository

CoreMedia Studio needs to know the connection parameters of the *Content Management Server* and *Workflow Server* to connect to. See Section 3.12, "UAPI Client Properties" in *Deployment Manual*.

CoreMedia Studio also needs to know the URL of Apache Solr and the name of the index collection for searching the repository content. Configure the URL in the property `solr.url` and the name of the index collection in the property `solr.content.collection` in the same file.

```
solr.url=http://<Host>:<Port>/solr
solr.content.collection=studio
```

CoreMedia Studio needs an additional relational database connection to store editorial comments. The properties `editorial.comments.data source.url` and `editorial.comments.datasource.driver-class-name` have to bet set according to the RDBMs to connect to. Furthermore, for each RDBMs there are differences in the configuration of the JDBC connection. For more details see Section 3.4, "Editorial Comments Database Configuration" [22].

CoreMedia Studio offers connectivity to the *CoreMedia Workflow Server*. Therefore, a *Workflow Server* has to run when starting *CoreMedia Studio*. If this is not desired, set the property `repository.workflow.connect` in the file `WEB-INF/application.properties` to `false`.

```
repository.workflow.connect=false
```

Studio supports "Simple Publication" and "Two Step Publication" publication workflows. To use these workflows, upload the workflow definitions `studio-simple-publication.xml` and `studio-two-step-publication.xml` to the *Workflow Server* with the *cm upload* tool. See section Section 5.6.1.2, "Predefined Publication Workflows" in *Blueprint Developer Manual* for more information on these workflows.

3.2 Control Room Configuration

The Control Room consists of the following components:

- Control Room Plugin is a *Studio* plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other *Studio* users.
- *User Changes Application* is a repository listener, which collects content modified by a user working with *Studio*. To this end, the modified content can be managed in the Control Room plugin as projects, shared and used in workflows, for example.
- Extensions of the *Workflow Server* – Control Room comes with adapted workflow definitions that among other things persist finished workflows.

NOTE

Perform all configurations concerning the *User Changes Application* in the module `user-changes-app` in *CoreMedia Blueprint* before building or later on during deployment of the *User Changes Application*.

The Control Room stores content sets and finished workflows, commonly specified as collaboration data in an SQL database. See following sections for detailed configuration options:
<itemizedList>
<listitem>Section 3.4.12, "Projects/To-Dos SQL Persistence Configuration" in *Deployment Manual*</listitem>
<listitem>Notifications SQL Persistence Configuration for CoreMedia Studio in *Deployment Manual*</listitem>
<listitem>Section 3.8, "My Edited Content and Workflow Lists Properties" in *Deployment Manual*</listitem>
</itemizedList>

3.3 Basic Preview Configuration

The configuration options regarding the studio preview are listed in the Deployment Manual.

3.4 Editorial Comments Database Configuration

CoreMedia Studio Server stores Editorial Comments in an SQL database, so it needs to be configured accordingly.

The default schema, username and password is: `cm_editorial_comments`.

In case you want to provide a schema or username, different to `cm_editorial_comments`, use the properties `editorial.comments.db.username`, `editorial.comments.db.schema` and `editorial.comments.db.password`.

The datasource url has to be set for the respective database with the property `editorial.comments.datasource.url` like this:

MySQL	`jdbc:mysql://${host}:${port}/cm_editorial_comments`
PostgreSQL	`jdbc:postgresql://${host}:${port}/coremedia`
MariaDB	`jdbc:mariadb://${host}:${port}/cm_editorial_comments`
Oracle	`jdbc:oracle:thin:@${host}:${port}:COMMENTS`

MySQL / MariaDB: Ensure Proper Character Set And Collation

To ensure proper encoding and collation behavior, ensure using character set `utf8mb4` and collation `utf8mb4_bin` for your created database.

MariaDB: Respect Password Check Plugins

MariaDB Enterprise Server comes with a password check plugin enabled by default. Ensure, that your password meets the requirements of that password check plugin.

3.5 Development Setup

During development, it may be convenient to specify the property `content server.host` and optionally the property `contentserver.port` for connecting to the *Content Server* as system properties on the command line when starting the Studio servlet container.

4. Quick Start

This chapter presents the basic steps to set up a *CoreMedia Studio* development environment quickly.

4.1 Setting Up the Workspace and IDE

Setting Up the Workspace

CoreMedia Content Cloud comes with a fully preconfigured, Maven and pnpm based development workspace. Details on how to get and set up your development environment are described in the [Blueprint Developer Manual]. You will find guidance for the following topics:

1. Required third-party software, such as Maven and pnpm.

2. Getting *CoreMedia Blueprint*.

3. Installing *CoreMedia Blueprint*.

4. Configuring all components.

5. Building the workspace.

6. Starting the components.

The recommended development setup is to use the workspace `apps/studio-client` for client-side changes and `apps/studio-server` for server-side customizations. For the latter, you may additionally need to change shared code in workspace `shared/middle` or, in rare cases, `shared/common`.

Setting Up the IDE

Once you have set up the workspace, you may configure your IDE as described in Chapter 7, *Developing with the Studio Client Workspace* [93].

4.2 Building Studio Server

A detailed description on how to build *CoreMedia Studio* can be found in Chapter 8, *Using the Development Environment* [99]. If you are using IntelliJ IDEA and the IDE is set up correctly, you can build the whole project via Maven from within the IDE. If you prefer building from the command line, you can do it by using standard Maven commands like

```
mvn clean install -DskipTests
```

The *CoreMedia Studio* server application can then be launched either directly from your IDE or via Maven from the commandline by calling `mvn spring-boot:run` in module `studio-server-app`.

4.3 Building Studio Client

Building the *CoreMedia Studio* client application can be achieved via pnpm from the `apps/studio-client` folder using the following commands:

```
pnpm install
pnpm -r run build
```

Next, start the *CoreMedia Studio* client application by changing into the `apps/studio-client/global/studio` directory and using the following command:

```
pnpm run start
```

More details on how to build and start *CoreMedia Studio*, as well as how to run tests with it, are described in Chapter 7, *Developing with the Studio Client Workspace* [93]. Additionally, see Section 8.3, "Debugging" [105] for details on how to debug.

4.4 Creating a Simple Studio Client Extension

You can customize many features of *Studio* with plugins. This section shows the deployment of a simple plugin into the Blueprint workspace. The plugin is only intended as an example and adds a string property to the *Content Items Linking to this Content Item* field of the *System* tab. The aim of this tutorial is to give you a working starting point, from which you can start exploring all details and features of Studio customization.

The required CoreMedia and third-party components, such as Content Servers, CAE and databases are running in the CoreMedia Docker environment.

Each of the following steps link to chapters which give more information about the described task. CoreMedia also recommends attending the CoreMedia Studio Customization training. See https://www.coremedia.com/training for details.

1. In order to check the prerequisites, get the Blueprint workspace, get licences, build the workspace and start the Docker environment. Follow the instructions in Section 3.2, "Quick Start" in *Blueprint Developer Manual* .

   When you are finished with these tasks, you should have a Blueprint workspace where you can develop your plugin and a CoreMedia system running in Docker containers on your local machine.

2. Prepare your IDE for Studio development as described in Chapter 7, *Developing with the Studio Client Workspace* [93].

3. Create your plugin in the `apps/studio-client/apps/main/exten sions/mycompany/myplugin` directory, using the pnpm Starter Kit (see Section 9.3, "Studio Plugins" [130] for an in-depth description of Studio plugins and Section 4.4.3, "Developing with Studio" in *Blueprint Developer Manual* for pnpm configuration).

   ```
   pnpm create @jangaroo/project
   apps/studio-client/apps/main/extensions/mycompany/myplugin
   ```

   The default choices of the command line tool should be sufficient. As a package name it makes sense to stick to the following naming pattern: `@MY COMPANY/studio-client.main.MYPLUGIN`. Regarding the versioning stick to Semantic Versioning.

   See Chapter 6, *Structure of the Studio Client Workspace* [90] for more details on the created structure and files.

4. The functionality of your plugin will be defined via `*.ts` files (in this case `ExampleStudioPlugin.ts`). The example adds a string property to the

Content Items Linking to this Content Item field of the *System* tab. See the complete Chapter 9, *Customizing CoreMedia Studio* [118] and the TSDoc for more customization features.

Add dependencies to `@coremedia/studio-client.main.editor-components`, `@coremedia/studio-client.ext.ui-components` and `@jangaroo/runtime` to the recently created package by using:

```
pnpm add @coremedia/studio-client.main.editor-components
@coremedia/studio-client.ext.ui-components @jangaroo/runtime
```

After you have added and installed the dependencies, make sure to initially build to package using `pnpm run build` so the `tsconfig.json` is properly setup for syntax assist.

Copy the code into the `src/ExampleStudioPlugin.ts` file.

```
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import StringPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/StringPropertyField";
import ReferrerListPanel from
"@coremedia/studio-client.main.editor-components/sdk/premular/ReferrerListPanel";
import AddItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/AddItemsPlugin";
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";

interface SimplePluginExample extends Config<StudioPlugin> {
}

class SimplePluginExample extends StudioPlugin {
  declare Config: SimplePluginExample;

  constructor(config: Config<SimplePluginExample> = null) {
    super(ConfigUtils.apply(Config(SimplePluginExample, {

      rules: [
        // add your rules here...
        Config(ReferrerListPanel, {
          plugins: [
            Config(AddItemsPlugin, {
              items: [
                Config(StringPropertyField, {
                  propertyName: "title",
                }),
              ]
            }),
          ],
        }),
      ],
    }), config));
  }
}

export default SimplePluginExample;
```

Example 4.1. SimplePluginExample.ts

To actually load the `Studio Client Plugin` on startup you need to add the following entry to the `jangaroo.config.js` file:

```
...
module.exports = jangarooConfig({
  ...
  sencha: {
    ...
    namespace: "mycompany.myplugin",
    studioPlugins: [
      {
        mainClass: "mycompany.myplugin.SimplePluginExample",
        name: "Asset Management Extensions",
      },
    ],
  }
});
...
```

Example 4.2. jangaroo.config.js

To properly mark the package as an extension to be handled by the *CoreMedia Extension Tool* you need to add the following entry to the package.json file:

```
{
  ...
  "coremedia": {
    "projectExtensionPoint": "studio-client.main"
  },
}
```

Example 4.3. package.json

5. Call the *CoreMedia Extension Tool* from the commandline in the workspace root directory. See Section 4.1.5, "Project Extensions" in *Blueprint Developer Manual* for a description of extensions and the extensions tool.

```
mvn -f workspace-configuration/extensions extensions:sync -Denable=mycompany
```

The tool will add your plugin to the following files:

- apps/studio-client/pnpm-workspace.yaml
- apps/studio-client/apps/main/extension-config/exten sion-dependencies/package.json

6. Install and build the studio-client from the root of your workspace:

```
pnpm install
pnpm -r run build
```

7. Start Studio locally on your machine from the apps/studio-client/glob al/studio directory:

```
pnpm run start --proxyTargetUri http://docker.localhost:41080
```

8. Enter **http://localhost:3000** in your browser. Studio should open. Log in and open an article. You will see an additional property field.
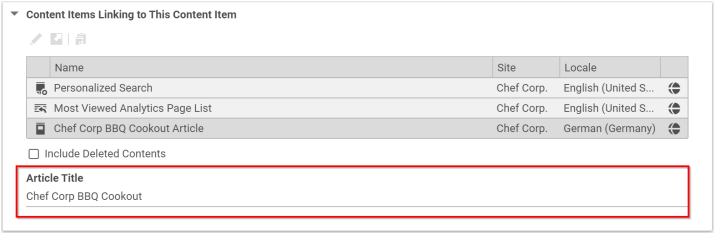
Figure 4.1. Added string property with the title of the content

Now, you have created your first – very simple – running Studio extension and learned about the required structure and tools. From this starting point, you might now extend your plugin. See Section 8.3, "Debugging" [105] for details on how to debug the application. When you are finished, you only have to build your plugin, not the complete studio-client package. From the plugin directory simple call:

```
pnpm run build
```

5. Concepts and Technology

This chapter describes the basic concepts and technologies on a more detailed level than in the overview chapter. It is not a prerequisite for the subsequent chapters, but it will give you valuable insight into the underlying concepts.

5.1 Ext JS Primer

Ext JS is a JavaScript library for building interactive web applications. It provides a set of UI widgets like panels, input fields or toolbars and cross-browser abstractions (Ext core).

CoreMedia Studio uses Ext JS (Classic Toolkit) on the client side. With plain Ext JS, widgets are defined in JSON format as displayed in the following example:

```
{
 xtype: "panel",
 title: "Teaser Properties",
 items: [
  {
   xtype:
    "com.coremedia.cms.editor.sdk.config.stringPropertyField",
   itemId: "linktextEditor",
   propertyName: "linktext"
  },
  {
   xtype:
    "com.coremedia.cms.editor.sdk.config.cke5RichTextPropertyField",
   propertyName: "teaserText",
   anchor: "98%",
   height: 300
  }
 ],
 defaults: {
  bindTo: config.bindTo
 }
}
```

Example 5.1. Ext JSON

The above code example defines a component of `xtype` "panel" with two *property editors* for editing a string and a richtext property, respectively. The `xtype` of the surrounding panel, like that of all Ext JS components, is a simple string without a namespace prefix. The `xtype` of a plain Ext JS component is, in most cases, the name of the component class, in all lowercase characters.

The property editors shown above are *CoreMedia Studio* components, that are based on plain Ext JS components, but add Studio-specific functionality. Their `xtype` is a qualified name. See Section 5.2, "Ext TS: Developing Ext JS in TypeScript" [38] for details. Instead of the `xtype` attribute you can also use the `xclass` attribute, which uses the fully qualified class name of the component.

The optional `itemId` property can be understood as a per-container id which identifies the component uniquely within its container. Note that `itemIds` are not to be confused with DOM element ids or Ext JS component ids which are unique within the entire application.
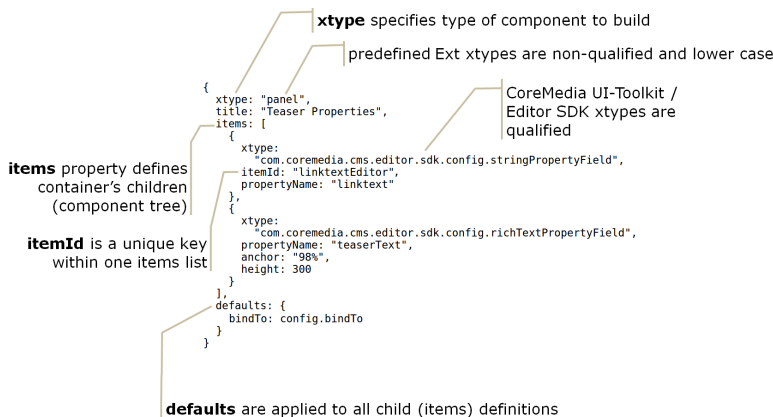
Figure 5.1. Ext JSON

When developing *CoreMedia Studio* extensions, you don't need to use the Ext JS `xtype` and memorize or look up all possible `xtype` values and their supported configuration properties. Instead, you're encouraged to specify components using the much more convenient and type-safe `Config` notation that takes advantage of TypeScript's type checking and IDE support. It is available from CoreMedia's Jangaroo project. The example below shows the Studio TypeScript code corresponding to the Ext JSON from above:

```
import Config from "@jangaroo/runtime/Config";
import Panel from "@jangaroo/ext-ts/panel/Panel";
import StringPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/StringPropertyField";
import RichTextPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/richtext/RichTextPropertyField";
import PropertyField from
  "@coremedia/studio-client.main.editor-components/sdk/premular/PropertyField";

Config(Panel, {
  title: "Teaser Properties",
  items: [
    Config(StringPropertyField, {
      itemId: "linktextEditor",
      propertyName: "linktext"
    }),
    Config(RichTextPropertyField, {
      propertyName: "teaserText",
      anchor: "98%",
      height: 300
    })
  ],
  defaults: Config<PropertyField>({
```

```
    bindTo: config.bindTo
  })
})
```

Example 5.2. Ext JSON in TypeScript

As you can see, the `xtype` properties with string values are replaced by import-ing the corresponding component class and using the (also imported) Jangaroo utility function `Config` with the component class and a corresponding config-uration object. Each component class defines a Config type which the given configuration object must match. Calling the `Config` function with a component class adds the corresponding `xtype` at run-time, calling it with just a configur-ation object, like in the example for the value of `defaults`, only takes care of the type check. For details, see Section 5.2, "Ext TS: Developing Ext JS in TypeScript" [38].

The following sections describe Ext JS components, plugins, and actions in more detail.

Ext JS-specific examples of advanced components are available on the official Ext JS examples page. The full Ext JS API documentation is also available at sencha.com.

5.1.1 Components

Ext JS defines three basic types of components

- `Ext.Component`
- `Ext.container.Container`
- `Ext.container.Viewport`

The base class for Ext JS UI controls is `Ext.Component`. Components are re-gistered with the `Ext.ComponentManager` at construction time. They can be referenced at any time by id using the `Ext.getCmp` utility function. For more sophisticated searches like by xtype or component structure the `Ext.ComponentQuery` can be used as well as methods provided by `Ext.mixin.Queryable` as for example in `Ext.container.Container`. Component classes are required to define a static property named "xtype" that is used by the component manager to determine the runtime type of a compon-ent given in JSON notation.

Components are nested in containers of class `Ext.container.Container` which is a subclass of `Ext.Component`. Containers manage the lifecycle (that is, control creation, rendering and destruction) of their child components.

The top-level component of *Studio*'s component tree is `Ext.contain er.Viewport`, which represents the viewable application area of the browser.

The API documentation of Ext JS is available at sencha.com. Specifically, the documentation of `Ext.Component` provides a list of component types available in Ext JS. It is also worth looking into the API documentation of `Ext.Com ponentManager`, `Ext.dom.Element`, and the `Ext` namespace/utility class which contains many useful singletons like for example the `Ext.Compon entQuery`.

5.1.2 Component Plugins

In general, the recommended strategy for extending Ext JS components is to use the component plugin mechanism, rather than subclassing. Reusable functionality should be separated out into component plugins, and can then be used by components of completely different types, without requiring them to inherit from a common base class.

Plugins are configured in a component's `plugins` property. A plugin must provide an `init` method accepting the component it is plugged into as parameter. This method is called by the component when the component is initialized.

The following code defines a `field` component and adds the plugin `BindProp ertyPlugin`.

```
{
  xtype: 'field',
  name: 'properties.' + config.propertyName,
  plugins: [
    {
      bindTo: config.bindTo.extendBy('properties',
        config.propertyName),
      bidirectional: true,
      xclass: 'com.coremedia.ui.plugins.BindPropertyPlugin',
    }
  ]
}

// The same declaration in TypeScript:

import Config from "@jangaroo/runtime/Config";
import BaseField from "@jangaroo/ext-ts/form/field/Base";
import BindPropertyPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/BindPropertyPlugin";

Config(BaseField, {
  name: 'properties.' + config.propertyName,
  plugins: [
    Config(BindPropertyPlugin, {
      bindTo: config.bindTo.extendBy('properties',
        config.propertyName),
      bidirectional: true
    })
```

```
   ]
})
```

Example 5.3. Plugin usage in Ext JSON

Refer to blog post Using Plugins and Mixins in Your Sencha Apps by Seth Lemmons (October 23, 2014) for further details on Ext JS plugins.

5.1.3 Actions

Actions combine some functional parts of your application with UI details to be attached to a component. Buttons, for example, are commonly associated with an action. The difference between designing an action and attaching a mere event handler to a component is that an action combines the handler code with UI details such as a name or a button icon, which simplifies reuse. *CoreMedia Studio* defines actions that work on content objects, for example for creating new content objects or publishing contents.

It is not recommended instantiating an action just to invoke it once programmatically. For such tasks, use the corresponding API method instead. For example, when you write a piece of code that needs to publish content, use the API method `PublicationService#publish(content, callback)` instead of creating a temporary `PublishAction`.

5.2 Ext TS: Developing Ext JS in TypeScript

While the *CoreMedia Studio* code you see at runtime is all JavaScript, *CoreMedia Studio* is completely written in *TypeScript*. CoreMedia calls this combination of tools and approach *Ext TS*, where obviously, "TypeScript" replaces the "JavaScript" in *Ext JS*.

While Sencha, the vendor of Ext JS, provides basic TypeScript typings for the configuration API of their components in order to use them from other frameworks like React or Angular, CoreMedia/Jangaroo supports the full Ext JS API in TypeScript, generated from the official Sencha Ext JS documentation. With Jangaroo Ext TS, TypeScript source code is compiled to Ext-JS-compatible JavaScript.

Ext TS is designed to provide a statically typed way to implement Ext JS applications. Typed object literals, so-called Config objects, are used to declaratively describe Ext UI components (or component trees). During the build process, Ext TS TypeScript files are compiled to JavaScript using the Ext JS class and Config system.

While it is possible to extend *CoreMedia Studio* with components written in JavaScript, it is recommended to use Ext TS. With the Jangaroo project, CoreMedia offers tools and libraries that provide complete support for this development approach. All public *CoreMedia Studio* APIs as well as the original Ext JS API are available as TypeScript `*.d.ts` files, so that you can set up your IDE to provide code completion, validation and documentation lookup.

This section states the rationale for using Ext TS, gives you a rough overview of the approach and tools, and explains in detail how Ext TS TypeScript sources translate to "pure" Ext JS.

Ext TS: the Typed Version of Ext JS

In contrast to JavaScript and JSON, TypeScript is a typed language. While originally, typed languages were chosen to find errors early at compile time, the more important advantage today is that much better tools can be built to ease and speed up development. In a good IDE, errors and possible mistakes are detected as you type, and the IDE even makes suggestions as to what to type next, how to resolve errors, and lets you look up documentation easily. Using a typed language is important for the IDE to be able to derive what the code is referring to. With an untyped language, only limited IDE support is possible, and the IDE must

use more or less imprecise heuristics, and will in many cases make ambiguous (or even erroneous) suggestions.

Source File Types and Compilers

CoreMedia Studio is an Ext TS application and as such uses four different kinds of source files:

• Ext TS TypeScript files that compile to Ext JS classes
• TypeScript files representing properties for localized texts and labels
• Standard TypeScript files for all other, Ext-JS-independent application code
• A few JavaScript files for bootstrap or low-level code

The Jangaroo build process invokes the Jangaroo compiler to translate TypeScript source file types to JavaScript and then proceeds to handle all JavaScript files. The compiler is invoked through *pnpm* and based on *Babel*.

TypeScript Documentation

The following sections go into the details of some Ext TS concepts. They explain how Jangaroo represents Ext JS concepts in TypeScript and compiles such TypeScript back to Ext JavaScript.

5.2.1 Classes

As a start, compile a simple TypeScript class to Ext JS code. To enforce that it is treated as Ext TS code, the example class inherits from the Ext class `Base`, which is the base class of all Ext JS classes. To focus on how class features are translated, this example ignores import/export and the corresponding Ext JS code is slightly simplified.

TypeScript	Ext JS

```
class SimpleClass extends Ext.Base {
  foo: string;
  #bar: number = 0;

  constructor(newBar: number) {
    super();
    this.#bar = newBar;
  }

  protected hook(): boolean {
    return false;
  }

  get bar(): number {
    return this.#bar;
  }
```

```
Ext.define("SimpleClass", function
(SimpleClass) {
  return {
    extend: "Ext.Base",
    foo: undefined,
    bar$mgcE: 0,
    constructor: function (newBar) {

Ext.Base.prototype.constructor.apply(this,
arguments);
      this.bar$mgcE = newBar;
    },
    hook: function () {
      return false;
    },
    __accessors__: {
```

TypeScript	Ext JS
```\nset bar(value: number) {\n this.#bar = value;\n}\n\nstatic readonly FOO: any = "FOO";\n\nstatic #static = (() => {\n Registry.register(SimpleClass);\n})();\n}\n```	```\n bar: {\n get: function () {\n return this.bar$mgcE;\n },\n set: function (value) {\n this.bar$mgcE = value;\n }\n }\n },\n inheritableStatics: {\n FOO: "FOO",\n __initInheritableStatics__: function ()\n{\n Registry.register(SimpleClass);\n }\n }\n};\n});\n```

Table 5.1. TypeScript class to Ext JS example

This example illustrates the following mappings of ECMAScript/TypeScript features to Ext JS:

- The ECMAScript class syntax is not supported by Ext JS. It uses the `Ext.define()` utility function to declare classes. This function receives the (fully-qualified) name of the class to define and a class descriptor object, or a function receiving the (not yet initialized) class object and returning the class descriptor object.

- The ECMAScript class `extends` clause goes into the Ext class descriptor object's `extend` property and, instead of the super class itself, specifies the super class *name*.

- ECMAScript class fields create an entry in the Ext class descriptor object. Simple initializer values go into the corresponding value. If there is no initializer, in Ext, such fields are initialized using `undefined`, so that at least the property is present.

- The new ECMAScript private member syntax using the hash prefix (here: `#bar`) is not supported by Ext JS. The Jangaroo compiler simulates private members by renaming them. While this does not make them technically inaccessible, it avoids inadvertent name clashes in subclasses when the superclass introduces new private members.

  Jangaroo complements the private member name by a postfix `$` plus a hash computed from the fully-qualified name of the containing class.

- Like in normal TypeScript compilation, all TypeScript access modifiers (`public`, `protected`, `private`) generate no JavaScript code.

All TypeScript type annotations (`: SomeType`) also generate no JavaScript code.

- ECMAScript accessors are not supported by Ext JS, but its class system is extensible, so Jangaroo added a meta-property `__accessors__` to define properties with custom get/set logic.

- ECMAScript static class members are defined by the Ext JS meta-property `inheritableStatics`. Since static members are always inherited in ECMAScript, it is not possible to use Ext's (non-inheriting) `statics` from TypeScript. The value of `inheritableStatics` is a mapping from static member name to simple initial value.

  Ext JS initializes static members very early, so for custom static initialization logic (here: `Registry.register(SimpleClass);`), Jangaroo adds another meta-property `__initInheritableStatics__`, which specifies a function that is called later, when this class is used for the first time.

5.2.2 Interfaces

TypeScript has a notion of interfaces, but uses different semantics than other statically typed languages like Java or ActionScript.

In TypeScript, a class "automatically" implements an interface when it defines the same member signatures (*duck typing*). You can, however, use the keyword `implements` to explicitly state that your class intends to implement some interface. A TypeScript interface defines a so-called *ambient* type, that is a type that is only relevant for the compiler/type checker, but not at runtime. Consequently, there is no built-in way to do an instance-of check with an interface. To simulate this, you have to provide a custom function that tests whether a given object is of the interface type (*type guard*).

Studio used to be implemented in ActionScript, where it can be checked at runtime whether an object is an instance of a given interface, using the ActionScript built-in operator `is`. This means that in ActionScript, interfaces *do* have some run-time representation.

When converting code from ActionScript to TypeScript, we wanted to keep the ActionScript interface semantics, so we had to find some way to represent interfaces and the `is` operator in TypeScript.

Since Jangaroo ActionScript was compiled to JavaScript using the Ext class system, too, there already was a solution at run-time. Interfaces are represented as "empty" Ext classes, that is, classes that have no members, but an identity. When a class A implements an interface I, in Ext, the class corresponding to I is *mixed into* A. The `is` check is implemented by looking up the `mixins` hierarchy of the object's class.

We use a similar approach in TypeScript. A "runtime interface" is represented as a *completely abstract class*, that is, an abstract class that only has abstract members. At runtime, again, only an empty class with an identity remains. When implementing an interface, this abstract class is implemented *and* mixed in. TypeScript allows to "implement a class", because a class actually defines two entities: a value (the "class object" that exists at runtime) and a type (only relevant for the type checker / compiler). If you use a class in an `implements` clause, only its type is used. The *mixin* aspect is represented in TypeScript by calling the Jangaroo runtime function `mixin(Clazz, Interface1, ..., InterfaceN)` after the class declaration.

The following example illustrates how "runtime interfaces" are specified in TypeScript and how they translate to Ext JS.

TypeScript	Ext JS

```
abstract class IFoo extends Base {
  abstract foo: string;

  abstract get bar(): number;
  abstract set bar(value: number);

  abstract isAFoo(obj: any): boolean;
}

class Foo extends Base implements IFoo {
  foo: string;
  #bar: number;

  get bar(): number {
    return this.#bar;
  }

  set bar(value: number) {
    this.#bar = value;
  }

  isAFoo(obj: any): boolean {
    return is(obj, IFoo);
  }
}

mixin(Foo, IFoo);
```

```
Ext.define("IFoo", {
  extend: "Ext.Base"
});

Ext.define("Foo", {
  extend: "Ext.Base",
  mixins: ["IFoo"],
  requires: ["IFoo"],
  foo: undefined,
  bar$fPTk: undefined,
  __accessors__: {
    bar: {
      get: function () {
        return this.bar$fPTk;
      },
      set: function (value) {
        this.bar$fPTk = value;
      }
    }
  },
  isAFoo: function (obj) {
    return is(obj, IFoo);
  }
});
```

Table 5.2. Runtime Interfaces in TypeScript and Ext JS

The utility functions `is` and `mixin` are imported from `@jangaroo/runtime`. Section 5.2.3, "Imports and Exports" [43] explains how importing and exporting works in Ext TS.

The main takeaways here are that runtime interfaces are represented by abstract classes in TypeScript and by empty classes in Ext JS, implementing a runtime interface means mixing-in that class, and Ext JS's `mixins` class definition property is represented in TypeScript by calling the utility function `mixin` with

the class that implements the runtime interface as the first argument, and the runtime interface itself as the second argument.

5.2.3 Imports and Exports

In Ext JS, each *compilation unit* (usually a class, but Jangaroo allows global variables, constants or functions to also be compilation units) has a *fully-qualified name* that is *globally unique*. Ext JS uses this name to reference other compilation units when *importing* them. The name consists of a (hierarchical, dot-separated) namespace and the local name of the compilation unit.

Ext JS organizes compilation units in *packages*. A package has a name and can have a namespace, which is used as a prefix for all fully-qualified names of its compilation units. The package name and namespace prefix need not to be the same identifier.

As TypeScript is an extension of ECMAScript, it uses the ECMAScript module system. Since ES5, any source file that contains `import`s and/or `export`s is a *module*. In `import` directives, modules are references by file path without extension. This file path may either be relative to the current source file, starting with `./` or `../`, or it refers to an *npm package name* and then specifies the relative path within that package.

5.2.3.1 Imports

Ext TS maps TypeScript *default* imports, consisting of npm package name plus relative path, to Ext JS fully-qualified names, consisting of Ext namespace and local name, separated by a dot.

Note that whenever you see *named* imports in Ext TS, the source must be a non-Ext JS, standard ECMAScript module. For example, the Jangaroo Runtime utility functions are named exports and as such require named imports (see complete class example below).

For backwards-compatibility, for each npm package, the Ext package name and namespace prefix to use can be customized via the file `jangaroo.config.js`, which must be located next to the corresponding `package.json` file, like so:

```
const { jangarooConfig } = require("@jangaroo/core");

module.exports = jangarooConfig({
  type: "code",
  sencha: {
    name: "com.coremedia.blueprint__blueprint-forms",
    namespace: "com.coremedia.blueprint.studio",
```

```
  },
});
```

This example shows the (shortened) `jangaroo.config.js` file of npm package `@coremedia-blueprint/studio-client.main.blueprint-forms`, located in `apps/main/blueprint/blueprint-forms/`, taking care of the corresponding Ext package being called `com.coremedia.blueprint__blueprint-forms` (like in *CoreMedia Content Cloud* v10) and all Ext classes in this package using the Ext namespace prefix `com.coremedia.blueprint.studio`.

For example, in Ext TS, the class `util/ContentInitializer.ts` in the `blueprint-forms` package must be imported as follows:

```
import ContentInitializer from
"@coremedia-blueprint/studio-client.main.blueprint-forms/util/ContentInitializer";
```

In Ext JS, using the `namespace` configuration from above, it is then "required" or "used" like so:

```
  requires: ["com.coremedia.blueprint.studio.util.ContentInitializer", ...],
  ...
```

which is exactly the former fully-qualified Ext JS name of that class in *CoreMedia Content Cloud* v10 (2107).

5.2.3.2 Export

In Ext JS, each compilation unit contains exactly one declaration that is visible from the outside, usually a class. In TypeScript modules, it is possible to export multiple identifiers, but there is a *default export*. So when converting code to Ext JS, it is straight-forward to use this default export to export the primary declaration of the compilation unit.

While it is possible to combine the declaration and the (default) export of a class, the code style in the Blueprint workspace is to separate them, because later you'll see cases where TypeScript's *declaration merging* is used, which would lead to redundant export directives. So the recommended code style is to always end each source file with the default export, like in this example class:

```
import { is, mixin } from "@jangaroo/runtime";
import SuperFoo from "./SuperFoo";
import IFoo from "../api/IFoo";

class Foo extends SuperFoo implements IFoo {
  static readonly FOO: any = "FOO";
  foo: string;
  #bar: number;
```

```
  constructor(newBar: number) {
    super();
    this.#bar = newBar;
  }

  get bar(): number {
    return this.#bar;
  }

  set bar(value: number) {
    this.#bar = value;
  }

  isAFoo(obj: any): boolean {
    return is(obj, IFoo);
  }

  protected hook(): boolean {
    return false;
  }
}

mixin(Foo, IFoo);

default export Foo;
```

Example 5.4. Using the default export for Ext TS classes

5.2.4 Mixins

Ext JS allows mixins to achieve multiple inheritance between classes. Since neither ECMAScript nor TypeScript supports mixins out of the box, we had to find some way to represent them.

5.2.4.1 Mixins in TypeScript

In ECMAScript/TypeScript, a class can only extend one other class, but in TypeScript, it can implement multiple interfaces.

To understand how mixins work, it helps to know that in TypeScript, a class consists of its runtime JavaScript *value* and a *type*, which is only relevant for type checking, that is at compile-time. The class identifier represents both aspects. Depending on context, it is clear whether the value, the type, or both are meant. When a class `A` extends another class `B`, in the `extends` clause, `B` refers to both the value (JavaScript class `A` will at runtime extend JavaScript class `B`) and the type (TypeScript type `A` will at compile time be a subtype of type `B`). When using a class identifier behind a colon or in the `implements` clause of a class, only its type aspect it used. This allows to use a *class* in an `implements` clause! This equals implementing the interface extracted from that class.

Another TypeScript concept that is relevant here and closely related is *declaration merging*. In TypeScript, a type with the same identifier can be declared multiple times, and all declarations are merged. Since a class declares a value and a type, and an interface only declares a type, you cannot declare the same class twice, but you can declare a class and an interface using the same identifier. What happens is that the interface extracted from the class is merged with the additionally declared interface. This is how we tell TypeScript *not* to complain about the class not implementing the additional interface methods from the mixin. We call such an interface a *companion interface* of the class, as it comes together with the class and adds more declarations (the ones implemented in the mixin).

Using these ingredients, we can declare mixins in TypeScript as follows.

As in Ext JS, a mixin is a common TypeScript class. A mixin client class implements the interface automatically extracted from the mixin class, in other words, it directly *implements the mixin class*.

But that does not suffice: We have to specify that we do not only want to use the interface, but also want to mix in the mixin's methods at runtime. You learned about the `mixin()` utility function in the interface chapter. Maybe now it becomes clear why it is called like that: it can do more than just mix in the identity of an interface: it actually mixes in any class with all its members into the client class.

Last thing to do is again to prevent the type checker from complaining about missing implementations of the mixin interface, since it does not know about the mixin magic. We declare a *companion interface* of the mixin client class and let that extend the mixin class interface. We could even leave out the `imple ments` clause of the mixin client class itself. However, to emphasize what's going on (and to help some IDEs that don't really support declaration merging completely), we recommend specifying both clauses.

The following TypeScript code is an example of how an Ext mixin looks like in Ext TS.

```
// ./acme/MyMixin.ts
class MyMixin {
  #mixinConfig: string = "";

  get mixinConfig(): string {
    return this.#mixinConfig;
  }

  set mixinConfig(value: string) {
    this.#mixinConfig = value;
  }

  doSomething(): number {
    return this.#mixinConfig.length;
  }
}
```

```
export default MyMixin;


// ./MixinClient.ts
import { mixin } from "@jangaroo/runtime";
import Component from "@jangaroo/ext-ts/Component";
import MyMixin from "./acme/MyMixin";

class MixinClient extends Component implements MyMixin {
  constructor(config: any = null) {
    super(config);
    this.doSomething();
  }
}

// companion interface, so we don't need to re-declare all mixin members:
interface MixinClient extends MyMixin {}

// use Jangaroo utility method to perform mixin operation:
mixin(MixinClient, MyMixin);

export default MixinClient;
```

Example 5.5. Ext Mixin in TypeScript example

5.2.5 Using the Ext Config System

A major part of the Ext JS infrastructure deals with components, plugins, actions, and other classes that have in common that they use the Ext Config system.

5.2.5.1 How the Ext Config System Works

The Ext Config system is quite a beast, but we'll try to keep things as simple as possible here.

Simple Ext JS Config System (Version 3.4)

When we started with Ext JS 3.4, Configs were a simple concept: To specify the properties of some object to create, plain JavaScript object literals are used – a bit more than JSON, because their values may be more complex. These objects are passed around and eventually used to derive a class to instantiate, in Ext 3.4 based on their xtype property. The class constructor is then called with the Config object and essentially "applies" (copies) all properties onto itself (this).

For example, you could specify a button with a label as a config object and then let Ext create the actual Ext.Button instance from that Config:

```
var buttonCfg = {
  xtype: "button",
  label: "Click me!"
};
var button = Ext.create(buttonCfg);
console.log(button.label); // logs "Click me!"
```

Example 5.6. Ext Config example

So in Ext 3.4, Configs were nothing but properties/fields of the target class which were "bulk applied" through a JSON-like object.

Advanced Ext JS Config System

Things became more complicated with the new class and Config system intro-duced with Ext 4 (CoreMedia skipped Ext 4 and 5, but upgraded directly to Ext 6, later to 7).

The new Ext Config system supports additional indicators of which class to instantiate. The three different special Ext properties available to specify the target class are:

xtype The "classic" class hint. Each Ext class may specify a unique `xtype`, which is registered and referenced here to identify the class to instantiate. This indirection is meant to separate usage and implementation (a bit).

alias When Ext extended their Config System to more than just components, they thought it would make sense to introduce prefixes for the different groups of classes. Components use `widget.`*<xtype-value>*, plugins use `plugin.`*<type-value>*, GridColumns use `gridcolumn.`*<type-value>*. The `type` property used for that purpose before introducing `alias` has been deprecated.

xclass Introduced last, this is the most straightforward way to specify the target class: Just give its fully-qualified name! Unfortunately, this property does not work everywhere in Ext's Classic Toolkit (the one CoreMedia Studio uses), so if a class has an `xtype` / `alias`, you should better use that, or even better, *all* possible meta-properties the class offers.

Configs now can be declared explicitly for an Ext class and then trigger some magic: For every Config property `foo`, Ext generates methods `getFoo()` and `setFoo(value)`. Such Configs are called *bindable*, because they can be bound to a model that would read and write their value through these methods.

Bindable Configs

Note that bindable Configs in Ext JS do *not* use ECMAScript accessors, but "normal" methods, as Ext 4 came out when browser support for accessors was

not yet mainstream. Sencha never managed to update the Ext JS Config system to "real" accessors.

Only for sake of completeness, it should be mentioned that the generated `setFoo(value)` method looks for two optional "hook" methods that allow the following:

- Transform the value *before* it is stored:

```
updateFoo(value) { return transform(value) }
```

- Trigger side-effects *after* the value has been set:

```
applyFoo(value, oldValue) { /* side effect */ }
```

Since these hook methods do not add much value, but rather make code harder to read, we do not recommend using them. Rather, simply provide a custom implementation of `setFoo()`, calling `super.setFoo()` if and where needed.

As an example, here is how you could define a Config `text`, prevent anything that is not a `string` from being set into that Config (at least not when everybody uses the `setText(value)` method), and update the DOM of your component whenever the text is changed:

```
Ext.define("acme.Label", {
  extend: "Ext.Component",
  xtype: "acme.label",
  config: {
    text: ""
  },
  setText(value) {
    this.value = typeof value === "string" ? value : value ? String(value)
: "";
    if (this.rendered) {
      // update my DOM node with 'this.value' ...
    }
  }
};

var label = Ext.create({ xtype: "acme.label", text: "Hi!"});
label.setText(null);
console.log(button.getText()); // logs the empty string (""), not "null"
```

Example 5.7. Ext JS Bindable Configs

5.2.5.2 Using the Ext Config System in TypeScript

This section describes the TypeScript syntax for using the Ext Config system.

Declaring the Config Type in TypeScript

In TypeScript, each class using the Ext Config system needs an additional interface that describes its Config options. The design goal for the representation of this Config interface is to only declare and document Config properties once, although they usually re-appear on the class itself. Also, we need to distinguish simple Configs and advanced ("bindable") Configs. Last but not least, Config objects usually only specify a subset of all possible properties.

Here, the TypeScript utility types `Pick` and `Partial` come in handy. `Pick` allows to pick a list of specified member declarations from another type. `Partial` creates a new type that is exactly like the source type, only that all members are optional, as if they were declared with the `?` modifier.

All Config properties are declared in the class itself. "Simple" Config properties are just properties with an optional default value, while bindable Config properties must be specified as an *accessor* pair, typically encapsulating a private field. The additional Config type is then declared as an interface using the partial type of picking those Config properties from the class. By convention, we name this interface like the class, suffixed with `Config`.

```
import Component from "@jangaroo/ext-ts/Component";

interface MyClassConfig extends Partial<Pick<MyClass,
  "configOption1" |
  "configOption2">> {
}

class MyClass extends Component {
  /**
   * Simple Config property.
   */
  configOption1: string = "foo";

  #configOption2: number[] = [42];
  /**
   * Bindable Config property.
   */
  get configOption2(): number[] {
    return this.#configOption2;
  }
  set configOption2(value: number[]) {
    this.#configOption2 = value;
  }

  constructor(config: MyClassConfig) {
    super(config);
  }
}

export default MyClass;
```

Example 5.8. Simple and Bindable Config Properties in TypeScript

To also *export* the additional interface, the most straightforward option seemed to be using a named export. But this has disadvantages:

- When a class declares no additional Config properties, but just reuses the Config type of its superclass, it would have to re-export the super Config type.
- When using both the class and its Config type, you need two import identifiers, which is especially cumbersome when there is a name clash, because you need to rename both.

So we decided to assign the Config type to the class, which can be done in TypeScript by declaring a "virtual" class member, and use the name `Config` for it.

```
interface MyClassConfig ...

class MyClass ... {
  declare Config: MyClassConfig;
  ...
}
```

Example 5.9. Declaring Config type as virtual class member

This allows to access the Config type by importing the class and then use the utility type called `Config` (imported from `@jangaroo/runtime/Config`). As this pattern is followed by all classes using the Ext Config System, also the Ext TS declarations of all framework components, we can complement the example by extending the superclass Config type. The pattern should also be used to refer to the Config type for the constructor parameter.

```
import Config from "@jangaroo/runtime/Config";
import Component from "@jangaroo/ext-ts/Component";

interface MyClassConfig extends Config<Component>, Partial<Pick<MyClass,
  "configOption1" |
  "configOption2">> {
}

class MyClass extends Component {
  declare Config: MyClassConfig;

  //...

  constructor(config: Config<MyClass>) {
    super(config);
  }
}

export default MyClass;
```

Example 5.10. Extending superclass Config type

Specifying Strictly Typed Config Objects in TypeScript

Having a Config type allows to specify typed Config objects in TypeScript by using a *type assertion* (we use the $<...>$ syntax here rather than the `as` keyword to place the type in front), taking advantage of type checks and IDE support. The following example shows that type errors are detected for existing properties, however, arbitrary undeclared properties can still be added without a type error:

```
import Config from "@jangaroo/runtime/Config";
import MyClass from "./MyClass";

...
    const myClassConfig = <Config<MyClass>>{
        // inherited from Config<Component>:
        id: "4711",
        // MyClass Config property:
        configOption1: "bar",
        // an undeclared property does *not* lead to a type error:
        untyped: new Date(),
        // type error: '"42" is not assignable to type number[]':
        configOption2: "42",
    };
...
```

Example 5.11. TypeScript detecting type errors for existing properties

Being able to use undeclared properties without warning is not desirable. Fortu‐nately, in TypeScript, it is possible to specify the signature of a generic Config type‐check function to prevent using untyped properties. You get access to this function through the same imported `Config` identifier (remember, TypeScript allows to declare a *value* and a *type* with the same identifier).

```
import Config from "@jangaroo/runtime/Config";
import MyClass from "./MyClass";

...
    // first 'Config' is the utility type, second the utility function:
    const myClassConfig: Config<MyClass> = Config<MyClass>({
        // inherited from Config<Component>:
        id: "4711",
        // MyClass Config property:
        configOption1: "bar",
        // an undeclared property now *does* lead to a type error:
        untyped: new Date(),
        // type error: '"42" is not assignable to type number[]':
        configOption2: "42",
    });
...
```

Example 5.12. Preventing use of untyped properties

We just added a type annotation to `myClassConfig` for clarity. You can omit it and leave that to TypeScript's *type inference*.

The first `Config` (after the colon) is the utility type from above, but the second `Config` is a call to the generic Config type-check function, which takes as argument a Config object of the corresponding Config type `MyClassConfig` and returns exactly that Config object.

Since TypeScript is more strict when checking the type of function arguments than when a type assertion is used, this solution prevents accidental access to untyped properties. In the example, the property `untyped` would now be marked as an error, because it does not exist in the Config type.

Creating Ext Config Objects in TypeScript

Now, we have strictly typed Config objects, but they lack `xclass`/`alias`/`xtype` properties, which Ext uses to determine the target class when instantiating a Config object later (see Section "Advanced Ext JS Config System" [48]). Since we do not want to specify the Config type twice, once as a TypeScript type and once as a utility function that add the target class indicator, we combine both into one.

To this end, the generic `Config` function supports an overloaded signature which takes as first argument the target class which must define a Config type and as second (optional) argument a Config object of the corresponding Config type, and returns that Config object complemented by `xclass`/`alias`/`xtype` properties taken from the class.

With this new usage of the `Config` function, you can now create Ext Config objects like so:

```
import Config from "@jangaroo/runtime/Config";
import MyClass from "./MyClass";

...
    // use Config function with target class + config object:
    const myClassConfig: Config<MyClass> = Config(MyClass, {
      // inherited from Config<Component>:
      id: "4711",
      // MyClass Config property:
      configOption1: "bar",
      // an undeclared property now *does* lead to a type error:
      untyped: new Date(),
      // type error: '"42" is not assignable to type number[]':
      configOption2: "42",
    });
...
```

Example 5.13. Create Ext Config objects with Config function

As you can see, the syntax is very similar to using `Config` for a strict type-check. The crucial difference is that `MyClass` is not a type parameter (which is just a compiler hint and only relevant for type checking), but an argument of the function call. The class reference is needed at runtime to determine the `xclass` etc. and add it to the config object. Although this `Config` signature still has a type parameter, is should never be necessary to specify it explicitly, just leave it to TypeScript's type inference.

If you use a class as first argument, but leave out the second one, the `Config` function returns an empty Config object with just the target class marker (`xclass`, `xtype`, …). This comes in handy for simple components like `Config(Separator)`. TypeScript automatically distinguishes the two one-argument usages of `Config` by overloaded signatures, one with a Config object, the other with a class that declares a Config type.

In the rare case that you need to instantiate the "real" object from a given Config object, you have different options:

```
import { cast } from "@jangaroo/runtime";
import Ext from "@jangaroo/ext-ts";

    // Alternatives
    // ============

    // using constructor directly
    // xclass of Config object is ignored:
    const myClassInstance: MyClass = new MyClass(myClassConfig);

    // using Ext.create() with class and Config object
    // xclass of Config object is ignored:
    const myClassInstance: MyClass = Ext.create(MyClass, myClassConfig);

    // using Ext.create() with Config object only, type on left-hand side
    // must repeat target class, but incompatible class and Config type would
 be reported:
    const myClassInstance: MyClass = Ext.create(myClassConfig);

    // using Ext.create() with type parameter and Config object
    // must repeat target class, but incompatible class and Config type would
 be reported:
    const myClassInstance = Ext.create<MyClass>(myClassConfig));
```

Example 5.14. Instantiate object from Config object

The first two usages are when you know which target class to create, anyway, so you would construct `myClassConfig` without any `xclass`, but just use the strict Config type function.

The latter two usages are when the Config object might have its own `xclass` of some `MyClass` subclass. `Ext.create()` uses the `xclass` to instantiate the corresponding class, and the resulting object is type-compatible with `MyClass`. This is the kind of mechanism used by `Ext.Container` to instantiate its `items`.

But the best thing is, that if you want to create an instance directly, you can do so in a strongly typed fashion with full IDE support using an inline, ad-hoc Config object, which does not need any `Config` usage:

```
const myClassInstance: MyClass = new MyClass({
  id: "4711",
  configOption1: "bar",
  configOption2: [42, 24]
});
```

Example 5.15. Inline ad-hoc Config object

In other words, the difference between creating a Config object and creating an instance is just using `Config(MyClass, ...)` versus using `new MyClass(...)`.

Note that when creating the component tree, you usually use Config objects, while certain elements like `Action`s require instantiation. Any Ext `Container` takes care that its `items` are instantiated if they are Config objects, but for example a `Component`'s `baseAction` property does not support Config objects. This is reflected in the Ext TS API by declaring

`Container.items: (Component | Config<Component>)[],`

but

`Component.baseAction: Action` (not `Config<Action>`).

In other words, inadvertently using a Config object for a `baseAction` results in a type error.

Merging Config Objects

When receiving a Config object, the typical things a constructor does is:

- Apply the received `config` on its own Config defaults
- Hand through the resulting Config to its super constructor

In TypeScript code, this could be done like this:

```
constructor(config: Config<MyClass>) {
  super(ConfigUtils.apply(Config(MyClass, {
    id: "4711",
    configOption1: "bar",
    configOption2: [42, 24]
  }), config));
}
```

Example 5.16. Typical work of constructor done in TypeScript

However, there is a special utility class named `ConfigUtils` that helps implementing a specific merge logic. For array-valued properties, it should be possible to, instead of replacing the whole array, append or prepend to the existing value. The concrete use cases where this often makes sense are Ext component's `plugins` and `items` properties. So at least if your class has any array-valued properties, you should use the following in your constructor:

```
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";

  //...

    constructor(config: Config<MyClass>) {
      super(ConfigUtils.apply(Config<MyClass>({
        id: "4711",
        configOption1: "bar",
        configOption2: [42, 24]
      }), config));
    }

  //...
```

Example 5.17. Using ConfigUtils utility class

Any client using such a component can then use the following:

```
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";

  //...

    Config(MyClass, {
      id: "4711",
      configOption1: "bar",
      ...ConfigUtils.append({
        configOption2: [12]
      }
    }), config));
  }

  //...
```

Example 5.18. Component with utility class in client

The resulting value of `configOption2` after merging via `ConfigUtils.apply()` will be `[42, 24, 12]`. There is an analogous utility method `ConfigUtils.prepend()`. Both return an object, handing through the given property, complementing it by an internal marker property that specifies where to insert the value into the previous value. To "lift" these properties into the surrounding object literal, the spread operator `...` is used.

5.3 Client-side Model

The *CoreMedia Studio* user interface is implemented following the Model-View-Controller (MVC) pattern. The widgets provided by Ext JS are considered the view, whereas Ext JS actions take the role of controllers. To deal with the model layer efficiently, the Studio framework provides the key concepts of *beans* and *value expressions*.

MVC pattern

A bean is an object that aggregates a number of properties, where property values may be arbitrary JavaScript objects, including arrays or even other beans. Beans are capable of sending events when one of their properties changes, making it possible to update the view components dynamically when a bean changes.

Beans

While wiring up a UI component property to a plain bean property is mostly straightforward and can be as simple as connecting a button label to a simple string bean property, you will inevitably run into situations where you need to "compute" a UI component property based on complex model state that might span different bean properties, or even completely separate beans.

Simple and complex wiring

Both the simple and the complex case can be conveniently solved using *value expressions*, which can encapsulate the computation of mutable values on the bean level. A frequently used value expression takes a start bean and follows property references from beans to beans to arrive at a target bean or value. Value expressions, too, generate events whenever their value changes, and you can attach event listeners to them to dynamically update the UI.

value expressions

While it is possible to hand code the view response to model changes, you are encouraged to make use of the *Studio SDK's* predefined Ext JS plugins. Plugins are available for setting UI component properties, selections, displayed values, and so on. All these plugins transfer state between a value expression and an Ext JS component, sometimes in both directions ("bidirectional").

Using Ext JS plugins

For experienced Ext JS developers, it may seem strange that an explicit model in the form of beans is used, instead of widget-internal state as an implicit model. However, the chosen approach allows for a more consistent representation of the model. By wrapping remote data sources as beans, a uniform access layer throughout *CoreMedia Studio* is achieved. In other words, from a developer's perspective, it is transparent whether model state is wired up to remote (server-side) or local (client-side) data. This also means that as a developer, you don't need to manually write code to make Ajax calls in order to update server-side data - you make sure that your model is properly wired up to your UI, and the framework takes care of the details for you.

Uniform access layer

For details about the TypeScript classes mentioned in the following sections, refer to the TypeScript documentation as found on the Studio release page, available at the CoreMedia download section.

5.3.1 Beans

Beans are objects with an arbitrary number of properties. Properties can be updated, generating events for each change. The name "bean" originates from the concept of Java Beans, which are also characterized by their properties and event handling capabilities. Unlike Java beans, the Studio beans do not enforce a strict typing and naming policy, whereby each property must be represented by individual getter and setter functions. Instead, untyped generic methods for getting and setting properties are provided. Specific bean implementations are allowed to add typed accessors, but are not required to do so.

All beans implement the abstract class `@coremedia/studio-client.client-core/data/Bean`. Remote beans, which encapsulate server-side state, conform to the more specific class `@coremedia/studio-client.client-core/data/RemoteBean`. Refer to Section 5.3.2, "Remote Beans" [60] for more details about these concepts. At first, the more generic `Bean` class is described, which is agnostic of a potential backing by a remote store.

Remote beans

Properties

Individual properties of any bean can be retrieved using the `get(property Name)` method, which receives the name of the property as an argument. Arbitrary objects and primitive values are allowed as property values. The set of property names is not limited, but it is good practice to document the properties and their semantics for any given bean. If non-string values are used as property names, they will be converted to a string.

Retrieving bean properties

Beans may reference other beans. For example, the `Content` bean contains a property `properties` that contains a bean with schema-specific properties, whereas the `Content` bean itself contains the predefined content metadata, such as creation and publication date, which are defined implicitly for all CoreMedia content objects.

By calling `set(propertyName, value):boolean`, a property value can be updated. The method returns `true` if (and only if) the bean was actually changed. Generally, the new value is considered to equal the old value if the `===` operator considers them equal. There are a number of exceptions, though:

Updating properties

- Arrays are equal if they are of the same length and if all elements are equal according to the bean semantics. That is, arrays are treated as values and not as modifiable objects with state.
- Date and Calendar values are equal if they denote the same date and time, with time zone information taken into account.
- Blobs as stored in the CMS are equal if they contain the same content with the same content type. As long as the blobs are not fully loaded from the server, a conservative heuristic is used that considers the blobs equal if it is known that they will ultimately represent the same value when loaded.

By using the method updateProperties(newValues), you can set multiple properties at once. The argument object must contain one TypeScript property per bean property to be set. Bean properties not mentioned in the argument object are left unchanged. Consider the following example:

```
bean.updateProperties({
    a: 1,
    b: ["a", "b"],
    c: anotherBean
});
```

Example 5.19. Updating multiple bean properties

The above code sets the three properties a, b, and c simultaneously, but the property d keeps its previous value if it was set. Apart from convenience, the main difference compared to three calls like bean.set("a", 1) is that events will be sent only after all properties have been updated. This can be useful when you want to update a bean atomically.

Calling toObject() on a bean will return a snapshot of the current bean state in the form of an object that contains one TypeScript property per bean property.

Events

Property event listeners for a single property are registered with addProper tyChangeListener(propertyName, listener) and removed with removePropertyChangeListener(propertyName, listener). The listener argument must be a function that receives a simple argument of type @coremedia/studio-client.client-core/data/Proper tyChangeEvent. This event object contains information about the bean, the changed property and the old and the new value.

Register and remove property event listener

A listener function registered with addValueChangeListener(listener) receives events for all properties of the respective bean. When multiple properties are updated, the listener receives one call per updated property. Such listeners can be removed by calling removeValueChangeListener(listener).

Listener for all property events

For beans, events are dispatched synchronously, before the update call returns.

Bean State

Beans, especially remote beans, may enter different states. The possible states are enumerated in the class `@coremedia/studio-client.client-core/data/BeanState`. The method `getState()` provides the current state of the bean. State changes are also reported to all listeners. The event object provides the old and the new bean state.

The possible states are:

- `UNKNOWN`: The bean is still being set up.
- `NON_EXISTENT`: The bean represents an entity that does not exist. Typically, the entity existed at one time in the past, but has been destroyed.
- `UNREADABLE`: The bean represents an entity that exists, but authorization to access it is missing.
- `READABLE`: The bean can be accessed without restrictions.

Local beans are always in state `READABLE`.

Singleton Bean

The interface `IEditorContext`, whose default instance can be accessed as the package field `@coremedia/studio-client.main.editor-components/sdk/editorContext`, provides the method `getApplicationContext()`, which returns a singleton local bean. This bean is provided as a starting point for navigating to other singletons and for sharing system-wide state. Individual APIs document the properties of the singleton bean that are set by that API. Be careful when adding custom properties and avoid name clashes.

5.3.2 Remote Beans

A remote bean encapsulates the state of a server-side object in the client-side application. Its properties are loaded on demand - most commonly by invoking the `RemoteBean#load` or `RemoteBean#invalidate` methods, respectively.

The SDK provides more specialized subclasses of remote beans, for example beans of type `Content`, which represents CoreMedia CMS content items and folders.

Bean values may change when the remote bean is invalidated and reloaded. Some remote beans, in particular content object and workflow objects, are invalidated automatically after server-side changes.

In the class `@coremedia/studio-client.client-core/data/RemoteBean`, the method `getUri()` provides access to the URI from which its state is loaded. Its sibling method `getUriPath()` returns a URI path relative to the base URI of the remote service from which the bean is loaded. The latter value provides a more concise and still unique identification of the remote bean. There is only ever one remote bean for each URI path.

By calling `load(AnyFunction)`, the bean is instructed to load its properties, using an asynchronous HTTP request. Note that this is transparent to the developer and you never need to manually construct an XHR.

Asynchronous HTTP request

Once the call has returned, an optional callback function is invoked, indicating the new state of the bean. A remote bean is also loaded as soon as any of its properties are read. However, the bean will report properties as `undefined` initially and fire an event as soon as the property is updated to a different value after loading.

To reload the bean state, call the method `invalidate(AnyFunction)`, which takes an optional callback function which is invoked after all properties have been reloaded.

Please note that computed bean properties may still be `undefined` when the callback functions are invoked. For example, the `Content` bean contains a property `path` that requires all the content's parents to be loaded recursively. Although the `Content` bean itself might be completely loaded, the `path` property remains `undefined` until all the content's parents have finished loading. Listen to the change events for the computed property to find out when the property is ready or use a `ValueExpression`. See Section 5.3.6, "Value Expressions" [65] for details.

Listen to events until property is ready to use

When properties of a remote bean are set, they are eventually written back to the server. The remote bean may bundle any number of writes before making its update request. At least all updates made in the same JavaScript execution without an intervening `setTimeout()` are bundled in one write. You can call the method `flush(AnyFunction)` to ensure that a callback function is invoked after the update call for all previously updated properties has completed, either successfully or with an error. The callback function can determine the success status of a flush call by its single argument, a `FlushResult` object. This object also carries a reference to the flushed bean and, in the case of an error, to a `RemoteError` object indicating the source of the problem.

Update properties on server

Remote beans may be unreadable or even nonexistent, which is indicated by the method `getState()`. A bean's state can be observed by usual property change listeners (see previous section), since bean state changes trigger property

change events and report the current state (see `PropertyChan geEvent#newState`). Working with remote beans generally requires more attention to error conditions than working with local beans.

5.3.3 Issues

CoreMedia Studio has built-in support for server-side validation of content objects. You can leverage the validation framework for your own (non CMS) data resources, but for content objects managed in the *CoreMedia Content Server*, the framework already offers convenient support (see Section 5.4.2, "Content" [76] for a general description of the Studio Content API.)

Server-side validation always works on values already saved (persisted) - in other words, a validator will never prevent the user from saving data, so that the risk of data loss is minimal. You can however set up *Studio* to prevent the user from approving or checking in documents that have validation issues with severity `ERROR` (see Section 9.23.1.5, "Tying Document Validation to Editor Actions" [265] for details on how to configure this).

The client can ask the server to compute *issues* of an entity (most commonly Content), where they become accessible as a `@coremedia/studio-client.client-core/data/validation/Issue` object. Once received, the client can do things like highlight a property field that contains an invalid value, or open a dialog. *Studio* offers built-in support for marking standard property fields invalid, and offers the user a convenient interface to step through and correct detected validation issues in one go.

Getting issues from the server

The issues object provides access to individual `Issue` objects through a number of methods:

- `getAll()` returns all issues of the entity in a single array.
- `getByProperty()` returns a sub bean whose properties match the properties of the entity. Each property contains an array of issues that affect exactly that property.
- `getGlobal()` returns an array of issues that do not affect a specific property, but that describe the state of the entity as a whole. A common example for this is a validator that checks for the correct folder path of a content item - you could set up a validator to raise a `WARNING` when a content item is created in a folder that is not appropriate for its type, for example.

An issue links back to its entity by means of the `entity` property. The `severity` property indicates a level of `"INFO"`, `"WARN"`, and `"ERROR"`. You can freely define the severity level for any validator. An issue may belong to one or more categories. Issues are grouped according to their category when displayed in the client.

The property `property` stores the name of the property whose value causes the issue. If `null`, this indicates a global issue that affects the entity as a whole, rather than one of its properties. In the property `code`, each issue stores a string identifier indicating the type of issue detected. Applications are expected to localize this identifier as needed. Depending on the code, the array property `arguments` might store additional data in a specific layout.

The issue code identifiers depend on the type of entity that has been validated. In fact, each server-side validator may introduce its own code and you have to refer to the documentation of the validators for details. Some validators allow you to configure the error code that they report. In custom validators, you can also pass on additional ("runtime") information describing the error in more detail, and use this additional information to present user-friendly descriptions of the problem in the UI. See Section 9.23.1, "Validators" [252] for details.

Error codes and further information

5.3.4 Operation Results

Complex remote operations typically allow you to specify a callback function. The callback function is called after the operation has completed, either successfully or unsuccessfully. This allows you to postpone subsequent steps until a remote resource is in a defined state again.

Callback functions

Callback functions often receive an *OperationResult* argument. Such objects indicate in their `success` attribute whether the attempted operation was successful. In the case of errors, the attribute `error` points to a `RemoteError` object further detailing the problems. Individual operations may return richer result objects. For example, the previous section already mentioned the `FlushResult`, which also references the modified bean in the `remoteBean` property.

5.3.5 Model Beans for Custom Components

When creating complex GUI components, it is good practice to provide an abstract model in the form of a bean to back the view. In combination with `ValueExpression`s this allows an easy dependency tracking between the different widgets of a complex GUI. The best practice here, is to lazy initialize the model bean through a getter method. The bean itself is created there using the call `beanFactory.createLocalBean()` upon first access:

```
import Config from "@jangaroo/runtime/Config";
import Panel from "@jangaroo/ext-ts/panel/Panel";
```

```
import Bean from "@coremedia/studio-client.client-core/data/Bean";
import beanFactory from
"@coremedia/studio-client.client-core/data/beanFactory";

class MyComponent extends Panel {
  #model: Bean;

  constructor(config: Config<MyComponent>) {
    super(config);
    this.#initModel(config);
    //...
  }

  getModel(): Bean {
    if (!this.#model) {
      this.#model = beanFactory._.createLocalBean();
    }
    return this.#model;
  }

  #initModel(config: Config<Panel>): void {
    this.getModel().set("myProperty", config.title);
    //...
  }

  //...
}

export default MyComponent;
```

Example 5.20. Model bean factory method

The model can then be used inside a `ValueExpression` and be bound to components:

```
import Config from "@jangaroo/runtime/Config";
import TextField from "@jangaroo/ext-ts/form/field/Text";
import BindPropertyPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/BindPropertyPlugin";
import ValueExpressionFactory from
"@coremedia/studio-client.client-core/data/ValueExpressionFactory";

// inside items of some container:
Config(TextField, {
  itemId: "...",
  plugins: [
    Config(BindPropertyPlugin, {
      bindTo: ValueExpressionFactory.create("myProperty", this.getModel()),
      bidirectional: true,
    }),
  ],
})
//...
```

Example 5.21. Model bean access

Here a text field is configured to display the value of a property, but of course arbitrary widgets can be used.

In fact, the property is not directly accessed by the plugin, but indirectly through a value expression that, in this case, simply evaluates to a property value. Value expressions will be discussed in the next section.

5.3.6 Value Expressions

The class `@coremedia/studio-client.client-core/data/Value Expression` describes objects that provide access to a possibly mutable value and that notify listeners when the value changes. They may also allow you to receive a value that can then become the next value of the expression. Value expressions may be as simple as defining a one-to-one wiring of a widget property to a model property, but they may encapsulate complex logic that accesses many objects to determine a result value. As an application developer, you can think of value expressions as an abstraction layer that hides that potential complexity from you, and use a common, simple class when wiring up UI state to complex model state.

The Studio SDK offers the following primary implementations of the abstract `ValueExpression` class. You can use the factory methods from `@coremedia/studio-client.client-core/data/ValueExpressionFactory` to create a `ValueExpression` programmatically from TypeScript.

- `PropertyPathExpression`. This is meant to be used in simple scenarios, where you want to attach a simple bean property to a corresponding widget property. It starts from a bean and navigates through a path of property names to a value. Long paths can be separated with a dot. You can obtain this value expression flavor using `ValueExpressionFactory#create(expression, bean)`.
- `FunctionValueExpression`. Use this in scenarios where your UI state requires potentially complex calculations on the model, using multiple beans (remote or local). This value expression object wraps an TypeScript function computing the expression's value. When a listener is attached to the returned value expression, the current value of the expression is cached, and dependencies of the computation are tracked. As soon as a dependency is invalidated, the cached value is invalidated and eventually a change event is sent to all listeners (if the computed value has actually changed). You can use `ValueExpressionFactory#createFromFunction(AnyFunction, ...args)` to create this flavor. See below for details on how to use `FunctionValueExpression`s.

In many cases, you can use the facilities provided by plugins without ever constructing a value expression programmatically. Nevertheless, value expressions are a vital part of the Studio SDK's data binding framework, so it is helpful to understand how they work.

Values

The method `getValue()` returns the current value of the expression. How this value is computed depends on the type of value expression used. Like bean properties, value expressions may evaluate to any TypeScript value.

When a value expression accesses remote beans that have not yet been loaded, its value is `undefined`. Getting the value or attaching a change listener (see below) subsequently triggers loading all remote bean necessary to evaluate the expression. If you need a defined value, you can use the `loadValue(AnyFunction)` method instead. The `loadValue` method ensures that all remote beans have been loaded and only then calls back the given function (and, in contrast to change listeners, only once, see below) with the concrete value, which is never `undefined`.

Be sure that the value is not undefined

Like remote beans, value expressions may turn out to be unreadable due to missing read rights. In this case, `getValue()` returns `undefined`, too, and the special condition is signaled by the method `isReadable()` returning `false`.

Events

A listener may be attached to a value expression using the method `addChangeListener(listener)` and removed using the method `removeChangeListener(listener)`. The listener must be a function that takes the value expression as its single argument. The listener may then query the value expression for the current value.

Contrary to bean events, value expression events are sent asynchronously after the calls modifying the value have already completed. The framework does however not guarantee that listeners are notified on all changes of the value. When the value is updated many times in quick succession, some intermediate values might not be visible to the listener.

The listener is also notified when the readability of the value changes.

As long as you have a listener attached to a value expression, the value expression may in turn be registered as a listener at other objects. To make sure that the value expression can be garbage collected, you must eventually remove all listeners added to it.

A common pattern when adding a listener to a value expression involves an upfront initialization and subsequent updates on events:

```
import { bind } from "@jangaroo/runtime";
import Config from "@jangaroo/runtime/Config";
```

```
import Panel from "@jangaroo/ext-ts/panel/Panel";
import ValueExpression from
"@coremedia/studio-client.client-core/data/ValueExpression";
import ValueExpressionFactory from
"@coremedia/studio-client.client-core/data/ValueExpressionFactory";

class MyComponent extends Panel {
  #valueExpr: ValueExpression<number>;

  constructor(config: Config<MyComponent>) {
    super(config);
    this.#valueExpr = ValueExpressionFactory.create<number>(/*...*/);
    this.#valueExpr.addChangeListener(bind(this, this.#valueExprChanged));
    this.#valueExprChanged(this.#valueExpr);
  }

  protected override onDestroy(): void {
    this.#valueExpr && this.#valueExpr.removeChangeListener(bind(this,
this.#valueExprChanged));
    super.onDestroy();
  }

  #valueExprChanged(valueExpr: ValueExpression<number>): void {
    const value:number | undefined = valueExpr.getValue();
    //...
  }
}

export default MyComponent;
```

Example 5.22. Adding a listener and initializing

By calling the private function once immediately after adding the listener, it is possible to reuse the functionality of the listener for initializing the component. By removing the listener on destroy, memory leaks due to spurious listeners are avoided.

Property Path Expressions

The most commonly used value expression is the *property path expression*. It allows you to navigate from an object to a value by successively reading property values on which the next read operation takes place. For example, a property path expression may operate on the object obj and be configured to read the properties a, b, and then c. If the property a of obj is obj1, the property b of obj1 is obj2, and the property c of obj2 is 4, then the expression will evaluate to 4. A path of property names is denoted by a string that joins the property names with dots, in this case "a.b.c". If you want to address array elements you have to add the index of the element with another dot, such as a.b.c.3, and not use the more obvious but false a.b.c[3] notation.

You can create a property path expression manually in the following way:

```
import ValueExpression from
"@coremedia/studio-client.client-core/data/ValueExpression";
import ValueExpressionFactory from
"@coremedia/studio-client.client-core/data/ValueExpressionFactory";
```

```
//...
const ppe: ValueExpression = ValueExpressionFactory.create("a.b.c", obj);
```

Example 5.23. Creating a property path expression

The dot notation above might suggest that property path expressions operate exactly like TypeScript expressions, but that is not quite correct. Property path expressions support the following access methods for properties:

- read the property of a bean using the `get(property)` method;
- call a publicly defined getter method whose name consists of the string "get" followed the name of the property, first letter capitalized;
- call a publicly defined getter method whose name consists of the string "is" followed the name of the property, first letter capitalized;
- read from a publicly defined field of an object. This is the classic TypeScript case.

At different steps in the property path, different access methods may be used.

Even if there are many properties in the path, changes to any of the objects traversed while computing the value will trigger a recomputation of the expression value and potentially, if the value has changed, an event. This is only possible, however, for objects that can send property change events.

- For beans, a listener is registered using `addPropertyChangeListener()`.
- For components using `@jangaroo/ext-ts/mixin/Observable`, a listener is registered using `addListener()`.

Property path expressions may be updated. When invoking `setValue(value)`, a new value for the value expression is established. This will only work if the last property in the property path is writable for the object computed by the prefix of the path. More precisely, a value may be

- written into a property of a bean using the `set(property,value)` method;
- passed to a publicly defined setter method that takes the new value as its single argument and whose name consists of the string "set" followed by the name of the property, first letter capitalized;
- written into a publicly defined field of an TypeScript class.

At various points of the API, a value expression is provided to allow a component to bind to varying data. Using the method `extendBy(extensionPath)` adds further property dereferencing steps to the existing expression. For example, `ValueExpressionFactory.create("a.b.c", obj)` is equivalent to `ValueExpressionFactory.create("a", obj).extendBy("b.c")`.

Function Value Expressions

Function value expressions differ from property path expressions in that they allow arbitrary TypeScript code to be executed while computing their values. This flexibility comes at a cost, however: such an expression cannot be used to update variables, only to compute values. They are therefore most useful to compute complex GUI state that is displayed later on.

To create a function value expression, use the method `createFromFunction` of the class `ValueExpressionFactory`.

```
ValueExpressionFactory.createFromFunction(() => {
  return ...;
});
```

Example 5.24. Creating a function value expression

The function in the previous example did not take arguments. In this case, it can still use all variables in its scope as starting point for its computation or it might access global variables. To make the code more readable, you might want to define a named function in your TypeScript class and use that function when building the expression.

```
import ValueExpression from
"@coremedia/studio-client.client-core/data/ValueExpression";
import ValueExpressionFactory from
"@coremedia/studio-client.client-core/data/ValueExpressionFactory";

class MyClass {
  //...

  getExpr(): ValueExpression<number> {
    return ValueExpressionFactory.createFromFunction(calculateSomething);

    function calculateSomething(): number {
      return 42; // calculate some number with dependency tracking
    }
  }

}
```

Example 5.25. Creating a value expression from a private function

If you want to pass arguments to the function, you can provide them as additional argument of the factory method. The following code fragment uses this feature to pass a model bean to a static function.

```
import ValueExpression from
"@coremedia/studio-client.client-core/data/ValueExpression";
import ValueExpressionFactory from
"@coremedia/studio-client.client-core/data/ValueExpressionFactory";

class MyClass {
```

```
  //...

  getExpr(): ValueExpression<number> {
    return
ValueExpressionFactory.createFromFunction(MyClass.#calculateSomething);
  }

  static #calculateSomething(): number {
    return 42; // calculate some number with dependency tracking
  }
}
```

Example 5.26. Creating a value expression from a static function

Function value expressions fire value change events when their value changes. To this end, they track their dependencies on various objects when their value is computed. For accessed beans and value expressions, the dependency is taken into account automatically: whenever the bean or the value expression changes, the value of the function value expression changes automatically, and an event for the function value expression is fired.

value change events

If you access other mutable objects, you should make sure that these objects inherit from Observable, so that you can register the dependencies yourself. To this end, you can use the static methods of the class ObservableUtil. In particular, the method dependOn(Observable, String) provides a way to specify the observable and the event name that indicates a relevant change. As a shortcut, the method dependOnFieldValue(Field) allows you to depend on the value of an input field.

```
import ObservableUtil from
"@coremedia/studio-client.ext.ui-components/util/ObservableUtil";
import Observable from "@jangaroo/ext-ts/mixin/Observable";
import BaseField from "@jangaroo/ext-ts/form/field/Base";

class MyClass {
  #observable: Observable;
  #field: BaseField;

  #calculateSomething(): number {
    ObservableUtil.dependOn(this.#observable, "fooEvent");
    ObservableUtil.dependOnFieldValue(this.#field);
    //...
    this.#observable.fooMethod();
    //...
    return this.#field.getValue() as number;
  }
}
```

Example 5.27. Manual dependency tracking

If you register a dependency while no function value is being computed, the call to ObservableUtil is ignored. This means that you can register dependencies in your own functions, and the methods will work whether they are called in the context of a function value expression or not.

The following listing contains a comprehensive example of a function value expression with detailed code comments concerning where and why dependency tracking is active or not. In the function, a list of titles is gathered from different sources. For each of the titles, a panel is searched and its height is put into a map. This map is the return value of the function.

```
import {bind} from "@jangaroo/runtime";
import ValueExpression from
"@coremedia/studio-client.ext.client-core/data/ValueExpression";
import ValueExpressionFactory from
"@coremedia/studio-client.client-core/data/ValueExpressionFactory";
import RemoteBeanUtil from
"@coremedia/studio-client.client-core/data/RemoteBeanUtil";
import Content from "@coremedia/studio-client.cap-rest-client/content/Content";
import ObservableUtil from
"@coremedia/studio-client.ext.ui-components/util/ObservableUtil";

class MyClass {
  //...

  #listenToChanges(): void {
    const firstContent: Content =
this.#getFirstContentValueExpression().getValue();
    const secondContentVE: ValueExpression =
this.#getSecondContentValueExpression();

    const panelHeightsVE: ValueExpression =
      ValueExpressionFactory.createFromFunction(
        bind(this, this.#getPanelHeights),
        firstContent,
        secondContentVE);
  }

  // First content is directly passed to the function.
  // => No dependency tracking for changes to
this.#getFirstContentValueExpression().
  // Second content is accessed via ValueExpression.
  // => Dependency tracking for changes to
this.#getSecondContentValueExpression().
  #getPanelHeights(
    firstContent: Content,
    secondContentVE: ValueExpression): Object {

    // 'additionalTitles' is just a class field.
    // => No dependency tracking for changes to its value.
    let titles = this.additionalTitles || [];

    // Accessing a Bean property.
    // => Dependency tracking for changes to the bean property.
    // Normal beans as opposed to the RemoteBeans below are not asynchronous,

    // so we do not need to wait until they are loaded.
    const model = this.getModel();
    titles = titles.concat(model.get('additionalTitles') || []);

    // Contents are of type Bean (RemoteBean).
    // RemoteBeanUtil.isAccessible() checks if loaded and readable.
    // If not:
    // (1) A 'load' call is automatically triggered.
    // (2) A dependency for a Bean state change is registered.
    // => dependency tracking for the content beans being loaded.
    switch (RemoteBeanUtil.isAccessible(firstContent)) {
    case undefined:
      // Not loaded yet.
      // => Interrupt computation. Wait for firstContent being loaded.
      return undefined;
    case true:
      // Loaded and unreadable.
```

```
      // => Abort
      return null;
      // Otherwise: RemoteBean loaded, just continue ...
    }

    // Dependency tracking for changes to secondContentVE.
    const secondContent = secondContentVE.getValue();
    if (!secondContent) {
      // Interrupt computation.
      // Wait for secondContentVE holding a content.
      return undefined;
    }
    // See above: Wait for secondContent being loaded.
    switch (RemoteBeanUtil.isAccessible(secondContent)) {
    case undefined:
      return undefined;
    case true:
      return null;
    }

    // From here on, both contents are loaded
    // Their properties can be accessed.
    // Properties of contents are SubBeans => no need to wait
    // for them being loaded.
    let properties = firstContent.getProperties();
    titles.push(properties.get("title"));
    properties = secondContent.getProperties();
    titles.push(properties.get("title"));

    const panelHeights: Object = {};

    // For all gathered titles, find a panel with the corresponding title
    // and get its height.

    var panelsParentContainer = this.#getPanelsParentContainer();

    let addDependencyAdded: Boolean = false;

    for (let i = 0; i < titles.length; i++) {
      const title = titles[i];
      const panel = panelsParentContainer.getPanelWithTitle(title);
      if (!panel) {
        // Panel with title does not exist yet.
        // Dependency tracking for new childs being added to the container.
        // 'add' is a component Event of Ext.container.Container
        if (!addDependencyAdded) {
          ObservableUtil.dependOn(panelsParentContainer, "add");
          // Only add one dependency for 'add'.
          addDependencyAdded = true;
        }
        // Continue with next title.
        continue;
      }
      if (panel.rendered) {
        // If panel is rendered, just get its height.
        panelHeights[panel.getId()] = panel.getHeight();
      } else {
        // If panel is not rendered:
        // => Dependency tracking for the panel being rendered.
        // 'afterrender' is component event of Ext.Component
        ObservableUtil.dependOn(panel, "afterrender");
      }
    }

    // Alternative:
    // According to the code above, also partial values for
    // 'panelHeights' are computed: Not found or not rendered
    // panels are just skipped. Alternatively, we could wait
    // until all panels are present and rendered. In that case
    // we need to return 'undefined' each time we encounter
    // a missing part. It really depends on what 'panelHeightsVE'
    // is supposed to deliver.
```

```
    return panelHeights;
  }
}
```

Example 5.28. Comprehensive example of a FunctionValueExpression

5.4 Remote CoreMedia Objects

For accessing content, users and groups from *CoreMedia Studio*, a rich API is provided on top of the Bean/RemoteBean API. In particular, the interfaces `Content`, `User`, and `Group` all inherit from `RemoteBean`. The API aims at being similar to the *Unified API*, which provides access to the CoreMedia servers from Java. However, some adjustments were necessary to support the different flavor of concurrency found in JavaScript/TypeScript.

Accessing content on the server

Please refer to the TypeScript documentation for details about the individual interfaces and methods listed in the following overview.

5.4.1 Connection and Services

Usually, the *Studio* framework will already have taken care of the login when your code is invoked.

In special cases, for example if you are not in *CoreMedia Studio*, you can use the static method `CapImpl.prepare(AnyFunction)` to create a connection to the remote server. The URL of the CMS remote service to use is read from the global variable `coremediaRemoteServiceUri`. The `prepare` method calls the callback function when the connection has been established, passing a `@coremedia/studio-client.cap-rest-client/common/CapConnection` as the single argument. This connection is not yet bound to a user, but it provides the method `getLoginService()`. On the returned `@coremedia/studio-client.ext.cap-rest-client/common/CapLoginService` you can call the `login(string, string, string, AnyFunction)` method to authenticate the current user, which enables access to other services of the connection.

Creating a connection when not logged in

Once a connection is established, the current session is stored under the key *session* in the application scope bean (obtainable from the current `editorContext` instance). The session provides access to the current user and back to the connection.

The methods `getContentRepository()`, `getUserRepository()`, and `getWorkflowRepository()` of the connection return objects of type `@coremedia/studio-client.cap-rest-client/content/ContentRepository`, `@coremedia/studio-client.cap-rest-client/user/UserRepository`, and `@coremedia/studio-client.cap-rest-client/workflow/WorkflowRepository`, respectively. These repositories serve the same purpose as the identically named objects of the

Unified API. However, the supported functionality is limited to the use cases required for content editing.

The `ContentRepository` provides access to the `PublicationService` and the content `AccessControl` through the method `getPublication Service()` and `getAccessControl()`, respectively.

Content repository and services

Unlike the *Unified API*, approval operations using the publication service also approve all folders on the path to a content. Publication is very similar to the *Unified API* counterpart, but withdrawals are performed in a single step without the need to successively set a mark, approve it, and publish the withdrawal.

The `@coremedia/studio-client.ext.cap-rest-client/con tent/authorization/AccessControl` class allows you to check whether certain operations on contents are permitted for the current user. Some methods like `mayMove()` and `mayCreate()` are provided for special cases, but most checks are made using the method `mayPerform`, which takes a `Right` enumeration value to indicate the intended operation.

All these methods track the dependencies and can be used from within a `FunctionValueExpression`, even though you cannot register change listeners directly.

The `WorkflowRepository` provides access to the `WorklistService` and the workflow `AccessControl` through the method `getWorklistSer vice()` and `getAccessControl()`, respectively.

Workflow repository and services

The `WorklistService` corresponds closely to the `WorklistService` of the *Unified API*. It provides access to all user-specific lists, but not the administration lists. In particular, you can retrieve the list of process definition that the current user may instantiate, the processes the user has created, but not started, the processes the user has created and started, the offered task and the accepted tasks. You can also obtain lists of tasks that encountered problems during their execution.

All these methods track the dependencies and can be used from within a `FunctionValueExpression`, even though you cannot register change listeners directly.

The `@coremedia/studio-client.ext.cap-rest-client/con tent/authorization/AccessControl` class allows you to check whether certain operations on workflow objects are permitted for the current user. The methods match the methods defined in the *Unified API*. While the rights are being retrieved, the methods will return `undefined`. Afterwards a Boolean value is answered. Note, however, that no changes of rights are propagated to the client. This is not normally a problem, because the built-in rights policies depend on the current user, only, and not on the workflow state.

5.4.2 Content

A `@coremedia/studio-client.cap-rest-client/content/Content` object represents a content item or folder in the CoreMedia system. It can be obtained through the methods `getChild(...)` or `getContent(string)` of the content repository. Note that unlike in *Unified API*, the *string* parameter to the latter method is not an ID, but a URI path. You can get the URI path of a `Content` with the `Content#getUriPath()` method (inherited from `@coremedia/studio-client.client-core/data/RemoteBean`).

Content on the server

You can also initiate a search request using the search service returned by `getSearchService()` or by navigating to a content from the root folder returned by `getRoot()`.

Using `getProperties()`, it is possible to navigate to a secondary bean of type `@coremedia/studio-client.cap-rest-client/content/ContentProperties` that contains all schema-defined properties of a content item. When updating properties, use the inherited, generic `set(property, value)` method of `@coremedia/studio-client.client-core/data/Bean` with *Calendar*, *string*, or *number* objects or arrays of `Content` objects as appropriate for the individual properties. Refrain from setting blob-valued and XML-valued properties at this time. As for all remote beans, the method `flush(callback)` can be called to force properties to be written to the server immediately.

Accessing properties of content

The `Content` object itself is only responsible for the meta properties that are the same for all contents, for example the name property. The class `ContentPropertyNames` lists all these property names for your reference. As usual, these are also the property names for the events that are sent when a content changes.

The property `lifecycleStatus` is a special property that does not correspond to any *Unified API* feature. It indicates the simplified way in which *Studio* represents the approval, deletion, and publication flags to the user. The class `LifecycleStatus` contains constants for the supported states.

Following the *Unified API*, every content object is associated to a `ContentType` object by means of the `getType()` method. You can also retrieve types by name from the content repository. Given a type, you can create new instances of the type by means of the `create(Content, string, AnyFunction)` method.

The `move()` and `rename()` methods are shortcuts for setting the `parent` and `name` properties. As such, a callback provided with these calls receives a `FlushResult` as its single argument. The methods `copy()`, `checkIn()`,

checkOut(), revert(), and doDelete() correspond to the equivalent *Unified API* methods. (The unusual name of the doDelete() method is caused by delete being a reserved word in TypeScript.)

All operations receive result objects indicating whether the operation was successful. The result of a delete operation is recorded in a DeleteResult, with result codes being documented in DeleteResultCodes. Similarly, there are CopyResult and CheckInResult objects. Please see the TSDoc for details.

Getting result objects

Through the method getIssues(), a Content object provides access to issues detected by the server-side validators. See Section 5.3.3, "Issues" [62] for details about the issue API.

5.4.3 Workflow

A WorkflowObject represents a Task or Process in the *Workflow Server*. Tasks provide the method getContainingProcess() to navigate to its process. Each task and process links to a definition object by means of its getDefinition() method. Definition objects are either instances of TaskDefinition or ProcessDefinition. Each task definition indicates a TaskDefinitionType through the method getType(), for example USER or AUTOMATED.

Workflows on the server

Using the methods getTaskState() and getProcessState() the current state of a task or process can be obtained as an enumeration value.

The methods available for workflow objects and definitions correspond to the equivalent *Unified API* methods.

Accessing properties of workflow objects

Using getProperties() on a task or process, it is possible to navigate to a secondary bean of type WorkflowObjectProperties that contains all schema-defined properties of a workflow object. When updating properties, use the inherited, generic set(property, value) method of Bean with boolean, string, number, User, Group, Content, or Version objects or arrays of such objects as appropriate for the individual properties. At the moment, timer are not supported. As for all remote beans, the method flush(callback) can be called to force properties to be written to the server immediately.

5.4.4 Structs

Structs are part of the *Unified API* and are thus a core product feature.

Implemented by the interfaces `Struct` and `StructBuilder` in the Java API, structs provide a way to store dynamically typed, potentially nested objects in the content repository, and thus add the possibility of storing dynamically structured content to the *Content Server*'s static content type system. To this end, the content type schema may define XML properties with the grammar name `coremedia-struct-2008`. This grammar should use the XML schema `http://www.coremedia.com/2008/struct` as defined in `coremedia-struct-2008.xsd`.

Storing dynamically structured content with Structs

In the TypeScript API, structs are modeled as `Bean` objects. They are directly modifiable. They implement the additional abstract class `@coremedia/studio-client.cap-rest-client/struct/Struct` to provide access to their dynamic type.

Like every content property value, struct beans are provided as properties of the `ContentProperties` beans. If a struct bean contains a substruct at some property, that substruct is again represented as a struct bean.

Atomic properties of structs may be accessed just like regular bean properties. Structs can store strings, integers, Boolean, and links to content items as well as lists of these values. All struct properties can be read and written using the ordinary `Bean` interface. As usual, lists are represented as TypeScript `Array` objects. Do *not* modify the array returned for a list-valued property. To modify an array, clone the array, modify the clone, and set the new value at the bean.

In the special case of lists of structs, use the methods `addAt()` and `removeAt()` (of the struct containing the struct list) to insert or delete individual entries in the struct list. Note that `Struct` objects in struct lists represent a substruct at a fixed position of the list. For example, the `Struct` objects at position 2 will contain the values of the struct previously at position 1 after you insert a new struct at position 0 or 1.

Structs and substructs support property change events. Substructs do *not* support value change events. You can only listen to a single property of a substruct.

Top-level structs in the TypeScript API are never `null`. If a content property is bound to an empty XML text, a struct without properties is still accessible on the client. This makes it easier to fill initially empty struct properties.

The most convenient way to access a struct property is by means of a value expression. For example, for navigating from a content property bean to the property `bar` of the struct stored in the content property `foo`, you would use the property path `foo.bar`. You can use these property paths in the standard property fields provided by *CoreMedia Studio*. This case is shown in the following code fragment:

```
import Config from "@jangaroo/runtime/Config";
import DocumentTabPanel from
"@coremedia/studio-client.main.editor-components/sdk/premular/DocumentTabPanel";
import StringPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/StringPropertyField";

//...
Config(DocumentTabPanel, {
  //...
  items: [
    //...
    Config(StringPropertyField, {
      propertyName: "foo.bar",
    }),
  ],
  //...
})
```

Example 5.29. Property paths into struct

Structs support the dynamic addition of new property values. To this end, the interface `Struct` provides access to a type object implementing `@core media/studio-client.cap-rest-client/struct/StructType` through the method `getType()`. You can call the `addXXXProperty()` methods for various property types during the initialization code that runs after the creation of a content item.

Dynamic addition of new property values

```
import { cast } from "@jangaroo/runtime";

import Content from "@coremedia/studio-client.cap-rest-client/content/Content";
import Struct from "@coremedia/studio-client.cap-rest-client/struct/Struct";

import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import IEditorContext from
"@coremedia/studio-client.main.editor-components/sdk/IEditorContext";

class MyStudioPlugin extends StudioPlugin {

  override init(editorContext: IEditorContext): void {
    //...
    editorContext.registerContentInitializer("MyDocumentType",
      MyStudioPlugin.#initStruct);
    //...
  }

  static #initStruct(content: Content): void {
    const properties = content.getProperties();
    let struct = cast(Struct, properties.get("foo"));
    struct.getType().addStringProperty("bar", 200);
  }
}
```

Example 5.30. Adding struct properties

While it is possible to add a property automatically during the first write, this is not recommended. Some property fields cannot handle an initial value of `un defined`. You should therefore only bind property fields to initialized properties.

5.4.5 Types and Property Descriptors

Both `Content` and `Struct` are derived from a common parent interface `CapStruct`, which takes the same responsibilities as its *Unified API* equivalent. It augments `Bean` objects by providing a type in the form of a `CapType`, the common parent of, for example, `ContentType` and `StructType`. Types can be arranged in a type hierarchy and they can be given a name.

A `CapType` provides access to `CapPropertyDescriptor` objects, which describe the individual properties allowed for a `CapObject`. In the `type` property a property descriptor indicates which value the property can take according to the constants defined in `CapPropertyDescriptorType`: string, integer, markup, and so on. Each property descriptor also declares whether the property is `atomic` and accepts plain values or is a `collection` and accepts arrays of appropriate values.

For certain descriptor types, more specific interfaces provide access for additional limitations on the property. A `StringPropertyDescriptor` declares a `length` attribute indicating the maximum length of a string stored in the property. A `BlobPropertyDescriptor` can limit the `contentType` (a MIME type string) of the property values. A `LinkPropertyDescriptor` specifies the type of linked objects and a `MarkupPropertyDescriptor` the grammar of stored XML data.

5.4.6 Concurrency

Being remote beans, the `Content` objects inherit the concurrent behavior of the bean layer. A request to load content data is issued upon first querying any property except for `isDocument()` and `isFolder()`). However, since the response arrives asynchronously and is handled in a subsequent execution, the getter methods will initially return `undefined`. You must therefore make your code robust to handle this situation – which commonly is done by attaching a value change listener that is invoked once the content properties become available, or create a property path expression and use its `loadValue(Any Function)` method (see Section 5.3.6, "Value Expressions" [65]). Depending on the execution sequence, content may be loaded due to some other, potentially unrelated request before you access it – but your code must not rely on it.

All singletons (`Cap`, `CapConnection`, `CapLoginService`, `session`/`CapSession`, `ContentRepository`, `UserRepository`) and all `ContentType` objects, however, *are* fully loaded before the *Studio* application's initialization process is finished (which is why these interfaces do not extend `RemoteBean`).

When you want to make sure that values have actually hit the server after an update, you can use `RemoteBean#flush(AnyFunction)`, and register a callback function.

5.5 Web Application Structure

CoreMedia Studio uses a web application for delivering both static content (like the JavaScript code defining the application) and dynamic content stored in the CMS.

Dynamic content is provided by means of a REST service embedded in a Spring web application context. See http://www.springsource.org/ for details about the Spring framework. In the following section, it is assumed that you know about the essential concepts of the Spring inversion of control (IoC) container.

As described in ????, *CoreMedia Studio* is assembled from application and component artifacts. To change and extend the default context configuration you can modify the config files `application.xml` and `component-<componentname>.xml` and their corresponding property files `application.properties` and `component-<componentname>.properties` in the `/WEB_INF/` file system folder.

You must modify the application context to configure your content validation setup. See Section 9.23.1, "Validators" [252] for the details.

5.6 Localization

> **Localizing CKEditor**
>
> This section is about localizing Ext JS components. For embedded rich text editing component *CKEditor* a different approach is required. For details, see Section 10.3.2, "Localizing CKEditor 5" [430].

ⓘ

Creating Resource Bundles

Text properties in *CoreMedia Studio* can be localized. English and German are supported out of the box; you can add your own localization bundles if required. To do so, proceed as follows:

1. Add the new locale to the `studio.locales` property in your Studio application's `application.properties` file.

   This property contains a comma-separated list of locales. The first element in the list is `en` and specifies the locale of values in the default properties files (that is, the files without a locale suffix). Therefore, you must not change this first entry; it must always remain `en` (see below).

2. Add properties classes that follow the naming scheme for your added locale, as explained below.

Localized texts are stored in TypeScript classes as constants. The naming scheme of these files is:

`<FileName>_<IsoLanguageCode>_properties.ts`

A TypeScript properties class with no language code contains properties in the default language English. Note that English is only a technical default. The default locale used for users opening *CoreMedia Studio* for the first time is determined by the best match between their browser language settings and all supported locales.

When properties are missing in a locale-specific properties class or the complete properties class is missing, the values of the properties are inherited from the default language (that is, they will appear in English rather than in the locale the user has set).

Accessing Resource Bundles

Resource bundles can be accessed via the ResourceManager or by directly accessing the constant of the properties class:

```
import Config from "@jangaroo/runtime/Config";
import Panel from "@jangaroo/ext-ts/panel/Panel";
import MyPropertiesClass from "./MyPropertiesClass";

Config(Panel, {
    title: MyPropertiesClass.my_constant,
    //...
})
```

The ResourceManager can be accessed via the constant `resourceManager` (lower case) which has the type `IResourceManager`. It is mostly used when values of other property classes should be overwritten or a value from another language is should be read.

Overriding existing properties

If you want to change predefined labels, tooltips or similar, you can override properties from existing properties classes. To this end, you should first define a new properties class and then declare a `CopyResourceBundleProperties` inside the `configuration` section of your plugin rules. This plugin will copy all key-value-pairs from the `source` properties class to the `target` properties class, overwriting entries with the same keys.

```
import resourceManager from "@jangaroo/runtime/l10n/resourceManager";
import CopyResourceBundleProperties from
"@coremedia/studio-client.main.editor-components/configuration/CopyResourceBundleProperties";
import SomeStudio_properties from "@coremedia/studio-client.main...";
import MyCustomized_properties from "./MyCustomized_properties";

// inside the 'configuration' property:
new CopyResourceBundleProperties({
  destination: resourceManager.getResourceBundle(null, SomeStudio_properties),

   source: resourceManager.getResourceBundle(null, MyCustomized_properties),
})
```

Generally, each Studio plugin module will contain at least one set of properties class for localizing its own components or for adapting existing properties classes.

For details on UI localization through properties classes see Section 9.4, "Localizing Labels" [141].

5.7 Multi-Site and Localization Management

CoreMedia provides a concept to handle multi-site and multi-language in a standardized way.

Configuration

The CoreMedia Site Model is defined via the bean `siteModel` of the *CoreMedia Studio* web application. Please refer the to the Blueprint Developer Manual to know, how CoreMedia has designed multi-site and multi-language support.

SitesService

To access all the features of multi-site and multi-language, you can use the `sitesService`. The `sitesService` is available via the `IEditorContext` with its `getSitesService()` method or can access directly via the global constant `sitesService`.

With this, you have access to all available Sites and their properties - the root folder, the site indicator, etc. Furthermore, you have access to the Site Model specifications like the properties for master relations or of which content type the Site Indicator is. For a detailed understanding, you are asked to read the Studio API documentation as well.

5.8 Jobs Framework

A job is an execution of code that you can monitor, track and cancel. The jobs framework lets you execute jobs within the Studio client (local job) or Studio backend (remote job). Using the jobs framework you can track the progress of a job, show it to the client and cancel it.

> **NOTE**
>
> You should use a *remote job*, whenever you only need to get data from the Studio backend for the execution of your code.

5.8.1 Defining Local Jobs

If you want to create a job that only runs within the studio client, you need to implement the interface `Job`. Within the `execute` method you can perform the wanted operation. You can use the methods of the `JobContext` object to notify about a success, failure, progress or abort.

When a job gets aborted by the user (or because a job with the same groupId is already running), the method `requestAbort` will be called and you can stop the execution of your job and then call the `notifyAbort` method from the `JobContext.`

5.8.2 Defining Remote Jobs

A remote job is executed within the Studio backend and can be triggered and monitored by the Studio client. The client can pass parameters to the job and will receive the progress of the job's execution and the result, once the job is finished.

> **CAUTION**
>
> You should use this framework for any backend calls that need some time to deliver their result, in order to prevent timeouts for your request.

Defining a Remote Job in the Studio Backend

In order to define a remote job you need to implement the interface `com.coremedia.rest.cap.jobs.JobFactory`. The implementation has to be defined as a bean within the `studio-lib` extension.

The method `accepts` needs to define for which job type the factory will return a `Job` object. The method `createJob` has to return an implementation of `com.coremedia.rest.cap.jobs.Job`. Within the implementation of your `Job` class you can perform your execution and return the result within the `call` method.

If you want to pass parameters from the Studio client to your job implementation, you need to define those parameters as local variables and define setters for them. Together with that setter, you need to annotate the variables with `@JsonProperty("variableName")` with a variable name that matches the parameter key, passed to the job from the Studio client. You may also leave the `@JsonProperty` annotation, if your setters are named correctly: `variableName => setVariableName`

If you want to send the progress of your job to the Studio client, you need to call the method `notifyProgress` of the `JobContext` Object with a value between 0 and 1. You get the `JobContext` instance within the `call` method of your `Job` object.

A job can be aborted by the client. You can use the result of the method `isAbortRequested` of the `JobContext` object as a break condition within your execution in order to react to the abortion of your job.

Defining a RemoteJob in Studio Client

If you want to execute your remote job, you need to create a job in the Studio client that extends the class `RemoteJobBase`. Your extension has to override the method `getJobType` and return the value that has to match the `jobType` within the `accepts` method in your `JobFactory` in the Studio backend.

In addition, you can override the method `getParams`. The object you return in this method will be passed as parameters in your job implementation in the Studio backend. Note that the keys in your parameters object have to match the value that you defined within your backend job via the annotation `@JsonProperty("variableName")`.

5.8.3 Executing Jobs

After defining and instantiating your job, you need to execute the job via the globally defined variable `jobService`. Together with the job itself you can pass a success, a failure and an abort function. Additionally, you can pass a groupId. If a job with a certain groupId is already running and another job with the same groupId gets executed by the `IJobService`, the first job will be automatically canceled.

> **NOTE**
>
> You can always rely on the fact that one of the callbacks (success, failure or abort) is triggered after the job execution has finished. After that, no additional callbacks will be triggered.
>
> The same job instance can be executed multiple times as long as it is stateless.

The `jobService` returns a `TrackedJob` object, which can be used to receive the status of a job and its result when the execution was successful. In case the job fails the result contains the error message. An abortion of the job yields no result.

5.8.4 Visualize Jobs Within the BackgroundJobsWindow

If you want your jobs to be displayed in the *BackgroundJobsWindow*, which can be opened via the TabsPanel, your job needs to implement the interface `BackgroundJob`. This ensures that the progress is also visualized via a progress bar and the corresponding action buttons are shown for the optional success and error callback functions.

5.9 Further Reading

At http://docs.sencha.com/extjs/7.2.0/ you can find the API documentation of Ext JS 7.2.

ckeditor.com provides information about the rich text editor *CKEditor*.

The documentation of the TypeScript API is linked from the documentation page of *CoreMedia Content Cloud*. The overview page can be found at **https://docu-mentation.coremedia.com/**. Note that classes or interfaces not mentioned in the API documentation pages are not public API. They are subject to change without notice.

The remote API for content is closely related to the *Unified API* provided for Java projects, although there are changes to accommodate for the different semantics of the base languages. Still, the *Unified API Developers Guide* gives a good overview of the involved concepts when dealing with content. Documents, folder, versions, properties, types, and the like are explained in detail as well as the structuring of the API into repositories, identifiable objects and immutable values.

6. Structure of the Studio Client Workspace

The studio-client workspace is a pnpm workspace consisting of various packages. It has the following file structure:

File structure of the workspace

```
apps/studio-client/
    ├── apps/                  // app specific packages
    ├── global/                // global packages
    ├── node_modules/          // dependencies managed by the package
    │                             manager generated during installation
    ├── shared/                // packages shared between apps
    ├── tools/                 // helper tools
    ├── check-pnpm.js          // script making sure pnpm is used
    ├── Dockerfile             // Build studio-client image
    ├── docker-bake.hcl        // convenience scripting for Docker build
    ├── entrypoint.sh          // entrypoint triggered while starting client
    ├── nginx.conf.template    // nginx configuration for studio-client
    │                             image
    ├── package.json           // meta data about the workspace for pnpm
    ├── pnpm-lock.yaml         // pnpm lock file to fixate versions
    ├── pnpm-workspace.yaml    // pnpm workspace configuration
    ├── README.adoc            // general information
    ├── .dockerignore          // files to ignore during docker build
    └── .gitignore             // files to ignore by Git
```

> **NOTE**
>
> Depending on the used IDE additional files or folders might be existing. The `node_modules` folder is created after running `pnpm install` for the first time.

The workspace structure follows a strict pattern regarding the location of packages and their responsibilities:

Subfolders and their responsibilities

- As the name implies, the `shared` folder contains libraries that are or can be shared across multiple other packages in the workspace. None of these packages should have a dependency to any package in the `apps`, `global` or `tooling` folders.

The first level of sub folders inside the `shared` folder declares if the package utilizes Ext JS (`ext`) or if all code contained inside the package is written in plain JavaScript (`js`). While plain JavaScript packages can be used like any other npm package the former require special treatment by the Jangaroo tooling and are most likely exclusive to being used inside applications built with Ext JS.

- The `apps` folder contains apps or libraries shared among the apps. While they can have a dependency to packages in the `shared` folder, none of these packages should have a dependency to any package in the `global` or `tooling` folders.

  Inside the `apps` folder, the first level of subfolders identifies the app that the corresponding package belongs to, for example, `main` or `workflow`.

  The `extension-config` and `extensions` folders on the second level have special relevance for the extensions tool (see Section 4.1.5, "Project Extensions" in *Blueprint Developer Manual*).

- The `global` folder contains only the studio package which aggregates all available apps into a single bundle, which can be deployed on a web server. It may depend on every other package except packages in the `tooling` folder.

- Packages inside the `tooling` folder are used to provide some helper tools when migrating from an older *CoreMedia* studio-client workspace. Their usage is usually explained in the corresponding release notes.

  You should never depend on any package inside this folder after you have successfully migrated as they might change or be removed entirely in an *AEP*.

NOTE

There is no check inside the workspace that enforces that the patterns described above are actually applied CoreMedia. However, CoreMedia highly suggest to stick to this structure as CoreMedia might provide helper tools that will not adapt to any custom workspace layout and which you might not be able to use without manual adjustments otherwise.

File structure of a Jangaroo package

While every package needs at least a `package.json` file containing the meta data for pnpm, packages that represent a Jangaroo project usually have the following file structure:

```
some-package/
    ── build/              // output folder for builds and tests
    ── dist/               // distribution folder for publishing
                           //   into the npm registry
    ── node_modules/       // dependencies managed by the package
                           //   manager generated during installation
    ── jest/               // contains unit tests for jest
```

```
├── joounit/              // contains unit tests for joounit
├── sencha/               // additional files for sencha build
├── src/                  // the actual sources of the package
├── .eslintrc.js          // linter configuration
├── jangaroo.config.js    // jangaroo configuration
└── package.json          // meta data about the package for pnpm
```

Not all folders exist for all packages in the workspace. The `build` and `dist` folders are only created after executing certain pnpm scripts.

The `jest`, `joounit` and `src` folders each contain a `tsconfig.json` file for TypeScript which is generated after building the package for the first time. Part of this configuration will mirror your (transitive) dependencies and will be modified if there were are changes in the dependency tree. CoreMedia recommends to check in these files to have code completion immediately after checking out a branch without having to build everything first.

NOTE

Currently it is not possible to add custom configuration to the `tsconfig.json` file.

---

7. Developing with the Studio Client Workspace

This workspace is based on TypeScript and the package manager pnpm. The following sections describe how to build and develop with it.

Required Tools

You have to install the following tools to build the workspace:

Node.js

You need Node.js in a supported version (see https://releases.coremedia.com/cm-cc-13/artifacts/CMCC 13 - Supported Environments.pdf) to build the studio-client workspace. See https://nodejs.org/en/

pnpm

While Node.js provides a build-in package manager for npm packages, alternative package managers are also supported. CoreMedia workspaces use an alternative package manager called pnpm. See https://pnpm.io/installation for details or install it directly via npm (replace <VERSION> with the supported pnpm version, for example 8.1, see https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf):

```
npm install -g pnpm@<VERSION>
```

> **NOTE**
> Make sure that all these tools are available in your `PATH` variable.

Configuration

Your pnpm client first needs to be authenticated to the CoreMedia npm registry in order to download CoreMedia packages (see Section 3.1, "Prerequisites" in *Blueprint Developer Manual*).

Building the Workspace

Once you have installed and configured the required tools, you can build the Studio client packages. Invoke the following commands from the `apps/studio-client` folder of your Blueprint workspace:

```
pnpm install
pnpm -r run build
```

The `install` command will download all dependencies required to actually build the workspace while the `build` command will compile all the sources into the corresponding output folders.

Starting the Studio Client

After the build was successful start the Studio client using the `start` script:

```
cd global/studio
pnpm run start
```

This requires a local Studio server running at http://localhost:41080. In order to provide a custom location for the studio server, you can provide a custom URL to the `start` script:

```
pnpm run start --proxyTargetUri=http://some-host/studio
```

For a list of all available parameters call the `start` script with `--help`.

The start script will output a message including the URL under which you can access the studio-client from the browser for local development.

Rebuild on changes

If you made file changes you will need to call the `build` script again. In most cases, you only need to build the package which contains the changed files. The whole workspace has only to be rebuild after checking out a new branch, for example.

Build a single package by running the `build` script from the folder of the package:

```
cd apps/main/blueprint-forms
pnpm run build
```

> **NOTE**
>
> There also is a way to build a package and/or its dependencies/dependents with a single command. Please consult https://pnpm.io/filtering for further information. The following chapters in this manual might make use of these filters.

Keep in mind that building a package with the `build` script does not automatically clean up deleted files in the output folders. To clean up the output folder of a package, use the `clean` script:

```
cd apps/main/blueprint-forms
pnpm run clean
```

However, depending on what has been changed it might be necessary to rebuild all packages or at least the package including its dependents. Typical situations are:

- Changing any SASS file in `sencha/sass` requires (at least) also building the corresponding (base) apps.

  Adding/Removing dependencies via `package.json` or `pnpm-lock.yaml` as well as changing the workspace structure via `pnpm-workspace.yaml` usually requires running `pnpm install` in addition to rebuilding the packages. If a dependency has been used for the first time, it is also necessary to build all `app`, `app-overlay` and `apps` packages.

  Rare case: Changing the base class of a class so that is being compiled to a Ext JS class instead of plain JavaScript and vice versa has a major impact on all derived classes and how a class is included in the app build. Such a change requires rebuilding not only the package but also all its dependent packages.

Most changes can be immediately seen after a browser reload. However, general changes in configuration and dependencies (including the workspace) require rerunning the `start` script.

Automatically rebuild on changes

All Jangaroo projects also have a `watch` script which can be used to automatically track changes inside a package (and optionally its dependencies inside the workspace). You can start the watch task for a single package using the following command:

```
cd apps/main/blueprint-forms
pnpm run watch
```

This will automatically rebuild the project if any changes have been detected inside the `src` or `sencha` folders.

By using the command line parameter `--skipInitialBuild` you can prevent that the package is build initially, for example, if you have already built the whole workspace and did not make any changes yet.

The `watch` script can not only track changes inside a single package but also track changes of its dependencies inside the workspace if the parameter `--recursive` is passed. As the watch task only knows about Jangaroo projects this however is limited to packages containing a Jangaroo project. The watcher will not trigger any custom `build` scripts.

The most common case is watching the apps packages in `global/studio` including its dependencies. To avoid rebuilding the whole dependency tree first, the `--skipInitialBuild` comes in handy here:

```
cd global/studio
pnpm run watch --recursive --skipInitialBuild
```

NOTE

As a convenience feature, the watcher will recompile the CSS of a (base) app contained inside the workspace if any changes to SCSS files inside `sencha/sass` have been detected. This comes in handy when making many changes to styling as building the CSS of an Ext JS application requires only a fraction of the normal build time.

As the `watch` task itself can be configured by the `jangaroo.config.js` file and it is contained inside a dependency it has some limitations:

- Changing the `jangaroo.config.js` file will not have any effect until the watcher is restarted.

  It will not trigger `pnpm install` to update any dependencies. So changes to the workspace or the dependency tree requires performing a manual rebuild and restarting the watcher in most cases.

Running tests

Tests are not automatically run when triggering the `build` script. You need to invoke the `test` script provided in every package containing Jest tests and/or JooUnit tests. To run all the tests of all packages in the workspace use the following command:

```
pnpm -r run test
```

If a package does not contain a `test` script it will be ignored.

The execution will immediately exit with a non-zero exit code as son as any test error occurs. In case you want to execute all tests, regardless of previous failures, you can pass the parameter *--testFailureExitCode* to the test:

```
pnpm -r run test --testFailureExitCode 0
```

Test setup failures will still lead to a non-zero exit code in that case. The only difference is that the execution will not be interrupted because there were test failures. This might become handy in CI environments to collect the JUnit test reports every package provides in the `build` folder.

The Jest test report can be found in `build/jest/junit.xml` and the JooUnit test report can be found in `build/joounit/junit.xml` accordingly.

IDE Support

One of the rationales behind using TypeScript is to make the good parts of static typing, such as getting reliable and useful IDE support, available for the dynamic language JavaScript. This section shows how to properly configure syntax assist for *JetBrains* products but also for *Microsoft Visual Studio Code*.

JetBrains

Recent versions of the JetBrains IDEs *IntelliJ IDEA Ultimate* and *WebStorm* have built-in support for TypeScript and JavaScript development. Make sure that you activate the plugin providing support for TypeScript and JavaScript. It might also be handy to activate support for Node.js.

Also make sure that the setting `Node interpreter` is properly set up in both plugins and points to Node.js in the supported version (see https://re-leases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environ-ments.pdf).

The TypeScript path of the corresponding plugin should be set to `apps/studio-client/node_modules/.pnpm/typescript@x.x.x/node_modules/typescript` where `x.x.x` is the TypeScript version used inside the workspace (usually, there is only one). This folder is created after `pnpm install` has been called for the first time.

> **NOTE**
>
> In case the IDE support does not properly work it might help to restart the TypeScript support. Usually this can be done via the footer toolbar by clicking *TypeScript x.x.x* and clicking *Restart TypeScript Service*.
>
> If the footer item does not exist or does not show a version this usually indicates that something is not properly configured.

i

Visual Studio Code

In contrast to *JetBrains* products, this IDE is available for free and more lightweight by sacrificing some features for code assist (for example, more complex code refactoring).

Make sure to add/enable at least the extensions for JSON, Npm and TypeScript.

> **NOTE**
>
> Just like when using *JetBrains* products it might be helpful to restart the TypeScript support if the IDE's does not work as expected. You can achieve this by opening the *Command Palette* from the *View* menu item and executing the *TypeScript: Restart TS Server* command.

i

8. Using the Development Environment

This section describes how to connect the *Content Server* and the *Preview CAE*. It provides pointers to information on Jangaroo tools supporting the build process. Furthermore, some basic information on debugging Studio customizations is given.

8.1 Configuring Connections

CoreMedia Studio's Server application needs to be connected with the *Content Management Server* to access the repository and with the preview *CAE* to show the preview of the opened form. If you use *CoreMedia Blueprint*, everything is already configured properly for your local workspace. If you use a distributed environment you can either use the `local` Spring profile and provide the `in stallation.host` property with the value of the host name where your CMS services and databases are running.

Connecting with the Content Server

When you start the *Studio server* Spring Boot application (`apps/studio-server/spring-boot/studio-server-app`) with `mvn spring-boot:run` during development, you may configure the CMS connection by configuring the `installation.host` property in the `private` Spring profile (`src/main/resources/config/application-private.proper ties`) as described in the documentation of how to start Blueprint applications.

Refer to Chapter 2, *Docker Setup* in *Deployment Manual* to learn about building deployable artifacts.

Connecting with the Editorial Comments Database

When running `mvn spring-boot:run` you can configure the properties `editorial.comments.db.host=DB_HOST` or `editorial.com ments.datasource.url=DATASOURCE_URL` in the `application-private.properties` file to configure the *Editorial Comments* Database.

Connecting with the Preview CAE

When you start the *Studio* server application locally during development, you can configure the connection to the preview *CAE* in the `application-private.properties` file in the `src/main/resources` directory of `studio-server-app`. Simply change the value of the property `studio.pre viewUrlPrefix` to the URL prefix of your *CAE*.

The property `studio.previewControllerPattern` contains the configurable preview controller pattern. If it is empty or not defined, then *Studio* will use the default preview controller pattern `preview?id={0}`. If you want to use simple numeric IDs instead, then you can configure in the `studio.prop erties` file as follows: `studio.previewControllerPattern=pre`

`view?id={1}`. The placeholder `0` and `1` are representing the CoreMedia ID and the numeric ID, respectively.

Note that *Elastic Social* users and user comments do not have numeric IDs. Hence, you should configure `preview?id={0}`. However, when using `pre view?id={1}`, the placeholder `1` is replaced with the non-numeric ID as well and the preview application has to handle this special case or will fail to deliver.

8.2 Build Process

While the *CoreMedia Studio* server provides artifacts for use with Maven the client part provides packages for use with pnpm.

In the following section, you will find a description of some of the typical use cases that appear during *CoreMedia Studio* development using the *CoreMedia Project* workspace.

All following commands assume you have opened a command shell at the *CoreMedia Project* root directory.

Compiling the Studio Project

To create a clean build of all *CoreMedia* project modules, including all Studio server modules, run the following:

```
mvn clean install -DskipTests
```

To build *only* Studio server modules, run

```
mvn install -DskipTests -pl :studio-server-app -am
```

Studio Client: Base Apps and App Overlays

The *Studio* client packages are packaged into *apps*, where the so-called *base app* and *app overlay* are distinguished. A base app is a Sencha Ext JS app and includes the Ext JS framework, *Studio* core packages and generally all packages that participate in theming. Modules of a base app are included in the build of the Sencha Ext JS app and are thus *statically* linked into the app. An app overlay in contrast references a base app and adds further modules to this base app. These modules are not included in the build of the Sencha Ext JS app and instead can be loaded at runtime. Consequently, they are *dynamically* linked into the app.

The *CoreMedia Blueprint* features one *Studio* base app, namely `@coremedia-blueprint/studio-client.main.base-app` package with Jangaroo type `app`. In addition, there are two app overlays, the `@coremedia-blueprint/studio-client.main.app` packag and the `@coremedia-blueprint/studio-client.workflow.app` package with Jangaroo type `app-overlay`. While the former references `@coremedia-blueprint/studio-client.main.base-app` the latter references `@coremedia/studio-client.workflow.app` which is part of the *CoreMedia Core*.

Both app overlays are aggregated in a so called *apps* package which bundles all apps as static resources so they can be served via a web server.

To build *all Studio* client apps including their dependencies (mind the dots), run

```
pnpm -r --filter @coremedia-blueprint/studio-client.studio... run build
```

To build only the *Studio* client modules that are part of the main.base-app, run

```
pnpm -r --filter @coremedia-blueprint/studio-client.main.base-app... run
build
```

Running Studio

CoreMedia Studio consists of Studio Client, a client-side (browser) application, and Studio Server, a REST service, implemented in Java. For client-side-only development, it is recommended to only use workspace `apps/studio-client` and only run Studio Client locally, connecting to a Studio Server on some reference system. For full Studio development, you run Studio Server and Studio Client locally.

Running Studio Client

Unlike most CoreMedia application, Studio Client is not a Spring Boot application, so it is started differently, using the pnpm script `start` (for development purposes only). Prerequisite for this is that a complete *Studio* web application is already running *somewhere*. The `start` script starts an embedded Web server (express), serves some Studio client packages from your developer workspace and proxies all other requests to the remote *Studio Server* web application.

- Run complete Studio Client app from your machine, proxy all REST requests to Studio Server:

```
pnpm -r --filter ./apps/main/app run start
--proxyTargetUri=https://<studio-server-host>:<studio-server-port>
```

- Run app overlay from your machine, proxy everything else to the remote Studio:

This a special treat for app overlays. Consequently, it works for the *Blueprint*'s `studio-client.main.app` but also for every other (lightweight) app overlay that you define yourself for development purposes in your workspace. Here you have the option to just serve the app overlay's own Studio modules from your local machine and proxy everything else (REST calls, other client module code) to the remote *Studio*.

```
pnpm -r --filter ./apps/main/app run start --proxyPathSpec="/*"
--proxyTargetUri=https://<studio-server-host>:<studio-server-port>
```

In this development mode, resources are read from target directories of the individual Studio client packages. When TypeScript files are recompiled, the `start` script automatically serves the updated compiled JavaScript files. There is no need to stop and restart the process.

8.3 Debugging

CoreMedia Studio components and plugins consist of static resources (images, style sheets, JavaScript files) and JavaScript objects. Debugging a custom *CoreMedia Studio* component or plugin involves the following tasks:

- Check whether the static resources have been loaded

- Explore the runtime behavior of the customization, that is, the relevant JavaScript code or DOM nodes

In the following sections, tools and best practices for debugging your *CoreMedia Studio* customizations are described.

8.3.1 Browser Developer Tools

All modern browsers provide tools for web application debugging. These are usually simply called "Developer Tools" and can be invoked via a menu entry, a toolbar button, the **F12** key or the key combination **Ctrl+Shift+I**.

> **NOTE**
>
> While in the past CoreMedia did recommend to use Google Chrome, as of today, both Chrome and Firefox are very powerful for debugging. Just use the one you prefer. The documentation will feature descriptions for using the Google Chrome debugger.

All modern browser developer tools provide tabs for different tools:

- DOM Explorer / Element / Inspector — Inspect the page's actual DOM elements as a DOM tree, with the option to select an element on the rendered page to reveal it in the tree. Selected DOM tree nodes are highlighted on the rendered page. The DOM can be watched for changes and modified interactively.

- Console — All JavaScript messages and errors are logged to this console, and it provides a read–eval–loop for JavaScript expressions.

- Network — Inspect all HTTP network traffic between the client-side application and the server, static resources as well as Ajax (XHR) requests. Most developer tools offer to disable the cache while they are active, to make sure that you always load the most recent version of code and other resources you just changed.

- Debugger / Sources — Inspect all loaded JavaScript and CSS sources, set breakpoints to debug in step by step mode. Most modern developer tools allow you to change sources interactively with immediate effect.

- Profiles / Profiler / Audits / Memory / Analysis — Diverse tools to measure your web application's client-side and network performance and memory usage. Helpful to find memory leaks (see below) and track performance issues.

All browser developer tools offer a convenient way to navigate to a certain script file or Ext JS class: With the Sources / Debugger tab active, press **Ctrl-P** (note that this invokes the print dialog when the focus is not on the developer tools!) and just start typing the name of the class (file) you want to debug, and the list is filtered incrementally. Some tools even support typing camel case prefixes of the class name, for example to find the class `PreviewPanelToolbarBase` in Google Chrome, press **Ctrl-P** and type "PrevPaToBa" to quickly reduce the number of suggestions.

Opening a JavaScript file

To navigate to the desired line in the file, you can add a colon (:) and the line number directly after the file search term. To jump to a certain column in the line append another colon (:) followed by the column number. To navigate to (a column in) a different line, press **Ctrl-L** or **Ctrl-G** (Goto Line) and enter the line number (and a colon with column number).

A very efficient way to locate a certain line of a file in Google Chrome's Developer Tools (to set a breakpoint, for instance) when working with IntelliJ IDEA is as follows. In IDEA, jump to the very start of the line (press Pos 1 repeatedly until there). Then, press **Ctrl-Alt-Shift-C** ("Copy Reference"). IDEA's status line shows a message that the file/line reference has been copied to the clipboard. Switch to Chrome Developer Tool's **Sources** tab (**Alt-Tab** suffices when changing back and forth) and press **Ctrl-P**. Now paste the file/line reference and remove the file extension (also see section about source maps). Hitting Return, Chrome accepts the syntax file-path:line and takes you to the exact file and line.

Opening a *CoreMedia Studio* file in the debugger requires the source maps feature to be enabled in the developer tools settings. This is the default in Google Chrome. If not enabled, a file lookup with **Ctrl-P** will fail. To enable source maps in Google Chrome open the Developer Tools settings by pressing the **F1** key or by selecting in the control menu, see Figure 8.1, "Open Chrome Developer Tools settings" [107].
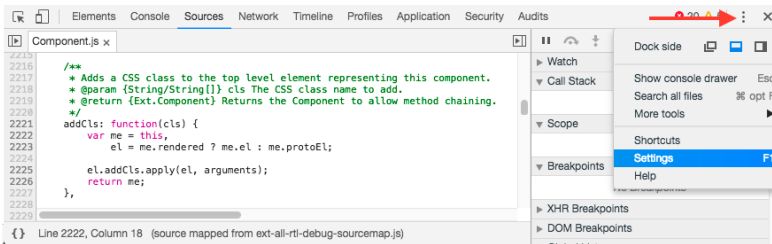
Figure 8.1. Open Chrome Developer Tools settings

Then enable the checkboxes marked red.
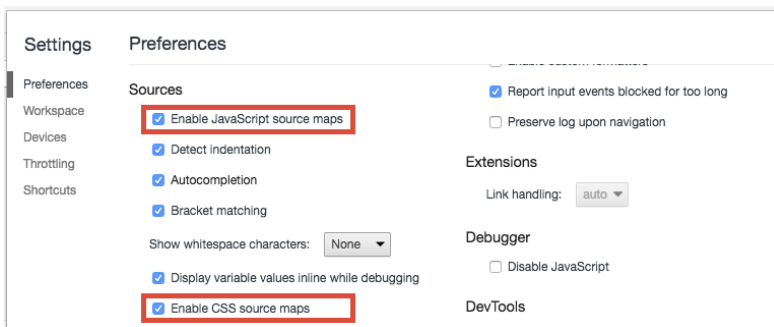
Figure 8.2. Enable Source Maps in Chrome Developer Tools settings

> **NOTE**
>
> While the lines of plain TypeScript source files will match the lines of the files your see in the browser this is not the case for classes compiled to Ext JS. The former will be shown as TypeScript files in the browser while the latter will be shown as JavaScript files.
>
> The reason for this difference is the transformation that both of these source files undergo when being compiled. While plain TypeScript source files will basically keep their structure when being transformed to JavaScript, all TypeScript files transformed to Ext JS will receive major structural changes so that using the TypeScript source files would debugging does not properly work. This is why you will see them as (non-minified) JavaScript files in the browser that are already transformed to Ext JS.

The debugger allows you to set breakpoints, to automatically pause on errors, to step through the script at runtime and to evaluate expressions in the current

scope of the script. In this context, the **Console** tab, see Figure 8.3, "Google Chrome Console" [108], is also very helpful, because it offers a JavaScript shell for direct interaction with the current script. The console displays the results of the expressions evaluated in the shell and also messages generated by the current script runtime. In Google Chrome you can also open and close the console in the **Sources** tab by pressing the escape key.
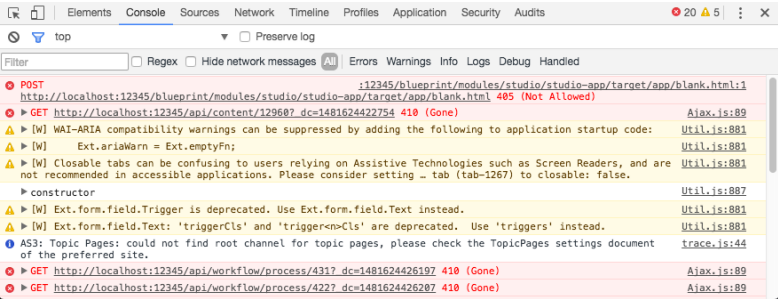
Figure 8.3. Google Chrome Console

Visit the Google Chrome Developer Tools website for more details.

8.3.2 Debugging Tips and Tricks

Studio Console Logging

By default, all JavaScript console errors that occur in Studio are logged in the backend as well. The errors are logged into the file `studio-console.log`. Additionally, the user can enable the *Log* button for debugging purposes. When using the hash parameter `joo.debug` a button with a counter will appear next to the user menu,which captures all log messages that are sent to the server as well.
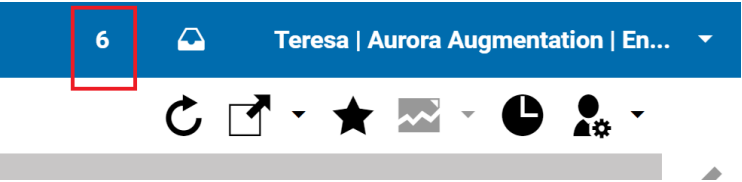
Figure 8.4. The Browser Console Log Button

A click on the button shows the console log messages. Longer messages provide a tooltip so that the full stacktrace of errors can be seen. The amount of stored messages is limited to the last 300 by default.

The logging is configurable via the Studio resource bundle `LogSettings_prop erties.ts` which may be overwritten. The properties file contains the following configuration options:

- *whitelist*: a comma separated list of messages. If a log message matches a part of one of these values, it is ignored for logging.

- *cache_size*: the number of messages kept in the browser log window (*100* by default).

Dump content to browser console

You can use a shortcut to dump a readable representation of a content item to the browser console. Open the content item in a form and press the shortcut **CTRL+ALT+D**.

```
******************************************************************************************************
Name:    Responsive Image Settings
Type:    CMSettings
Path:    /Settings/Options/Settings/Responsive Image Settings
Id:      coremedia:///cap/content/162
******************************************************************************************************
lifecycleStatus: in-production
previewUrl:      //preview.toko-ci-01-calau-tomcat-3-cms.coremedia.vm/preview?id=coremedia:///cap/content/162
------------------------------------------------------------------------------------------------------
locale:
master:
masterVersion: null
settings:
<Struct xmlns="http://www.coremedia.com/2008/struct" xmlns:xlink="http://www.w3.org/1999/xlink">
  <StringProperty Name="responsiveImageSettingsDescription">These settings are global fallback settings, if site specific settings are mi
  <BooleanProperty Name="enableRetinaImages">false</BooleanProperty>
  <StructProperty Name="responsiveImageSettings">
    <Struct>
      <StructProperty Name="portrait_ratio3x4">
        <Struct>
          <IntegerProperty Name="widthRatio">3</IntegerProperty>
          <IntegerProperty Name="heightRatio">4</IntegerProperty>
          <IntegerProperty Name="minWidth">300</IntegerProperty>
          <IntegerProperty Name="minHeight">400</IntegerProperty>
          <IntegerProperty Name="previewWidth">300</IntegerProperty>
          <StructProperty Name="0">
            <Struct>
              <IntegerProperty Name="width">300</IntegerProperty>
              <IntegerProperty Name="height">400</IntegerProperty>
            </Struct>
          </StructProperty>
          <StructProperty Name="1">
            <Struct>
              <IntegerProperty Name="width">600</IntegerProperty>
              <IntegerProperty Name="height">800</IntegerProperty>
            </Struct>
          </StructProperty>
          <StructProperty Name="2">
            <Struct>
              <IntegerProperty Name="width">1200</IntegerProperty>
              <IntegerProperty Name="height">1600</IntegerProperty>
            </Struct>
          </StructProperty>
        </Struct>
      </StructProperty>
```

Figure 8.5. Example of a content dump

Inspecting an Ext JS component in the developer tools console

The DOM elements of Ext JS components can be identified in the Studio DOM tree. The value of the *id* attribute of a DOM element resembles the xtype of the corresponding Ext JS component, for example, the issues window has xtype

`com.coremedia.cms.editor.sdk.config.issuesWindow`. The ID value is `com-coremedia-cms-editor-sdk-config-issuesWindow-nnnn` where `nnnn` is an arbitrary unique integer value. Be careful, the DOM element often contains subelements with similar id values, for example, there is a subelement with id value `com-coremedia-cms-editor-sdk-config-issuesWindow-nnnn-bodyWrap`. This DOM element does not represent an Ext JS component.

Now select and copy the id value from the DOM element. You get an Ext JS component from the id value by invoking the method `Ext.getCmp(id)` in the console. For example to inspect the issues window component enter:

```
c=Ext.getCmp("com-coremedia-cms-editor-sdk-config-issuesWindow-nnnn");
c.items.items;
```

Trigger the debugger when a component property is modified

Sometimes you want to know why a property of a certain Ext JS component was modified. You can trigger the Chrome debugger to stop at a breakpoint you define for the property change in the console of the developer tools. In the following example the debugger stops when you change the height of the issues window.

```
c=Ext.getCmp("com-coremedia-cms-editor-sdk-config-issuesWindow-nnnn");
c._height = c.height;
Object.defineProperty(c, "height",
                      {get: function() {return this._height;},
                       set : function(val) {debugger; this._height = val;}})
```

The first line assigns the issues window component to the variable `c` as described in the component inspection section above. The second line defines a new variable `_height` to store the height property value. The last lines define the getter and setter methods of the height property. The **debugger** command in the setter tells the debugger to stop at the same position. Now the user can analyze the call hierarchy, inspect other component values and continue debugging.

Debug and Understand CKEditor Data Processing

To debug the richtext editing component based on *CKEditor*, you may add a hash parameter `ckdebug` to the *CoreMedia Studio* URL. For details, see Section 10.4, "Debugging CKEditor 5" [458].

8.3.3 Tracing Memory Leaks

Ext JS applications can consume high amounts of memory in the browser. As long as memory is de-allocated when UI elements are disposed, the user has the choice to limit memory usage. But it becomes a problem when there are *memory leaks*. Fortunately, reloading the application's page (F5), with a few exceptions, frees memory again, but still, frequent reloading is undesirable for the user.

Memory leaks occur when an object is supposed to be no longer used, but undesired references to that object remain that keep it "alive", that is, from being garbage-collected. Such references are called *retainers*. In an Ext JS application, such retainers are typically

- Ext's component manager. It maintains a global list of all active components. See below how to tackle memory leaks caused by the component manager (component leaks).
- Event listeners. When attaching your event listener function to some object, that object retains the event listener function and every object in the scope of that function, typically at least `this`.
- Drop zones. Like for components, Ext keeps a global list of all active drop zones. So when your custom component creates a drop zones, remember to explicitly destroy it together with your component.

8.3.3.1 Component Leaks

If a component is destroyed, it is removed from the Ext component manager registry. If the component is a container, all its items are removed as well. But there are cases when components fail to be destroyed:

- If two items of the same container use the same itemId, Ext does not complain, but one of them is kept even if the container is destroyed.
- Components that are created manually via ComponentMgr.create() have to be destroyed manually unless they are added to the items of a container.

8.3.3.2 Memory Leaks Caused by Non-Detached Listeners

Always remove any listeners that you attach to an `Observable`, `Bean`, `ValueExpression`, or any other object that emits events. Even when using

the option `{single: true}`, the event might not have been fired at all when your component is destroyed.

A typical error pattern is to attach some method `handleFoo` as event listener, but by mistake hand in another method with a similar name `handleFuu` when intending to remove the listener. No error whatsoever is reported, because trying to remove a function as listener that is not in the current set of listeners is silently ignored by `Observable#removeListener()` and all other event emitters.

A useful utility to automate removing listeners is to use `Observable#mon()` instead of `Observable#on()` (alias: `Observable#addListener()`). `mon` does not attach the listener to the caller, but to the first parameter, but binds it to the lifetime of the caller. For example, when your custom component creates a DOM element `elem` and registers a `click` listener like so: `this.mon(elem, "click", handleClick)`, the listener is automatically detached when your component (the caller, `this`) is destroyed.

> ### CAUTION
>
> It never makes sense to call `comp.mon(comp, ...)`, because when a component is destroyed, it removes its own listeners, anyway. Using `comp.mon(comp, "destroy", handleDestroy)` even leads to the handler *never* being called, because a component removes all `mon` listeners already in its `beforedestroy` phase. In contrast, `comp.on("destroy", handleDestroy)` works as expected.

Not only components, but any objects that register event handlers, most prominently actions, have to detach all event handlers again.

As actions do not have a `destroy` event and `onDestroy` method like components, you have to override `addComponent()` and `removeComponent()` to detect when an action starts and ends being used by any component. Introducing a simple counter field starting with zero, you should acquire resources (for example, register event listeners, populate fields) when `addComponent()` is called while the counter is zero before increasing, and release resources (remove event listeners, set fields to `null`) when `removeComponent()` is called while the counter is zero after decreasing.

To minimize the impact in case event listeners are not detached, and to avoid cyclic dependencies, keep the scope of any event handler function or method as small as possible. In the optimal case, the event handler function is a private static method, for example if it just toggles a style class of the DOM element given in the event object:

```
#attachListeners():void {
  const el = this.getEl();
```

```
    // bad style: using an anonymous function that
    // does not need its outer scope at all:
    el.addListener("mouseover", e => e.getTarget().addClass("my-hover"));
    // good style: for such cases, use a static method:
    el.addListener("mouseout", this.#removeHoverCls);
}

static #removeHoverCls(e:IEventObject):void {
    e.getTarget().removeClass("my-hover");
});
```

If your event handler only needs access to `this` (this current component instance), declare it as a method as opposed to an anonymous function:

```
private #hoverCounter:int = 0;

#attachListeners():void {
    const el = getEl();
    // bad style: using an anonymous function that
    // only needs to access "this":
    el.addListener("mouseover", e => ++this.++hoverCounter);
    // good style: for such cases, use a (non-static) method:
    el.addListener("mouseout", bind(this, this.#countHoverEvent));
}

#countHoverEvent(e:IEventObject):void {
    ++this.hoverCounter;
});
```

In TypeScript, like in JavaScript, anonymous or inline functions have lexical scope, that is they can access any variable declared in the surrounding function or method. Since this scope usually contains a reference to the object that emits events (here: `el`), and that object stores your event handler function in its listener set, you create a cyclic reference between the two. Cyclic references are not bad per se, because garbage collection can handle them if all objects contained in the cycle are not referenced from "outside". But firstly, as long as any of the objects is kept alive, all others are retained, too, and secondly, as discussed below, this makes finding the real culprit for memory leaks harder.

8.3.3.3 Memory Leaks Caused by Other References

Any reference to an object can cause it to stay alive. Thus, to find unwanted retainers, it makes sense to null-out all references a component keeps in its `onDestroy()` method, like in this code sketch:

```
class MyComponent extends Component {

    #foo:SomethingExpensive;

    constructor(config:Config<MyComponent> = null) {
        super(config);
        this.#foo = new SomethingExpensive();
    }

    protected onDestroy():void {
```

```
    this.#foo = null;
    super.onDestroy();
  }
}
```

You have to be careful that even after your component has been destroyed, certain asynchronous event callbacks may occur. Your event handlers have to be robust against fields already being `null`. Consider this example using a fictitious `timeout` event:

```
class MyComponent extends Component {

  #foo:SomethingExpensive;

  constructor(config:Config<MyComponent> = null) {
    super(config);
    this.#foo = new SomethingExpensive();
    this.addListener("timeout", this.#handleTimeout);
  }

  #handleTimeout():void {
    // Although we remove the listener in onDestroy,
    // an event may already be underway, so foo may
    // already be null in time it arrives:
    if (this.#foo) {
      this.#foo.doSomething();
    }
  }

  protected onDestroy():void {
    this.removeListener("timeout", this.#handleTimeout);
    this.#foo = null;
    super.onDestroy();
  }
}
```

8.3.3.4 Detecting Memory Leaks

To check whether your customized Studio contains any component leaks, proceed as follows.

1. First, you need to prepare your Studio carefully.

   • Close all tabs in your Studio.

   • Reload Studio.

   • Before opening any tabs, get rid of any tab reuse configuration by entering the following in your browser's JavaScript console:

   ```
   com.coremedia.ui.util.reusableComponentsService.reset()
   ```

2. Open the suspicious UI, for example, a content tab containing your new property field. If you want to check a content tab as a whole, you need to click through all subtabs and expand all collapsible panels as they contain lazy items. Wait until everything is rendered correctly and close the UI again. This

is to ensure that helper components (a context menu, for instance) that are shared between instances and created with the first instance do not blur the view on real component leaks.

3. Store a snapshot of the current Ext component manager registry by executing the following command in the JavaScript console:

```
before = Ext.ComponentMgr.getAll()
```

4. Open and close the UI again like before. Take a second snapshot:

```
after = Ext.ComponentMgr.getAll()
```

5. In theory, the second snapshot should be exactly equal to the first. But some components are recreated occasionally, which is not bad if their old version is correctly destroyed. Thus, the first check is to simply compare the component count:

```
after.length - before.length
```

6. If there are more components in the second snapshot (positive difference), next goal is to determine their component type (xtype). This is achieved by the following code:

```
newComponents = after.filter(c => before.indexOf(c) === -1)
```

7. To get an overview of the new components, count how many components are of which type (xtype), using the following code:

```
byXtype = {};
newComponents.forEach(c => {
  const xtype = c.xtype;
  byXtype[xtype] = (byXtype[xtype] || 0) + 1;
});
byXtype
```

8. For custom Ext JS components, the xtypes in the resulting map indicate a unique identifier, from which you can derive the npm package.

To check whether your customized Studio contains any other memory leaks, proceed as follows.

1. Open the suspicious UI, for example, a content tab containing your new property field. Wait until everything is rendered correctly and close the UI again. In addition to what has been said regarding component leaks, this is to ensure that all needed data objects (remote beans) have been fetched from the server. In Studio, remote beans are cached, so they are not garbage-collected on purpose.

2. Take a heap snapshot. In Google Chrome, this is achieved as follows: In Developer Tools, select "Profiles". Under "Select profiling type", the option "Take Heap Snapshot" is preselected. The third option, "Record Heap Allocations", claims to be suitable for isolating memory leaks, but CoreMedia founds comparing heap snapshots simpler. Press the button "Take Snapshot". In the left column, Chrome adds an icon for the snapshot and shows a progress indicator while it is recorded. When recording is finished, the heap snapshot is shown as an expandable list of all JavaScript objects is shown, grouped by their (internal) type.

3. Repeat opening and closing the suspicious UI like in step 2.

4. Take a second heap snapshot. To do so, either you have to select "Profiles" on the left and proceed like in step 3, or simply click the "record" button (a gray filled circle).

5. Where the label "Summary" is shown, you can switch to "Comparison". The first snapshot is automatically selected for comparison. Now, you no longer see all objects, but only those that either have been removed ("Deleted") or have been created ("New") between snapshot one and two ("Delta").

Since the application is in the same state after opening and closing the suspicious UI, ideally, the comparison would be empty. In practice, however, this can never be achieved. What you have to look for are "expensive" objects, consuming lots of memory ("Alloc. Size", "Freed Size", "Size Delta"). The focus is "Size Delta", which tells you how much memory has leaked between snapshot one and two.

Since you cannot do much about memory leaks in Ext JS or in Studio Core, concentrate on your own extensions. Fortunately, Chrome's Profiler manages to find the Ext JS class names of objects. Thus, you can filter the comparison by the name of your TypeScript class, and it will only show objects of that class whose set of instances has changed.

Each entry in the upper part represents the set of all object. To inspect a concrete instance and its retainers, you have to expand the entry using the triangle / arrow, and select an instance from the expanded list. For the selected instance, all retainers are now shown in the lower part of the heap analyzer.

Each root node in the "Retainers" tree represents the property of the instance directly referencing (retaining) the instance selected in the upper part. By expanding any node, you can drill down into its retainers, until you reach an instance that is globally retained, usually by the global JavaScript object `window`.

By default, the heap analyzer sorts child nodes by "Distance" (first column), so that you inspect the longest path when always expanding the first child node. This most likely, but not necessarily leads you to the "culprit" retainer, that is the instance that should no longer refer to the inspected instance. Many other retainers result from cyclic references, that is, they would have been garbage-

collected together with the inspected object, if the "culprit" did not reference the inspected object. This is why it is recommended to reduce the number of references by cleaning up fields and listeners, even if this would not have been necessary without the memory leak (see above).

Hopefully, by inspecting retainers, you'll find a listener that has not been detached or a global reference that should be removed on destroy. If not, you can still clean up your component or action so that it at least leaks less memory.
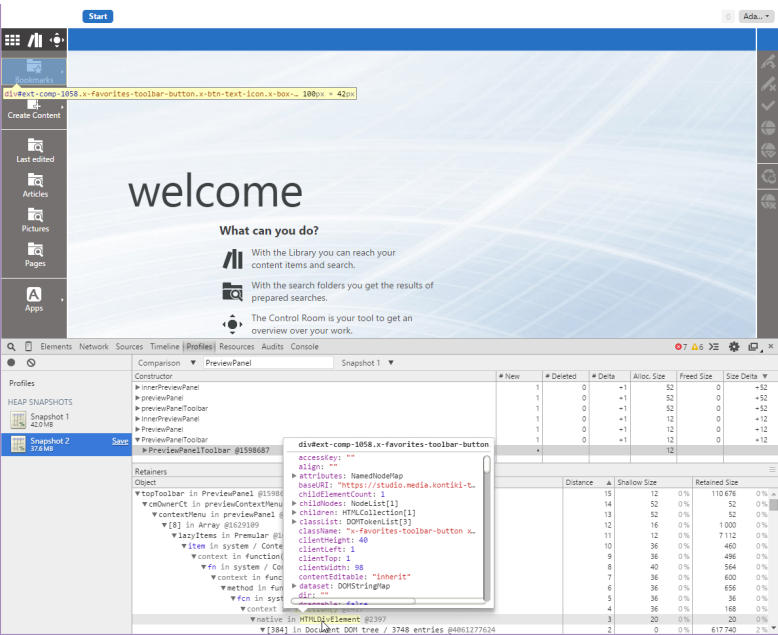
Figure 8.6. Google Chrome's Developer Tools Support Comparing Heap Snapshots

The screenshot shows Google Chrome's developer tools in action. Blueprint Studio has been loaded in debug mode. A content tab has been opened and closed again, "Snapshot 1" has been taken, and after repeating this, "Snapshot 2" has been added. Then, both snapshots have been compared as described above and the developer has filtered for "PreviewPanel". The only retained instance of `PreviewPanelToolbar` has been selected, so that its retainers are shown in the lower part. In the expanded path, the mouse hovers over the almost-leaf `HTMLDivElement`, which is also automatically highlighted in the Studio UI. This reveals the culprit of the memory leak: The highlighted "Bookmarks" button in the favorites toolbar is the one who keeps an indirect reference to the `Pre viewPanel` through its context menu.

9. Customizing CoreMedia Studio

This chapter describes different customization tasks for *CoreMedia Studio*.

- Section 9.16, "Customizing the Library Window" [203] describes how you can customize the Library Window.
- Section 9.17, "Studio Frontend Development" [211] describes how to work with the frontend framework to create own studio styles or customize the existing studio appearance.
- Section 9.18, "Work Area Tabs" [226] describes how to integrate your own tab to *CoreMedia Studio*. how to determine which tabs are opened at start time and how to add actions to the work area tab context menu.
- Section 9.22, "Configuring Maximum Length for TextFields" [251] describes how you can set a limit to the length of a `TextField`'s or a `TextArea`'s input value.
- Section 9.19, "Re-Using Studio Tabs For Better Performance" [233] describes how to configure the reusability of `WorkArea` content form tabs for better performance.
- Section 9.20, "Dashboard" [237] describes how to configure the dashboard of *CoreMedia Studio*.
- Section 9.21, "Configuring MIME Types" [248] describes how to configure MIME types for additional file types for *CoreMedia Studio*.
- Section 9.23, "Server-Side Content Processing" [252] describes how the processing of content can be influenced by custom strategies and how inconsistencies in the content structure can be detected or avoided.
- Section 9.24, "Available Locales" [272] describes how *CoreMedia Studio* assists the user in choosing a locale and how to configure the available locales.
- Section 9.25, "Toasts and Notifications" [273] describes how to enrich *CoreMedia Studio* with custom notifications.
- Section 9.26, "Annotated LinkLists" [278] describes how to enrich LinkLists with custom properties.
- Section 9.27, "Thumbnails" [284] describes how thumbnails are resolved and how to enrich LinkLists with images.
- Section 9.28, "Custom Workflows" [289] describes how you can customize Workflows by adding custom parameters.
- Section 9.31, "User Manager" [353] describes how you can customize the Studio's user manager.
- Section 9.35, "Quick Search Configuration" [383] describes how to configure Studio's quick search dialog.
- Section 9.38, "Developing Studio Apps" [390] describes how to extend the *CoreMedia Studio* with additional custom apps which run in their own browser window.

9.1 General Remarks On Customizing (Multiple) Studio Apps

Since *CoreMedia v11*, *Studio* consists of two client apps, the *Main / Content App* and the *Workflow App*. It is important to note that these are distinct apps that are customized separately from each other. As for all CoreMedia applications, customizations may be carried out in terms of classical Blueprint extensions or in terms of the newer concept of application plugins. Application plugins are covered in detail in Section 4.1.6.2, "Plugins for Studio Client" in *Blueprint Developer Manual* . Blueprint extensions are covered in this section

> ### Restricted Set of Supported Customizations for the Workflow App
>
> For the *Workflow App* only a very restricted set of customizations is supported although other customizations might be technically possible. Most of what is described in Chapter 9, *Customizing CoreMedia Studio* [118] only applies to the Studio *Main App*. More precisely, only *Apps Menu* customizations (see Section 9.2, "Customizing Entries to the Apps Menu" [123]), content type customizations (see Section 9.5, "Document Type Model" [145]) and workflow customizations (see Section 9.28, "Custom Workflows" [289]) are supported for the *Workflow App*.

Studio Client Apps Extension Points

The Studio *Main App* and the *Workflow App* are customized separately. Section 4.1.6.2, "Plugins for Studio Client" in *Blueprint Developer Manual* describes that you can add separate application plugins to both apps. If you customize in terms of classical *Blueprint* extensions, you need to be aware of separate extension points.

- *studio-client.main* (defined in `blueprint/apps/studio-client/apps/main/extension-config/extension-dependencies/package.json`):

  Main App extension point for extensions that can de dynamically linked into the application (no Ext JS theming is done in the extension modules). This corresponds to the `studio-dynamic` Maven extension point in *CoreMedia v2107* and earlier.

- *studio-client.main-static* (defined in `blueprint/apps/studio-cli ent/apps/main/extension-config/static-extension-depend encies/package.json`):

  Main App extension point for extensions that need to be statically linked into the application (Ext JS theming is done in the extension modules). This corresponds to the `studio` Maven extension point in *CoreMedia v2107* and earlier.

- *studio-client.workflow* (defined in `blueprint/apps/studio-cli ent/apps/workflow/extension-config/extension-dependen cies/package.json`):

  Workflow App extension point. The distinction between dynamic and static linking does not apply here because for the *Workflow App*, only certain customizations are supported, see above.

Extension modules for both apps are located under `blueprint/apps/stu dio-client/apps/{APP_NAME}/extensions`. They have a `core media.projectExtensionFor` entry in their `package.json` file, for example:

```
"coremedia": {
  "projectExtensionFor": "studio-client.workflow"
},
```

Example 9.1. Marking a module as an extension for the Workflow App

Shared Customization Code

Although the *Main App* and the *Workflow App* are customized separately, it of course makes sense for certain use cases to develop shared code that customizes both apps in the same way. A good example of this are content type localizations (see Section 9.5, "Document Type Model" [145]). For the customization mechanism of application plugins, this is straightforward: Just add the same plugin to both apps. For the customization mechanism of *Blueprint* extensions it is only slightly more complex and described here.

As described above, extension modules for different apps are located under different `.../{APP_NAME}/extensions` folders. To have shared customization code it is recommended to have non-extension modules under `blue print/apps/studio-client/shared/ext/extensions` (for Ext JS modules) or under `blueprint/apps/studio-client/shared/js/ex tensions` (for non-Ext JS modules) and then let the extension modules of each app depend on these shared modules.

Customization Entry points

There are two ways to bootstrap customization code. The first (traditional) one is *StudioPlugins* as described in Section 9.3, "Studio Plugins" [130]. This approach has been around for as long as Studio itself and is the way to go when Ext JS components are to be customized. In addition, some of CoreMedia's pre-defined customization options only work as *StudioPlugin* configurations.

A newer and more light-weight way to bootstrap custom code are auto-loaded scripts. The usage is much simpler than for *StudioPlugins* and it is the preferred way for any customizations other than customizing Ext JS components. An auto-loaded script is simply set up by putting a corresponding entry into the `jangaroo.config.js` file of a module as in the following example (where the script is named `initMyCustomCode.ts` and is located under the `src` folder of the module):

```
module.exports = jangarooConfig({
  ...
  autoLoad: [
    "./src/initMyCustomCode",
  ],
  ...
});
```

Example 9.2. Bootstrapping auto-loaded scripts

9.2 Customizing Entries to the Apps Menu

The *Apps Menu* is part of the *Side Bar* that each Studio app has. It can be opened via the burger menu button in the top-left corner of a Studio app.
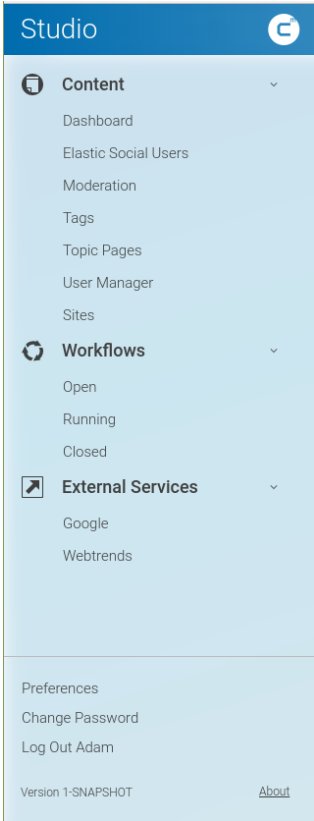
Figure 9.1. The Apps Menu inside the Side Bar of Each Studio App

One speciality of the *Apps Menu* is that it includes entries from all Studio apps. For example, the entries under *Workflows* are shortcuts for the *Workflow App* while all others are shortcuts for the *Main App.* The entries should fulfil certain conditions:

- The entries need to be dynamic in the way that they are tied to the existence of their respective app. If for example the *Workflow App* was removed from the Studio client build, its *Apps Menu* shortcuts should also vanish.
- The shortcuts for an app need to be in the *Apps Menu* even if this app is not yet opened. For example, if a user worked with the main Studio and the *Workflow App* was not yet opened in a browser window/tab, the *Workflow App* shortcuts should still be present in the menu.
- The complete set of *Apps Menu* shortcuts should not be configured for each app separately as this does not scale with more apps. Instead, each app should just declare, which individual shortcuts it adds to the menu.

App Manifests

To meet the conditions from above, a customization approach based on *app manifests* was chosen. It is based on the Web standard Web App Manifests but adds some *CoreMedia*-specific attributes. Via its manifest, each app defines its shortcuts and all of them appear in all *Apps Menus* of all apps.

The manifest for a Studio app is assembled by the build process. To this end, multiple modules can add app manifests fragments which are deep-merged to obtain the complete manifest. Modules add their manifest fragments as part of their `jangaroo.config.js` file in the module's root folder. The complete assembled manifest for an app lies under `APP_MODULE_PATH/build/manifest.webmanifest`. In addition, the manifests are locale-specific so that you also find the files `manifest-de.webmanifest` and `manifest-ja.webmanifest` for German and Japanese.

App manifests contain a lot information but this section focuses on the *shortcuts* part of the manifests.

The grouping of shortcuts under the collapsible sections of the menu mainly follows the question, which app defines the shortcuts. So they are grouped under *Content* for the *Main App* and *Workflows* for the *Workflow App*. However, a `cm Category` can be defined for a shortcut (see below). For the apps menu, categories normally do not have an impact. The exception is when you use the config option `topLevelCategories` of the `AppsMenu`. In that case shortcuts of the configured categories are assembled under a joint section alongside the sections for the apps. For example, even though *Google* is a shortcut of the *Main App*, it appears under *External Services* in the menu because this category is configured to be a top level category.

Defining App Shortcuts

There are two kinds of app shortcuts, (1) app path shortcuts and (2) service shortcuts.

App Path Shortcuts

App path shortcuts assume that the app can deal with different app sub-paths. For example, if you switch to the *Pending Workflows* overview list of the app, you can see that the browser URL has the hash parameter `#path=pending`. So an app path shortcut simply sets the `path` hash parameter of the app to a specific value and assumes that the app reacts to this in some way.

Examples for app path shortcuts can be found in the manifest fragment for the *Workflow App* module (part of the core).

```
module.exports = jangarooConfig({
  ...
  appManifests: {
    de: {
      shortcuts: [
        {
          name: "Offen",
        },
        {
          name: "Laufend",
        },
        {
          name: "Abgeschlossen",
        },
      ],
    },
    en: {
      ...
      categories: [
        "Workflow",
      ],
      ...
      shortcuts: [
        {
          cmKey: "cmInbox",
          name: "Open",
          url: "inbox",
          icons: [
            {
              src: "appIcons/inbox_24.svg",
              sizes: "24x24",
              type: "image/svg",
            },
            {
              src: "appIcons/inbox_192.png",
              sizes: "192x192",
              type: "image/png",
            },
            {
              src: "appIcons/inbox_512.png",
              sizes: "512x512",
              type: "image/png",
            },
          ],
```

```
        },
        {
          cmKey: "cmPending",
          name: "Running",
          url: "pending",
          ...
        },
        {
          cmKey: "cmFinished",
          name: "Closed",
          url: "finished",
          ...
        },
      ],
    },
  },
  additionalLocales: [
    "de",
    "ja",
  ]
});
```

Example 9.3. App Path Shortcuts for the workflow app

An app path shortcut defines an `url` property that is exactly the value that will be set for for the `path` hash parameter of the app's URL. In addition a `name` and a unique `cmKey` are set. Icons in different sizes for the shortcut are optional. If they are provided they need to reside in the *APP_MODULE_ROOT/sencha/ap-plcons* folder of the module.

The example also shows how different locales are handled. Only selected properties need to be overwritten, everything else is kept from the manifest of the base locale.

Service Shortcuts

The *Main App* does not handle app paths. Instead, service shortcuts are used. Service shortcuts do not change the app path in any way. Instead, an action inside the corresponding app is triggered to display something. An example is the *Tags* view of the *Main App*. Instead of setting a sub-path of the app, a new *Studio* tab for the *Tags* sub-app is opened.

To obtain this behaviour, first of all a corresponding service needs to be set up in the associated app. For the *Tags* sub-app this is done in the `TaxonomyStudioPlugin` in the *Blueprint*.

```
studioAppsContext._.getShortcutRunnerRegistry().registerShortcutRunner({
cmKey: "cmTaxonomy" }, (): void => {
  const openTagsAction = new OpenTaxonomyEditorAction();
  openTagsAction.execute();
});
```

Example 9.4. Registering a Service Method to Trigger the Tags App

A sub-app launcher is registered for the key `cmTaxonomy` which simply triggers the `OpenTaxonomyEditorAction`.

With such a sub-app launcher service in place, service shortcuts can be added to the manifest. For the example of the *Tags* sub-app, this is done in the `jangaroo.config.js` file of the `blueprint/apps/studio-client/apps/main/extensions/taxonomy` module itself.

```
module.exports = jangarooConfig({
  ...
  appManifests: {
    en: {
      ...
      cmServiceShortcuts: [
        {
          cmKey: "cmTaxonomy",
          cmOrder: 30,
          cmCategory: "Content",
          name: "Tags",
          url: "",
          cmAdministrative: true,
          cmGroups: ["global-manager", "taxonomy-manager", "developer"],
          cmService: {
            name: "launchSubAppService",
            method: "launchSubApp",
          },
        },
      ],
    },
  },
});
```

Example 9.5. Service Shortcut for the Tags Sub-App

The *cmKey* parameter must match the key that was used above when registering a sub-app launcher. Under `cmService` you define that the sub-app launcher mechanism should be used to bring the *Tags* sub-app to life.

Postprocessing App Manifests

As mentioned above, the app manifests are assembled from multiple fragments, coming from different modules. Some of the modules are part of the core - like the user manager, others are part of the project. App manifests as part of the project code can be customized to meet the project use cases. Unfortunately, app manifests from the core cannot be customized directly.

To customize the app manifests from the core, we recommend to post-process the assembled app manifests using a node script. The script can be part of the project code and can be executed after the build process. To this end, extend the `build` property in the `scripts` section of the `package.json` file of the project:

```
"build": "download-plugins ./build/additional-packages && jangaroo build &&
 node ./postprocess-manifest.js",
```

Example 9.6. Extending the build script by a postprocessor

The script `postprocess-manifest.js` can then be used to modify the assembled app manifests. The following example shows how to set the `cm Groups` property of the Moderation app so that only the users of the group `moderators` can see the app in the menu.

```
#! /usr/bin/env node
"use strict";

const fs = require("fs");
const path = require('path');

console.log("overwrite moderation service manifest");

// Array of JSON files to manipulate. The files are located in the dist
folder. Add more files if needed.
const files = ['manifest.webmanifest', 'manifest-de.webmanifest',
'manifest-ja.webmanifest'];

// Function to postprocess JSON data. Customize for your use cases
const postprocessJsonData = (jsonData) => {
  jsonData.cmServiceShortcuts = jsonData.cmServiceShortcuts.map(shortcut =>
 {
    // Allow only the group "moderators" to the Moderation app
    if (shortcut.cmKey === 'cmModeration') {
      shortcut.cmGroups = ['moderators'];
    }
    return shortcut;
  });

  return jsonData;
}

files.forEach(fileName => {
  const filePath = path.join(__dirname, 'dist', fileName);

  fs.readFile(filePath, 'utf8', (err, data) => {
    if (err) {
      console.error("Error reading the file:", err);
      return;
    }
    try {
      // Parse the JSON data
      let jsonData = JSON.parse(data);

      // postprocess the JSON data using the function
      jsonData = postprocessJsonData(jsonData);

      // Convert back to JSON string
      const updatedData = JSON.stringify(jsonData, null, 2);

      // Write the updated data back to the file
      fs.writeFile(filePath, updatedData, 'utf8', (err) => {
        if (err) {
          console.error("Error writing the file:", err);
          return;
        }

        console.log("File successfully updated!");
      });
```

```
    } catch (parseErr) {
      console.error("Error parsing the JSON:", parseErr);
    }
  });
});
```

Example 9.7. Script to Postprocess the Assembled App Manifests

The script reads the assembled app manifests (of different locales), modifies them and writes them back to the file system. The postprocessJsonData function can be used to modify the manifests in any way. The following example shows how to exclude the *Elastic Social Users* app from the apps menu.

```
const postprocessJsonData = (jsonData) => {
  jsonData.cmServiceShortcuts = jsonData.cmServiceShortcuts
    // Exclude Elastic Social Users App for all users
    .filter(shortcut => shortcut.cmKey !== 'cmElasticSocialUsers')
    .map(shortcut => {
    // Allow only the group "moderators" to the Moderation app
    if (shortcut.cmKey === 'cmModeration') {
      shortcut.cmGroups = ['moderators'];
    }
    return shortcut;
  });

  return jsonData;
}
```

Example 9.8. Exclude an App using Postprocessor

9.3 Studio Plugins

In Section 9.1, "General Remarks On Customizing (Multiple) Studio Apps" [120], two ways of bootstrapping custom code were introduced, Studio plugins and auto-loaded scripts. While auto-loaded scripts are a more light-weight and easy to use approach, Studio plugins come with more utility and pre-fabrication for customizing Ext JS components. In addition, many of CoreMedia's pre-defined Studio customizations are only available as Studio plugin configurations. The `extension` modules in the *CoreMedia Blueprint workspace* demonstrate the usage of the Studio plugin mechanism, and define several plugins for Studio.

> **CAUTION**
>
> Note that a Studio plugin is not to be confused with an *Ext JS* component plugin. The former is an application-level construct; Studio plugins are designed to aggregate various extensions (custom UI elements and their functional code, together with the required UI elements to trigger the respective functionality). The latter means a per-component plugin and is purely an *Ext JS* mechanism. This section deals with Studio plugins; *Ext JS* plugins are described in Section 5.1.2, "Component Plugins" [36]. In this manual, the terms *Studio plugin* and *component plugin* are used, respectively, to avoid ambiguity.

Examples for *CoreMedia Studio* extension points that plugins may hook into are:

- Localization of content types and properties
- Custom forms for content types
- Custom collection *thumbnail view*, and custom columns in collection *list view*
- Custom tab types (example in Blueprint: Taxonomy Manager tab)
- Custom library search filters
- Allowed image types and respective blob properties for drag and drop into rich text fields
- Additional extensions to extension menu
- Content types without a valid preview

A plugin for *CoreMedia Studio* usually has the following structure:
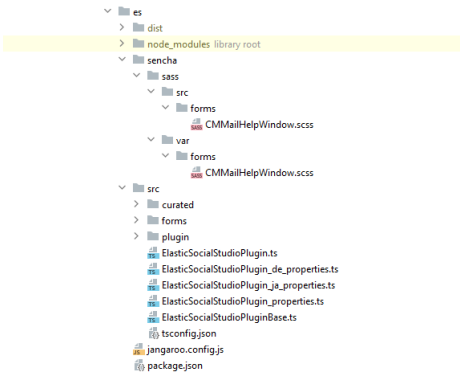
Figure 9.2. Plugin structure

The example above depicts the layout of a typical Studio module in the *Core-Media Blueprint workspace*. All plugins contain a `package.json` file that defines the dependencies of the plugin. The actual source code goes into the subdirectories `src` and `sencha`. The former contains TypeScript code, the latter Sass files in the `sass` subfolder and additional static resources such as images or CSS files in the `resources` subfolder not shown in the example. The `jangaroo.config.js` file registers the Studio plugin, see further below.

Structure of example

For example, the module `es-studio` holds a resource bundle `ElasticSocialStudioPlugin_properties` and the mein plugin file `ElasticSocialStudioPlugin.ts` (declaring the plugin and its applicable rules and configuration) under `src`. In addition, further Typescript source code files are held in several sub-folders under `src`.

Each plugin is described in a TypeScript file like `ElasticSocialStudioPlugin.ts`. This file declares the plugin's rule definitions (that is the various *Studio* extension points that this plugin hooks into) and configuration options. For many defining these rules and configuration TypeScript file is sufficient for a plugin declaration. However, you can of course also run arbitrary further Typescript code as part of your plugin's initialization.

The Main Class

The main class of a plugin shall be defined as TypeScript code. In the example in Figure 9.2, "Plugin structure" [131] the main class is `ElasticSocialStudioPlugin`. For your own plugins, it is recommended to use a name schema like <your plugin name>StudioPlugin.

The main class for a plugin must implement the interface `EditorPlugin`. The interface defines only one `init()` method that receives a context object im-

plementing `IEditorContext` as its only parameter, which is supposed to be used to configure *CoreMedia Studio*.

You can simply implement the interface in your source code. However, Studio also provides a base TypeScript class to inherit from, namely `StudioPlugin` which not only implements the `EditorPlugin` interface, it also delegates the `init()` call to all Studio plugins specified in its `configurations` config option.

The `IEditorContext` instance handed in to the `init()` method can be used for the following purposes:

- Configure which content types can be instantiated by the *CoreMedia Studio* user. This basically restricts the list of content types offered after clicking on the Create Document Icon in the Collection View (see Section 9.5.6, "Excluding Content Types from the Library" [160] for details). Note that only those content items are offered in the create content menu that the current user has the appropriate rights for in the selected folder - excluded content types will be placed on top of that rule (that is, you can exclude content type X from the menu even when the user has technically the rights to create content items of type X).
- Configure image properties for display in the thumbnail view and for drag and drop
- Register hooks that fill certain properties after initial content creation (see Section 9.5.7, "Client-side initialization of new content items" [161] for details)
- Add properties to the localization property bundles, or override existing properties (see Section 9.4, "Localizing Labels" [141] for details)
- Get access to the central bean factory and the application context bean
- Get access to the REST session and indirectly to the associated repositories
- Register content types for which *Studio* should not attempt to render an embedded preview
- Register a transformer function to post-process the preview URL generated for an existing content item for use in the embedded preview
- Get access to persistent per-user application settings, such as the tabs opened by the user or custom search folders
- Register symbol mappings for pasting external text from the system clipboard into a RichText property field, which can be useful when you have to paste content from Microsoft Word with special non-standard characters

Note that a Studio plugin's `init()` method is allowed to perform asynchronous calls, which is essential if it needs server-side information (access user, groups, Content, and so on) during initialization. *CoreMedia Studio* waits for the plugin to handle all callbacks, only then the next plugin (if any) is initialized and eventually, *CoreMedia Studio* is started. However, you cannot use `setTimeout()` or `setInterval()` in Studio plugin initialization code!

Plugin Rules

The other essential part of a *CoreMedia Studio* plugin is the plugin rules it declares in its `rules: []` element. Plugin rules are applied to components whenever they are created, which allows you to modify behavior of standard *CoreMedia Studio* components with component plugins. The `ElasticSocialStudioPlugin` plugin, for example, declares rules that add content forms for elastic social.

The studio plugin file consists of one "rules" element that contains component elements. The components can be either identified by their global id or by namespace and xtype. For the latter case, you need to declare the required namespace(s) in the root tag of the plugin file. You can read a Studio plugin rule like this: "Whenever a component of the given xtype is built, add the following component plugin(s)."

The rules element

You can use predefined Ext JS component plugins to modify framework components. The `ElasticSocialStudioPlugin` plugin, for example, uses the `AddItemsPlugin` to add content forms to the `CommentExtensionTabPanel`.

In the `ElasticSocialStudioPlugin`, custom forms for the elastic social content types are added by using the `AddTabbedDocumentFormsPlugin` (which is a component plugin).

!

CAUTION

While in simple cases, the items to add can be specified directly inline in the Studio plugin TypeScript file, this is generally not recommended.

The reason is that the Studio plugin class is instantiated as a singleton, and all TypeScript objects that are not components or plugins, most prominently Actions, are instantiated immediately, too. This means that Actions are instantiated (too) early, and that a plugin rule may be applied several times with the same Action instance, leading to unexpected results.

The best practice is to move the whole component plugin to a separate TypeScript file and reference this new plugin subclass from the Studio plugin rule. Since the new plugin is referenced by its ptype, a new plugin instance and thus a new Action instance is created for each application of the plugin rule as expected.

The Ext JS plugins of any component are executed in a defined order:

Execution order

1. Plugins provided directly in the component definition are initialized

2. Plugins defined in Studio plugin rules, starting with the plugins for the most generic applicable `xtype`, then those with successively more specific `xtypes`.

3. Plugins configured for the component's ID

If that specification does not unambiguously decide the order of two plugins, plugins registered earlier are executed earlier. To make sure that a certain module's Studio plugins are registered after another module's Studio plugin, the former module must declare a package.json dependency on the latter module. This way, the Studio plugins run and register in a defined order.

For your own Studio plugin, you might want to use the file from the *CoreMedia Project* workspace as a starting point. The name of the Studio plugin file should reflect the functionality of the plugin, for example `<My-plugin-Name>Studio Plugin.ts` for better readability.

The following example shows how a button can be added to the actions toolbar on the right side of the work area:

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";

import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import ActionsToolbar from
"@coremedia/studio-client.main.editor-components/sdk/desktop/ActionsToolbar";

import AddActionsToolbarItemsPlugin from "./AddActionsToolbarItemsPlugin";

class AddButtonToActionsToolbarPlugin extends StudioPlugin {

  constructor(config: Config<AddButtonToActionsToolbarPlugin>) {
    super(ConfigUtils.apply(Config(AddButtonToActionsToolbarPlugin, {
      //...
      rules: [
        Config(ActionsToolbar, {
          plugins: [
            Config(AddActionsToolbarItemsPlugin, {}),
          ],
        }),
      ],
    }), config));
    //...
  }
}

export default AddButtonToActionsToolbarPlugin;
```

Example 9.9. Adding a plugin rule to customize the actions toolbar

Because it is embedded in the element `ActionsToolbar` in the above declaration, your custom plugin `AddActionsToolbarItemsPlugin` will be added to all instances of the `ActionsToolbar` class.

Your custom plugin is defined in a separate TypeScript file `AddActionsToolbarItemsPlugin.ts` that configures an `addItemsPlugin` to add a sep-

arator and a button with a custom action to the `ActionsToolbar` at index 5:

```
import Config from "@jangaroo/runtime/Config";
import AddItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/AddItemsPlugin";
import Separator from "@jangaroo/ext-ts/toolbar/Separator";
import Button from "@jangaroo/ext-ts/button/Button";
import MyAction from "./MyAction";

//...
Config(AddItemsPlugin, {
    index: 5,
    items: [
        Config(Separator),
        Config(Button, {
            baseAction: new MyAction({text: "hello"})
        })
    ]
})
```

Example 9.10. Adding a separator and a button with a custom action to a toolbar

While you can insert a component at a fixed position as shown above, it might also make sense to add the component after or before another component with a certain (global) ID, `itemId`, or `xtype`. To that end, the `AddItemsPlugin` allows you to specify pattern objects so that new items are added before or after the represented objects. If the component you want to use as an "anchor component" is not a direct child of the component you plug into, you can set the *recursive* attribute in your rules declaration to `true`.

Relative position of new component

When the component you want to modify is located inside a container that is also a public API extension point, you might have to access that container's API to provide context for your customizations. A typical use case for this is that you want to add a button to a toolbar that is nested below a container, but you need to apply your plugin rule to the container (and not the toolbar), because you need to access some API of that Container to configure the items to add (for example, access to the current selection managed by that container), or because the toolbar is reused by other containers, and you want your button to only appear in one specific context. Some *Studio* components define public API interfaces for accessing the runtime component instance, for example `CollectionView` creates a component that is documented to implement the public API interface `ICollectionView`.

Nested extension points

To express such nested extension point plugin rules, there is the plugin `NestedRulesPlugin`. Its usage is similar to *CoreMedia Studio* plugin rules, namely is must contain an element `rules` that again contains nested plugin rules. A nested plugin rule consists of the element of the subcomponent to locate with an optional `itemId`, which in turn contains a `plugins` element with the plugins to add to that component. Typical plugins to use here are `AddItemsPlugin`, `RemoveItemsPlugin`, and `ReplaceItemsPlugin`, all located in namespace `exml:com.coremedia.ui.config`.

For example, assume that to every LinkList property field, you want to add a custom toolbar action that needs access to the current selection of items in the LinkList given via *LinkListPropertyField#getSelectedValues-Expression()* of type `ValueExpression`. Like in the example above, you have to add a custom plugin to a *CoreMedia Studio* extension point in your *CoreMedia Studio* plugin TypeScript file:

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import LinkListPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/LinkListPropertyField";
import CustomizeLinkListPropertyFieldPlugin from
"./CustomizeLinkListPropertyFieldPlugin";

class MyPlugin extends StudioPlugin {

  constructor(config:Config<MyPlugin>){
    super(ConfigUtils.apply(Config(MyPlugin, {
      //...
      rules:
          [
            Config(LinkListPropertyField, {
              plugins: [
                Config(CustomizeLinkListPropertyFieldPlugin),
              ],
            }),
          ],
    }), config));
    //...
  }
}

export default MyPlugin;
```

Example 9.11. Adding a plugin rule to customize all LinkList property field toolbars

Now, in your plugin `CustomizeLinkListPropertyFieldPlugin.ts`, instead of using `AddItemsPlugin` directly, you apply `NestedRulesPlugin` to locate the toolbar you want to customize. Still, the component you plug into is a LinkList property field, and when your custom plugin is instantiated, that component is instantiated, too, and handed in as the config option `config.cmp`. It is good practice to assign the LinkList property field component as well as its initial configuration (when needed) to typed local TypeScript variables to avoid repeating longish expressions and type casts in inline code.

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import Component from "@jangaroo/ext-ts/Component";
import Separator from "@jangaroo/ext-ts/toolbar/Separator";
import AddItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/AddItemsPlugin";
import IconButton from
"@coremedia/studio-client.ext.ui-components/components/IconButton";
import NestedRulesPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/NestedRulesPlugin";
import LinkListPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/LinkListPropertyField";
```

```
import LinkListPropertyFieldToolbar from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/LinkListPropertyFieldToolbar";
import MyAction from "./MyAction";

class CustomizeLinkListPropertyFieldPlugin extends NestedRulesPlugin {

  static override readonly xtype: string =
"com.coremedia.blueprint.studio.template.config.CustomizeLinkListPropertyFieldPlugin";

  constructor(config: Config<NestedRulesPlugin>) {

    const linkListPropField = as(config.cmp, LinkListPropertyField);

    super(ConfigUtils.apply(Config(CustomizeLinkListPropertyFieldPlugin, {
      ...ConfigUtils.append({
        rules: [
          Config(LinkListPropertyFieldToolbar, {
            plugins: [
              Config(AddItemsPlugin, {
                items: [
                  Config(Separator),
                  Config(IconButton, {
                    baseAction: new MyAction({
                      contentValueExpression:
linkListPropField.getSelectedValuesExpression(),
                    }),
                    contentValueExpression:
linkListPropField.getSelectedValuesExpression(),
                    forceReadOnlyValueExpression:
linkListPropField.forceReadOnlyValueExpression,
                  }),
                ],
                before: Config(Component, {
                  itemId:
LinkListPropertyFieldToolbar.LINK_LIST_SEP_FIRST_ITEM_ID,
                }),
              }),
            ],
          })
        ],
      })
    }), config));
  }
}

export default CustomizeLinkListPropertyFieldPlugin;
```

Example 9.12. Using NestedRulesPlugin to customize a subcomponent using its container's API

Note how the above code makes use of the TypeScript element `LinkList PropertyFieldToolbar` to locate the toolbar inside the `LinkListProp ertyField`, as well as to use an `..._ITEM_ID` constant from that config class to specify the new items' location.

As another example, assume you want to create your own component inheriting from `LinkListPropertyField`. You want to reuse the default toolbar that the standard link list component defines, but you want to add one additional button to that toolbar. In a very similar fashion to the example above concerning *CoreMedia Studio* plugins, you can then write your custom component's TypeScript file like this:

Customizing nested components

```
// ----------------
// TODO: find another example: LinkListPropertyField already got a config
additionalToolbarItems, this clashes!
// ----------------
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import Component from "@jangaroo/ext-ts/Component";
import AddItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/AddItemsPlugin";
import NestedRulesPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/NestedRulesPlugin";
import LinkListPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/LinkListPropertyField";
import LinkListPropertyFieldToolbar from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/LinkListPropertyFieldToolbar";

interface UsingNestedRulesPluginConfig extends Config<LinkListPropertyField>
 {
  additionalToolbarItems?: Component;
}

class UsingNestedRulesPlugin extends LinkListPropertyField {

  static override readonly xtype: string =
"com.coremedia.blueprint.studio.template.config.UsingNestedRulesPlugin";
  declare Config: UsingNestedRulesPluginConfig;

  constructor(config: Config<UsingNestedRulesPlugin>) {
    super(ConfigUtils.apply(Config(UsingNestedRulesPlugin, {
      ...ConfigUtils.append({
        plugins: [
          Config(NestedRulesPlugin, {
            rules: [
              Config(LinkListPropertyFieldToolbar, {
                plugins: [
                Config(AddItemsPlugin, { items: config.additionalToolbarItems
 }),
                ],
              }),
            ],

          }),
        ],
      })
    }), config));
  }
}

export default UsingNestedRulesPlugin;
```

Example 9.13. Using NestedRulesPlugin to customize a subcomponent

Note that when you inherit from a component and use the `plugins` element
to declare the plugins you want to apply to this component, you overwrite the
`plugins` definition of the component you inherit from. That means that all the
plugins that the super component defines would not be used in your custom
component. To avoid that, you have to wrap your additional `plugins` definition
into a `...ConfigUtils.append()` or `...ConfigUtils.prepend()`
call. This will then add your custom plugin definitions to the end of the super
component's declarations, or insert them at the beginning, respectively.

You might also want to remove certain components from their containers. In
that case, you can add the `RemoveItemsPlugin` to the container component

Removing components

and remove items, again identifying them by pattern objects that can specify id, item id, or xtype.

In order the replace an existing component, you can use the `ReplaceItems Plugin`. For this plugin, you specify one or more replacement components in the `items` property. Each item must specify an id or an item id and replaces the existing component with exactly that id or item id.

Finally, a custom *CoreMedia Studio* plugin needs to be registered with the *Studio* application. This is done in the `jangaroo.config.js` file in the module root folder. The purpose of this file is to add the fully qualified main plugin class to the list of Studio plugins as shown in the following example:

Register the plugin

```
module.exports = jangarooConfig({
  type: "code",
  ...
  sencha: {
    studioPlugins: [
      {
        mainClass: "com.acme.AcmeStudioPlugin",
        name: "Ac me!",
      },
    ],
  },
  ...
});
```

Example 9.14. Registering a plugin

The object created in the `jangaroo.config.js` file may use the attributes defined by the class `EditorPluginDescriptor`, especially `name` and `mainClass` as shown above. In addition, the attributes *requiredGroup* and *requiredLicenseFeature* may be used.

Group-specific plugin

You can also implement group specific and own conditions using the `OnlyIf` plugin.

OnlyIf plugin

To recapitulate, this is a brief overview of the configuration chain:

1. NPM dependencies introduce Studio plugin modules to *CoreMedia Studio*.

2. Studio plugin modules register Studio plugins in the `jangaroo.config.js` file.

3. Studio plugin rules definitions denote components by ID or xtype and add Ext JS plugins to those components.

4. The Ext JS plugins shown here change the list of items of the components. Any other Ext JS plugins can be used in the same way.

Load external resources

If you want to load external style sheets or JavaScript files into *Studio*, you have to place them below the folder `src/main/sencha/resources` in your module and add the file paths to the `sencha` entry of your module's jangaroo.config.js file with the configuration options *css* and *js* as follows:

```
/** @type { import('@jangaroo/core').IJangarooConfig } */
module.exports = {
  type: "code",
  sencha: {
    css: [
      {
        path: "resources/path/to/myStylesheet1.css",
      },
      {
        path: "resources/path/to/myStylesheet2.css",
      },
    ],
    js: [
      {
        path: "resources/path/to/myJavascript1.js",
      },
      {
        path: "resources/path/to/myJavascript2.js",
      },
    ],
  },
};
```

Example 9.15. Loading external resources

9.4 Localizing Labels

Many labels besides content types and property names can also be localized. Typical cases are labels or button texts, error messages or window titles. The localized texts are stored in property files. To use these property values, classes are generated by the TypeScript compiler following the singleton pattern. Property classes can be adapted as described in Section 5.6, "Localization" [83], typically overriding the existing value with values from a new customizing property class.

CKEditor: Note that in contrast to this, the *Richtext Editing* component based on *CKEditor* requires a different approach for localization. For details have a look at Section 10.3.2, "Localizing CKEditor 5" [430].

Predefined property classes of CoreMedia Studio

The following classes are some of the predefined property classes defining labels and messages used throughout *CoreMedia Studio*.

- `Actions_properties`
- `DeviceTypes_properties`
- `Editor_properties`
- `EditorErrors_properties`
- `Publisher_properties`
- `Validators_properties`
- `ContentTypes_properties`
- `ContentActions_properties`

See the TypeScript documentation for a list of defined properties.

Predefined property files of Blueprint Studio

The module *@coremedia-blueprint/studio-client.main.blueprint-forms* contains several property files with localization entries. These files are used to localize several features of *Studio*, for example tab titles, content type names or validator messages.

You can simply change the value of any of the properties as needed. While you can also add new properties to these files when building extensions of *CoreMedia Studio*, it is preferable to put new localization keys into new property files.

Adding a new resource bundle

If you want to add a new property file which contains your own localization keys, proceed as follows:

1. Create a directory corresponding to the desired location of your resource bundle, for example `<ModuleName>/src/bundle/`.

2. Create new properties files following the naming schema: `<PropertyFile Name>_properties` and `<PropertyFileName>_de_properties`.

3. Add one or more keys and values in the form shown in the example below.

4. Optionally, add the same key to each locale-specific properties file, using an appropriate translation.

5. Import the resource bundle in other Typescript files like importing any other class.

6. Address the resource bundle and key in the text attribute of the component where you want to use the label: `BundleName_properties.KEY_NAME`. You will get code completion in a properly configured IDE for the keys of your resource bundle.

Example: Adding a Search Button

In order to introduce a new localized button to the favorites toolbar you could add the following component to the file `BlueprintFormsStudioPlugin.ts` for the component `FavoritesToolbar`.

```
import Config from "@jangaroo/runtime/Config";
import Component from "@jangaroo/ext-ts/Component";
import AddItemsPlugin from "@coremedia/studio-client.ext.ui-components/plugins/AddItemsPlugin";
import ShowCollectionViewAction from
"@coremedia/studio-client.main.editor-components/sdk/actions/ShowCollectionViewAction";
import BlueprintStudio_properties from
"@coremedia-blueprint/studio-client.main.blueprint-forms/BlueprintStudio_properties";
import EditorMainNavigationToolbar from
"@coremedia/studio-client.main.editor-components/sdk/desktop/maintoolbar/EditorMainNavigationToolbar.ts"

//...
Config(Component, {
  plugins: [
    Config(AddItemsPlugin, {
      items: [
        Config(EditorMainNavigationToolbar, {
          baseAction: Config(ShowCollectionViewAction, {
            published: false,
            editedByMe: true,
            contentType: "CMArticle",
            text: BlueprintStudio_properties.doc_example_txt,
          }),
        }),
      ],
      after: [
        Config(Component, { itemId: EditorMainNavigationToolbar.NEW_MENU_BUTTON_ITEM_ID }),
      ],
```

```
    }),
  ]
});
```

Example 9.16. Adding a search button

The attribute *text* of the ShowCollectionView Element defines the text to be displayed in the *Studio* web application.

In order to have the label you want, you need to add it to the properties file. The BlueprintStudio_properties file starts like this after adding a string for the label:

```
interface BlueprintStudio_properties {
  //...
  doc_example_txt: string;
  //...
}
const BlueprintStudio_properties: BlueprintStudio_properties = {
  //...
  doc_example_txt: "My Example Button",
  //...
};
```

Example 9.17. Example property file

Override Standard Studio Labels

It is also possible to override the standard *Studio* labels, like so:

1. Create a property file with all labels you want to override, for example Cus
   tomLabels_properties and CustomLabels_de_properties.

2. Search for the key of the property that should be changed. All the keys are
   documented in the TypeScript API, such as Action_withdraw_tooltip
   in the resource bundle class Actions_properties.

3. In your CustomLabels bundle, set the new value for the key.

4. In the configuration section of your *Studio* plugin, override the Ac
   tions_properties bundle with the following code:

```
import resourceManager from "@jangaroo/runtime/l10n/resourceManager";
import CopyResourceBundleProperties from
"@coremedia/studio-client.main.editor-components/configuration/CopyResourceBundleProperties";
import Actions_properties from
"@coremedia/studio-client.main.editor-components/sdk/Actions_properties";
import CustomLabels_properties from
"@coremedia-blueprint/studio-client.main.ec-studio/CustomLabels_properties";

//...

//override the standard studio labels with custom properties
new CopyResourceBundleProperties({
```

```
    destination: resourceManager.getResourceBundle(null, Actions_properties),
    source: resourceManager.getResourceBundle(null, CustomLabels_properties),
})
```

Example 9.18. Overriding properties

This can be done with every property of *Studio*. Examples for this can also be found in the BlueprintFormsStudioPlugin.

9.5 Document Type Model

Each CoreMedia CMS content application is based on an object-oriented content type model. Documents of different types often require different treatment. By tailoring CoreMedia Studio to the content type model, the support for dealing with content items is greatly improved.

- Section 9.5.1, "Localizing Types and Fields" [145] describes how to localize the names of content types and content properties.
- Section 9.5.2, "Customizing Content Forms" [148] describes how you can add or remove property fields to or from a content form.
- Section 9.5.3, "Image Cropping and Image Transformation" [154] describes how to enable the image cropping feature.
- Section 9.5.5, "Disabling Preview for Specific Content Types" [159] describes how you can disable the preview for a specific content type.
- Section 9.5.6, "Excluding Content Types from the Library" [160] describes how you can exclude content types from the dropdown lists for content item creation and content type search filtering.
- Section 9.5.7, "Client-side initialization of new content items" [161] describes how you can initialize newly created content items.

9.5.1 Localizing Types and Fields

You can localize the display of content types and their properties in terms of *type name, description and icon* and in terms of *property names and descriptions*. To this end, the global registry `contentTypeLocalizationRegistry` is used. The registration code can be placed in an auto-loaded script as described in Section 9.1, "General Remarks On Customizing (Multiple) Studio Apps" [120]. The following figure shows the example of localizing an article and a media content type.

```
import { contentTypeLocalizationRegistry } from "@coremedia/studio-client.cap-base-models";
import BlueprintDoctypesDocTypes_properties from "./BlueprintDoctypesDocTypes_properties";
import typeArticle from "./icons/type-article.svg";
import typeMedia from "./icons/type-media.svg";

contentTypeLocalizationRegistry.addLocalization("CMArticle", {
  displayName: BlueprintDoctypesDocTypes_properties.CMArticle_displayName,
  description: BlueprintDoctypesDocTypes_properties.CMArticle_description,
  svgIcon: typeArticle,
  properties: {
    title: {
      displayName: BlueprintDoctypesDocTypes_properties.CMArticle_title_displayName,
      description: BlueprintDoctypesDocTypes_properties.CMArticle_title_description,
      emptyText: BlueprintDoctypesDocTypes_properties.CMArticle_title_emptyText,
    },
    detailText: {
      displayName: BlueprintDoctypesDocTypes_properties.CMArticle_detailText_displayName,
      description: BlueprintDoctypesDocTypes_properties.CMArticle_detailText_description,
      emptyText: BlueprintDoctypesDocTypes_properties.CMArticle_detailText_emptyText,
    },
  },
```

```
});
contentTypeLocalizationRegistry.addLocalization("CMMedia", {
  displayName: BlueprintDoctypesDocTypes_properties.CMMedia_displayName,
  description: BlueprintDoctypesDocTypes_properties.CMMedia_description,
  svgIcon: typeMedia,
  properties: {
    localSettings: {
      properties: {
        playerSettings: {
          properties: {
            muted: { displayName:
              BlueprintDoctypesDocTypes_properties
                .CMMedia_localSettings_playerSettings_muted_displayName },
            loop: { displayName:
              BlueprintDoctypesDocTypes_properties
                .CMMedia_localSettings_playerSettings_loop_displayName },
            autoplay: { displayName:
              BlueprintDoctypesDocTypes_properties
                .CMMedia_localSettings_playerSettings_autoplay_displayName },
            hideControls: { displayName:
              BlueprintDoctypesDocTypes_properties
                .CMMedia_localSettings_playerSettings_hideControls_displayName },
          },
        },
      },
    },
    alt: {
      displayName: BlueprintDoctypesDocTypes_properties.CMMedia_alt_displayName,
      description: BlueprintDoctypesDocTypes_properties.CMMedia_alt_description,
      emptyText: BlueprintDoctypesDocTypes_properties.CMMedia_alt_emptyText,
    },
    caption: {
      displayName: BlueprintDoctypesDocTypes_properties.CMMedia_caption_displayName,
      description: BlueprintDoctypesDocTypes_properties.CMMedia_caption_description,
    },
    copyright: {
      displayName: BlueprintDoctypesDocTypes_properties.CMMedia_copyright_displayName,
      description: BlueprintDoctypesDocTypes_properties.CMMedia_copyright_description,
      emptyText: BlueprintDoctypesDocTypes_properties.CMMedia_copyright_emptyText,
    },
  },
});
```

Example 9.19. Localizing content types

For the content type itself and all of its properties, *displayName* and *description* can be set. The description is mostly used for tooltips. For properties, an additional *emptyText* can be specified. As can be seen for the media type localization, it is also possible to localize properties nested in structs and sub-structs. Just as described for labels in Section 9.4, "Localizing Labels" [141], resource bundles are generally used to localize the content type texts. This allows to comfortably account for multiple locales.

It is possible to localize the same property differently for a content type and its sub-types. If the localization for a concrete type instance is accessed, the localization of the most specific fitting type or super-type is used.

The icon for a content type is given as an SVG icon. For this to work, the module where the localization takes place needs to have a `custom.d.ts` file in its root folder with the following content:

```
declare module "*.svg" {
  const content: string;
  export default content;
}
```

Example 9.20. Allows the import of SVG icons in a typescript file

CoreMedia Studio works with icon sizes in 16px, 24px or 32px. Despite the scalability of SVG icons, it might happen that an icon looks blurry in some sizes. In addition, it might be that the 16px icon looks slightly different than the 32px version. To fully optimize your icons for the different icon sizes, you can create an SVG that embeds the SVG code for all three sizes as shown for the article type icon in the following example.

```
<?xml version="1.0" encoding="utf-8"?>
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
    <style type="text/css">
        @media screen {
          #small {
            display: initial;
          }
          #medium, #large {
            display: none;
          }
        }
        @media screen and (min-width: 24px) {
          #small {
            display: none;
          }
          #medium {
            display: initial;
          }
        }
        @media screen and (min-width: 32px) {
          #medium {
            display: none;
          }
          #large {
            display: initial;
          }
        }
    </style>
    <svg id="small" version="1.1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
        viewBox="0 0 16 16" enable-background="new 0 0 16 16">
        <g>
            <rect x="3" y="1" fill="#3D4242" width="10" height="1"/>
            <path fill="#3D4242" d="M3,3v12h10V3H3z M11,9H5V5h6V9z"/>
        </g>
    </svg>

    <svg id="medium" version="1.1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
        viewBox="0 0 24 24" enable-background="new 0 0 24 24">
        <g>
            <rect x="4" y="1" fill="#3D4242" width="16" height="2"/>
            <path fill="#3D4242" d="M4,4v19h16V4H4z M17,14H7V7h10V14z"/>
        </g>
    </svg>

    <svg id="large" version="1.1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
        viewBox="0 0 32 32" enable-background="new 0 0 32 32">
        <g>
            <rect x="5" y="1" fill="#3D4242" width="22" height="2"/>
            <path fill="#3D4242" d="M5,5v26h22V5H5z M23,19H9V9h14V19z"/>
        </g>
    </svg>

</svg>
```

Example 9.21. Content type icon optimized for the sizes 16px, 24px and 32px

The same `contentTypeLocalizationRegistry` registry that is used to add new content type localizations can also be used to override existing ones. The method `contentTypeLocalizationRegistry.addLocalization(contentType: string, localization: ContentTypeLocalizationConfig)` checks whether an existing localization already exists for the given `contentType`. If this is the case, a deep merge of the existing localization and the passed `localization` is carried out, giving the latter precedence in case of a conflict.

9.5.2 Customizing Content Forms

The following section describes how to customize the content forms, which constitute the main working component that your users will use. *Studio* allows you to organize a - potentially quite big - set of property fields into horizontal tabs.

To register your custom content form, you need to register your TypeScript component to the `TabbedDocumentFormDispatcher` inside the initialization of a *Studio* plugin, like so:

Multi-tab content forms

```
import Config from "@jangaroo/runtime/Config";
import TabbedDocumentFormDispatcher from
"@coremedia/studio-client.main.editor-components/sdk/premular/TabbedDocumentFormDispatcher";
import AddTabbedDocumentFormsPlugin from
"@coremedia/studio-client.main.editor-components/sdk/plugins/AddTabbedDocumentFormsPlugin";
import MyCMArticleForm from "./MyCMArticleForm";

//...
Config(TabbedDocumentFormDispatcher, {
  plugins: [
    Config(AddTabbedDocumentFormsPlugin, {
      documentTabPanels: [
        Config(MyCMArticleForm, { itemId: "CMArticle" }),
        //...
      ],
    }),
  ],
})
```

The above code plugs into the `TabbedDocumentFormDispatcher`, and registers custom content forms with the plugin namespace `bpforms`. Note that the `itemId` still corresponds to the name of the content type you want to apply your form for.

The content forms registered with the dispatcher are automatically used for both the regular content form and for the left-side form of the version comparison view and the master side-by-side view. When used on the left side, the `forceReadOnlyValueExpression` passed to the form is set to `true`, allowing your form to switch into a read-only mode.

To customize a form, you need to adapt the respective form definition file (a TypeScript component) in `@coremedia-blueprint/studio-cli`

ent.main.blueprint-forms (under src/forms). The following code shows a simple example for a standard CMArticle form definition:

```
import DocumentForm from
"@coremedia/studio-client.main.editor-components/sdk/premular/DocumentForm";
import DocumentTabPanel from
"@coremedia/studio-client.main.editor-components/sdk/premular/DocumentTabPanel";
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import BlueprintTabs_properties from "../BlueprintTabs_properties";
import CMArticleSystemForm from "./components/CMArticleSystemForm";
import DefaultExtraDataForm from "./components/DefaultExtraDataForm";
import MultiLanguageDocumentForm from "./containers/MultiLanguageDocumentForm";
import PropertyFieldGroup from
"@coremedia/studio-client.main.editor-components/sdk/premular/PropertyFieldGroup";
import StringPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/StringPropertyField";
import RichTextPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/richtext/RichTextPropertyField";

class CMArticleForm extends DocumentTabPanel {
  static override readonly xtype: string =
"com.coremedia.blueprint.studio.config.cmArticleForm";

  constructor(config: Config<CMArticleForm>) {
    super(ConfigUtils.apply(Config(CMArticleForm, {
      items: [
        Config(DocumentForm, {
          title: BlueprintTabs_properties.Tab_content_title,
          items: [
            Config(PropertyFieldGroup, {
              title: "My Field Group",
              itemId: "myFieldGroup",
              items: [
                Config(StringPropertyField, {
                  propertyName: "title",
                }),
                Config(RichTextPropertyField, {
                  propertyName: "detailText",
                  initialHeight: "200",
                }),
              ],
            }),
          ],
        }),
        Config(DefaultExtraDataForm),
        Config(MultiLanguageDocumentForm),
        Config(CMArticleSystemForm),
      ],
    }), config));
  }
}

export default CMArticleForm;
```

Example 9.22. Article form

Collapsible Property Field Groups

To add several property fields to a group with an additional title, the component PropertyFieldGroup can be used. All content forms of *CoreMedia Blueprint* use it to provide a better overview about related fields.
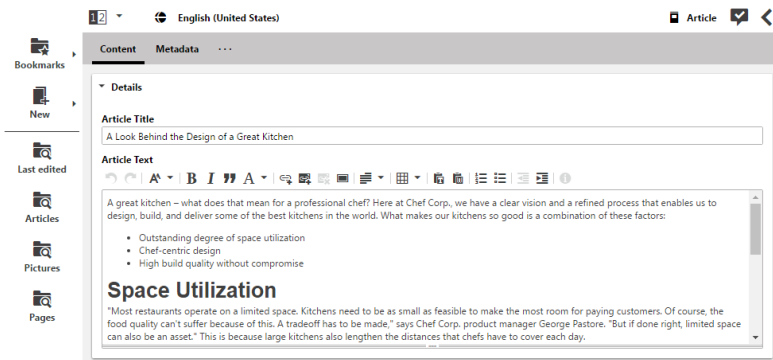
Figure 9.3. Document form with a collapsible property field group

Additionally, the collapsible property field group persists the collapsed status. For example, when the group is collapsed for the teaser title and teaser text of an article, the group is collapsed for all newly opened article content items too (except it contains an invalid field). This status information is stored in the user preferences of the user, so if the user logs into *Studio* on another computer, the same state will be restored.

```
import Config from "@jangaroo/runtime/Config";
import PropertyFieldGroup from
"@coremedia/studio-client.main.editor-components/sdk/premular/PropertyFieldGroup";
import CustomLabels_properties from
"@coremedia-blueprint/studio-client.main.blueprint-forms/CustomLabels_properties";
import StringPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/StringPropertyField";
import RichTextPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/richtext/RichTextPropertyField";

Config(PropertyFieldGroup, {
  title: CustomLabels_properties.PropertyGroup_Details_label,
  itemId: "detailsDocumentForm",
  items: [
    Config(StringPropertyField, {
      propertyName: "title",
    }),
    Config(RichTextPropertyField, {
      propertyName: "detailText",
      initialHeight: "200",
    }),
  ],
})
```

Example 9.23. Collapsible Property Field Group

Each declaration of a `PropertyFieldGroup` element should contain the attributes `title` and `itemId`. The `title` attribute applies a title to the panel (and also provides a meaning to the group). It is also used as click area for collapsing the panel. The `itemId` should be applied to persist the state of the group. If no `itemId` is provided, the collapsible state is not stored in the user

preferences and therefore not applied when new content items of the same type are opened.

Property Fields

CoreMedia Studio offers at least one predefined property field for each property type available for CoreMedia content items. See Table 9.1, "Property Fields" [151] for a list of all provided field types.

Each property field of this table has at least an attribute `propertyName` which corresponds to the property name of the content type. The property name must be specified for each field. The content form also provides three additional properties to all fields without specifying them explicitly: `bindTo`, `hideIssues`, and `forceReadOnlyValueExpression`. The standard property fields recognize these options and custom property fields are encouraged to so, too. See Section 9.6, "Customizing Property Fields" [162] for details about developing new property fields.

- `bindTo`: A value expression that evaluates to the content object to show in the form. The content may change when the form content changes.

- `hideIssues`: This attribute is used to disable the highlighting of property fields with issues originating from validators. Validators will be described in Section 9.23.1, "Validators" [252]. If set on the content form, it applies to all property fields.

- `forceReadOnlyValueExpression`: A value expression that evaluates to true when the content form and all of its property fields should be shown in read-only mode, for example when showing the content form on the left side in master comparison mode.

Other attributes might vary depending on the property type. The `BlobProp ertyField` editor, for example, has a property `contentType` that defines the MIME type. If you want to hide a property, you can simply remove the related `<PropertyType>PropertyField` element. The order of the editor elements defines the order in the form.

Property Field	Used for	Description
StringProperty Field	String property	Shows string data.
IntegerProperty Field	Integer property	Shows integer number.

Property Field	Used for	Description
SpinnerProperty Field	Integer property	Shows integer number, with arrow buttons to increase/decrease the current value, and mouse wheel.
BooleanProperty Field	Integer property with 0/1 boolean values	Shows a checkbox indicating checked=1, unchecked=0.
DateTimeProper tyField	Date property	Shows date, time and time zone and provides appropriate picker elements.
LinkListProper tyField	Link List property	Allows drag and drop.
XmlProperty Field	Generic XML property	Shows the raw XML text.
BlobProperty Field	Blob property for all MIME types	Shows the image and provides an upload dialog.
TextAreaString PropertyField	String property	Shows the text represented in the content repository as a StringProperty in a text area.
TextAreaProper tyField	CoreMedia RichText (XML) property	Shows the text represented in the content repository as a XmlProperty as plain text in a text area.
RichTextProper tyField	CoreMedia RichText (XML) property	Shows the text represented in the content repository as a XmlProperty in a WYSIWYG style and provides a fully featured toolbar.
TextBlobProper tyField	Blob property of MIME type text/plain	Shows the blob as plain text in a text area.
StructProperty Field	CoreMedia Struct property	Shows a generic editor for structs.

Table 9.1. Property Fields

Customizing Columns in Link List Properties

By default, the `LinkListPropertyField` shows a content type icon, the name and the lifecycle status for each linked content item. Additionally, the boolean property `showThumbnails` can be set to `true` to enable a thumbnail preview for the referenced content item. Also, you can configure an array of columns to be shown using the `columns` property of the field component. Each array element must be an Ext JS grid column object. The available fields of the store backing the grid panel are `name`, `status`, `type`, and `typeCls`. These fields represent the name, the lifecycle status, the content type name and a style class for a content type icon, respectively.

Showing more columns

If you need additional fields for your custom columns, you can add them using the `fields` property. Each field should be a `@coremedia/studio-client.ext.ui-components/store/DataField`. The following example shows how a new column uses a custom field to display the `locale` property of linked content items.

```
import Config from "@jangaroo/runtime/Config";
import Column from "@jangaroo/ext-ts/grid/column/Column";
import LinkListPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/LinkListPropertyField";
import DataField from
"@coremedia/studio-client.ext.ui-components/store/DataField";
import NameColumn from
"@coremedia/studio-client.ext.cap-base-components/columns/NameColumn";
import StatusColumn from
"@coremedia/studio-client.ext.cap-base-components/columns/StatusColumn";
import TypeIconColumn from
"@coremedia/studio-client.ext.cap-base-components/columns/TypeIconColumn";

//...
Config(LinkListPropertyField, {
    fields: [
        Config(DataField, {
            name: "locale",
            mapping: "properties.locale",
            ifUnreadable: null,
        }),
    ],
    columns: [
        Config(TypeIconColumn),
        Config(NameColumn),
        Config(StatusColumn),
        Config(Column, {
            header: "Locale",
            width: 270,
            dataIndex: "locale",
        }),
    ],
})
```

Whereas the configured fields are added to the default fields, the configured columns completely replace the default columns. That is, if you want to keep the predefined fields, you have to repeat their definitions as shown in the example.

Customizing Suggestions and Search Strategy in Link List Properties

The `LinkListPropertyField's` drop area displays suggestions and search results for new list entries. Suggestions and search results can be adjusted in the `LinkListPropertyField` by configuring a custom `linkSuggester`, a corresponding `linkSuggesterTemplate` and `linkSuggesterTemplateExtraFields`. The following code example shows how to customize the field:

```
import Config from "@jangaroo/runtime/Config";
import LinkListPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/LinkListPropertyField";
import DataField from
"@coremedia/studio-client.ext.ui-components/store/DataField";
import CustomUtil from "./CustomUtil";
import MyCustomLinkSuggester from "./MyCustomLinkSuggester";

//...
Config(LinkListPropertyField, {
    linkType: "CMTeasable",
    linkListSuggesterTemplate: CustomUtil.getMyTpl(),
    linkSuggester: Config(MyCustomLinkSuggester),
    linkSuggesterTemplateExtraFields: [
        Config(DataField, {
            name: "customField",
            mapping: "",
            convert: CustomUtil.getCustomField,
            encode: false,
        }),
    ],
})
```

9.5.3 Image Cropping and Image Transformation

The Image Editor provides various image transformations which are stored in a separate struct property of the content item. It also holds the original image data which is never modified – the transformations are applied only when previewing or delivering the image.

The Image Editor uses the same Image Transformation Framework to display the image within the image form as the CAE uses for delivering images to web sites, for example, within the preview panel. See the Content Application Developer Manual for further details on image transformations.

The `ImageEditorPropertyField` is defined in the `CMPictureForm.ts` of the *Blueprint* and can be defined by using the config properties listed below. Properties marked with * are mandatory.

Config Property	Type	Description
bindTo*	ValueEx pression	A property path expression leading to the content Bean whose properties are edited.
propertyName*	String	The name of the BLOB property containing image data.
imageSettings PropertyName*	String	The name of the Struct property containing image transformation data.
hideIssues	boolean	If true, no validation issues on this property field are shown. Defaults to false.
forceReadOnly ValueExpression	String	An optional ValueExpression which makes the component read-only if it is evaluated to true.

Table 9.2. `ImageEditorPropertyField` Configuration Settings

The `ImageEditorPropertyField` can be configured as follows:

```
Config(ImageEditorPropertyField, {
  bindTo: config.bindTo,
  propertyName: "data",
  imageSettingsPropertyName: "localSettings",
})
```

Example 9.24. Configuring the Image Editor

A crop is a subset of the image with a fixed aspect ratio and minimum size. Crops in the Image Editor are represented by variants. There are two different ways to configure variants: via Spring or as site specific variants directly in the content.

Spring Configuration for Variants

To configure global variants for all `CMPicture` content items, beans of type `com.coremedia.cap.transform.Transformation` can be added to the Spring application context which are automatically picked up by the `transformImageService` bean. Each variant is defined by one `Transform ation` which holds all the information for that variant.

```
<bean class="com.coremedia.cap.transform.Transformation">
  <property name="name" value="large4x3"/>
  <property name="widthRatio" value="4"/>
  <property name="heightRatio" value="3"/>
  <property name="minWidth" value="640"/>
  <property name="minHeight" value="480"/>
  <property name="previewWidth" value="400" />
</bean>
```

Example 9.25. Configuring an image variant

The configuration of variants via Spring is the default used by the `Transfor mImageService`.

Theme Specific Image Variants

In the Blueprint, the image variants are by default defined in the themes of the frontend workspace. The CSS and the templates create the HTML elements with their widths and heights on the website. As such, they are the first choice for placing the image variant settings. Have a look at the Example 5.8, "Responsive Images.settings.json" in *Frontend Developer Manual* and at the `Responsive Images.settings.json` files in themes of the Blueprint workspace.

Site-specific Image Variants

If a site does not have a theme and if not all sites should have the same fixed set of image variants, you can configure site-specific image variants via content instead. Define a `CMSettings` content item named `responsiveImageSet tings` with the struct property `linkedSettings` for every site (see also section "Content Configuration" [157] below).

The feature for site-specific variants is enabled by default. To disable it, set the property `imagetransformation.dynamic-variants` to `false`.

In addition to the site-specific variants, the default variants configured in the Spring configuration (see section " Spring Configuration for Variants " [155]) will always be applied.

Rendering Site Specific Image Variants

When rendering images, the `TransformImageService` is used to access the variants of an image. You can find an example for this in the `CMPicture.as Preview.ftl` file. In this template, the `previewWidth` and `previe wHeight` attributes of the `Transformation` class are used to calculate the image size in the preview. If these attributes are not set, `minWidth` and `min Height` are used instead.

CAE Configuration

For the CAE, the class `TransformImageService` is responsible for loading site specific cropping information. Disable the feature by configuring `imagetransformation.dynamic-variants=false`.

The `TransformImageService` will automatically look up the linked settings of the root channel and search for the "Responsive Image Settings" struct which contains the variant information.

Content Configuration

The "Responsive Image Settings" content item not only contains image variants, but also various resolutions which may be used on different devices. The breakpoint values defined in the CSS for the corresponding theme are used to determine which resolution should be used. With the introduction of site specific image crops, you can configure additional struct properties for variants.

Variant Properties, the following are mandatory:

- `widthRatio`: minimum integer which defines the width of the aspect ratio
- `heightRatio`: minimum integer which defines the height of the aspect ratio
- `minWidth`: this value is the minimum variant width the studio demands while uploading an image (integer property)
- `minHeight`: this value is the minimum variant height the studio demands while uploading an image (integer property)

Predefined image sizes (resolutions), at least one pair should be defined per variant and must match the aspect ratio:

- `width`: defines the width of the image (integer property)
- `height`: defines the height of the image (integer property)

So `minWidth` and `minHeight` should at least be as high as the largest pre-defined image size.

Properties for variant and predefined image sizes (properties listed within the predefined image size properties will always override the more general variant properties):

- `jpegQuality`: the default JPEG quality of the picture (string value with numeric value from 0 (excluded) to 1).
- `webpQuality`: the default WebP quality of the picture (string value with numeric value from 0 (excluded) to 1). This value is ignored in the built-in image transformation and may only be used with the CoreMedia Image Transformation Service.

- `avifQuality`: the default AVIF quality of the picture (string value with numeric value from 0 (excluded) to 1). This value is ignored in the built-in image transformation and may only be used with the CoreMedia Image Transformation Service.

- `sharpen`: boolean value to enable/disable sharpening of the picture.

- `removeMetadata`:boolean value to enabled/disable metadata removal of the transformed image.

MIME Type Mapping

Another optional setting in the content is a struct property called `linkMime TypeMapping` which can be used to adjust the MIME type / file extension of links that are created for image variants. For details see Section 5.4.14, "Images" in *Blueprint Developer Manual* .

9.5.4 Enabling Image Map Editing

The image map editor comes as a panel component embedding an image view. The editor allows users to create hot zones (image map areas) and to attach content items to hot zones via drag and drop. The image map editor uses a configurable struct property name to store the image map configurations to a struct property of an image map content item. It also offers a configuration option for the image to display. This allows you to store image map configurations in content item that do not have an image blob property themselves.

To enable image map editing in your project, include an image map editor component in your content item's TypeScript form (*Blueprint* shows this in its `CMImageMapForm.ts` definition).

```
Config(ImageMapEditor, {
        imageBlobValueExpression:
config.bindTo.extendBy("properties.pictures.0.properties.data"),
        structPropertyName: "localSettings",
        })
```

Example 9.26. Configuring an Image Map Editor

In the example above, the source content item has a link list property name `pictures` of cardinality `1`. So the image editor component is bound to the image stored at the `data` property of the linked image content item. The map configuration is stored at the source content item's `localSettings` property.

Enabling validation

Configure the `ImageMapAreasValidator` in the Studio server's Spring application context to enable validation of the image map content item. The validator generates an error issue if there is no image blob or if at least one of the defined image map areas does not have a valid link target. See also Section 9.23.1, "Validators" [252] for validation in general.

```
  @Bean
  @ConditionalOnProperty(name =
"validator.enabled.image-map-areas-validator.cm-image-map", matchIfMissing
= true)
  ImageMapAreasValidator cmImageMapAreasValidator(CapConnection connection)
  {
    return new ImageMapAreasValidator(type(connection, "CMImageMap"), true,
  "localSettings", "pictures.data");
  }
```

Example 9.27. Configuring a validator for image maps

In the example above, the validator is configured for the content type `CMImageMap` and its subtypes. The image is stored in the blob property `data` of the first content item of link list property `pictures` of the image map content item. The image map configuration is stored in the struct property `localSettings`.

9.5.5 Disabling Preview for Specific Content Types

For some content types a suitable preview representation is not easily generated. This applies to some built-in content types like `Dictionary` and `Editor Preferences`, but also to very technical content types storing CSS or script code.

The method `getDocumentTypesWithoutPreview()` from the global `@coremedia/studio-client.main.editor-components/sdk/editorContext` grants access to an array of content type names for which no preview should be shown. Like in the case of content types excluded from creation as shown in the previous section, you can simply push additional content types into the mutable array returned from the method.

You can also use the `ConfigureDocumentTypes` plugin to specify content types without preview, like in the following excerpt from `BlueprintFormsStudioPlugin.ts`.

```
import ConfigureDocumentTypes from
"@coremedia/studio-client.main.editor-components/configuration/ConfigureDocumentTypes";

//...
new ConfigureDocumentTypes({
    names: "CMAction,CMCSS,...",
    preview: false,
})
```

Example 9.28. Defining content types without preview

9.5.6 Excluding Content Types from the Library

The CoreMedia content type model is a very powerful concept to tailor *Core-Media CMS* to your needs. However, in any typical project, there are at least a couple of content types mainly designed to manage technical metadata, such as site settings. In many cases you want to hide these content types from casual users of *CoreMedia Studio*, thereby keeping the interface simple and avoiding clutter. To do so, you can remove choices from the dropdown content type selector in the Library's create content menu, and from the dropdown used to restrict search results to certain content types.

You can add the content types that should not be shown to the list of excluded content types using the `@coremedia/studio-client.main.editor-components/sdk/editorContext`. The methods `getExcludedDocumentTypes()` and `getContentTypesExcludedFromSearch()` return an array holding the names of all content types excluded from the create content dropdown and search filter dropdown, respectively. Using the array's `push` method, you can add additional content types you wish to hide: `editorContext._.getExcludedDocumentTypes().push('<DocType1>', ...)`

```
editorContext._.getExcludedDocumentTypes().push('Dictionary',
  'Preferences', 'Query',
  'CMDynamicList', 'CMVisual',
  'EditorPreferences');
```

Example 9.29. Defining excluded content types

This call gets the array of excluded content types and adds Strings containing the names of the content types to exclude.

You can also use the `ConfigureDocumentTypes` plugin from the previous section to achieve the same in a more declarative manner.

```
import ConfigureDocumentTypes from
"@coremedia/studio-client.main.editor-components/configuration/ConfigureDocumentTypes";

//...
new ConfigureDocumentTypes({
        names:
"Dictionary,Preferences,Query,CMDynamicList,CMVisual,EditorPreferences",
        exclude: true,
        excludeFromSearch: true,
})
```

Example 9.30. Defining excluded content types in TypeScript

9.5.7 Client-side initialization of new content items

With a content initializer you can initialize the properties of a newly created content item. A content initializer will be called while a new content object is being created by the `NewContentAction`. Only one initializer can be defined for each content type. You must register custom initializers with the global `@coremedia/studio-client.main.editor-components/sdk/editorContext`. Simply call the `registerContentInitializer(contentTypeName, initializer)` method.

The following code defines a simple initializer that sets the content's language property to German by default:

```
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import IEditorContext from
"@coremedia/studio-client.main.editor-components/sdk/IEditorContext";
import ContentInitializer from
"@coremedia-blueprint/studio-client.main.blueprint-forms/util/ContentInitializer";
import Content from "@coremedia/studio-client.cap-rest-client/content/Content";

class MyStudioPlugin extends StudioPlugin{

  //...

  init(editorContext: IEditorContext): void {
    editorContext.registerContentInitializer("CMTeaser",
MyStudioPlugin.initLanguage);
  }

  static initLanguage(content:Content):void {
    ContentInitializer.setProperty(content, "locale", "de");
  }
}
```

Example 9.31. Defining a content initializer

Client-side initialization might be sufficient for simple initialization scenarios. If you have complex requirements, consider using server-side initialization: Refer to Section 9.23.2, "Intercepting Write Requests" [266] for details.

9.6 Customizing Property Fields

While *CoreMedia Studio* provides predefined property fields for strings, dates, link lists (including those handling images), and many others, you might want to use an own widget to display and edit a property according to your specific re-quirements.

Ext JS offers many components that can be used for this purpose. Often, some configuration will get you a long way to an appropriate widget. The main task that is always necessary is the binding of the new component to your data ("the model"). *Studio*'s client-side models are explained in more detail in Section 5.3, "Client-side Model" [57] and Section 5.4, "Remote CoreMedia Objects" [74]. While you could theoretically implement property fields in any way, adhering to certain conventions as described in the following section helps to make the property fields reusable.

Also, there are a number of standard plugins that simplify the task of writing a property field. These are introduced by way of an example in Section 9.6.2, "Standard Component StringPropertyField" [163]. Here you will find a simple recipe for creating property fields that use a predefined plugin to handle the data binding.

For editing of richtext properties an extra section exists, not only telling about the underlying architecture but also about possible customization options. Find more details in Chapter 10, *Rich Text Editing* [416].

9.6.1 Conventions for Property Fields

Property field are intended for use in content forms as described in Section 9.5.2, "Customizing Content Forms" [148]. To ensure the most convenient usage, custom property fields should adhere to the standard name for config options.

The option `propertyName` should define the name of the property to show and edit in the property field. While you can use a different name for this option, your content form definition become more readable when you use the `propertyName` option uniformly.

Further conventions arise, because a content form forwards a number of config-uration option to all included components, that is, to all included property fields. By using the standard option names, you avoid repetitions and accidental omissions.

The option `bindTo` is a value expression that evaluates to the object that defines the property. If possible, the field should not assume that this object

implements the `Content` interface, but rather that it is a bean with a property `properties` that stores another bean that contains the property given as `propertyName`. That will eventually make it possible to reuse the field for workflow forms.

For the same reason, a property field should not access built-in properties like `creationDate` and others. It should also refrain from performing other operations like `checkIn` on the returned bean. This is no significant limitation, because property fields are typically reading and writing schema-defined properties, only. When property fields are used in the left half of the version comparison view, they are bound to an object that does implement the `Content` interface, but that is actually wrapping a version. In this case, the built-in properties of `Content` are present, but might not always return the value you expect. It always claims to be checked in and it returns the properties of the historic version, even though it reports the id of the versioned content. When accessing only the schema-defined properties, property field will behave as expected.

If the value expression provided through the option `forceReadOnlyValue Expression` evaluates to true, the property field should switch to a read-only mode. In this mode it should be possible to view property values and preferably to copy them, but it should be impossible to make updates. The value expression is set to true when a content form is used on the left side of a master side-by-side view or a version comparison view. The property field itself must take other reasons into account that might make the field read-only. To this end, the utility methods `isReadOnly` and `createReadOnlyValueExpression` in the class `PropertyEditorUtil` support you in making a property field read-only.

The class `PropertyEditorUtil` also contains methods for localizing property names, types, and so on.

9.6.2 Standard Component StringPropertyField

The task attempted in this section is to replicate the behavior of the standard `StringPropertyField`.

Create the new property field as a TypeScript component. You inherit directly from the Ext JS component `TextField` that is used for displaying the property. Before you can start, you must set the stage for the TypeScript file.

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";

import FieldContainer from "@jangaroo/ext-ts/form/FieldContainer";
```

```
class ExampleFieldContainer extends FieldContainer {

  constructor(config: Config<ExampleFieldContainer>) {
    super(ConfigUtils.apply(Config(ExampleFieldContainer, {
      // add default Config property values here
    }), config));
  }
}

export default ExampleFieldContainer;
```

Example 9.32. Custom property field

You are now ready to configure a property of your base class, for example the label alignment.

```
import Config from "@jangaroo/runtime/Config";
import FieldContainer from "@jangaroo/ext-ts/form/FieldContainer";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";

class ExampleFieldContainer extends FieldContainer {

  constructor(config: Config<ExampleFieldContainer>) {
    super(ConfigUtils.apply(Config(ExampleFieldContainer, {
      labelAlign: "top",
    }), config));
  }
}

export default ExampleFieldContainer;
```

The additional Config options supported by your `CustomPropertyField` are now declared. You can think of the set of these fields as the configuration API description of your component. Any component inherits the Config options from its superclass(es).

The following things are required to declare config options for your custom field:

1. You declare public variables (without `#` modifier) in your field class.

2. You define an interface `<YOUR_COMPONENT_CLASS>Config extends Config<<SUPERCLASS>>` that expose these variables as config options.

3. You declare the config interface in your class: `declare Config: <YOUR_COMPONENT_CLASS>Config`.

```
import Config from "@jangaroo/runtime/Config";
import Content from "@coremedia/studio-client.cap-rest-client/content/Content";
import FieldContainer from "@jangaroo/ext-ts/form/FieldContainer";
import ValueExpression from
"@coremedia/studio-client.ext.client-core/data/ValueExpression";

interface CustomPropertyFieldConfig extends Config<FieldContainer>,
Partial<Pick<CustomPropertyField,
  "bindTo" |
  "propertyName"
>> {
}
```

```
class CustomPropertyField extends FieldContainer {
  declare Config: CustomPropertyFieldConfig;

  /**
   * A value expression evaluating to the Bean whose property (path) is
edited.
   */
  bindTo:ValueExpression<Content>;

  /**
   * The property to bind.
   */
  propertyName: string;

  constructor(config: Config<CustomPropertyField>) {
    super(config);
  }
}

export default CustomPropertyField;
```

The two properties `propertyName` and `bindTo` are mandatory for all property
fields. The former declares the name of the property to be edited, which is used
both for accessing the model and for localizing the property field. The latter
declares a value expression evaluating to the `Content` object.

```
import Config from "@jangaroo/runtime/Config";
import Content from "@coremedia/studio-client.cap-rest-client/content/Content";
import FieldContainer from "@jangaroo/ext-ts/form/FieldContainer";
import ValueExpression from
"@coremedia/studio-client.ext.client-core/data/ValueExpression";

interface CustomPropertyFieldConfig extends Config<FieldContainer>,
Partial<Pick<CustomPropertyField,
  "bindTo" |
  "propertyName" |
  "readOnly" |
  "hideIssues"
  >> {
}

class CustomPropertyField extends FieldContainer {
  declare Config: CustomPropertyFieldConfig;

  /**
   * A value expression evaluating to the Bean whose property (path) is
edited.
   */
  bindTo:ValueExpression<Content>;

  /**
   * The property to bind.
   */
  propertyName: string;

  /**
   * Set the <code>readOnly</code> config option of the contained field.
   */
  readOnly: boolean;

  /**
   * Don't show any validation issues on this property field.
   */
  hideIssues: boolean;

  constructor(config: Config<CustomPropertyField>) {
    super(config);
  }
```

```
}
export default CustomPropertyField;
```

Another Config option is to hard-wire the property field to be read-only. As a fourth configuration option, you can disable the visual indication of content errors or warnings via configuration. These options will later on be passed to the appropriate plugins.

Several plugins are available to customize the behavior of your custom property field. For example, the property label is used when displaying the component in a form. Using the following plugin, you can make sure that the label is localized according to the standard localization pattern.

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import FieldContainer from "@jangaroo/ext-ts/form/FieldContainer";
import SetPropertyLabelPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/plugins/SetPropertyLabelPlugin";

...

class CustomPropertyField extends FieldContainer {
  declare Config: CustomPropertyFieldConfig;

  ...

  constructor(config: Config<CustomPropertyField>) {
    super(ConfigUtils.apply(Config(FieldContainer, {
      ...
      ...ConfigUtils.append({
        plugins: [
          Config(SetPropertyLabelPlugin, {
            bindTo: config.bindTo,
            propertyName: config.propertyName,
          }),
        ],
      })
    }), config));
  }
}

export default CustomPropertyField;
```

Now, the actual input property editor is added to the custom field. It needs some configuration and a bunch of plugins of its own. `tabIndex` is set to 1 to force the text field into the standard focus tab order. The `readOnly` flag is simply handed through.

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import FieldContainer from "@jangaroo/ext-ts/form/FieldContainer";
import SetPropertyLabelPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/plugins/SetPropertyLabelPlugin";

...

class CustomPropertyField extends FieldContainer {
  declare Config: CustomPropertyFieldConfig;
```

```
...
  constructor(config: Config<CustomPropertyField>) {
    super(ConfigUtils.apply(Config(FieldContainer, {
      ...
      items: [
        Config(TextField, {
          tabIndex: 1,
          readOnly: config.readOnly,
        }),
      ],
      ...ConfigUtils.append({
        plugins: [
          Config(SetPropertyLabelPlugin, {
            bindTo: config.bindTo,
            propertyName: config.propertyName,
          }),
        ],
      })
    }), config));
  }
}

export default CustomPropertyField;
```

To register the property field properly with *Studio* for the purposes of preview-base editing and navigating directly to property field, you need to declare the following plugin:

```
import Config from "@jangaroo/runtime/Config";
import PropertyFieldPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/PropertyFieldPlugin";

...

      items: [
        Config(TextField, {
          tabIndex: 1,
          readOnly: config.readOnly,
          ...ConfigUtils.append({
            Config(PropertyFieldPlugin, {
              propertyName: config.propertyName
            }),
          }),
        }),
      ],
...
```

Using this plugin lets *Studio* know that your component is authoring a content property. Among other things, this will set up your component to cooperate properly with the content errors and warnings navigation window, and with content shortcuts from the embedded preview.

In order to support content validation, a field should also be highlighted in red (when content errors are present), or orange (when content warnings are present). See Section 9.23.1, "Validators" [252] for information on how to set up server-side content validators. On the client side, the ShowIssuesPlugin as shown below handles all the work. It reads the issues generated on the server and attaches one of the style classes issue-error and issue-warn if an

issue is present. Pass all relevant configuration options from the property field to the plugin, especially the options `bindTo` and `propertyName`.

Additionally, this plugin highlights the property field in differencing mode when the property value has changed. To this end, it attaches a style class `issue-change` to its component if the property is reported as changed by the server.

For struct properties, a dot-separated property path can be used as the property name to visualize issues and differences of a property nested in a struct value.

Because the string property field shown here is based on a plain `TextField`, all formatting rules are already provided in the standard style sheets. For custom components, it might be necessary to add CSS rules for the style classes `issue-error`, `issue-warn`, and `issue-change` in order to visualize issues and changes correctly.

The `PropertyFieldPlugin` and the `showIssuesPlugin` are often, but not always attached to the same component. In some cases it may appropriate to designate an outer component as the component to scroll into view when navigating to a property, but to select an inner component to be tagged with issue style classes.

```
import Config from "@jangaroo/runtime/Config";
import PropertyFieldPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/PropertyFieldPlugin";
import ShowIssuesPlugin from
"@coremedia/studio-client.main.editor-components/sdk/validation/ShowIssuesPlugin";

...

    items: [
      Config(TextField, {
        tabIndex: 1,
        readOnly: config.readOnly,
        ...ConfigUtils.append({
          Config(PropertyFieldPlugin, {
            propertyName: config.propertyName
          }),
          Config(ShowIssuesPlugin, {
            bindTo: config.bindTo,
            ifUndefined: "",
            propertyName: config.propertyName,
            hideIssues: config.hideIssues
          })
        }),
      }),
    ],
...
```

When the string field is empty, you want to display a message instructing the user to enter a text. Also, the component should be made read only (meaning that the user cannot enter any text but still can mark and copy the content) when the edited content is checked out by another user or is forced to be read only by the content panel. Consequently, two further plugins are added.

Show default text and set read-only state

```
import Config from "@jangaroo/runtime/Config";
import BindReadOnlyPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/plugins/BindReadOnlyPlugin";
```

```
import PropertyFieldPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/PropertyFieldPlugin";
import ShowIssuesPlugin from
"@coremedia/studio-client.main.editor-components/sdk/validation/ShowIssuesPlugin";
import SetPropertyEmptyTextPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/plugins/SetPropertyEmptyTextPlugin";

...
      items: [
        Config(TextField, {
          tabIndex: 1,
          readOnly: config.readOnly,
          ...ConfigUtils.append({
            Config(PropertyFieldPlugin, {
              propertyName: config.propertyName
            }),
            Config(ShowIssuesPlugin, {
              bindTo: config.bindTo,
              ifUndefined: "",
              propertyName: config.propertyName,
              hideIssues: config.hideIssues
            }),
            Config(SetPropertyEmptyTextPlugin, {
              bindTo: config.bindTo,
              propertyName: config.propertyName,
            }),
            Config(BindReadOnlyPlugin, {
              forceReadOnlyValueExpression:
config.forceReadOnlyValueExpression,
              bindTo: config.bindTo
            }),
          }),
        }),
      ],
...
```

Lastly, the most important plugin is added. Editor changes to the field's value need to be passed to the server. The other way around, the field's value should be synchronized to changes of the server-side value. This bi-directional data binding is typically done using the versatile `BindPropertyPlugin` as shown below.

Data binding

```
import Config from "@jangaroo/runtime/Config";
import BindPropertyPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/BindPropertyPlugin";
import BindReadOnlyPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/plugins/BindReadOnlyPlugin";
import PropertyFieldPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/PropertyFieldPlugin";
import ShowIssuesPlugin from
"@coremedia/studio-client.main.editor-components/sdk/validation/ShowIssuesPlugin";
import SetPropertyEmptyTextPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/plugins/SetPropertyEmptyTextPlugin";

...
      items: [
        Config(TextField, {
          tabIndex: 1,
          readOnly: config.readOnly,
          ...ConfigUtils.append({
            Config(PropertyFieldPlugin, {
              propertyName: config.propertyName
            }),
            Config(ShowIssuesPlugin, {
              bindTo: config.bindTo,
              ifUndefined: "",
```

```
               propertyName: config.propertyName,
               hideIssues: config.hideIssues
            }),
            Config(SetPropertyEmptyTextPlugin, {
               bindTo: config.bindTo,
               propertyName: config.propertyName,
            }),
            Config(BindReadOnlyPlugin, {
               forceReadOnlyValueExpression:
config.forceReadOnlyValueExpression,
               bindTo: config.bindTo,
            }),
            Config(BindPropertyPlugin, {
               bindTo: config.bindTo.extendBy('properties',
config.propertyName),
               ifUndefined: config.ifUndefined,
               bidirectional: config.readOnly,
            })
         }),
      }),
   ],
...
```

While the list of plugins may appear quite long at first, it is very helpful to be able to separate the different aspects of a property field in different plugins. If you want to provide a custom algorithm of reacting to an empty value, for example, you can easily do so by just omitting the respective plugin declaration, and providing custom handling code – either in the base class or possibly extracted into your own reusable plugin.

9.6.3 Compound Field

The following code example shows a more complex scenario, where a field for a URL is created that lets the user open a browser window or tab for the linked page with a single click.

```
import {bind} from "@jangaroo/runtime";
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import Button from "@jangaroo/ext-ts/button/Button";
import TextField from "@jangaroo/ext-ts/form/field/Text";
import BindPropertyPlugin from "@coremedia/studio-client.ext.ui-components/plugins/BindPropertyPlugin";
import PropertyFieldPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/PropertyFieldPlugin";
import SetPropertyLabelPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/plugins/SetPropertyLabelPlugin";
import SetPropertyEmptyTextPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/plugins/SetPropertyEmptyTextPlugin";
import BindDisablePlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/plugins/BindDisablePlugin";
import UrlPropertyFieldBase from "./UrlPropertyFieldBase";
import PropertyFieldExample_properties from "./PropertyFieldExample_properties";

interface UrlPropertyFieldConfig extends Config<UrlPropertyFieldBase>,
  Partial<Pick<UrlPropertyField,
    "readOnly" |
    "hideIssues"
  >> {
}

class UrlPropertyField extends UrlPropertyFieldBase {
```

```
    declare Config: UrlPropertyFieldConfig;

    constructor(config: Config<UrlPropertyField> = null) {
        super(((()=> ConfigUtils.apply(Config(UrlPropertyField, {
            items: [
                Config(TextField, {
                    itemId: "urlTextField",
                    name: "properties." + this.propertyName,
                    plugins: [
                        Config(PropertyFieldPlugin, {
                            propertyName: config.propertyName,
                        }),
                        Config(SetPropertyLabelPlugin, {
                            bindTo: config.bindTo,
                            propertyName: config.propertyName,
                        }),
                        Config(SetPropertyEmptyTextPlugin, {
                            bindTo: config.bindTo,
                            propertyName: config.propertyName,
                        }),
                        Config(BindDisablePlugin, {
                            bindTo: config.bindTo,
                        }),
                        Config(BindPropertyPlugin, {
                            bindTo: config.bindTo.extendBy(
                              'properties',
                              config.propertyName
                            ),
                            ifUndefined: "",
                            bidirectional: true,
                        }),
                    ]
                }),
                Config(Button, {
                    itemId: "urlOpenButton",
                    text: PropertyFieldExample_properties.UrlPropertyField_open_text,
                    handler: bind(this, this.openFrame),
                }),
            ],
        }), config))());
    }
}

export default UrlPropertyField;
```

The base class:

```
import {as} from "@jangaroo/runtime";
import Config from "@jangaroo/runtime/Config";
import Container from "@jangaroo/ext-ts/container/Container";
import ValueExpression from "@coremedia/studio-client.client-core/data/ValueExpression";

interface UrlPropertyFieldBaseConfig extends Config<Container>, Partial<Pick<UrlPropertyFieldBase,
    "bindTo" |
    "propertyName"
>> {
}

class UrlPropertyFieldBase extends Container {

  declare Config: UrlPropertyFieldBaseConfig;

  constructor(config: Config<UrlPropertyFieldBase> = null) {
    super(config);
  }

  /**
   * A property path expression leading to the Bean whose property is edited.
   */
  bindTo: ValueExpression = null;
```

```
  /**
   * The property of the Bean to bind in this field.
   */
  propertyName: string = null;

  /**
   * Try to open a new window with the string currently stored in the property used as the URL.
   */
  openFrame(): void {
    const url = as(this.bindTo.extendBy('properties', this.propertyName).getValue(), String);
    if (url) {
      window.open(url, 'externalLinkTarget');
    }
  }
}

export default UrlPropertyFieldBase;
```

The above is an example of a compound field, where you need to wrap multiple Ext JS components in a container. This is possible, but you must take care to declare and pass around all configuration properties that need to be set on subcomponents.

There is also some application logic, which is what the base class is for. While you could technically embed any code into the TypeScript file itself, it is good practice to separate out application code in a base class.

```
import { bind } from "@jangaroo/runtime";
import Config from "@jangaroo/runtime/Config";
import Button from "@jangaroo/ext-ts/button/Button";

Config(Button, {
    itemId: "urlOpenButton",
    text: "...",
    handler: bind(this, this.openFrame),
})
```

Example 9.33. Using a base class method

9.6.4 Complex Setups

Keep in mind that somewhat counter-intuitively, the base class constructor has not run while the component tree is built in the constructor of the TypeScript class. In particular, this means that methods calls in the TypeScript file (not mere usages of methods as event handlers) will find the fields of the base class uninitialized. For example, calling `Config(TextField, { name: compute Name() }` would enter the method `computeName` before the base class constructor has run, so that some initialization would have to be done early on demand. On the other hand, in `<Button handler="{handleButton}"/>` the method `handleButton` is only invoked after the component is initialized. If a method that is called early needs access to the configuration, you must pass

the `config` object as a parameter: `Config(TextField, { name: computeName(config) }`.

9.7 Hiding Components on Content Forms

Editors can hide fields of a content item so that the content form is cleaned up for their daily use. See Section 3.6, "Hiding Studio Form Components" in *Studio User Manual* for details.

For the standard Blueprint Studio forms the feature works out of the box. To use this feature for your customized Studio forms you have to adapt your forms as described below.

9.7.1 Code Customization for the HideService

In order to hide components on a content form, the service `HideService` is used. This service deals only with Studio components which implement `HidableMixin`:

```
import Config from "@jangaroo/runtime/Config";
import Mixin from "@jangaroo/ext-ts/Mixin";

interface HidableMixinConfig extends Config<Mixin>, Partial<Pick<HidableMixin,

  "hideText" |
  "hideId"
>> {
}

/**
 * Adds hide properties feature to the component this mixin is mixed into.
 */
declare class HidableMixin extends Mixin {
  Config: HidableMixinConfig;

  /**
   * Sets the text used to display this component in the hide service dialog.

   */
  set hideText(newHideText: string);

  /**
   * @returns the text used to display this component in the hide service
dialog.
   */
  get hideText(): string;

  /**
   * Sets the optional id to identify this component.
   * If not available the mixin demands the presence of the component's item
 id
   * The id might be used to persist the state of this component and
   * should be hence permanent.
   */
  set hideId(newHideId: string);
```

```
  /**
   * Gets the optional id to identify this component.
   * The id might be used to persist the state of this component and
   * should be hence permanent.
   */
  get hideId(): string;
}
export default HidableMixin;
```

Example 9.34. HidableMixin.ts

Any Studio component not implementing this mixin will be ignored by the `HideService`. In the standard Blueprint Studio all relevant fields on the content forms already implement the mixin. Your customized fields, though, must imple-ment the mixin so that they are considered by `HideService`.

hideId   An ID which must be global for a given content type. Usually you don't have to set it for yourself. But it is internally set to `propertyName` when the given component is a property field. See Section 9.6, "Cus-tomizing Property Fields" [162] for details about property fields.

For the HideService to persist the hidden state of a component the component itself and its parent up to the `DocumentForm` must have an `itemId` or a `hideId`.

hide-Text   The text is used to display the corresponding component in the hide service dialog. For example, for a `FieldContainer` it is recommen-ded to set it to the function call `getFieldLabel()`.

Blueprint example code of hideable components

Have a look at a Blueprint example of the requirements. The first level children of a content form are all of the type `DocumentForm` which implements the mixin in its base class. The following code of `DocumentFormBase.ts` shows the implementation of the mixin.

```
import FloatingToolbarContainer from
"@coremedia/studio-client.ext.ui-components/components/FloatingToolbarContainer";
import HidableMixin from
"@coremedia/studio-client.ext.ui-components/mixins/HidableMixin";
import { mixin } from "@jangaroo/runtime";
import Config from "@jangaroo/runtime/Config";
import DocumentForm from "./DocumentForm";

interface DocumentFormBaseConfig extends Config<FloatingToolbarContainer>,
Config<HidableMixin>, Partial<Pick<DocumentFormBase,
  "title" |
  "hideText"
  >> {
}

class DocumentFormBase extends FloatingToolbarContainer {
  declare Config: DocumentFormBaseConfig;
```

```
  // The title of this form when used as a tab.
  title: string = null;

  constructor(config: Config<DocumentForm> = null) {
    super(config);
  }

  /** @private */
  set hideText(newHideText: string) {
    // The hideText is determined by the getter. Nothing to do.
  }

  /** @inheritDoc */
  get hideText(): string {
    return this.title;
  }
}

interface DocumentFormBase extends HidableMixin{}

mixin(DocumentFormBase, HidableMixin);

export default DocumentFormBase;
```

Example 9.35. DocumentFormBase.ts

The content form uses the `title` property for the tab label, therefore, the mixin implementation of `hideText` uses the same as well. Note that the setter of `hideText` has an empty block as you can change the `hideText` only by changing the title.

Take a look into the code snippet of `CMArticleForm.ts` which renders the content form for the content type `CMArticle`:

```
import Container from "@jangaroo/ext-ts/container/Container";
import DocumentForm from
"@coremedia/studio-client.main.editor-components/sdk/premular/DocumentForm";
import Config from "@jangaroo/runtime/Config";
import BlueprintTabs_properties from "../BlueprintTabs_properties";
import CMArticleSystemForm from "./components/CMArticleSystemForm";
import DefaultExtraDataForm from "./components/DefaultExtraDataForm";
import AuthorLinkListDocumentForm from
"./containers/AuthorLinkListDocumentForm";
import DetailsDocumentForm from "./containers/DetailsDocumentForm";
import ExternallyVisibleDateForm from "./containers/ExternallyVisibleDateForm";
import MediaDocumentForm from "./containers/MediaDocumentForm";
import MultiLanguageDocumentForm from "./containers/MultiLanguageDocumentForm";
import RelatedDocumentForm from "./containers/RelatedDocumentForm";
import TeaserDocumentForm from "./containers/TeaserDocumentForm";
import ValidityDocumentForm from "./containers/ValidityDocumentForm";
import ViewTypeSelectorForm from "./containers/ViewTypeSelectorForm";

//...
Config(Container, {
  //...
  items: [
    Config(DocumentForm, {
      title: BlueprintTabs_properties.Tab_content_title,
      itemId: "contentTab",
      items: [
        Config(DetailsDocumentForm, { bindTo: config.bindTo }),
        Config(TeaserDocumentForm, {
          bindTo: config.bindTo,
          collapsed: true,
        }),
```

```
        Config(MediaDocumentForm, { bindTo: config.bindTo }),
        Config(AuthorLinkListDocumentForm, { bindTo: config.bindTo }),
        Config(RelatedDocumentForm, { bindTo: config.bindTo }),
        Config(ViewTypeSelectorForm, { bindTo: config.bindTo }),
        Config(ExternallyVisibleDateForm, { bindTo: config.bindTo }),
        Config(ValidityDocumentForm, { bindTo: config.bindTo }),
      ],
    }),
    Config(DefaultExtraDataForm),
    Config(MultiLanguageDocumentForm, { bindTo: config.bindTo }),
    Config(CMArticleSystemForm, { bindTo: config.bindTo }),
  ],
  //...
})
```

Example 9.36. CMArticleForm.ts

The code shows four children of the type `DocumentForm` which represent the four tabs of the content form as seen in the following screenshot:
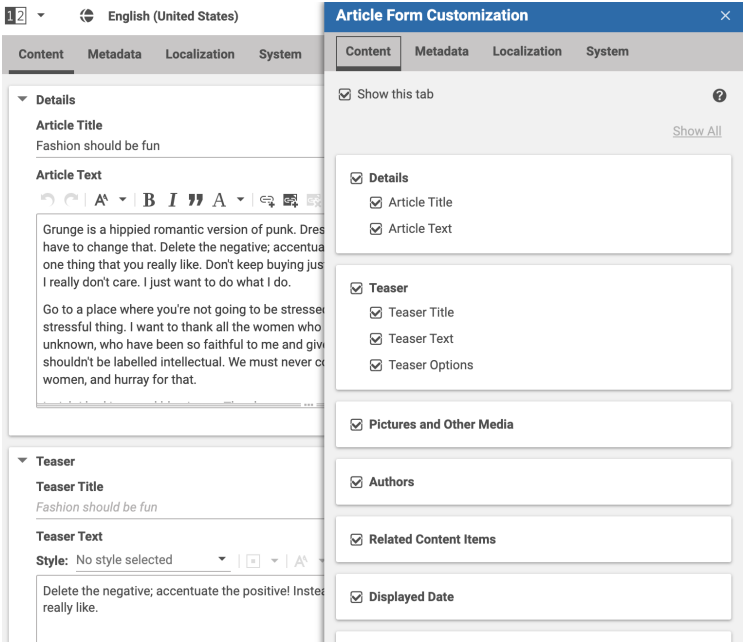
Figure 9.4. Hide Service Dialog

You can see that the first content form has the `itemId` "ContentTab". The other content forms `DefaultExtraDataForm`, `MultiLanguageDocu` `mentForm` and `CMArticleSystemForm` have all their own `itemId` defined in the respective TypeScript files.

The first child on the "Content" content tab is the property field group "Details" with sub children "Article Title" and "Article Text". The fields are defined in `DetailsDocumentForm.ts` which is a subtype of `CollapsiblePanel` which also implements the `HidableMixin`:

```
import Container from "@jangaroo/ext-ts/container/Container";
import Config from "@jangaroo/runtime/Config";
import StringPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/StringPropertyField";
import RichTextPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/richtext/RichTextPropertyField";

//...
Config(Container, {
  //...
  items: [
    Config(StringPropertyField, {
      bindTo: config.bindTo,
      itemId: "title",
      propertyName: "title",
    }),
    Config(RichTextPropertyField, {
      bindTo: config.bindTo,
      itemId: "detailText",
      propertyName: "detailText",
      initialHeight: 200,
    }),
  ],
  //...
})
```

Example 9.37. DetailsDocumentForm.ts

Again, each item has its own `itemId`. In addition both `StringProperty Field` and `RichTextPropertyField` are the subtype of `AdvancedField Container` which implements the `HidableMixin`.

9.7.2 Studio Logging

When preparing your custom code you should check if all relevant components appear in the hide service dialog. If some components are missing, you can use the Studio logging which logs the components which are ignored by the hide service.

To this end, append the hash parameter `loglevel=warn` to your Studio URL. When now opening the hide service dialog, there will be various warnings in the console log of the browser you should pay attention to:

```
HideService: com-acme-config-ExampleDataForm-1660 has no hideId or itemId.
```

The component with the ID `com-acme-config-ExampleDataForm-1660` has no hideId or itemId. Find out in your Studio code where the component with

the xtype `com-acme-config-ExampleDataForm` is used and configure an item ID to the component.

```
HideService: com-acme-config-MyProperyField-123 is not a hidable.
```

Your custom component with the xtype `com-acme-config-MyPropery Field` must implement the `IHidableMixin`.

```
HideService: com-acme-config-MyContainer-456 has no hide text.
```

Your custom component with the xtype `com-acme-config-MyContainer` and with the ID `com-acme-config-MyContainer-456` implements the mixin but has no `hideText`. Therefore, the hide service dialog doesn't now how to display the combobox for this component. Configure a suitable text for the component.

Not all warnings are relevant as long as relevant components for your hide service dialog are now recognized.

9.7.3 Configuration Options

The following table describes the available Spring properties that you can configure for the Hide Service.

`studio.hideservice.enabled`	
Type	Boolean
Default	true
Description	If set to `false`, the hide service is disabled and the hide service dialog will not be accessible. No components will be hidden by the service.

`studio.hideservice.hideDepth`	
Type	Integer
Default	3
Description	The depth of the component hierarchy to which the hide service will provide the hide option. The root of the component hierarchy is the `DocumentTabPan`

`el`. For example, if set to 1 only the tabs of `DocumentTabPanel` will be provided but not its children.

Table 9.3. Hide Service Spring Properties

9.8 Hiding Components for Sites

With the `SiteAwareVisibilityPlugin`, you can show or hide content form elements (for example, property fields) depending on the activation of a "feature" for a specific site.

The `SiteAwareVisibilityPlugin` takes a parameter called *feature*, which is a name for the feature. You can group two or more plugins by giving them the same feature name. The second optional parameter is *contentValue-Expression*, which calculates the value of the site from the current content. If not set, the values from the global settings will be used instead.

If you configure any Ext JS component to use this plugin, that component only becomes visible when this feature is configured to be active for the site that the current content belongs to.

By default, the configuration for features of a site is done in a `CMSettings` content item, which has to be named `<SITE_ROOT_FOLDER>/Options/Settings/Studio Features`. Global settings can be configured too in the settings document `/Settings/Option/Settings/Studio Features` in case features should not be site-specific.

This settings bundle consists of a `StringList` property named "features" and contains the string values that in turn need to be configured as desired in the `SiteAwareVisibilityPlugin`:

∨ Settings

PROPERTY	VALUE	TYPE
∨ ...		Struct
∨ features		String List
#1	Article Title	String

Figure 9.5. Studio Feature Settings

The given example shows how to hide the "Title" field of an article for all sites that do not have the feature "Article Title" activated.

```
...
Config(DetailsDocumentForm, {
```

```
 title: CustomLabels_properties.PropertyGroup_Details_label,
 itemId: "detailsDocumentForm",
 propertyNames: ["title", "detailText"],
 expandOnValues: "title,detailText",
 items: [
Config(StringPropertyField, {
  itemId: "title",
  propertyName: "title",
  ...ConfigUtils.append({
 plugins: [
   Config(SiteAwareVisibilityPlugin, {
   feature: "Article Title",
   ifUndefined: false,
   contentValueExpression: config.bindTo,
   }),
 ],
  }),
}),
...
```

Example 9.38. SiteAwareVisibilityExample.ts

9.9 Coupling Studio and Embedded Preview

In Section 4.3.5, "Adding Document Metadata" in *Content Application Developer Manual* it is described in detail how to use the *Content Application Engine* to include metadata in Web documents.

This section explains how to access metadata of content items that are shown in the Studio's embedded preview.

9.9.1 Built-in Processing of Content and Property Metadata

CoreMedia Studio automatically accesses and interprets content and property metadata in order to connect preview and content form. When the user edits a content property that is mapped to a preview DOM element via metadata, all changes are reflected in the embedded preview, either instantly (for simple content properties like strings) or through automatically reloading the preview.

Moving the mouse cursor over the preview will highlight elements with attached content and/or property metadata. Right-clicking one of these elements in the preview focuses the corresponding form field, if possible. If the clicked element belongs to a content object different from the content object currently displayed in the content form, a context menu is opened that shows a breadcrumb to navigate through the metadata hierarchy down to the clicked content object, and it offers the options to open the content in a new tab or in the library.

9.9.2 Using the Preview Metadata Service

As described in Section 4.3.5, "Adding Document Metadata" in *Content Application Developer Manual*, it is possible to include arbitrary metadata in Web documents by means of the FreeMarker macro `<@preview.metadata>`. In the rendered Web document, the different metadata chunks are included as JSON-serialized values of the custom HTML attribute `data-cm-metadata` of different DOM nodes.

9.9.2.1 The Metadata Service Interface

In Chapter 3, *Deployment* [18] it is described that the preview *CAE* web application and *Studio* communicate via an internal messaging system. This messaging system is also used to transfer metadata from the preview side to the *Studio* side. To hide this low-level layer from the Studio developer, CoreMedia offers a *metadata service* for each instance of a preview panel that runs in *CoreMedia Studio*. Given a preview panel, its metadata service can be obtained as follows (please see the API documentation of `PreviewPanel` for further information on how to obtain a preview panel component).

Communication between Studio and CAE web application

```
import PreviewPanel from
"@coremedia/studio-client.main.editor-components/sdk/preview/PreviewPanel";

//...
const previewPanel:PreviewPanel = ... ;
const metadataService = previewPanel.getMetadataService();
```

The metadata service interface currently offers just one method, namely:

```
getMetadataTree(filterProperties?: String[]): MetadataTree;
```

Via this method, the metadata of the associated preview panel's content item can be retrieved. Metadata embedded in the preview content item is represented in terms of a tree. This *metadata tree* originates from the DOM tree of the preview content item: Hierarchical relationships between the metadata tree nodes correspond to hierarchical relationships between the DOM tree nodes that the respective metadata chunks are attached to. Consequently, the metadata tree is basically a projection of the DOM tree to its metadata information.

It is possible to further filter the metadata tree by means of the method's optional parameter, namely an array of properties. If such properties are supplied, the metadata tree contains only nodes that have at least one of these properties. In addition, other properties than the given properties are filtered out. Such a filtered metadata tree is a projection of the metadata tree that contains all metadata. The above statement about the correspondence of hierarchical relationships in the metadata tree and the DOM tree still holds.

9.9.2.2 Working with the Metadata Tree

When working with the metadata tree, you have two data structures to your convenience:

- `@coremedia/studio-client.main.editor-components/sdk/preview/metadata/MetadataTree`: This data structure represents the whole tree and, for example, offers methods for accessing

specific nodes (by their ID) or getting a list of all tree nodes (in breadth-first order).

- `@coremedia/studio-client.main.editor-compon ents/sdk/preview/metadata/MetadataTreeNode`: This data structure represents a single metadata tree node. It offers a range of methods like retrieving the parent or the children of a node, finding specific parent nodes upwards in the hierarchy or specific child nodes downwards in the hierarchy or accessing properties of a metadata tree node.

In the following you will find two examples of how to use the metadata tree. Suppose that the FTL templates on the *CAE* side have been prepared to include metadata about content. At different points throughout the FTL templates the code might look as follows:

```
...
<#assign contentMetadata = { "contentInfo": {
  "title": self.content.title,
  "keywords": self.content.keywords }
}/>

<div <@preview.metadata contentMetadata />>
  ...
</div>
...
```

For more details on how to include metadata in FTL templates, please refer to the documentation Section 6.5.2, "Preview (preview)" in *Frontend Developer Manual*.

In a preview content item there might be multiple of such content-related metadata chunks attached to different DOM nodes. Suppose you want to gather the titles of all the contents that are included in such metadata chunks. One way to gather these titles in an array is the following:

```
import MetadataTree from
"@coremedia/studio-client.main.editor-components/sdk/preview/metadata/MetadataTree";
import MetadataTreeNode from
"@coremedia/studio-client.main.editor-components/sdk/preview/metadata/MetadataTreeNode";

//...
const metadataService = null;
const metadataTree:MetadataTree = metadataService.getMetadataTree();
const result = [];
let nodesToProcess = metadataTree.getRoot() ? [metadataTree.getRoot()] : [];

let arrayIndex = 0;
while (arrayIndex < nodesToProcess.length) {
  const currentNode:MetadataTreeNode = nodesToProcess[arrayIndex];
  if (currentNode.getProperty("contentInfo")) {
    const title = currentNode.getProperty("contentInfo").title;
    result.push(title);
  }
  if (currentNode.getChildren()) {
    nodesToProcess = nodesToProcess.concat(currentNode.getChildren());
  }
```

```
  arrayIndex++;
}
```

In this example, the whole metadata tree is traversed in a breadth-first manner. For each node it has to be checked whether it has the `contentInfo` property as there might be metadata nodes with completely other information.

The code can be simplified considerably if a filtered metadata tree is retrieved:

```
const metadataService = ... ;
const metadataTree = metadataService.getMetadataTree(["contentInfo"]);
let result = [];
const metadataNodesList = metadataTree.getAsList();
metadataNodesList.forEach((node:MetadataTreeNode) => {
  result.push(node.getProperty("contentInfo").title);
});
```

In this case, the metadata tree is filtered on retrieval, namely for metadata nodes that contain the `contentInfo` property. Now it is sufficient to get all metadata tree nodes as an array, walk through it and gather the content titles.

9.9.2.3 Listening to Metadata Availability/Changes

A metadata service is always associated with a specific preview panel. When a content item is opened in a preview panel, it takes some time until its metadata is loaded. This happens asynchronously via the above mentioned message service. Consequently, it is necessary to have a mechanism to listen to the availability of a conten item's metadata. In addition, changes to the metadata may occur when the displayed content item of the preview panel changes. Thus, it is also necessary to listen to metadata changes.

To this end, the method `IMetadataService.getMetadataTree()` is dependency-tracked. This means that it is possible to listen to changes to the returned metadata tree by using a function value expression (see `@core media/studio-client.client-core/data/dependencies/Depend encyTracker` and `@coremedia/studio-client.client-core/data/ValueExpressionFactory`). The following example is provided to illustrate this process:

```
import ValueExpressionFactory from
"@coremedia/studio-client.client-core/data/ValueExpressionFactory";
import PreviewPanel from
"@coremedia/studio-client.main.editor-components/sdk/preview/PreviewPanel";

const previewPnl:PreviewPanel // = ...
ValueExpressionFactory.createFromFunction(()=> {
  const metadataTree = previewPnl.getMetadataService().getMetadataTree();
  return metadataTree.getRoot() ? metadataTree : undefined;
}).loadValue(metadataTree => {
  // metadata tree loaded!
```

```
  metadataTree.getAsList() //...
});
```

In this example `MetadataTree.getRoot()` is used as an indicator of whether the metadata has already been loaded (if not, the method returns `null`). A function value expression is created around a function that simply determines the existence of a metadata root node, returning `undefined` as long as it does not exist. Afterwards the value expression is loaded, which automatically retries to invoke the function until it returns a non `undefined` value. As soon as it does, the metadata has been loaded and the callback function can now process the metadata tree.

9.10 Storing Preferences

A custom component may have to store user preferences persistently. To this end, the global `@coremedia/studio-client.cap-base-models/preferences/editorPreferences` offers the method `getPreferences` of the interface `IEditorPreferences`. The method returns a `Struct` object that is stored in the `EditorPreferences` content item of the current user. You can modify this struct using the standard struct API as described in .

To offer a bit more utility, the class `@coremedia/studio-client.cap-base-models/preferences/PreferencesUtil` provides two handy methods for reading and writing complex objects in the preferences struct: `getPreferencesJSONProperty` and `updatePreferencesJSONProperty`. These methods support strings, numbers, Boolean, contents, and complex objects and arrays containing such values. The Studio API uses these methods internally for persisting saved searches (including custom filters), open tabs, dashboard widget states, and bookmarks.

9.11 Customizing Central Toolbars

Toolbars contain buttons for making functionality quickly accessible. There are the following central toolbars that you might want to customize:

- The `@coremedia/studio-client.ext.frame-components/Main NavigationToolbar` toolbar on the top left of Studio containing the "Favorites" and "Create" menu buttons.
- The `@coremedia/studio-client.main.editor-components/sdk/desktop/HeaderToolbar` toolbar on the top right of Studio containing the site selector, buttons for the main Studio functionalities (library, Control Room, dashboard) and the menu buttons for jobs and notifications.
- The `@coremedia/studio-client.main.editor-components/sdk/desktop/ActionsToolbar` on the right of each content form for completing the work on the current content.

The following section describes how you can use `AddItemsPlugin` to add your custom button to an existing toolbar.

It is good practice to wrap the custom UI component's actual functionality (that is, what your button will do when clicked) in `Action` objects, so that these actions can be reused for other buttons. Actions are described in Section 5.1.3, "Actions" [37].

9.11.1 Adding Buttons to the Header Toolbar

Customizing `MainNavigationToolbar` and `HeaderToolbar` is very similar. You will get an example for the latter here.

If you want to add fixed buttons to `HeaderToolbar` (that is, buttons that can not be modified or removed by the user), you need to add them to either the top or the bottom section of the toolbar.

The given example shows how to use the `AddItemsPlugin` plugin to add your own buttons after the site selector. Simply add the following code to the plugin rules section of your Studio plugin:

```
import Config from "@jangaroo/runtime/Config";
import Component from "@jangaroo/ext-ts/Component";
import HeaderToolbar from
"@coremedia/studio-client.main.editor-components/sdk/desktop/HeaderToolbar";
import AddItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/AddItemsPlugin";

//...
```

```
Config(HeaderToolbar, {
  plugins: [
    Config(AddItemsPlugin, {
      items: [
        //...add your component/button here...
      ],
      after: [
        Config(Component, { itemId:
HeaderToolbar.HEADER_MENU_BAR_SEPARATOR_ITEM_ID_1 }),
      ],
    }),
  ],
})
```

Ensuring a proper order of the items in toolbars helps significantly in making the application usable. Note how an `after` constraint is used to put the new button to a specific place. It uses the framework-predefined `itemId` of the toolbar separator right to the site selector to describe the desired location of the added button.

To add a simple button with an action, enter the following code inside the `<items>` element (see Section 9.4, "Localizing Labels" [141] to learn how to localize the label of the button):

```
import Config from "@jangaroo/runtime/Config";
import Button from "@jangaroo/ext-ts/button/Button";
import ShowCollectionViewAction
  from
"@coremedia/studio-client.main.editor-components/sdk/actions/ShowCollectionViewAction";

...
    Config(Button, {
      baseAction: new ShowCollectionViewAction({
        text: "To be Published",
        published: false,
        editedByMe: true,
        contentType: "CMArticle",
      }),
    })
...
```

Example 9.39. Adding a search for content items to be published

This code snippet will create a search folder button with label text "To be Published" that uses a `ShowCollectionViewAction` action to open the Library window in a mode that searches for a restricted set of content items (please see the API documentation for `ShowCollectionViewAction` for more details).

9.11.2 Providing Default Search Folders

The first section of the *CoreMedia Studio*'s header toolbar contains user-defined search folders within the 'Favorites' menu. When you click a search folder, the collection view opens up in search mode showing the results of a predefined

query. The user can create custom search folders via the `Save Search` button of the Studio library toolbar in search mode. Users can also modify existing search folders, change their order, rename them, or delete them altogether.

As a developer, you can provide a default set of search folders to your first-time users, so that the favorites menu won't appear empty on a user's first login to Studio.

> **WARNING**
>
> The configuration option shown below explains solely the default set of search folders that users will see on their first login. When Studio detects that there are no custom search folders defined yet for the user logging in, this default set will be copied to this user's settings - from then on, management of the search folder section is completely up to the user, and your configuration will be ignored. If you want to permanently add buttons (including buttons representing search folders) to the **Favorites** menu or *Header* toolbar, please refer to Section 9.11.1, "Adding Buttons to the Header Toolbar" [189] above.

You can add default search folders by using the `AddArrayItemsPlugin` on the `FavoritesButton`. Each array item has to include the relevant search parameters that you want to pass to the library on opening. These parameters are modularized in terms of the different parts of the collection view in search mode. Thus, each array item is a nested JavaScript object literal that itself contains possibly multiple objects for the various parameter parts. These embedded objects can be accessed via unique keys (see below). In addition, each array item is given a unique name that will also be used as the display text for the resulting search folder in the favorites toolbar.

By default, the different search parameters of the collection view are divided into the following parts:

- The *main part* (key `_main`), featuring the search parameters `searchText`, `contentType`, `mode`, `view`, `folder`, `orderBy`, and `limit`.

  Note that for the `folder` property, it is possible to use both of the following notations:

  - `folder: {$Ref: "content/9"}` (Rest URI path)
  - `folder: {path: "/Sites/Media"}` (content repository path)
- The *status filter* (key `status`), featuring the search parameters `inProduction`, `editedByMe`, `editedByOthers`, `notEdited`, `approved`, `published` and `deleted`.
- The *last edited filter* (key `lastEdited`), featuring the search parameter `lastEditedBy`.

Further possible parameters may arise due to plugged in additional filters (see Section 9.16.5, "Adding Search Filters" [206]) where each of them makes up its own part of search parameters. In the source code example below, a default search folder is plugged in that shows all content items under the content repository path folder /Sites/Media that were last edited by the user. You can see that the array item is composed of two of the three parts listed above and has been given a name.

```
import AddArrayItemsPlugin
  from
"@coremedia/studio-client.ext.ui-components/plugins/AddArrayItemsPlugin";
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import FavoritesButton
  from
"@coremedia/studio-client.main.editor-components/sdk/desktop/maintoolbar/FavoritesButton";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";

interface MyStudioPluginConfig extends Config<MyStudioPluginBase> {
}

class MyStudioPlugin extends StudioPlugin {

  constructor(config: Config<MyStudioPlugin> = null) {
    super((() => {
      return ConfigUtils.apply(Config(MyStudioPlugin, {

        rules: [
          Config(FavoritesButton, {
            plugins: [
              new AddArrayItemsPlugin({
                arrayProperty: "defaultItems",
                items:
                  [
                    {
                      _main: {
                        contentType: "Document_",
                        folder: {path: "/Sites/Media"},
                        mode: "search",
                        view: "list",
                        limit: 50
                      },
                      lastEdited: {lastEditedBy: "me"},
                      _name: "Last edited"
                    },
                  ],
              }),
            ],
          }),
        ],

      }), config);
    })());
  }
}

export default MyStudioPlugin;
```

Example 9.40. Adding a custom search folder

If in doubt about the actual format for a default search folder entry, you can always customize a search manually in *CoreMedia Studio*, save it and have a look at the user's preferences where they get saved.

9.11.3 Adding a Button with a Custom Action

Sometimes it is necessary to develop a custom action, for example to open a special window or to start a wizard. In Section 5.1.3, "Actions" [37] you will find a more detailed explanation of actions, but the recipe shown here should be enough in many cases.

All actions inherit from `Action`. For example, an action `MyCustomAction` might look like this:

```
import Config from "@jangaroo/runtime/Config";
import Action from "@jangaroo/ext-ts/Action";

interface MyCustomActionConfig extends Config<Action> {
  amount?: number;
}

class MyCustomAction extends Action {
  declare Config: MyCustomActionConfig;

  constructor(config:Config<MyCustomAction>) {
    super(config);
    this.setHandler(this.#handleAction, this);
  }

  #handleAction():void {
    // do something, using `this.initialConfig.amount`
  }
}

export default MyCustomAction;
```

Example 9.41. Creating a custom action

The action can then be used inside a menu item or a button:

```
import Config from "@jangaroo/runtime/Config";
import Button from "@jangaroo/ext-ts/button/Button";
import MyCustomAction from "./MyCustomAction";

//...
Config(Button, {
  baseAction: new MyCustomAction({
    text: "do something",
  }),
})
//...
```

Example 9.42. Using a custom action

For example, such a button with a base action might by added to the Header toolbar or the Actions toolbar as shown in the previous sections.

Note that you can use all parameters inherited from `Action`, like `text` in the example above.

9.11.4 Adding Disapprove Buttons

You can revoke the status of the approved content using the disapprove action. The disapprove action can be enabled in *CoreMedia Studio* so that the disapprove action is part of the actions toolbar, the collection repository context menu and the collection search context menu.

You enable the disapprove action by using the plugin `EnableDisapprovePlugin`. For example by inserting the following code snippet inside `configuration` in your Studio plugin.

```
import Config from "@jangaroo/runtime/Config";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import EnableDisapprovePlugin from
"@coremedia/studio-client.main.editor-components/configuration/EnableDisapprovePlugin";

//...
Config(StudioPlugin, {
  //...
  configuration: [
    //...
    new EnableDisapprovePlugin({}),
    //...
  ]
});
```

Example 9.43. Adding disapprove action using enableDisapprovePlugin

9.12 Managed Actions

Managed actions are used to reuse the same action instance for different components, for example a button and a menu item, and even for a keyboard shortcut. This not only saves action instances, but can be crucial for keeping action state consistent.

Unlike previous examples, a managed action is not added to a button or menu item directly. Instead, a managed action is registered by giving it an `actionId` and adding it to the `actionList` property of a to a container. To add a managed action to an existing container, use a Studio plugin rule and the `AddArray ItemsPlugin` with `arrayProperty="actionList"`. Afterwards buttons that are located somewhere below this container may access the action using an `ActionRef` or execute them via a keyboard shortcut. The following example explains the implementation in detail.

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import AddArrayItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/AddArrayItemsPlugin";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import EditorMainView from
"@coremedia/studio-client.main.editor-components/sdk/desktop/EditorMainView";
import MyGlobalAction from "./MyGlobalAction";

class CustomStudioPlugin extends StudioPlugin {
  static MY_GLOBAL_ACTION_ID:string = "myGlobalAction";

  constructor(config:Config<CustomStudioPlugin> = null){
    super(ConfigUtils.apply(Config(CustomStudioPlugin, {

      rules: [
        Config(EditorMainView, {
          plugins: [
            Config(AddArrayItemsPlugin, {
              arrayProperty: "actionList",
              items: [
                new MyGlobalAction({
                  actionId: CustomStudioPlugin.MY_GLOBAL_ACTION_ID,
                }),
              ],
            }),
          ],
        }),
      ],
    }), config));
  }
}

export default CustomStudioPlugin;
```

The example shows how `MyGlobalAction` is added to the action list of the `EditorMainView` under a unique `actionId`. The action is now available for all child components of this main view container.

In the following, it is shown how a reference to this action is used for a button that is plugged into the `RepositoryToolbar` of the Studio's `Collection View` (library).

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import CollectionView from
"@coremedia/studio-client.main.editor-components/sdk/collectionview/CollectionView";

class CustomStudioPlugin extends StudioPlugin {
  static MY_GLOBAL_ACTION_ID:string = "myGlobalAction";

  constructor(config:Config<CustomStudioPlugin> = null){
    super(ConfigUtils.apply(Config(CustomStudioPlugin, {
      rules: [
        Config(CollectionView, {
          plugins: [
            Config(CollectionViewStudioPlugin),
          ],
        }),
      ],
    }), config));
  }
}

export default CustomStudioPlugin;

//--------------------

import { as } from "@jangaroo/runtime";
import ActionRef from "@jangaroo/ext-ts/ActionRef";
import AddItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/AddItemsPlugin";
import NestedRulesPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/NestedRulesPlugin";
import IconButton from
"@coremedia/studio-client.ext.ui-components/components/IconButton";
import ICollectionView from
"@coremedia/studio-client.main.editor-components/sdk/collectionview/ICollectionView";
import RepositoryToolbar from
"@coremedia/studio-client.main.editor-components/sdk/collectionview/RepositoryToolbar";

interface CollectionViewStudioPluginConfig extends Config<NestedRulesPlugin>
 {
}

class CollectionViewStudioPlugin extends NestedRulesPlugin {
  declare Config: CollectionViewStudioPluginConfig;

  #myCollectionView:ICollectionView = null;

  constructor(config:Config<CollectionViewStudioPlugin> = null){
    super(((()=>{
      this.#myCollectionView = as(config.cmp, ICollectionView);
      return ConfigUtils.apply(Config(CollectionViewStudioPlugin, {

        rules: [
          Config(RepositoryToolbar, {
```

```
            plugins: [
              Config(AddItemsPlugin, {
                items: [
                  Config(IconButton, {
                    baseAction: Config(ActionRef, {actionId:
CustomStudioPlugin.MY_GLOBAL_ACTION_ID}),
                  }),
                ],
              }),
            ],
          }),
        ],
      }), config);
    })());
  }
}

export default CollectionViewStudioPlugin;
```

For any action with an `actionId`, a keyboard shortcut can be defined, which
is described in the next section.

9.13 Adding Shortcuts

Once an action is registered in the `actionList` of a container, a shortcut can easily be applied to it via the `AddKeyboardShortcut`. Continuing from the example code of the previous section, this looks like follows.

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import AddKeyboardShortcut from
"@coremedia/studio-client.main.editor-components/sdk/shortcuts/AddKeyboardShortcut";
import Shortcut_properties from
"@coremedia/studio-client.ext.frame-components/shortcuts/Shortcut_properties";

class CustomStudioPlugin extends StudioPlugin {
  static MY_GLOBAL_ACTION_ID:string = "myGlobalAction";

  constructor(config:Config<CustomStudioPlugin> = null){
    super(ConfigUtils.apply(Config(CustomStudioPlugin, {
      configuration: [
        new AddKeyboardShortcut({
          shortcut: Shortcut_properties.Shortcut_my_key,
          description: Shortcut_properties.Shortcut_my_description,
          actionId: CustomStudioPlugin.MY_GLOBAL_ACTION_ID,
        }),
      ],
    }), config));
  }
}

export default CustomStudioPlugin;
```

The example shows how a shortcut is registered for `MyGlobalAction` that is already registered.

Shortcuts are defined inside the properties file `Shortcut_properties.ts`. For customizing existing shortcuts, a properties file has to be created that overrides the `Shortcut_properties.ts` file via the `CopyResource BundleProperties` class.

```
import resourceManager from "@jangaroo/runtime/l10n/resourceManager";
import CopyResourceBundleProperties from
"@coremedia/studio-client.main.editor-components/configuration/CopyResourceBundleProperties";
import Shortcut_properties from
"@coremedia/studio-client.ext.frame-components/shortcuts/Shortcut_properties";
import MyCustomShortcuts_properties from "./MyCustomShortcuts_properties";

//...under the 'configuration' property of a StudioPlugin:
new CopyResourceBundleProperties({
  destination: resourceManager.getResourceBundle(null, Shortcut_properties),

  source: resourceManager.getResourceBundle(null,
MyCustomShortcuts_properties),
})
```

To ensure that the documentation for newly created shortcuts is generated automatically and shown in the Studio preferences dialog, the key values inside the properties file must match the following format:

```
Shortcut_<SHORTCUT_NAME>_key: "<KEY_DEFINITION>",
Shortcut_<SHORTCUT_NAME>_description: "<SHORTCUT_DESCRIPTION>",
```

9.14 Inheritance of Property Values

The *CAE* sometimes renders fallbacks if a content property is not set, for example, by using a value of another property instead. To visualize this in *Studio*, you may use content of a property editor from another property editor as the default empty text.

This is currently possible for a few property fields. One is the `StringProper tyField` and the other one is the `TextAreaPropertyField`. While the `StringPropertyField` may inherit its content from another `StringProp ertyField`, the `TextAreaPropertyField` may inherit its content from a `StringPropertyField` or a `RichTextPropertyField`.

In order to use this visualization, you may use the `StringPropertyField DelegationPlugin` or the `TextAreaPropertyFieldDelegationPlu gin` attached to the property field that should inherit the value.

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import PropertyFieldGroup from
"@coremedia/studio-client.main.editor-components/sdk/premular/PropertyFieldGroup";
import StringPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/StringPropertyField";
import StringPropertyFieldDelegatePlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/plugins/StringPropertyFieldDelegatePlugin";
import TextAreaPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/TextAreaPropertyField";
import TextAreaPropertyFieldDelegatePlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/plugins/TextAreaPropertyFieldDelegatePlugin";

//...
Config(PropertyFieldGroup, {
  title: "...",
  items: [
    Config(StringPropertyField, {
      itemId: "teaserTitle",
      propertyName: "teaserTitle",
      ...ConfigUtils.append({
        plugins: [
          Config(StringPropertyFieldDelegatePlugin, { delegatePropertyName:
 "title" }),
        ],
      }),
    }),
    Config(TextAreaPropertyField, {
      propertyName: "teaserText",
      ...ConfigUtils.append({
        plugins: [
         Config(TextAreaPropertyFieldDelegatePlugin, { delegatePropertyName:
 "detailText" }),
        ],
      }),
    }),
  ]
})
```

Example 9.44. Configuring Property Inheritance

9.15 HTML5 Drag And Drop

Since *CoreMedia 11*, *Studio* supports HTML5 drag and drop. The main reason is to allow drag/drop operations between the *Main App* and the *Workflow App* which run in different browser windows or tabs.

The problem is that most of the *Studio's* drag/drop operations rely on the *Ext JS* framework (`DragSource` and `DropTarget` and sub-classes) which does not support HTML5 drag/drop. So an adapter was introduced by *CoreMedia* to mediate between Ext JS drag/drop and HTML5 drag/drop. To enable this adapter, a new configuration property `enableHtml5DD` was added to Ext JS' `Drag Source` and `DropTarget`.

Most of the *Studio*'s built-in drag sources and drop targets are now enabled for HTML5 drag/drop. For any custom sources and targets that should participate in HTML5 drag/drop, `enableHtml5DD` needs to be set. For many cases, just setting the property suffices. However, there are some potential problems to consider:

- The *Studio's* built-in drag sources and drop targets typically work with drag data that carries *CoreMedia* data objects like content items, products, categories or projects. This automatically works with HMTL5 drag/drop enabled. The prerequisite is that they are stored in the drag data's `contents` or `records` properties.
- If for a custom drag/drop operation other drag data is needed, you need to make sure that it is stored under the `additionalData` property of the drag data. Furthermore, this additional data must be JSON-serializable. Consequently, it is for example not possible to include Ext JS components (something that is sometimes done for Ext JS drag/drop classes). Instead, you could just include the component's `id` and use it to obtain the component in the drop handler of the drop target. However, if you encode app specific data like component ids make sure that your code is robust in a way the ids is only being interpreted from the same app.

> **CAUTION**
>
> The following explanations need to be taken with caution. Support for custom non-Ext JS drag sources and drop targets is still limited and experimental.

Using HTML5 drag/drop now allows using custom non-Ext JS drop targets that still receive drag data from *Studio's* "traditional" Ext JS drag sources. So it is even possible to drag from a *Studio* app into a custom non-Ext JS app. In the drop

handler of the drop zone, you can obtain the dragged *CoreMedia* objects and the additional data from the `DragEvent` by:

```
dragEvent.dataTransfer.getData("cm/uri-list")
// e.g. => '[{"$Ref":"content/720"},{"$Ref":"content/738"}]'

dragEvent.dataTransfer.getData("cm/additional")
// e.g. => '{"dragItem":"com-coremedia-cms-editor-sdk-tabProxy-26",\n
// "draggedTabStripEl":"ext-comp-2877"}'
```

Example 9.45. Obtaining The Dragged Objects from the DragEvent

A more sophisticated option opens up if your custom drop target runs inside an app that is connected to the *Studio* apps via the `serviceAgent` from the `@coremedia/service-agent` npm package (both apps need to run in the same context - host and port). The following code can be executed on *"dragover"* as well as on *"drop"*.

```
import { serviceAgent } from "@coremedia/service-agent";
import { DragDropService, DragDropServiceDescriptor } from
"@coremedia/studio-client.interaction-services-api";
import { as } from "@jangaroo/runtime";

const dragDropServiceDescriptor = new DragDropServiceDescriptor();
const service: DragDropService = serviceAgent.getService(dragDropServiceDescriptor);

// e.g. => '["ContentDD"]'
const dragGroups: string[] = as(JSON.parse(service?.dragGroups || null), Array) || [];

// e.g. => '{"content":[],"contents":[{"$Ref":"content/7120"},{"$Ref":"content/7328"}],\n
//   "additionalData":{"sourceViewId":"tableview-1479","viewId":"tableview-1479","copy":true}'
const dragData: Record<string, any> = as(JSON.parse(service?.dragData || null), Object) || {};
```

Example 9.46. Obtaining Drag Info Via the Service Agent

9.16 Customizing the Library Window

You can configure the library window in the following ways:

- by defining the columns that are displayed in the list view in the repository mode;
- by defining additional fields for the columns that should be displayed in the list views;
- by defining the columns that are displayed in the list view in the search mode and configuring the columns so that the results in the search mode can be sorted;
- by defining the blob properties that are displayed in the thumbnail view for different content types;
- by adding custom filters for the search mode of the library window.
- by making columns sortable and provide a detailed configuration how to sort.

If you are interested in opening the library from a toolbar button, see Section 9.11, "Customizing Central Toolbars" [189].

9.16.1 Defining List View Columns in Repository Mode

The list view of the library window is implemented using an Ext JS grid panel. A grid panel aggregates columns that refer to fields of an underlying store. For adding a new column, you usually have to add both a column definition and a field definition.

To define columns specify a `ConfigureListViewPlugin` and use it in a `StudioPlugin`. In the *CoreMedia Blueprint* the class `ConfigureCollectionViewColumnsPlugin` specifies a `ConfigureListViewPlugin` with column definitions you have to edit if additional columns are needed. `ConfigureCollectionViewColumnsPlugin` is then used in `BlueprintFormsStudioPlugin`. If you do not use the *CoreMedia Blueprint* take the class `ConfigureCollectionViewColumnsPlugin` as an example.

The property `repositoryListViewColumns` in `ConfigureListViewPlugin` lists **all** columns that should be displayed (not just the ones you want to add to the default) in the repository mode. Some columns in this example use predefined components from the Editor SDK, whereas some special columns use just a configured Ext JS standard grid column.

The ListViewTypeColumn, ListViewNameColumn, ListViewStatusColumn ListViewCreationDateColumn, and FreshnessColumn columns represent the standard columns that would be present without additional configuration (id and width of the column has to be defined if necessary), displaying a content item's type, name, lifecycle status, date of creation, and modification date, respectively. These columns can be made sortable by setting the attribute sortable to true. To enable sorting for other columns have a look at Section 9.16.6, "Make Columns Sortable in Search and Repository View" [209].

9.16.2 Defining Additional Data Fields for List Views

If you need additional fields in the underlying store, you can add fields using the listViewDataFields property of the ConfigureListViewPlugin. The standard columns do not need an explicit field configuration. But if, for example, you want to display the name of the user who created a content, the implementation would look like this:

```
import Config from "@jangaroo/runtime/Config";
import GridColumn from "@jangaroo/ext-ts/grid/column/Column";
import DataField from
"@coremedia/studio-client.ext.ui-components/store/DataField";
import ConfigureListViewPlugin from
"@coremedia/studio-client.main.editor-components/sdk/plugins/ConfigureListViewPlugin";

//...
Config(ConfigureListViewPlugin, {
  instanceName: "myListConfiguration",
  listViewDataFields: [
    Config(DataField, {
      name: "creator",
      mapping: "creator.name",
    }),
  ],

  repositoryListViewColumns: [
    Config(GridColumn, {
      width: 75,
      dataIndex: "creator",
      header: "Creator",
    }),
  ],
})
```

Example 9.47. Defining list view fields

In this case, an Ext JS gridcolumn is used for display, setting the column's attributes as needed. The definition of the field is slightly complex, because the property name of the property creator of each content in the search result should be accessed. To this end, a non-trivial mapping property will be added, but the name attribute of the data field and the dataIndex attribute of the

column will be kept simple and in sync. If the mapping property were identical to the name property of the field, it could have been omitted.

9.16.3 Defining List View Columns in Search Mode

The columns in the search mode are similarly configured but instead the property `searchListViewColumns` is used to list all columns of the search list. *CoreMedia Blueprint* defines custom columns of the search mode again in the file `ConfigureCollectionViewColumnsPlugin`:

> **WARNING**
>
> If you define columns by your own, make sure that the `FreshnessColumn` is configured because this column will be used as the default sort column. Otherwise, the Studio user will get this error message on the console:
>
> ```
> Invalid Saved Search Folder: Can not sort by sortfield
> freshness. It will be sorted by 'Last Modified' instead.
> ```

The freshness column is sortable but hidden. It means that the column will not be shown in the search list by default although freshness is used as the default sort criterion. Hidden columns can be unhidden by the user via the grid header menu.

The `ListViewNameColumn`, `ListViewCreationDateColumn` and `FreshnessColumn` columns are standard columns that can be configured to be sortable without additional configuration. To enable sorting for other columns have a look at Section 9.16.6, "Make Columns Sortable in Search and Repository View" [209].

9.16.4 Configuring the Thumbnail View

The thumbnail view of the library window can show a preview image of content items with a blob property holding the image data. If you want to do so, you need to register your content type and configure the name of the blob property you want the thumbnail preview to be generated from. One option is to use the `registerImageDocumentType` method of the `@coremedia/studio-client.cap-base-models/content/contentTypeContext`. You can also use the standard plugin `ConfigureDocumentTypes`, setting the `imageProperty` as shown below.

```
import ConfigureDocumentTypes from
"@coremedia/studio-client.main.editor-components/configuration/ConfigureDocumentTypes";

//...
new ConfigureDocumentTypes({
  names: "CMMedia,CMImage",
  imageProperty: "data",
  richTextImageBlobProperty: "data",
})
```

Example 9.48. Configuring the thumbnail view

The configured property applies to exactly the given content types, only. It is not inherited by subtypes.

9.16.5 Adding Search Filters

The search mode of the library offers a filter panel at the left side of the window in which you can for example select the editing state of content items to be included in the search result. Depending on your editorial needs, you can add custom search filters that further restrict the search result. For example, you might want to search only for recently edited content items or for content items in a particular language. A custom search filter is added to the library in three steps:

- Create a custom search filter component.
- Add the custom search filter component to the list of existing search filters. Additionally add filter state objects.
- Enable the new custom search filter in the `editorContext`.

9.16.5.1 Create a Custom Search Filter

For defining a custom filter, you can inherit from the class `FilterPanel`. This class implements the interface `SearchFilter` and provides the framework for implementing a custom filter easily. The state of a search filter is stored in a model bean provided by the method `getStateBean()` and is persisted in the preferences struct. See section Section 9.10, "Storing Preferences" [188] for details

Inheriting from Filter-Panel

In your `SearchFilter` class, you need to override two methods. The method `buildQuery()` can use the current state stored in the model bean to assemble a Solr query string. Query strings from individual filters will be combined using the `AND` operator. By returning an empty string or `null`, you can indicate that the filter should not currently impose any restrictions on the search result. The following example shows how a property `foo` is retrieved and how a query is built from it.

```
import FilterPanel from
"@coremedia/studio-client.main.editor-components/sdk/collectionview/search/FilterPanel";

class FooFilterPanelBase extends FilterPanel {

  //...

  override buildQuery(): string {
    const foo: Number = this.getStateBean().get("foo");
    if (foo === 0) {
      return null;
    }
    return "foo:" + foo + " OR foo:-1";
  }

  //...
}

export default FooFilterPanelBase;
```

The method `getDefaultState()` returns an object mapping all properties of the state bean to their defaults. It is used for initialization, for determining whether the current state of your UI represents the filter's default state, and for manually resetting the filter. In the above example, the respective filter's default state is represented by the special value "0", and consequently, you must use "0" as the filter's default value:

```
class FooFilterPanelBase /*...*/ {
  //...
  override getDefaultState():any {
    return { foo:0 };
  }
}
```

Because the `itemId` of the filter component is used when identifying the filter later on, it often makes sense to specify the `itemId` directly in the `Search Filter` subclass.

To synchronize your UI component(s) with the model state stored in the bean returned by `getStateBean()`, you might want to use the various existing bind plugins.

Synchronizing UI with model state

9.16.5.2 Add Custom Search Filter to Search Filter List

Use the `AddItemsPlugin` to add your custom filter to the Studio Library filter section. The component to configure is the `SearchFilters` class if a filter should be added for the content search.

Adding FooFilterPanel to the filter list

```
import Config from "@jangaroo/runtime/Config";
import Component from "@jangaroo/ext-ts/Component";
import SearchFilters from
"@coremedia/studio-client.main.editor-components/sdk/collectionview/search/SearchFilters";
import AddItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/AddItemsPlugin";
```

```
import FooFilterPanel from "./FooFilterPanel";

Config(SearchFilters, {
  plugins: [
    Config(AddItemsPlugin, {
      items: [
        Config(FooFilterPanel, {
          itemId: "fooFilter",
        }),
      ],
      after: [
        Config(Component, { itemId: SearchFilters.LAST_EDITED_FILTER_ITEM_ID
 }),
      ],
    }),
  ],
})
```

You can also open the library in a certain filter state, for example from a button in the favorites toolbar. To that end, the `ShowCollectionViewAction` provides a property `filters` that can take `SearchFilterState` objects. So that the action can configure the correct filter, the `filterId` attribute must be given, matching the `itemId` of the configured filter panel. The names and values of the attributes are exactly the property names and values of the state bean used by the filter set.

Opening the Library in certain filter state

9.16.5.3 Disable Default Search Filter

To disable a default search filter, you will have to remove it from the list of filters from the corresponding filters panel. The given example shows how to remove the status filter from the Content search filter list.

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import RemoveItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/RemoveItemsPlugin";
import SearchFilters from
"@coremedia/studio-client.main.editor-components/sdk/collectionview/search/SearchFilters";
import StatusFilterPanel from
"@coremedia/studio-client.main.editor-components/sdk/collectionview/search/StatusFilterPanel";

Config(SearchFilters, {
  ...ConfigUtils.append({
    plugins: [
      Config(RemoveItemsPlugin, {
        items: [
          Config(StatusFilterPanel, {
            itemId: $STATUS_FILTER_ID
          }),
        ],
      }),
    ],
  }),
```

```
})
```

9.16.5.4 Customize Search Filter for Issue Categories

The filter panel consists of a filter named `Issues`. You can customize the Category under Issues to search for content items efficiently. By default, two categories are present, "All Categories" and "Localization". You can add more issue categories in the `BlueprintIssueCategories` properties file. Those categories will then be available in the category dropdown in Studio. Note however, that issue categories must contain only alphabets and not consist of any special characters.

If you are adding more issue categories, you will have to configure the validators accordingly. For more details on configuration of validators, you can refer Section 9.23.1.3, "Custom Validators" [257]

9.16.6 Make Columns Sortable in Search and Repository View

Sorting can be enabled for custom columns by setting two mandatory attributes in the gridcolumn definition. The attribute `sortable` has to be set to `true` to enable sorting. The attribute `sortField` has to specify the Solr index column that should be used for sorting.

```
import Config from "@jangaroo/runtime/Config";
import Column from "@jangaroo/ext-ts/grid/column/Column";

//...
Config(Column, {
  sortable: true,
  dataIndex: "creator",
  header: "Creator",
  sortField: "creator",
})
```

Example 9.49. Two additional attributes for sorting.

The optional field `sortDirection` enables you to restrict the sort direction to only one direction. This is useful if sorting does only make sense in one direction. For example a user is usually not interested in the less relevant search result. So you want to disable sorting for relevance ascending. Possible values are "asc" or "desc" where the value is the enabled sort direction.

```
import Config from "@jangaroo/runtime/Config";
import Column from "@jangaroo/ext-ts/grid/column/Column";

//...
Config(Column, {
  sortable: true,
  dataIndex: "creator",
  header: "Creator",
  sortField: "creator",
  sortDirection: "desc",
})
```

Example 9.50. Optional `sortDirection` Attribute to enable only one sort direction.

You can make even hidden grid columns sortable. Hidden columns are not shown in the grid but users can select them from the sort drop down field. This is useful if columns do not have meaningful values (again relevance for example) or if you just do not want to blow up the grid too much. Hidden columns that do not have their `hideable` config option set to `false` can also be unhidden by the user using the grid header menu.

At last you can define one default sort column for each list in the collection view. The default sort column will be used when the user has not specified a sort criteria. To configure add the attribute `defaultSortColumn` with value `true`. For more fine grained configuration the attribute `defaultSortDirection` can be set to `asc` or `desc` to sort ascending or descending by default.

```
import Config from "@jangaroo/runtime/Config";
import Column from "@jangaroo/ext-ts/grid/column/Column";

//...
Config(Column, {
  sortable: true,
  dataIndex: "creator",
  header: "Creator",
  sortField: "creator",
  defaultSortColumn: true,
  defaultSortDirection: "desc",
})
```

Example 9.51. `defaultSortColumn` Attribute to configure one column as the default for sorting.

9.17 Studio Frontend Development

Frontend development in *CoreMedia Studio* is based on the Ext JS frontend API and makes applying styles in *CoreMedia Studio* easy. By using reusable and modular skins, passed to the `ui` configuration, the appearance of components can be changed. This way, styles can be created without knowledge of the concrete implementation of every component.

This chapter describes how *CoreMedia Studio* uses the Ext JS framework and how the basic concepts are extended to the needs of a complex software.

- Section 9.17.1, "Blueprint Studio Theme" [211] describes where you can put styles, resources and component skins to extend the *Studio Theme*.
- Section 9.17.2, "Studio Styling with Skins" [214] describes how *CoreMedia Studio* uses and extends the Ext JS theming concept, how existing skins can be switched off, changed or how new skins can be created.
- Section 9.17.3, "Styling of Custom Studio Components" [218] describes how to apply custom styles to components, where using Skins would not make sense.
- Section 9.17.4, "CoreMedia Svg Icons" [219] describes how to use icon fonts in *CoreMedia Studio*, how to apply icons to components and what needs to be done to display different other icons or images in *CoreMedia Studio*.
- Section 9.17.7, "Usage of BEM and Spacing Plugins" [222] describes how *CoreMedia Studio* uses BEM and which Plugins and Mixins exist to make the usage of BEM easier.
- Section 9.17.8, "Component States" [225] gives an overview about how to add additional states to components in *CoreMedia Studio*.

9.17.1 Blueprint Studio Theme

In order to use the Ext JS frontend framework properly, you will need to use the folder structure, as described by Sencha in the Theming Section of their API documentation. *CoreMedia* provides a blank package named *Blueprint Studio Theme* (or more precisely `@coremedia-blueprint/studio-client.main.blueprint-studio-theme`) which extends the Studio theme and should be used to define own skins and customize existing ones. Skins are written in a particular SASS dialect, compiled by Sencha's *Fashion* compiler, which offers most SCSS functionalities enriched by certain Sencha specific functions.
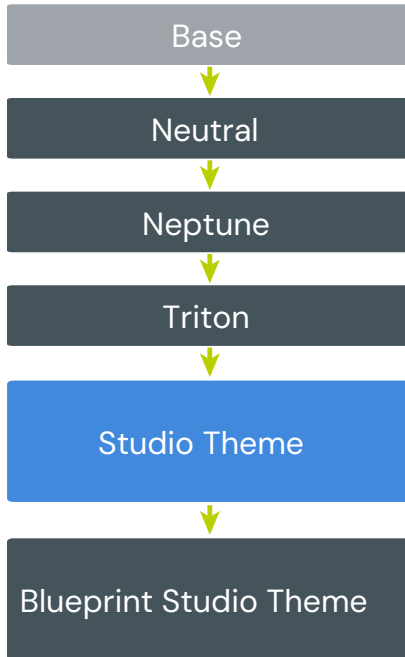
Figure 9.6. Theming Inheritance in Ext JS and CoreMedia

The *Studio Theme* is responsible for the default appearance of *CoreMedia Studio*. It extends the Triton Theme, provided by the Ext JS framework and offers certain variables and SCSS mixins. You can easily disable style rules, defined in the *Studio Theme* by setting the include variables for skins or custom components to false.

> **NOTE**
>
> The *Studio Theme* introduces many variables that are also used outside the theme in SCSS files for custom styling of particular components. CoreMedia recommends not changing the theming inheritance and extend from another theme than the *Studio Theme*, because those variables would not be initialized anymore. Also, all skins, introduced by the *Studio Theme*, would not be available, since the corresponding style rules would not be created anymore. The better way would be to still inherit from the *Studio Theme*, but disable undesired styles by setting the include variables to false.

In every package, there are different folders for SCSS files:

- `sencha/sass/etc` – contains utility functions or mixins
- `sencha/sass/src` – contains rules and UI mixin calls
- `sencha/sass/var` – contains global and private variables

The *Studio Theme* slightly differs from this structure by introducing a forth folder:

- `sencha/sass/mixins` – additional CoreMedia SCSS mixins

These additional mixins enhance the Ext JS framework and broaden the possibilities to style certain components. They are introduced in the *Studio Theme* and work as extensions of the Sencha SCSS mixins, as explained in Section 9.17.2, "Studio Styling with Skins" [214]. You can – and should – use them when creating new skins in the *Blueprint Studio Theme*. The include order of SCSS files from different folders is described in Organization of Generated Styles in the Sencha Ext JS API documentation.

The directory, in which the build tooling searches for SCSS files in the CoreMedia Workspace slightly differs from the path described in the Sencha API documentation. While the documentation states that the `sass` folder is in the root of the package our tooling requires using the `sencha/sass` folder.

Be aware that by default the sass namespace is generated. More precisely: if unset it will be the same as the normal namespace which – if unset – will be generated based on the npm package name. If you want to style components that are not part of your package (including the Ext JS base components), follow the advice at The Sass Namespace by setting the configuration in the package's `jangaroo.config.js` in the corresponding `sencha` entry, for example:

```
{
  ...
  sencha: {
    sass: {
      namespace: "",
    }
  }
}
```

Example 9.52. Sass namespace

If you also want to style the components contained in your own package you should explicitly define a namespace for your package:

```
{
  ...
  sencha: {
    namespace: "my.namespace",
    sass: {
      namespace: "",
```

```
   }
 }
```

Example 9.53. namespace + Sass namespace (only needed for parallel styling of own components and components of other packages)

In this example you would then put the styling of your own component `MyButton` located in `src/buttons/MyButton` in the subfolder `my/namespace/buttons/MyButton.scss` while styling, for example, the Ext JS Button components in `Ext/button/Button.scss`.

While the namespace of any third-party package can be found in the `sencha.namespace` entry of its `package.json` file (for example, `node_modules/@jangaroo/ext-ts/package.json`) for packages which are part of your workspace you need to check the `dist/package.json` instead. The `dist` folder is only available after building the package.

After adding or changing files in the *Blueprint Studio Theme*, you will need to rebuild the package and all apps using the theme. CoreMedia suggests using the `start` script (see Chapter 7, *Developing with the Studio Client Workspace* [93]) which will automatically rebuild the CSS of all apps in the dependency hierarchy when triggered. To just watch and rebuild the SCSS for all apps use the following command:

```
cd global/studio
pnpm run start
```

9.17.2 Studio Styling with Skins

Since *CoreMedia Studio* is based on the Sencha Ext JS framework, it uses and extends the provided skin concept. Styling rules are encapsulated in SCSS mixins and can be applied by using the `ui` configuration. There are SCSS mixins for almost every component and *CoreMedia Studio* also provides a huge set of skins, which create the visual appearance of said components.

If a component does not support the usage of skins, or the skin concept does not satisfy the requirements for certain situations you can learn about custom styles in Section 9.17.3, "Styling of Custom Studio Components" [218].

Please bear in mind that it is not always necessary to write a new skin if you want to change the appearance of a certain component. To change styles, you have multiple options:

• Change global styles by setting theme variables
• Change a skin by setting global CoreMedia variables
• Write a new skin and change the `ui` configuration of the component

Please make sure to read the Theming Section of the Ext JS API documentation to understand the core concepts of the theming system.

Change global styles by setting theme variables

To change the appearance of components Ext JS provides theme variables. If you want to change the style rules of a component, it can often be sufficient to simply override these variables in the *Blueprint Studio Theme*. Please keep in mind, that you will affect every skin of a component type by changing theme variables. Mixins use theme variables as default if a parameter is not set explicitly.

The following example shows how to set theme variables for panels. Please take a look at the Sencha Ext JS API documentation to get a list of available theme variables.

```
...
$panel-header-color: dynamic($cm-font-color);
$panel-header-padding: dynamic($cm-grid-100);
$panel-body-background-color: dynamic(transparent);
$panel-body-border-width: dynamic(0);
...
```

Example 9.54. Overriding theme variables

By assigning a SCSS variable with `dynamic(...)` you make sure that the new value is applied even in earlier usages of this variable. Please read the Dynamic Variables Section to learn more. Since the *Blueprint Studio Theme* is the last theme to be loaded, the value will not be overridden by another theme if you put the assignment in the theme's `sencha/sass/var/Ext/` folder.

Change a skin by setting global CoreMedia variables

CoreMedia Studio also provides own theme variables, which are used as default parameters in `cm-[component]-ui` mixins or provide a possibility to change styles of custom components. These CoreMedia variables begin with a `$cm-` prefix:

```
...
$cm-panel-show-validation: dynamic(true);
$cm-panel-box-shadow: dynamic($cm-elevation-box-shadow-100);
$cm-panel-ghost-background-color: dynamic($cm-grey-1);
$cm-panel-use-sub-collapsible-separator: dynamic(false);
...
```

Example 9.55. Overriding global CoreMedia variables

> **CAUTION**
>
> To prevent unpredictable component styling, it is not allowed to use the prefix `$_cm-` in your own variables, since it is reserved for private variables in the *Studio Theme*. Overriding these variables can lead to unwanted behavior and incorrect style rules for skins.

Write a new skin and change the `ui` configuration of the component

The *Studio Theme* creates styles by including SCSS mixins:

```
@if $cm-include-panel-accordion-ui {
  $_ui: "accordion";

  @include extjs-panel-ui(
    $ui: $_ui,
    $ui-header-color: $cm-font-color,
    ...
  );

  @include cm-panel-ui(
    $ui: $_ui,
    $background-color: $cm-white,
    ...
  );
}
```

Example 9.56. Simple Skin Example

As shown in Example 9.56, "Simple Skin Example" [216], the *Studio Theme* always includes two SCSS mixins per skin. In addition to the Ext JS mixin, *Studio Theme* provides own mixins, which extend the Ext JS framework. These mixins provide helpful functionality and enhancements, which are applied, even if only the ui parameter is passed to the mixin's parameter list (such as default styles for validation). Therefore, it is necessary to always include both mixins.

Please take a look at the Ext JS – Classic Toolkit API to get a list of theme mixins and possible parameters.

Please note that the *Studio Theme* wraps mixin includes in `if`-statements. You can easily switch off mixin includes by setting the corresponding `$cm-include-[COMPONENT-TYPE]-[SKIN-NAME]-ui` variables to false. Please keep in mind that switching a skin off, will remove all styles for components using the skin. The components will therefore be not styled. If the skin is still set in the `ui configuration`, not even the default styles will be applied.

```
// Switching off skin "accordion"
$cm-include-panel-accordion-ui: dynamic(false);
```

Example 9.57. Switching off skins

A skin should be switched off if you want to write an own skin or the skin is simply not used anymore. After switching it off, you can include the SCSS mixins in the *Blueprint Studio Theme* with the same `ui` parameter to create the style rules for your own skin.

CoreMedia Studio uses TypeScript classes to group skins for components. These classes contain constants for each skin, which provide a stable interface to use skins as `ui` configuration in components. It is recommended using this concept when applying skins to components. Otherwise, it can get very difficult to tell which skins are currently used in *CoreMedia Studio*.

```
import PanelSkin from
"@coremedia/studio-client.ext.ui-components/skins/PanelSkin";

class MyClass {
  static readonly DEFAULT: PanelSkin = new PanelSkin("default");
  static readonly DOCKED: PanelSkin = new PanelSkin("docked");
  static readonly ACCORDION: PanelSkin = new PanelSkin("accordion");
  static readonly CARD: PanelSkin = new PanelSkin("card");
  //...
}
```

Example 9.58. TypeScript Skin Constants

To apply a skin to a component, you just have to pass it to the `ui` configuration. If no `ui` configuration is applied, the used skin will be "default". The following example shows how to apply the toolbar skin to a button:

```
Config(Button, {
  itemId: "myToolbarButton",
  ui: ButtonSkin.TOOLBAR.getSkin(),
})
```

Example 9.59. Applying a Skin to a Component

Skins of the same component category are exchangeable without any other adjustments. If no skin is applied, the default skin will be used instead. Some containers can override this behavior. For example, a toolbar has the configuration `defaultButtonUI` (see Button documentation).

9.17.3 Styling of Custom Studio Components

It is important to understand, that your skins are a reusable set of styles and should be applied to components whenever possible. This not only keeps maintenance easy, but also keeps your layout simple and clear. Nevertheless, there can be different reasons why you would want to write custom styles in addition to existing skins:

• The component does not support skins
• You are using a custom template inside a component

CoreMedia recommends placing your files in the same package in which the component is located. To do this, create a `sencha/sass/` folder in the package's root folder. As long as the folder structure in your directories for TypeScript files and SCSS files match, the Sencha Microloader will find the SCSS files corresponding to the Ext JS components (as long a no custom namespace configuration is set in `jangaroo.config.js`).

> **NOTE**
>
> You can write styles in any SCSS file that will be found by the Sencha Microloader. However, it is possible that styles and variables can be overridden in other SCSS files, due to the order these files get loaded. As a rule of thumb you can assume, that all SCSS files in `sencha/var` folders will be loaded prior files in `sencha/src` folders. Take a look at the Sencha Documentation to learn more about Organization of Generated Styles.

Own custom styles should be an exception and only be used if writing a new skin is not an option. While skin mixins provide a robust way to apply styles you can never be sure if your own CSS selectors will work after updating the framework or changing the layout of the parent container. You should especially avoid applying styles to the following CSS classes:

• `x-box-target`
• `x-box-item`
• `x-form-item`
• `x-autocontainer-innerCt`
• `x-autocontainer-outerCt`
• icon classes, such as `cm-generated-icons`
• all classes containing a scale or ui (such as x-btn-default-small)

The recommended way to apply styles to custom components and keep your CSS robust is to add own classes by using the `cls` configuration or the BEM Plugin (see Section 9.17.7, "Usage of BEM and Spacing Plugins" [222]).

CoreMedia Studio defines own styles for custom components. If not needed, you can always disable these styles by setting the corresponding include variable to false.

> **NOTE**
>
> SCSS files for custom components will be included after SCSS files in themes. This makes it impossible to override a custom component variable in the *Blueprint Studio Theme*. If you want to disable custom style rules, you will have to override the variable after the custom styles get included.

9.17.4 CoreMedia Svg Icons

CoreMedia provides a complete set of SVG icons in the included `@coremedia/studio-client.common-icons` package. If the provided set of icons meets your requirements, you can make use of the icons by accessing them through the default export of the package.

The exports yield a string result with the contents of the SVG file. You can then pass the content to any function or render it directly into the DOM. The package `@coremedia/studio-client.base-models` provides a utility class `SvgIconUtil` which offers a function that generates an icon class out of a given SVG code so that it can be used inside ExtJS components:

```
import { search } from "@coremedia/studio-client.common-icons";

Config(Button, {
  iconCls: SvgIconUtil.getIconStyleClassForSvgIcon(search),
});
```

Example 9.60. Accessing CoreMedia Svg Icons

9.17.5 Custom SVG Icons

You can import your own SVG files directly in your TypeScript code:

```
import myIcon from "./icons/my-icon.svg";
console.log(`SVG code:\n${typeArticle}`);
```

Example 9.61. Importing SVG in TypeScript

Importing directly from an `*.svg` file requires the SVG to be located underneath the `src` folder and in addition to this a type definition file `custom.d.ts`:

```
declare module "*.svg" {
  const content: string;
  export default content;
}
```

Example 9.62. SVG definition

You can then use the icon in your ExtJS component by generating a CSS class for the SVG icon with the aforementioned `SvgIconUtil` utility class:

```
import { SvgIconUtil } from "@coremedia/studio-client.base-models";

Config(Button, {
  iconCls: SvgIconUtil.getIconStyleClassForSvgIcon(myIcon),
});
```

Example 9.63. Generating CSS class for SVG icon

Section 9.3, "Studio Plugins" [130] describes how to load external resources. You can also reference SVG icons in your SCSS code as follows:

```
.my-icon {
  background-image: url(get-resource-path("images/example.svg"));
}
```

Example 9.64. Get Resources in SCSS Code

Then pass the CSS class to the `iconCls` configuration of your component:

```
Config(Button, {
  iconCls: "my-icon",
})
```

Example 9.65. Use Image as IconClass

Due to technical limitations of the ExtJS SCSS compiler the SVG file needs to be located in the `sencha/resources` folder.

9.17.6 CoreMedia Icon Font (legacy)

For legacy reasons, *CoreMedia Studio* still includes an icon font that is used throughout the application. The icon font is provided by the `@coremedia/stu-dio-client.core-icons` package and contains a large set of icons that can be used in your own components. The look and feel of these icons does not match the icons in the `@coremedia/studio-client.common-icons` package.

The resource bundle `CoreIcons_properties.ts` of the package contains ready to use css classes which can e.g. be passed to the `iconCls` of your component as shown below:

```
import CoreIcons_properties from
"@coremedia/studio-client.core-icons/CoreIcons_properties";

Config(IconDisplayField, {
  itemId: "helpIcon",
  iconCls: CoreIcons_properties.help,
})
```

Example 9.66. Usage of CoreIcons_properties.ts

The `core-icons` package also defines SCSS variables that can be used to assign icons directly in your SCSS code. The following example sets the add icon for the StatusProxy:

```
$statusproxy-add-glyph: dynamic($cm-core-icons-100-var-plus 16px
$cm-core-icons-100-font-name);
```

Example 9.67. Usage of CoreMedia Icons in SCSS

`$statusproxy-add-glyph` - like any other Ext JS glyph variable - requires you to pass the content, size and font-family as a list of values. The variable in the example above are generated by the `core-icons` package. You can access the content of an icon by using its SCSS variable: `$cm-core-icons-[SCALE]-var-[ICON-NAME]`

There are 3 different scales in the CoreMedia icon font. These scales differ in details, shown in the icon. An icon with small scale is usually displayed in a size of 16px. Therefore, a lot of details have to be cut out, due to the lack of space to display them. The icon would otherwise be displayed blurry. Of course, you

can anyhow always determine the size of an icon for each usage. The following scales are available:

Scale	Size	Identifier	Example
Small	16px	100	`$cm-core-icons-100-var-help`
Medium	24px	200	`$cm-core-icons-200-var-help`
Large	32px	300	`$cm-core-icons-300-var-help`

Table 9.4. Different Icon Scales

> **NOTE**
>
> You don't need to worry about scales if you pass an icon as `iconCls` to a `Ext.button.Button`. If you make proper use of the `scale` configuration, the component will automatically choose the right scale for the icon, based on it.

9.17.7 Usage of BEM and Spacing Plugins

Block Element Modifier (BEM) is a methodology that helps to write neat, reusable CSS classes. Components usually consist of a block and multiple elements inside this block. To apply styles to a component you simply add a CSS class to the block. BEM also requires you to add classes to the elements to make sure styles apply even if the DOM changes. Modifiers can be used to describe a special mutation of a block. Learn more about BEM at https://en.bem.info. A typical BEM pattern looks like this:

```
<div class="bem-block">
  <div class="bem-block__item">...</div>
  <div class="bem-block__item">...</div>
  <div class="bem-block__item">...</div>
</div>

<div class="bem-block bem-block--error">
  <div class="bem-block__item">...</div>
```

```
   <div class="bem-block__item">...</div>
</div>
```

Example 9.68. BEM Example HTML Code

The corresponding SCSS file would look like this:

```
.bem-block {

  color: white;

  &__item {
    margin-bottom: 10px;
  }

  &--error {
    color: red;
  }
}
```

Example 9.69. BEM Example SCSS Code

The easiest way to apply all those CSS classes correctly is to use the `BEMPlugin`. To learn how to use Plugins, see Section 9.3, "Studio Plugins" [130]. You will have to apply the plugin to the container and all items will automatically be provided with the correct CSS classes. The Plugin even takes care of items that are added later on.

```
Config(Container, {
  items: [
    // ...
  ],
  plugins: [
    Config(BemPlugin, {
      block: "bem-block",
      defaultElement: "item",
      modifier: "error",
    })
  ],
})
```

Example 9.70. Usage of the BEM Plugin

BEM-Element classes can be applied by using the `bemElement` configuration provided by `BEMMixin` as long as the one of its parent containers utilizes a corresponding `BEMPlugin`:

```
Config(Container, {
  plugins: [
    Config(BEMPlugin, {
      block: "bem-block",
    }),
  ],
  items: [
    Config(Button, {
      itemId: "my-button",
      ...Config<BEMMixin>({
```

```
        bemElement: "my-button",
      }),
    })
  ]
})
```

Example 9.71. Using BEM Plugin with Element

The previous example adds the `.bem-block__my-button` CSS class to the button component. If the configuration of the `BEMMixin` is the only configuration that should be provided to the Button (for example, if you also want to omit the `itemId` in the previous example) you also need to cast the inner config to `Button` as otherwise the TypeScript compiler assumes that it must be an error:

```
Config(Container, {
  plugins: [
    Config(BEMPlugin, {
      block: "bem-block",
    }),
  ],
  items: [
    Config(Button, {
      ...Config<Button & BEMMixin>({
        bemElement: "my-button",
      }),
    })
  ]
})
```

Example 9.72. Usage of the BEM Mixin

If you want to add space between items of a container you can use the `HorizontalSpacingPlugin` or `VerticalSpacingPlugin`, which internally make use of the `BEMPlugin` and its `default Element` parameter.

```
Config(Container, {
  items: [
    // ...
  ],
  plugins: [
    Config(VerticalSpacingPlugin, {
      modifier: SpacingBEMEntities.VERTICAL_SPACING_MODIFIER_200,
    }),
  ],
})
```

Example 9.73. VerticalSpacing Plugin Example

The default spacing between items in a container that uses `VerticalSpacingPlugin` or `HorizontalSpacingPlugin` is small. By using a modifier, this spacing can be adjusted. You can also enhance the plugin by passing other strings as the modifier parameter, but you will obviously have to write own styles for the resulting CSS classes. Please inspect the constants exposed by the corresponding plugins to find possible modifiers.

9.17.8 Component States

CoreMedia Studio uses state mixins to support different component states, such as validation, read-only, highlighting, overflow behavior or text alignment. To apply a state to a component, simply implement the corresponding interface.

The following list contains state mixin interfaces, provided by *CoreMedia Studio*:

- `IValidationStateMixin`
- `IReadOnlyStateMixin`
- `IHighlightableMixin`
- `IOverflowBehaviourMixin`
- `ITextAlignMixin`

State mixins provide functions to dynamically add or change CSS classes to a component. This is more robust than simply adding a CSS class by passing it to the `cls` configuration. The following example shows how an error state can be added to a button:

```
import Panel from "@jangaroo/ext-ts/panel/Panel";
import Button from "@jangaroo/ext-ts/button/Button";
import ValidationState from
"@coremedia/studio-client.ext.ui-components/mixins/ValidationState";
import ValidationStateMixin from
"@coremedia/studio-client.ext.ui-components/mixins/ValidationStateMixin";

class MyForm extends Panel {

  #newButton: Button & ValidationStateMixin;

  //...

  updateButtonState(): void {
    if (this.#hasErrors()) {
      this.#newButton.validationState = ValidationState.ERROR;
    } else {
      this.#newButton.validationState = null;
    }
  }
}
```

Example 9.74. Set Validation State

The *Studio Theme* provides SCSS mixins that apply styles to CSS classes added by state mixins. Therefore, it is important to always include the *Studio Theme* SCSS mixin besides the Ext JS SCSS mixin. You can change styles by passing certain parameters or by setting global variables.

9.18 Work Area Tabs

CoreMedia Studio organizes working items in a so called work area. The work area is a tab panel with the tabs containing currently opened working items. *CoreMedia Studio* restores open tabs (and their content) after successful relogin or reload of the website. The tabs usually contain CoreMedia-specific content but you can integrate your own customized tab into the work area. This section shows how it can be done using an example code. The example introduces a browse tab which consists of a URL trigger field and an iFrame in which the content of the URL is displayed.

9.18.1 Configuring a Work Area Tab

First you have to configure the tab which should be displayed in the work area. This must be an `Panel` or any extended one. CoreMedia recommends that you configure your tab as a separate class. The rationale for this will be described below. In the example there are two such components: `BrowseTab.ts` and `CoreMediaTab.ts` (where the latter one uses the first one). Both have a configuration parameter `url` which is the key to persisting tab state across sessions and website reloads as explained below in Section 9.18.4, "Storing the State of a Work Area Tab" [228].

9.18.2 Configure an Action to Open a Work Area Tab

In most cases you will use an action to open your own tab. In the example, a button is plugged into the Favorites toolbar. Clicking the button triggers an `OpenTabAction` to open the browse tab.

```
import Config from "@jangaroo/runtime/Config";
import EditorMainNavigationToolbar from
"@coremedia/studio-client.main.editor-components/sdk/desktop/maintoolbar/EditorMainNavigationToolbar.ts"
import AddItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/AddItemsPlugin";
import Button from "@jangaroo/ext-ts/button/Button";
import OpenTabAction from
"@coremedia/studio-client.main.editor-components/sdk/actions/OpenTabAction";
import BrowseTab from "./BrowseTab";

Config(EditorMainNavigationToolbar, {
  plugins: [
    Config(AddItemsPlugin, {
      items: [
        //...
        Config(Button, {
```

```
            itemId: "browseTab",
            baseAction: new OpenTabAction({
              tab: Config(BrowseTab),
            }),
          }),
      ],
      //...
    }),
  ],
})
```

Example 9.75. Adding a button to open a tab

The `BrowseTab` from above is configured as the `tab` configuration parameter of `OpenTabAction`. A new browse tab is then opened every time when clicking the button. In addition, all open browse tabs will be reopened in the work area after the reload of *CoreMedia Studio*. For that *CoreMedia Studio* stores the xtypes of the open tabs as user preference when opening, closing or selecting tabs. When loading the work area instances of the xtypes are generated and added to the work area. This is basically why you should configure each tab in a separate TypeScript class. Nevertheless, you will see below in Section 9.18.4, "Storing the State of a Work Area Tab" [228] how you can save other state of the tab than the xtype in the user preference.

9.18.3 Configure a Singleton Work Area Tab

The previously shown `OpenTabAction` has an additional Boolean configuration parameter `singleton`. In the example a button that opens a `CoreMediaTab` is added, which is a browse tab with the fix URL of the CoreMedia homepage:

```
import Config from "@jangaroo/runtime/Config";
import EditorMainNavigationToolbar from
"@coremedia/studio-client.main.editor-components/sdk/desktop/maintoolbar/EditorMainNavigationToolbar.ts"
import AddItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/AddItemsPlugin";
import Button from "@jangaroo/ext-ts/button/Button";
import OpenTabAction from
"@coremedia/studio-client.main.editor-components/sdk/actions/OpenTabAction";
import CoreMediaTab from "./CoreMediaTab";

Config(EditorMainNavigationToolbar, {
  plugins: [
    Config(AddItemsPlugin, {
      items: [
        //...
        Config(Button, {
          itemId: "coremediaTab",
          baseAction: new OpenTabAction({
            singleton: true,
            text: "...",
            tab: Config(CoreMediaTab),
          }),
        }),
      ],
```

```
        //...
    }),
  ],
})
```

Example 9.76. Adding a button to open a browser tab

In the work area there will be no more than one opened `CoreMediaTab`: When clicking the button the already opened `CoreMediaTab` will be active instead of opening a new one.

9.18.4 Storing the State of a Work Area Tab

You probably want to persist the state of your tabs across sessions and website reloads. As described above, the xtype of all open tabs is stored automatically which allows you to create the correct tab instances when reloading. However, this does not help to persist the content of the tabs. You have to take care of persisting tab state yourself. For example, when the user sets the URL of the browse tab in the example the URL will be restored after reload. Such internal state of the tab can be stored implementing the interface `StateHolder` as `BrowseTabBase` of the example does:

```
import { mixin } from "@jangaroo/runtime";
import Panel from "@jangaroo/ext-ts/panel/Panel";
import StateHolder from
"@coremedia/studio-client.client-core/data/StateHolder";
import ValueExpression from
"@coremedia/studio-client.client-core/data/ValueExpression";
import ValueExpressionFactory from
"@coremedia/studio-client.client-core/data/ValueExpressionFactory";

class BrowseTabBase extends Panel implements StateHolder {

  #url: string;
  #stateValueExpression: ValueExpression;

  //...

  getStateValueExpression(): ValueExpression {
    if (!this.#stateValueExpression) {
      this.#stateValueExpression = ValueExpressionFactory
        .createFromValue({ url: this.#url });
    }
    return this.#stateValueExpression;
  }
}

mixin(BrowseTabBase, StateHolder);

export default BrowseTabBase;
```

Example 9.77. Base class for browser tab

To store the states of the open tabs *CoreMedia Studio* uses `getStateValue Expression` of each tab which implements the interface. See section Section 9.10, "Storing Preferences" [188] for details of how the state is persisted and for the limits on the allowed state structures. You must make sure that proper state is delivered via the state value expression. In `BrowseTabBase` this is achieved in the following way:

```
class BrowseTabBase /* ... */ {
  //...

  #reloadHandler(): void {
    const url = this.getTrigger().getValue();
    this.getBrowseFrame().setUrl(url);
    if (url) {
      this.setTitle(url);
    }
    //store the url as state in the user preference
    this.getStateValueExpression().setValue({ url: url });
  }

  //...
}
```

The `reloadHandler` is invoked when the user clicks on the trigger button. The value of the trigger becomes the URL of the iFrame of the tab. Finally, the state value is set to `{url: url}`: As described above, `url` is a configuration parameter of `BrowseTab` and consequently, `{url:url}` is a configuration object with the parameter `url` with the trigger value. This configuration object will be copied to the configuration object of `BrowseTab` when restoring it. So `BrowseTab`'s configuration parameter `url` is then set to the stored value.

9.18.5 Customizing the Start-up Behavior

After successful login, *Studio* restores the tabs of the last session. This default behavior can be disabled by calling the `setDefaultTabStateManagerEn abled(enable)` method of the singleton `@coremedia/studio-cli ent.main.editor-components/sdk/editorContext`.

When you set this value to `false`, *Studio* will start with a blank working area (that is, no content item or other tabs are open). This might be handy if you want to customize the startup behavior. When, for example, you want to open all conten items that a given search query finds on startup, you can do that with code like the following (a plugin attached to the `EditorMainView`):

```
import Config from "@jangaroo/runtime/Config";
import StringUtil from "@jangaroo/ext-ts/String";
import Component from "@jangaroo/ext-ts/Component";
import Container from "@jangaroo/ext-ts/container/Container";
import AbstractPlugin from "@jangaroo/ext-ts/plugin/Abstract";
import PropertyChangeEvent from "@coremedia/studio-client.client-core/data/PropertyChangeEvent";
import session from "@coremedia/studio-client.cap-rest-client/common/session";
```

```
import SearchParameters from
"@coremedia/studio-client.cap-rest-client/content/search/SearchParameters";
import editorContext from "@coremedia/studio-client.main.editor-components/sdk/editorContext";
import MessageBoxUtil from "@coremedia/studio-client.main.editor-components/sdk/util/MessageBoxUtil";
import EditorErrors_properties
  from "@coremedia/studio-client.ext.errors-validation-components/error/EditorErrors_properties";

class OpenCheckedOutDocumentsPlugin extends AbstractPlugin {
  readonly MAX_OPEN_TABS: int = 10;

  constructor(config: Config<AbstractPlugin>) {
    super(config);
  }

  init(component: Component): void {
    //get the top level container
    const mainView = component.findParentBy((container: Container) => {
      return !container.ownerCt;
    });

    mainView.on('afterrender', this.#openDocuments, null, {
      single: true
    });
  }

  #openDocuments(): void {
    // Perform query to determine content checked out by me.
    const searchParameters = this.#createSearchParameters();
    const searchResult =
session._.getConnection().getContentRepository().getSearchService().search(searchParameters);
    // When the query result is loaded ...
    searchResult.addPropertyChangeListener(SearchParameters.HITS,
      (event: PropertyChangeEvent) => {
        // ... open all content items in tabs.
        const searchResult = event.newValue;
        if (searchResult && searchResult.length > 0) {
          editorContext._.getContentTabManager().openDocuments(
            searchResult.slice(0, OpenCheckedOutDocumentsPlugin.MAX_OPEN_TABS));
          if (searchResult.length > OpenCheckedOutDocumentsPlugin.MAX_OPEN_TABS) {
            MessageBoxUtil.showInfo(
              EditorErrors_properties.editorStart_tooManyDocuments_title,
              EditorErrors_properties.editorStart_tooManyDocuments_message
            );
          }
        }
      });
    searchResult.getHits();
  }

  #createSearchParameters(): SearchParameters {
    const searchParameters = new SearchParameters();
    searchParameters.filterQuery = [this.#getQueryFilterString()];

    //searchParameters.contentType = ['Document_'];
    searchParameters.orderBy = ['freshness asc'];

    return searchParameters;
  }

  #getQueryFilterString(): String {
    const filterQueries = [];

    // retrieve user URI for parametrized filter expressions:
    const user = session._.getUser();
    const userUri = "<" + user.getUriPath() + ">";

    // filter documents checked out by me
    filterQueries.push("ischeckedout:true");
    filterQueries.push(StringUtil.format("editor:{0}", userUri));

    return filterQueries.join(" AND ");
  }
```

```
}
export default OpenCheckedOutDocumentsPlugin;
```

9.18.6 Customizing the Work Area Tab Context Menu

The context menu for work area tabs comes with several predefined actions like close operations and options for checking in or reverting contents. In addition, the `WorkAreaTabProxiesContextMenu` is an extension point for plugging in your own actions.

It is recommended to implement your custom actions as subclasses of `AbstractTabContextMenuAction` or `AbstractTabContextMenuContentAction`. In both cases, the context-clicked tab and tab panel can be accessed via the methods `getContextClickedTab():Panel` and `getContextClickedTabPanel():TabPanel` respectively. In addition, `AbstractTabContextMenuContentAction` provides the methods `getContextClickedContent():Content` and `getContextClickedContents():Array<any>` for obtaining the content of the context-clicked tab and all contents of work area tabs respectively. Note that only `Premular` tabs have content other than `undefined`.

Using these methods, subclasses should override the method `checkDisabled():boolean` to decide whether the action should be disabled. In addition, these methods should suffice to provide enough information to implement the action's behavior.

For example, the following two code samples show how to add an action for checking in all contents of opened work area tabs.

```
import Config from "@jangaroo/runtime/Config";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import WorkAreaTabProxiesContextMenu from
"@coremedia/studio-client.main.editor-components/sdk/desktop/reusability/WorkAreaTabProxiesContextMenu";
import AddItemsPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/AddItemsPlugin";
import Separator from "@jangaroo/ext-ts/toolbar/Separator";
import Item from "@jangaroo/ext-ts/menu/Item";
import CheckInAllContentTabsAction from "./CheckInAllContentTabsAction";

Config(StudioPlugin, {
  rules: [
    //...
    Config(WorkAreaTabProxiesContextMenu, {
      plugins: [
        Config(AddItemsPlugin, {
          items: [
            Config(Separator),
            Config(Item, {
              baseAction: new CheckInAllContentTabsAction({
```

```
                text: "Check in all contents",
              }),
            }),
          ],
        }),
      ],
    }),
    //...
  ]
});
```

```
import Config from "@jangaroo/runtime/Config";
import Content from "@coremedia/studio-client.cap-rest-client/content/Content";
import AbstractTabContextMenuContentAction from
"./AbstractTabContextMenuContentAction";

class CheckInAllContentTabsAction extends AbstractTabContextMenuContentAction
{
  constructor(config: Config<AbstractTabContextMenuContentAction>) {
    super(config);
    this.setHandler(this.#doCheckin, this);
  }

  #doCheckin():void {
    this.getContextClickedContents()
      .forEach((content:Content) => {
      if (content.isCheckedOutByCurrentSession()) {
        content.checkIn();
      }
    });
  }

  protected override checkDisabled():boolean {
    var atLeastOneContentTabInEditMode:Boolean = false;
    this.getContextClickedContents()
      .forEach((content:Content) => {
      if (content.isCheckedOutByCurrentSession()) {
        atLeastOneContentTabInEditMode = true;
      }
    });
    return !atLeastOneContentTabInEditMode;
  }
}

export default CheckInAllContentTabsAction;
```

9.19 Re-Using Studio Tabs For Better Performance

As stated in section Section 9.18, "Work Area Tabs" [226], *CoreMedia Studio* organizes working items in a so called `WorkArea`. The `WorkArea` is an Ext JS `TabPanel` with tabs containing currently opened working items, for example (content) content forms, commerce forms or singleton items like the *Studio Dashboard*. Normally, each working item is created and rendered as a separate tab.

As a `WorkArea` tab can be a very complex component (for example, a content form with various subtabs, collapsible subpanels and heavyweight property editors like richtext fields or image editors), its lifecycle management from creation to destruction can be quite costly. To increase performance you can reuse tabs for multiple working items instead of creating new tabs over and over again. You have to use the Studio plugin `ReusableDocumentFormTabsPlugin`.

> **NOTE**
> Currently, this possibility is only implemented for content form tabs that display content items: `Premulars`.

9.19.1 Concept

The following figure illustrates the concept of content form tab reusability.
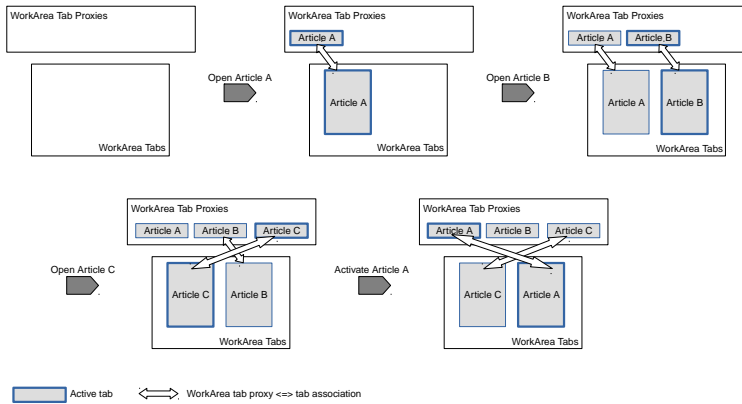
Figure 9.7. Premular Reusability (For A Reusability Limit Of 2 For Articles)

A `WorkArea` *Tab Proxy* is a lightweight representative of an actual `WorkArea` `Premular`. It basically just displays the title of its content item and otherwise has or does not have an active association with a real `Premular`. These proxies are what the Studio user *perceives* as the currently opened `WorkArea` tabs. However, under the hood there are possibly fewer tabs present in the `WorkArea`. Instead, `Premulars` are reused to display multiple content items over the course of a Studio session. For each content type, a *reusability limit* can be configured that limits the amount of actual `Premulars` for content items of this type.

The figure shows the case where the limit is *2* for articles. After the user opens the two articles *A* and *B*, the two created tab proxies are each associated with a corresponding `WorkArea` `Premular`. When the user opens the third article *C*, reusability takes place. No new `Premular` is created. Instead, the least recently used one is reused, which is the tab that currently holds the content for *A*. This `Premular` gets filled with its new content for *C*. When the user switches back to article *A*, the `Premular` currently holding the content of *B* is reused and filled with the content for *A*. So no matter how many more articles the user opens, there will only be more proxies but no more real `Premulars` than 2. Only if the user would open a content item of a different type (say, a picture) a new `Premular` would be created.

9.19.2 Prerequisites

`Premulars` are already designed to work with changing content items. For this purpose the `Premular`'s content item is held by the `bindTo ValueEx`

`pression`. However, if you have customized content forms (Section 9.5.2, "Customizing Content Forms" [148]) and / or property fields (Section 9.6, "Customizing Property Fields" [162]) you are advised to consider certain prerequisites for the *ReusableDocumentFormTabsPlugin* to work properly.

Problems may arise when some features depend on conditions that apply to some content forms but not to others. The following example illustrates this point. Let's say, a certain property editor should only be present for `Premulars` whose underlying content item belongs to a specific site. Depending on how this site check is carried out, the visibility of the editor might not be handled correctly on tab reusage.

- *Bad practice:* Some plugin checks only on creation time of the property editor if the underlying content item belongs to the site and makes the editor visible or not. This visibility is not reevaluated on change of the content item and thus does not go along with `Premular` reusage.
- *Good practice:* The `BindVisibilityPlugin` is used to determine the visibility of the property editor based on the underlying content item (`bindTo ValueExpression`). The plugin reevaluates as soon as the content item changes and thus works fine with `Premular` reusability.

As a rule of thumb, such dynamic content form features work fine with `Premular` reusability if either of the following two conditions holds:

- The conditions for the feature exclusively rest on the `Premular`'s underlying content item (`bindTo ValueExpression`) and are re-evaluated on content change.
- The conditions for the feature do not change until Studio reload (for example, the user's group memberships).

NOTE

Note that `ComboBoxes` cannot be reset to an empty value when reusing a tab, as `null`, `empty` string and `undefined` values are not valid stored values. Therefore, when implementing custom property editors, it is essential to ensure that these values are not assigned. A common solution to this issue is to provide a default value that appears when the combo box would otherwise be empty. For example, an additional record can be added using `re cords.push({value: "", name: ""})`. Additionally, the `BindProp ertyPlugin` should declared with `ifUndefined: ""`.

9.19.3 Usage

To enable the *ReusableDocumentFormTabsPlugin*, add the following (or something similar) to one of your Studio plugins:

```
import Config from "@jangaroo/runtime/Config";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import ReusableDocumentFormTabsPlugin from
"@coremedia/studio-client.main.editor-components/sdk/desktop/reusability/ReusableDocumentFormTabsPlugin";

// ...
Config(StudioPlugin, {
  //...
  configuration: [
    //...
    new ReusableDocumentFormTabsPlugin({
      defaultLimit: 2,
      limitsPerContentType: { "CMArticle": 3 }
    }),
    //...
  ];
//...
});
```

For each content type, a *reusability limit* can be configured that limits the amount of actual `WorkArea Premulars` for content items of this type. A limit of *0* effectively disables reusability for content items of the corresponding type. You can configure a default limit via the attribute `defaultLimit` or individually for each content type via the attribute `limitsPerContentType`.

The property `limitsPerContentType` overrides the property `default Limit`. So you can easily implement a white list for reusability (`defaultLimit` set to *0*, `limitsPerContentType` set for some content types) as well as a black list (`defaultLimit` set to some value, `limitsPerContentType` set to *0* for some values)

9.20 Dashboard

CoreMedia Studio provides a dashboard as a special tab type. On the dashboard, users may freely arrange so-called *widgets*, which display data that the user should be aware of. While your users may configure the dashboard according to their particular needs, it is your task as a developer to determine which widget types are available to them and to configure a suitable default dashboard for the first login.

9.20.1 Concepts

Studio dashboard widgets are organized in three columns of equal width that span the entire work area. Each widget may fill one or more fixed-height rows, depending on its `rowspan` attribute. Widgets cannot span multiple columns. Users can adjust the height of each individual widget when they adjust their widget configuration.

Three rows

There may be many fundamentally different widget types for various purposes. Generally, widgets are used to display current information that a user is likely to be interested in, without requiring immediate action. However, there may also be widgets that allow the user to make simple updates or interact with other users. Due to the limited size of a widget, complex interactions are likely moved to a tab or a separate dialog.

Each widget type must provide a user interface that displays the actual information for this widget. Additionally, each widget type *may* opt to provide a user interface to configure a particular instance of the widget type on the user's dashboard. Users can choose a "configuration mode" for each widget, and in this mode, the configuration UI is displayed, which can be used to modify the appearance and functionality of the widget. Multiple widgets of the same type may be shown on the dashboard and each such widget can be in a different configuration state. Note the "configurability" of a widget is optional. For non-configurable widget types, the widget may just show an explanatory text describing its functionality.

For each user, the set of widgets, their positions, sizes, and states are stored persistently, allowing you to restore the widgets when the dashboard is closed and reopened. Many widget types provide a corresponding state class that allows you to define the state of the widget when configuring an initial dashboard. Widget state object and widget types are matched with each other by means of a widget type id.

State is stored persistently

Besides creating the user interfaces, the widget type in the form of an object implementing the `WidgetType` interface is also responsible for providing a type name, description, icon, default `rowspan`, and for computing a title, possibly depending on the current widget state. Optionally, the widget type may also provide tools to be included in the header bar of the widget. Tools can allow the user to start operations based on the current widget state.

9.20.2 Defining the Dashboard

You can configure the dashboard by selecting which widgets the user may add to the dashboard and by describing the initial widget configuration of the dashboard.

To this end, the dashboard configuration is available through the method `get DashboardConfiguration()` of the `editorContext` object. It provides a list of `WidgetType` objects in the `types` property and a list of `Widget State` objects in the `widgets` property.

Usually, you will not access the configuration object directly, but rather through the `ConfigureDashboardPlugin`, which also offers a `types` and a `wid gets` property and takes care of merging these values into the global configuration at the correct time.

The widget state objects in the property `widgets` determine the widgets to be shown when the user first opens the dashboard. You should therefore select widgets that a typical novice user would find interesting.

Each widget state object must be an instance of the class `WidgetState`, or a subclass thereof. The class `WidgetState` itself defines only the properties `widgetTypeId`, `rowspan`, and `column`, indicating the widget type, the relative height of the widget and the placement of the widget, respectively.

Widget types for all initial widgets have to be provided, but you will typically add more widget types for advanced users. Widget types and widget state objects are matched by their id, which can be specified using the `widgetTypeId` property of the state object. Predefined state objects will typically provide the correct ID automatically.

The following example shows how the `ConfigureDashboardPlugin` is used inside a Studio plugin specification.

```
import Config from "@jangaroo/runtime/Config";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import ConfigureDashboardPlugin from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/ConfigureDashboardPlugin";
import SimpleSearchWidgetState from
```

```
"@coremedia/studio-client.main.editor-components/sdk/dashboard/widgets/search/SimpleSearchWidgetState";
import SimpleSearchWidgetType from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/widgets/search/SimpleSearchWidgetType";

class MyStudioPlugin extends StudioPlugin {
  constructor(config: Config<MyStudioPlugin>) {
    super(ConfigUtils.apply(Config(MyStudioPlugin, {
      //...
      configuration: [
        //...
        new ConfigureDashboardPlugin({
          widgets: [
            new SimpleSearchWidgetState({
              contentType: "CMArticle"
            }),
            new SimpleSearchWidgetState({
              contentType: "CMPicture",
              column: 1
            }),
          ],

          types: [
            new SimpleSearchWidgetType({}),
          ],
        }),
        //...
      ],
    }), config));
  }
}
```

Example 9.78. Dashboard Configuration

You can see a single widget type being configured, `SimpleSearchWidget Type`. In this example, the widget type provides no configuration option itself, but some widget type classes can be customized by configuration.

In the example, there are two widgets using the defined type. By specifying a `SimpleSearchWidgetState`, the widget type id is set to match the `SimpleSearchWidgetType`. The two widgets start off with a specific state. As a rule, any configuration options that can be provided using a state object should also be configurable when the widget is in edit mode.

For the second widget, a column is specified. Unless a column property is given, each widget is placed in the same column as the previous widget and the first widget is placed in the leftmost column. For the column property use either a numeric column id from 0 to 2 or one of the constants `SAME` or `NEXT` from the class `WidgetState`, indicating to stay in the same column or to progress one column to the right. The leftmost column is used as the next column of the rightmost column.
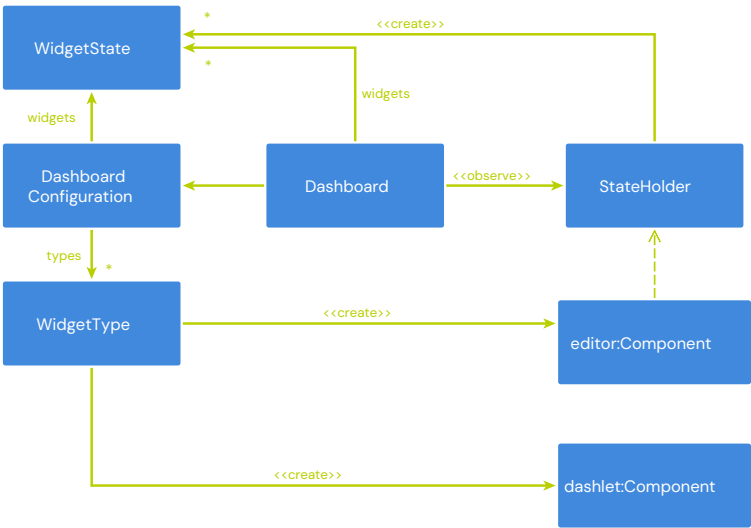
Figure 9.8. Dashboard UML overview

9.20.3 Predefined Widget Types

There are a number of predefined widgets that are immediately usable through simple configuration. The following table summarizes the existing widgets.

Name	Description
`WelcomeWidgetType`	Displays a welcome messages and the tip of the day. It requires no additional configuration.
`MyTodosWidgetType`	Allows to create a personal todo list. It requires no additional configuration.
`BookmarkWidgetType`	Displays the bookmarks of the user. It requires no additional configuration.
`FixedSearchWidgetType`	Displays the result of exactly one preconfigured search.

Name	Description
SimpleSearchWidgetType	Displays the result of a search for contents of a configurable type containing a configurable text.

Table 9.5. Predefined Widget Types

The individual types and their configuration options are subsequently explained in more detail.

9.20.3.1 Fixed Search Widget

Widget types based on the class `FixedSearchWidgetType` display the result of exactly one preconfigured search. Because this widget type does not offer any editable state, you should provide the search to execute when you define the widget type. In this way, you can define fixed search widget types showing checked-out content items or the most recently edited pages or arbitrary other searches.

For each type, you should at least specify the `name` under which the type can be selected in the dropdown box when adding a new widget. At your option, you may also set a `title` or a `description` to be shown for your type.

Because you can define multiple types, you must also provide different widget type IDs. You can then use a plain `WidgetState` element with the chosen type ID and placement attributes to instantiate the widget.

An example configuration of this widget might look like this:

```
import ConfigureDashboardPlugin from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/ConfigureDashboardPlugin";
import SearchState from
"@coremedia/studio-client.ext.library-services-toolkit/SearchState";
import FixedSearchWidgetType from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/widgets/search/FixedSearchWidgetType";

//...
new ConfigureDashboardPlugin({
  widgets: [
    new SearchState({
      editedByOthers: true,
      editedByMe: false,
      notEdited: false,
      approved: false,
      published: false,
    }),
  ],

  types: [
    new FixedSearchWidgetType({
      id_: "editedByOthers",
      name: "Edited by others",
    }),
```

```
    ],
})
```

Example 9.79. Fixed Search widget Configuration

9.20.3.2 Simple Search Widget

A widget of type `SimpleSearchWidgetType` displays the result of a search for contents of a configurable type containing a configurable text. By default, the search is limited to the preferred site, if such a site is set. Through the state class `SimpleSearchWidgetState`, the dashlet provides the associated configuration options `contentType`, `searchText`, and `preferredSite`.

An example configuration of this widget might look like this:

```
import ConfigureDashboardPlugin from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/ConfigureDashboardPlugin";
import SimpleSearchWidgetState from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/widgets/search/SimpleSearchWidgetState";
import SimpleSearchWidgetType from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/widgets/search/SimpleSearchWidgetType";

//...
new ConfigureDashboardPlugin({
  widgets: [
    new SimpleSearchWidgetState({
      contentType: "CMPicture",
    }),
  ],

  types: [
    new SimpleSearchWidgetType({}),
  ],
})
```

Example 9.80. Simple Search Widget Configuration

9.20.4 Adding Custom Widget Types

You can define your own widget types and add widgets of this type to the dashboard. This section will guide you through all the necessary steps, covering rather simple widgets as well as more sophisticated ones.

9.20.4.1 Widget Type and Widget Component

When creating own widgets, you typically start off by creating a custom *widget type*. As described in the previous sections, the dashboard is configured in terms

of columns and widget states. Each widget state carries a *widget type id* which associates it with its widget type. In order to get from widget states to the actual widget instances shown on the dashboard, the different widget types are consulted. A widget type is responsible for creating the widget components from their associated widget states.

You could define your own widget type by creating a class from scratch that implements the interface `WidgetType`. However, a convenient default implementation `ComponentBasedWidgetType`, is provided out of the box. For many cases it is sufficient to just use it or to let let your own widget type extend it. In order to do so, you have to define a *widget component* that defines the UI for widgets of your new widget type. For instance, the predefined `Simple SearchWidgetType` is simply defined as follows:

```
import Config from "@jangaroo/runtime/Config";
import ConfigureDashboardPlugin from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/ConfigureDashboardPlugin";
import ComponentBasedWidgetType from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/ComponentBasedWidgetType";
import SimpleSearchWidget from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/widgets/search/SimpleSearchWidget";

//...
new ConfigureDashboardPlugin({
  //...

  types: [
    new ComponentBasedWidgetType({
      name: "...",
      description: "...",
      iconCls: "...",
      widgetComponent: Config(SimpleSearchWidget),
    }),
  ],
})
```

Example 9.81. Simple Search Widget Type

Besides setting the parameters `name`, `description` and `iconCls`, the widget component `SimpleSearchWidget` is set. The `SimpleSearchWidget` can be configured with the parameters `searchText` and `contentType` in order to show a corresponding search result. Executing the search and obtaining the search results is carried out in the base class `SimpleSearchWidgetBase`. When extending that class, a value expression that references the search result can be obtained via `getContentValueExpression()` and is used by a `WidgetContentList` to display the result.

There is one further important aspect concerning the base class `SimpleSearchWidgetBase`. It implements the `Reloadable` interface. This indicates that a reload button should be placed in the widget header, calling the widget's `reload()` method for refreshing the widget's contents. In this case, the base class simply triggers a new search.

9.20.4.2 Configurable and Stateful Widgets

The `WidgetType` interface also features the creation of an editor component for a widget at runtime. Again, if you opt to implement the interface yourself, you have to provide this functionality from scratch. If you choose your type to extend `ComponentBasedWidgetType`, you simply have to add an editor component, just as you did for the widget component. Consequently, the TypeScript code for the `SimpleSearchWidgetType` for simple search widgets that are configurable at runtime looks as follows:

```
import Config from "@jangaroo/runtime/Config";
import ComponentBasedWidgetType from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/ComponentBasedWidgetType";
import SimpleSearchWidget from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/widgets/search/SimpleSearchWidget";
import SimpleSearchWidgetEditor from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/widgets/search/SimpleSearchWidgetEditor";

//...
new ComponentBasedWidgetType({
  name: "...",
  description: "...",
  iconCls: "...",
  widgetComponent: Config(SimpleSearchWidget),
  editorComponent: Config(SimpleSearchWidgetEditor),
})
```

Example 9.82. Simple Search widget Type with Editor Component

Now widgets of this type have their own editor component when a widget on the dashboard is in edit mode.

However, without further wiring, the changes a user makes in edit mode do not carry over to the widget component. For the simple search widget it is expected that the user can choose a search text and content type in edit mode and that the widget shows a corresponding search result in widget mode. To make this happen, `SimpleSearchWidgetEditor` has to implement the `StateHolder` interface. The method `getStateValueExpression()` has to be implemented in a way that the value expression refers to a simple JavaScript object containing the configuration properties to be applied to the widget component. Thus, for the simple search widget, these properties are `searchText` and `contentType`.

See section for details of how the state values are persisted and for the limits on the allowed objects.

You could just implement the `StateHolder` interface yourself. For convenience, CoreMedia recommends, that you let your editor component extend `StatefulContainer`. This component inherently implements `StateHolder`. It can be configured with a list of property names along with default values and

automatically takes care of building a *state model bean* from them. This state model bean is the basis for the evaluation of the value expression that is returned via `getStateValueExpression()`. Additionally, the bean can be consulted via `getModel()` from subclasses of `StatefulContainer`. This can be utilized for binding the model state to the user interface state. The following listing exemplifies this for the case of `SimpleSearchWidgetEditor`:

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import TextField from "@jangaroo/ext-ts/form/field/Text";
import ValueExpressionFactory from
"@coremedia/studio-client.client-core/data/ValueExpressionFactory";
import ContentTypeNames from
"@coremedia/studio-client.cap-rest-client/content/ContentTypeNames";
import BindPropertyPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/BindPropertyPlugin";
import VerticalSpacingPlugin from
"@coremedia/studio-client.ext.ui-components/plugins/VerticalSpacingPlugin";
import StatefulContainer from
"@coremedia/studio-client.ext.ui-components/components/StatefulContainer";
import ContentTypeSelector from
"@coremedia/studio-client.ext.cap-base-components/contenttypes/ContentTypeSelector";

class MyWidgetEditor extends StatefulContainer {
  static override readonly xtype: string =
"com.coremedia.cms.widget.config.myWidgetEditor";

  constructor(config: Config<StatefulContainer>) {
    super((() => ConfigUtils.apply(Config(MyWidgetEditor, {
      properties: "searchText,contentType,preferredSite",
      items: [
        Config(ContentTypeSelector, {
          fieldLabel: "...",
          anchor: "100%",
          itemId: "...",
          entries: ContentTypeSelector.getAvailableContentTypeEntries(),
          contentTypeValueExpression:
ValueExpressionFactory.create("contentType", this.getModel()),
        }),
        Config(TextField, {
          itemId: "...",
          anchor: "100%",
          plugins: [
            Config(BindPropertyPlugin, {
              bindTo: ValueExpressionFactory.create("searchText",
this.getModel()),
              bidirectional: true,
            }),
          ],
        }),
      ],
      plugins: [
        Config(VerticalSpacingPlugin, {}),
      ],
      propertyDefaults: { contentType: ContentTypeNames.DOCUMENT },
    }), config))());
  }
}

export default MyWidgetEditor;
```

Example 9.83. Simple Search Widget Editor Component

This editor component for the simple search widget extends `StatefulCon tainer` and is configured to build a state model for the two properties

searchText and contentType. For the content type property, a default is set. The editor component offers the user a combo box for selecting a content type and a text field for entering a search text. The user's input is tied to the state model via value expressions that use getModel() (inherited from StatefulContainer) as their context. This results in keeping the state model updated. Implementing the StateHolder interface yourself is not necessary. It is automatically taken care of by StatefulContainer on the basis of the always up-to-date state model.

All in all, this results in the simple search widget editor being stateful. When the user switches between widget mode and edit mode for this widget, the editor will keep its state (search text and content type). The state is only lost if the user selects a different widget type in edit mode.

In some cases, it might be useful to not only have the editor of a widget being stateful, but also the widget itself. This can be realized in the same way shown here for the editor: by implementing the StateHolder interface.

9.20.4.3 Custom Widget State Class

In many cases, it is not necessary to create your own widget state class for your custom widget type. As shown earlier in this chapter, the predefined class WidgetState allows you to set the dashboard column, the widget type and the widget's rowspan. This is sufficient unless you want to put widgets of your type into the default dashboard and at the same time use a configuration other than the default. However, if you want to do just that, CoreMedia recommends that you create your own widget state class as an extension to WidgetState. For the simple search widget, the custom state class SimpleSearchWidget State looks as follows:

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import WidgetState from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/WidgetState";
import SimpleSearchWidget from
"@coremedia/studio-client.main.editor-components/sdk/dashboard/widgets/search/SimpleSearchWidget";

class SimpleSearchWidgetState extends WidgetState {
  /**
   * The search text that is used for the collection view.
   * Default "".
   */
  searchText: string = null;
  /**
   * The content type that is used in the content type filter.
   * Default "Document_".
   */
  contentType: string = null;
  /**
   * Whether to restrict the search to the preferred site.
   * Default true.
   */
```

```
  preferredSite: boolean = false;

  constructor(config: Config<WidgetState>) {
    super(ConfigUtils.apply(Config(SimpleSearchWidgetState, { widgetTypeId:
 SimpleSearchWidget.xtype }), config));
  }
}

export default SimpleSearchWidgetState;
```

Example 9.84. widget State Class for Simple Search widget

This class allows you to launch simple search widgets initially with the configuration properties `searchText` and `contentType` being set. They are set via the dashboard configuration prior to the dashboard's launch instead of being set by the user via the `SimpleSearchWidgetEditor` component at runtime (although this is of course possible afterwards).

The `widgetTypeId` for the `SimpleSearchWidgetState` is set to the `xtype` of `SimpleSearchWidget`. This is because widget types that extend `ComponentBasedWidgetType` by default take the `xtype` of their widget component as their `id`.

9.21 Configuring MIME Types

When a blob is uploaded into a property field, *CoreMedia Studio* detects an appropriate MIME type based on name and content of the uploaded file. This is done with the help of the `mimeTypeService` bean, which is based on Apache Tika. This service is able to detect many common file types. If the file type is unknown, the MIME type suggested by the uploading browser will be used.

> ### MIME Type Service Configuration
>
> If you need to adapt the MIME Type Service Configuration, as proposed in subsequent paragraphs, find more details in section "MIME Type Mappings" in *Content Application Developer Manual*.

Adding new file types can be achieved by adding corresponding MIME type definitions to file `shared/common/modules/shared/custom-mime-types/src/main/resources/org/apache/tika/mime/custom-mimetypes.xml`.

The list of MIME type definition file names may be extended by setting `mimeTypeService.mimeTypesResourceNames`. If you want to place your MIME type definitions in a file other than `org/apache/tika/mime/custom-mimetypes.xml`, create a corresponding file in `shared/common/modules/shared/custom-mime-types/src/main/resources/`. Set `mimeTypeService.mimeTypesResourceNames` to include the pre-defined pathes plus your new file's relative path. See following example on how to add a new resource file `com/acme/project/acme-mimetypes.xml`.

```
mimeTypeService.mimeTypesResourceNames=org/apache/tika/mime/coremedia-tika-mimetypes.xml,org/apache/tika/mime/custom-mimetypes.xml,com/acme/project/acme-mimetypes.xml
```

Example 9.85. Add Custom Resource to MIME Type Definitions

You will find an example for a MIME type definition in Example 9.86, "Override `*.exe` MIME Type Detection" [248].

```xml
<?xml version="1.0" encoding="UTF-8"?>

<mime-info>

  <mime-type type="application/acme">
    <_comment>New MIME Type Mapping</_comment>
    <glob pattern="*.acme"/>
  </mime-type>

  <mime-type type="application/x-dosexec">
    <_comment>Override Tika Default</_comment>
```

```
    <sub-class-of type="application/x-msdownload"/>
    <glob pattern="*.exe" weight="100"/>
    <magic priority="100">
      <match value="MZ" type="string" offset="0"/>
    </magic>
  </mime-type>

</mime-info>
```

Example 9.86. Override `*.exe` MIME Type Detection

Details about the example:

- The first entry is about adding some new MIME type for files with `acme` extension.
- The second entry overrides the default Tika configuration enforcing all `*.exe` to be mapped to `application/x-dosexec`.

  While the default Tika configuration already maps `*.exe` to MIME type `application/x-dosexec`, it adds subsequent overrides to `application/x-msdownload` with `format` property, to distinguish for example 32bit from 64bit applications.

  To override it, you need to duplicate the `<magic>` pattern of the original definition and provide a higher priority than in Tika's default configuration. Valid priorities are from 0 to 100, where 50 is the default.

For a reference of all elements and attributes in `custom-mimetypes.xml` have a look at the API documentation of `org.apache.tika.mime.MimeTypesReader`. As stated in the documentation, the DTD is compliant to *freedesktop MIME-info DTD*. Note, though, that it only contains a subset of attributes and elements. Nevertheless, you may find some more valuable information in the official specification located at freedesktop.org: Shared MIME Info Specification.

If you need to override existing mappings, the approach via `custom-mimetypes.xml` may not be sufficient. In this case you may need to set `mimeTypeService.tikaConfig`. Note though, that, in contrast to `custom-mimetypes.xml`, this requires defining all MIME types by yourself. For a start, you may want to take `tika-mimetypes.xml` for reference, which can be found in the Apache Tika GitHub Repository.

Example where overriding may fail: You may struggle with Tika reporting duplicate definitions. For example, take the re-mapping of `*.exe` above. If you skipped the `<magic>` element, Tika would report about a duplicate definition for `*.exe` without being able to get the priorities straight. Thus, you need to *tune* your adaptations and have a deep understanding about the `<mime-info>` configuration. And as Tika does not support `<glob-deleteall>` and `<mime-deleteall>` as specified by *freedesktop MIME-info DTD*, there is no straightfor-

ward way to enforce your MIME-type detection, while trying to benefit from existing MIME-type detection configuration for types you want to keep as is.

9.22 Configuring Maximum Length for TextFields

When saving content, certain server-side limitations, such as restrictions in data storage or database structure, may affect the save operation. These limitations can result in client-side errors and may cause data loss during the saving process.

To prevent such issues, the length of input fields (`StatefulTextField` and `StatefulTextArea`) on the client side can be limited by configuring a maximum length. When the field is focused, both the defined maximum length and the current input length are displayed.

Note: this feature must be enabled manually for each input field by setting the corresponding properties:

Property	Description
`showCharacterCounter`	Whether to display a counter for the default or manually set character limit on the text field.
`characterLimit`	A manual character limit for the text field child. If set, editors will not be able to insert more characters.

Table 9.6. Character Limit Properties

The character counter can also be configured on `StringPropertyField`s directly. It considers the `Length` defined for the corresponding `StringProperty` in the `blueprint-doctypes.xml` file as the default, which can be overridden by setting the `characterLimit` property.

9.23 Server–Side Content Processing

Several operations on content can be implemented on the server side using the Unified API from Java. Especially, you may want to place restrictions on the content that is stored in your repository. This may be achieved by pointing the editors to invalid content, by normalizing content during writes or by inhibiting writes that violate your constraints.

- Section 9.23.1, "Validators" [252] describes how to add validation for values stored in the content repository.
- Section 9.23.2, "Intercepting Write Requests" [266] describes how to modify writes before they are executed.
- Section 9.23.3, "Immediate Validation" [269] describes how to inhibit undesirable writes.
- Section 9.23.4, "Post-processing Write Requests" [270] describes how to take additional action after a write has been completed.

9.23.1 Validators

CoreMedia supports server–side validation based on a project–specific configuration. Validators can analyze content and report issues which are available at the studio client side as described in Section 5.3.3, "Issues" [62]. Validators are implemented in Java and run in server applications, currently in the studio server and in the content feeder.

CoreMedia provides some predefined validator classes and an API to implement your own. Some validators are already declared in the Blueprint. You can disable them if they do not match your needs. You can declare custom validators and additional validators of the predefined classes.

9.23.1.1 Declaration of Validators

The declaration of validators is identical for our predefined validators and for your custom validators. There are two ways to declare validators:

- As Spring beans
- As Json configuration files

Validator Spring beans can be located in the application or in plugins. All relevant interfaces are extension points.

Json configuration files for validators must adhere to the naming pattern `validation/captype/**/*-validator-configuration.json`. They are detected in three locations:

- In the application class path
- As plugin resources
- In the file system, in directories to be configured by the list-valued application property `validator.configuration-directories`

If you know your validation requirements already at development time, it is easier to use Spring beans. Json configuration files are more suitable if validation decisions are made lately in the deployment process. They require the implementation and provision of validator factories, which means some extra effort when developing custom validators. You can mix validator Spring beans and Json configuration files in your application.

Most predefined validator classes of the Shared/Middle layer support declaration by Json files, except of a few which make only sense as singletons and are provided by default anyway. In case of doubt, check whether the API documentation of the validator mentions a factoryId for Json declaration at class level.

A Spring Boot actuator endpoint exposes a Json schema to assist in writing validator configuration files. The schema follows draft-07 and contains the schemas of all available validator factories. The endpoint id is `validatorschema`. As some IDEs only support schemas at URLs ending on .json, the schema is also available at `validatorschema/schema.json`.

A declared validator is active by default. However, you can deactivate each validator by an application property. So, you should not hesitate to declare a validator in case of doubt.

9.23.1.2 Predefined Validators

CoreMedia offers some predefined validators for common usecases and an API to implement your own, based on project-specific content validation requirements. The table below gives an overview of predefined validators. For details and more validators, see the Shared / Middle API documentation (available at the CoreMedia download area), especially the packages `com.core-`

media.rest.validators and com.coremedia.rest.cap.validat-
ors.

Name	Behavior
DayOfWeekValidator	checks that a date property contains only dates on certain days of the week
EmailValidator	checks for a valid email address according to RFC822
ImageMapAreasValidator	checks for non-empty image and correctly linked areas in an image map. See also Section 9.5.4, "Enabling Image Map Editing" [158]
ListMaxLengthValidator and ListMin LengthValidator	checks for maximum/minimum number of content items linked in a linklist
MaxIntegerValidator and MinInteger Validator	checks for a maximum/minimum integer value
MaxLengthValidator and MinLengthVal idator	checks for a maximum/minimum length of a String
NotEmptyValidator	checks whether a field is empty; works on strings, linklists, and blobs
RegExpValidator	checks whether a given (configurable) regular expression matches against the value given in the property
UniqueListEntriesValidator	checks against duplicate links in a linklist (that is, the same content item is linked at least twice in the same linklist)
UriValidator and UrlValidator	checks for valid URIs or URLs, respectively

Table 9.7. Selected predefined validators

The easiest way to declare a validator is to provide it as Spring Bean. For example, an ImageMapAreasValidator is declared like this:

```
@Bean
@ConditionalOnProperty(
  name = "validator.enabled.image-map-areas-validator.cm-image-map",
  matchIfMissing = true)
ImageMapAreasValidator cmImageMapAreasValidator(CapConnection cc) {
  ContentType type = cc.getContentRepository().getContentType("CMImageMap");

  return new ImageMapValidator(type, true, "localSettings",
"pictures.data");
}
```

Example 9.87. Declaring a validator as Spring bean

The `@ConditionalOnProperty` annotation allows you to disable this validator by the setting the application property `validator.enabled.image-
map-areas-validator.cm-image-map=false`. If you do not need this,
you can omit it. By convention, such validator enabling properties start with the
prefix `validator.enabled`, the third segment is the lowercase/hyphen
variant of the validator class, and the last segment is a short description.

The `ImageMapAreasValidator` validates a content as a whole. However,
most predefined validators are only property validators, which validate a single
property value of a content. Property validators must be wrapped into a `Con
tentTypeValidator`. For example, a `NotEmptyValidator`, which ensures
that the `title` property of a `CMArticle` content is not empty, is declared
like this:

```
@Bean
@ConditionalOnProperty(
  name = "validator.enabled.content-type-validator.article-validation",
  matchIfMissing = true)
ContentTypeValidator articleValidator(CapConnection cc) {
  ContentType type = cc.getContentRepository().getContentType("CMArticle")
  return new ContentTypeValidator(
          type,
          true,  // validate also subtypes of CMArticle
          List.of(new NotEmptyValidator("title")));
}
```

Example 9.88. Declaring a property validator as Spring bean

Here, the content type validator is configured to apply to all subtypes of the
given content type, too.

To provide multiple validators for a content type, you can declare multiple
`ContentTypeValidator` beans or, more commonly, multiple property validators in a single content type validator. Note that you can only disable the
whole content type validator. This may affect your decision how to arrange
property validators in content type validators.

For all property validators that inherit from `AbstractPropertyValidator`
(esp. all predefined property validators), you can set the field `code` to an issue
code of your choice. If you choose not to do so, the class name of the validator

implementation will be used as the issue code. For example, the validator `com.coremedia.rest.validators.RegExpValidator` creates issues with code `RegExpValidator` by default.

Alternatively, you can provide validator declarations as Json configuration files. The equivalent Json configuration for the above validators looks like this:

Declaration by Json

```
{
  "image-map-areas-validator": {
    "cm-image-map-areas": {
      "content-type": "CMImageMap",
      "subtypes": true,
      "struct-property": "localSettings",
      "image-property-path": "pictures.data"
    }
  },
  "content-type-validator": {
    "article-validation": {
      "content-type": "CMArticle",
      "subtypes": true,
      "property-validators": [
        {
          "not-empty-validator": {
            "property": "title"
          }
        }
      ]
    }
  }
}
```

Example 9.89. Json declaration of validators

The keys of the outer map, `image-map-areas-validator` and `content-type-validator`, denote validator factories. You find the factory ID in the API documentation of each validator. By convention, it is the lowercase/hyphen variant of the validator class (just like the third segment of the enabling properties).

The second level keys, `cm-image-map-areas` and `article-validation`, are validator IDs. You might encounter them in log messages, so you should use reasonable IDs that you can easily recognize. The validator maps contain the configuration data for the particular validator instance. The attributes `content-type` and `subtypes` are common for most validators. The other attributes depend on the particular validator class. Usually, it is the lowercase/hyphen variants of the fields that can be set by constructor arguments or setter methods of the validator.

The `property-validators` attribute of a `content-type-validator` is a list of maps, each of which denotes a property validator. Each map has exactly one entry, whose key (`not-empty-validator` in the example) is the factory ID of the property validator. The value is a configuration map for the property validator instance, which usually consists of the property to validate, the optional `code` field, and possibly additional fields like ranges or sizes to validate against.

Just like Spring bean validators, you can disable Json-declared validators by setting the application property `validator.enabled.<factoryID>.<validatorID>` to false.

Disabling Json-declared validators

9.23.1.3 Custom Validators

If there are no suitable predefined validator classes that match particular validation requirements, you can implement custom validators.

There are three levels of validators, each of which is represented by an interface:

Levels of Validators

Interface	Purpose
`PropertyValidator`	A `PropertyValidator` validates a single property value of a content, like the `LinkListMinLengthValidator`. If you want to validate multiple properties of a content independently of each other, use a `PropertyValidator` for each property. If the properties are related with respect to validity, use a `CapTypeValidator`. PropertyValidators are usually generic and can be used for various properties of different content types.
`CapTypeValidator`	A `CapTypeValidator` validates contents of a particular content type. CapTypeValidators usually take multiple properties of the content into account (for example `AtLeastOneNotEmptyValidator`) or verify contextual aspects of the content (for example `ChannelIsPartOfNavigationValidator`).
`Validator`	A `Validator` validates arbitrary contents. Such validators are often singletons, like the `AvailableLocalesValidator`. You will rarely need to implement a `Validator`, since `PropertyValidator` and `CapTypeValidator` of-

Interface	Purpose
	fer more development convenience and suffice for most usecases.

Table 9.8. Levels of Validators

Property Validators

For a property validator, you have to implement the interface `PropertyValidator`. The easiest way of doing this is by extending the class `AbstractPropertyValidator<T>` and implementing the method `isValid(T value)`.

Implementation of Property Validators

```
public class MyValidator extends AbstractPropertyValidator<String> {
  public MyValidator(@NonNull String property) {
    super(String.class, property);
  }

  @Override
  protected boolean isValid(String value) {
    return ...;
  }
}
```

Example 9.90. Implementing a property validator

The example shows a `PropertyValidator` for String properties. See `CapStruct` for the possible types of property values. You can also implement property validators for more general types, esp. for `Object`, and apply them to arbitrary properties. But the usecases for validators that are suitable for, let's say, Integer properties and Blob properties are probably rare, so you will implement property validators for particular property types most of the time.

Now, declare a `MyValidator` for the property `teaserTitle` of the content type `CMTeasable` as a Spring bean.

Declaration as Spring Bean

```
@Bean
@ConditionalOnProperty(
  name = "validator.enabled.content-type-validator.my-validator-teaser-title",

  matchIfMissing = true)
ContentTypeValidator myValidator(CapConnection con) {
  ContentType type = con.getContentRepository().getContentType("CMTeasable");

  return new ContentTypeValidator(type,
                                  true,  // include subtypes of CMTeasable
                                  List.of(new MyValidator("teaserTitle")));
}
```

Example 9.91. Declaring a property validator as Spring bean

As an alternative to the Spring bean declaration, you can declare validators by Json configuration files. If you want to support this option also for your custom validators, you must provide a factory to instantiate validators. In most cases, this is easy: first, you enhance the constructor of your validator class with some Jackson annotations. Your Json enabled `MyValidator` class would look like this:

Declaration by Json

```
public class MyValidator extends AbstractPropertyValidator<String> {

  @JsonCreator
  public MyValidator(@JsonProperty(value = "property", required = true)
                     @NonNull String property) {
    super(String.class, property);
  }

  @Override
  protected boolean isValid(String value) {
    return ...;
  }
}
```

Example 9.92. A Json-enabled property validator

The jackson annotations originate from the `com.fasterxml.jack` `son.core:jackson-annotations` library, which you must add to your maven dependencies.

All constructor arguments must be annotated with `@JsonProperty`. The supported types are `String`, `Boolean`, numbers, enums and nested maps and lists of these types. Be aware, that non-required constructor arguments must be nullable, that is, do not use primitive types but only the according wrapper classes like `Boolean` for such arguments. You can use the `@JsonProp` `erty` annotation also at *setter* methods (at the method, not at the argument!) or directly at the *field* declaration.

To generate a correct Json schema, all relevant properties have to be annotated with `@JsonProperty` directly at the *field* (can be private) or the *getter* method. Be aware that properties of simple types (for example, `boolean`) will automatically be marked as required.

So the summarized recommendation is: Add `@JsonProperty` to the field declarations of all relevant properties and to all constructor arguments and don't use simple types for non-required properties.

The `@JsonProperty` annotation has a `value` attribute, which denotes the field name in the Json representation of the object. By convention, the field name of the "property" constructor argument (corresponding to the second argument of the `AbstractPropertyValidator` constructor) is always `property`. When applied to setter methods, the field name should be the lowercase/hyphen variant. For example, if the method name is `setFooBar`, the field name should

be `foo-bar`. Adhering to these conventions, you spare a lot of documentation, and you make life much easier for those who want to use your validator.

Next, you provide the actual validator factory as a Spring bean. A property validator factory implements the interface `PropertyValidatorFactory`. For Jackson-annotated validator classes, there is a generic factory class `Class-BasedPropertyValidatorFactory` that you can use:

Property Validator Factories

```
@Bean
public PropertyValidatorFactory myValidatorFactory() {
  return new ClassBasedPropertyValidatorFactory(MyValidator.class);
}
```

Example 9.93. Providing a property validator factory

While the validator factory is a Spring bean, the validator instances are only simple POJOs. That means, that any Spring features of the validator class, like `@AutoWired` or `InitializingBean`, are not effective if a validator is instantiated by the ClassBasedPropertyValidatorFactory. Therefore, any mandatory configuration of a validator should be required as constructor arguments, and any state checks should be done in the constructor, in order to ensure a legal state of the validator.

If instantiating your validator is too complex to be expressed by Jackson annotations (for example, because it needs injections of unsupported types, or initialization methods must be invoked), you cannot simply use the `ClassBasedProp ertyValidatorFactory`, but you must implement a custom factory of type `PropertyValidatorFactory`. The `newInstance` method must return a property validator that is ready to use. The `configuration` map provides the parameters for the particular instance. Since the factory is a Spring bean, you can have injected any additional service beans you need to set up a property validator.

Custom Property Validator Factories

Finally, you provide a Json configuration file that declares concrete validators. The following Json declaration is equivalent to the above Spring bean declaration of a `MyValidator` validator for the `teaserTitle` property:

```
{
  "content-type-validator": {
    "teasable-validation": {
      "content-type": "CMTeasable",
      "subtypes": true,
      "property-validators": [
        {
          "my-validator": {
            "property": "teaserTitle"
          }
        }
      ]
    }
```

```
  }
}
```

Example 9.94. Declaring a property validator with Json

As you know already from the Spring bean configuration, property validators must be wrapped into content type validators. The outer map key `content-type-validator` denotes a predefined and provided factory to do this. The nested map key `teasable-validation` is the validator id. You might encounter it in log messages or exceptions, so you should choose a value that you can easily recognize. The three entries `content-type`, `subtypes` and `property-validators` constitute the configuration for the content type validator. The value of `property-validators` is a list of property validator configurations. A property validator configuration is a map with exactly one entry, whose key is the factory id for the property validator. The `ClassBased PropertyValidatorFactory`, that you use to create `MyValidator` instances, uses the lowercase/hyphen variant of the validator class as factory id, that is `my-validator`. The value of the map entry is another map which contains the configuration for the actual property validator. If you use the `ClassBasedPropertyValidatorFactory`, this map must contain at least values for all required constructor arguments, and optionally values for the other `@JsonProperty` annotated constructor arguments, setters or fields. `MyValidator` needs only the name of the property that is to be validated, `teaserTitle`.

Content Validators

If you want to validate a content as a whole, rather than a a single property value, you can provide a `CapTypeValidator`. You can implement it from scratch, or simply extend the `AbstractContentTypeValidator`, which leaves only the `validate(Content, Issues)` method to be implemented.

Implementation of Content Validators

```
public class MyContentValidator extends AbstractContentTypeValidator {
  private final SitesService sitesService;

  public MyContentValidator(@NonNull ContentType type,
                            boolean isValidatingSubtypes,
                            @NonNull SitesService sitesService) {
    super(type, isValidatingSubtypes);
    this.sitesService = sitesService;
  }

  @Override
  public void validate(Content content, Issues issues) {
    if (...) {
      issues.addIssue(Severity.ERROR, "myProperty", "myCode");
    }
  }
}
```

Example 9.95. Implementing a content validator

In this example, it is assumed that the validator needs the sites service for the validation. You can declare such a validator as a Spring bean:

Declaration as Spring Bean

```
@Bean
@ConditionalOnProperty(
  name = "validator.enabled.my-content-validator.cm-teasable",
  matchIfMissing = true)
CapTypeValidator myContentValidator(CapConnection con,
                                    SitesService sitesService) {
  ContentType type = con.getContentRepository().getContentType("CMTeasable");

  return new MyContentValidator(type, true, sitesService);
}
```

Example 9.96. Declaring a content validator as Spring bean

Just as for property validators, you should declare an application property to disable the validator. The name pattern is the same: the prefix `validator.en` `abled.`, followed by the lowercase/hyphen variant of the validator class and a short description.

Just like property validators, you can alternatively declare content validators by Json configuration files. This requires a factory for the validators. The validation framework provides the `ClassBasedCapTypeValidatorFactory` as a generic factory for Jackson-annotated content validators. For the `MyCon` `tentValidator` the annotations would look like this:

Declaration by Json

```
public class MyContentValidator extends AbstractContentTypeValidator {
  private final SitesService sitesService;

  @JsonCreator
  public MyContentValidator(
          @JsonProperty(value = "content-type", required = true) @NonNull
ContentType type,
          @JsonProperty(value = "subtypes") @Nullable Boolean
isValidatingSubtypes,
          @JacksonInject @NonNull SitesService sitesService)) {
    super(type, isValidatingSubtypes);
    this.sitesService = sitesService;
  }

  @Override
  public void validate(Content content, Issues issues) {
    if (...) {
      issues.addIssue(Severity.ERROR, "myProperty", "myCode");
    }
  }
}
```

Example 9.97. A Json-enabled content validator

`ClassBasedCapTypeValidatorFactory` has some more features compared to `ClassBasedPropertyValidatorFactory`. In addition to the simple types, you can also annotate `ContentType` arguments as `@JsonProp` `erty`. By convention, the Json field name of a `ContentType` argument is `content-type`. If you need the `SitesService` or the `CapConnection` to implement your validation logic, you can have them injected as `@Jackson`

Content Validator Factories

Inject annotated constructor arguments. The declaration of the factory looks like this:

```
@Bean
public CapTypeValidatorFactory myContentValidatorFactory(
        CapConnection connection, SitesService sitesService) {
  return new ClassBasedCapTypeValidatorFactory(
    MyContentValidator.class, connection, sitesService);
}
```

Example 9.98. Providing a content validator factory

Be aware, that validators instantiated by ClassBasedCapTypeValidator Factory are no Spring beans, but simple POJOs. Do not make use of Spring features, such as @Autowired or InitializingBean in your validator classes, but require any mandatory configuration as constructor arguments.

If the instantiation of your content validator is too complex to be expressed by Jackson annotations, you can provide a custom factory. It must implement the interface CapTypeValidatorFactory

Custom Content Validator Factories

The Json equivalent to the above Spring bean declaration of the validator looks like this:

```
{
  "my-content-validator": {
    "cm-teasable": {
      "content-type": "CMTeasable",
      "subtypes": true
    }
  }
}
```

Example 9.99. Declaring a content validator with Json

The factoryId my-content-validator is implied as the lowercase/hyphen variant of the validator class MyContentValidator. This validator can be disabled by setting the application property validator.enabled.my-content-validator.cm-teasable to false, just like the equivalent Spring bean validator.

Validators

If CapTypeValidator is still too specific, or you do not benefit from the features of AbstractCapTypeValidator, you can implement a Validat or. This interface is so generic that there is hardly more to say about it. Since it is rarely needed, CoreMedia does not provide any supporting convenience classes as for property validators or content validators.

You can provide validators as Spring beans or by Json configuration files. The possibility of Json configuration requires an according `ValidatorFactory`, which must be provided as a Spring bean. The factory pattern is the same as for property validators or content validators: The factoryId is used to reference the factory from the Json configuration, and the `newInstance` method is invoked with the innermost maps of the configuration. The configuration would look like this:

```
{
  "my-general-validator": {
    "an-instance": {
      "foo": "bar"
    }
    "another-instance": {
      "foo": "42"
    }
  }
}
```

Example 9.100. Declaring a general validator with Json

These general validators are technically decoupled from content validators and property validators. Therefore, configuration files for such validators have a different naming pattern: `validation/general/**/*-validator-configuration.json`.

9.23.1.4 Defining and Localizing Validator Messages

CoreMedia Studio ships with predefined validator messages for the built-in validators. The messages are defined in property files, following the idiom described in Section 5.6, "Localization" [83]. However, you might still want to add your own localized messages if you add custom validators or if you want to provide more specific message for individual properties.

To this end, you should start by adding a new set of `_properties.ts` files containing your localized messages. Make sure to add the base property file and an additional property file for each non-default language.

Augment the central validator property file with your own properties. The central property file is `Validators_properties.ts`, so that it can be updated as follows:

```
new CopyResourceBundleProperties({
  destination: resourceManager.getResourceBundle(null, Validators_properties),
```

```
  source: resourceManager.getResourceBundle(null, MyValidators_properties),
})
```

Example 9.101. Configuring validator messages

Now you can add localized message to the base property file and optionally to every language variant, using an appropriate translation.

There are three kinds of keys using the following schemes:

1. `Validator_<IssueCode>_text` is used as the generic message for the respective issue code.
2. `PropertyValidator_<PropertyName>_<IssueCode>_text` is used when the issue code appears for a property of a specific name.
3. `ContentValidator_<ContentType>_<PropertyName>_<IssueCode>_text` is used when the issue code appears for a property of a specific name for a content item with the given content type or any subtypes thereof. A localized message for a more specific content type takes precedence.

Generally, more specific settings take precedence over more general settings. For example `ContentValidator_*` keys take precedence over `Validator_*` keys, if applicable.

Each localized message may contain the substitution tokens `{0}`, `{1}`, and so on. Before being displayed, these tokens are replaced by the corresponding issue argument (counting from 0).

9.23.1.5 Tying Document Validation to Editor Actions

It is possible to tie the validation of a content item to editor actions via the `studio.validateBefore` property defined in `application.properties`. This property is to configure *Studio* to prevent certain activity on content items when they still contain errors. More specifically, you can specify that either checking in content or approving (and thus publishing) content will be not allowed in the presence of content errors. Setting the value of the `validateBefore` property to "CHECKIN" entails the check of both `Checkin` and Approve actions. Currently, the only supported options are "CHECKIN" or "APPROVE". Leaving the property value empty means that no such checks are imposed, and editors are allowed to check in, approve and publish even when content errors are detected.

9.23.2 Intercepting Write Requests

Write requests that have been issued by the client can be intercepted by custom procedures in the server. To this end, write interceptor objects can be configured in the Spring application context of the Studio Server. Typical use cases include:

- Setting initial property values right during content creation, ensuring that a completely empty content cannot be encountered even temporarily.
- Replacing the value to be written, for example, to automatically scale down an image to predefined maximum dimensions.
- Computing derived values, for example, to extract the dimensions (or other metadata) of an uploaded image and storing them in separate properties.

> **NOTE**
> Replacing values is not normally useful for text properties, because text values are saved continuously as the user enters data, and a write interceptor might not be able to operate appropriately during the first saves. For blobs or link lists, the impact on the user experience is typically less of a problem. In any case, when using interceptors, you need to make sure that the user experience is not impacted negatively.

9.23.2.1 Developing Write Interceptors

In order to process write requests as described above, create a class implementing the interface `ContentWriteInterceptor`. Alternatively, your class can also inherit from `ContentWriteInterceptorBase`, which already defines methods to configure the content type to which the write interceptor applies, and the priority at which the interceptor runs compared to other applicable interceptors.

This leaves only the method `intercept(ContentWriteRequest)` to be implemented in custom code. The argument of the `intercept` method provides access to all information needed for processing the current request, which is either an update request or a create request.

The method `getProperties()` of the `WriteRequest` object returns a mutable map from property names to values that represents the intended write request. Write interceptors can read this map to determine the desired changes. They may also modify the map (which includes the ability to add additional

Get values from write request

name/value pairs if required), thereby requesting modification of the original write request, and/or additional write operations. If multiple write interceptors run in succession, they see the effects of the previous interceptors' modifications in this map.

If a blob has been created in the write request by uploading a file via Studio, it is available as `UploadedBlob` in the properties of the `WriteRequest`, providing access to the original filename.

The method `getEntity()` returns the content on which an update request is being executed. A write interceptor may use this method to determine the context of a write request, for example to determine the site in which the content is placed in a multi-site setting or to determine the exact type of the content. Do not write to the content object. To modify the content, update the properties map as explained above.

Get content for request

The method `getEntity()` returns `null` for a create request, because a write interceptor is called before a content is created. So that the interceptor is able to respond to the context of a create request, the `ContentWriteRequest` object provides the methods `getParent()`, `getName()`, and `getType()`, which provide access to the folder, the name of the content item to be created, and the content type to be instantiated.

Finally, an issues object can be retrieved by calling `getIssues()`. This object functions as shown in Section 9.23.1, "Validators" [252]. In this context, it allows an interceptor to report problems observed in the write request. If a write interceptor reports any issues with error severity using the method `addIssue(...)` of the issues object, the write request will automatically be canceled and an error description will be shown at the client side. If issues of severity warn are detected, the write is executed, but a message box is still shown. In any case, the issues are not persisted, so that the only issues shown for a content permanently are the issues computed by the regular validators.

Reporting issues

If a write interceptor reports an error issue the write request is canceled but the whole chain of interceptors is still executed. To stop the interceptor chain immediately without further interceptor execution a write interceptor can throw an `InterceptionAbortedException` which is caught during interceptor iteration. In this case a new issue with severity error is created and added to the issues instance of the given write request. Currently only the `PictureUpload Interceptor` throws this exception if the picture to upload is too large and exceeds a given image size limit configured with the `uploadLimit` interceptor property in the Spring bean configuration. This reduces the possibility the Java virtual machine runs out of memory during image blob transformations.

Abort interceptor chain execution

The following example shows the basic structure of a custom interceptor for images. A field for the name of the affected blob property is provided. The `in`

tercept() method checks whether the indicated property is updated, retrieves the new value and provides a replacement value using the properties map.

```
public class MyInterceptor extends ContentWriteInterceptorBase {
  private String imageProperty;

  public void setImageProperty(String imageProperty) {
    this.imageProperty = imageProperty;
  }

  public void intercept(ContentWriteRequest request) {
    Map<String,Object> properties = request.getProperties();
    if (properties.containsKey(imageProperty)) {
      Object value = properties.get(imageProperty);
      if (value instanceof Blob) {
        ...
        properties.put(imageProperty, updatedValue);
      }
    }
  }
}
```

Example 9.102. Defining a Write Interceptor

9.23.2.2 Configuring Write Interceptors

A write interceptor is enabled by simply defining a bean in the Spring application context of the Studio web application. The interception framework automatically collects all interceptor beans and applies them in order whenever an update is requested. Interceptors with numerically lower priorities are executed first.

Enabling the interceptor

For a write interceptor implemented using the class `ContentWriteInterceptorBase`, the priority is configured through the `priority` property. Such interceptors also provide the property `type`, indicating that an interceptor should only run for instances of specific content types. While the setter `setType()` receives a `ContentType` parameter, it is possible to simply provide the content type name as a string in the Spring bean definition file. The type name will be automatically converted to a `ContentType` object.

Priority of interceptor

Furthermore, you need to configure whether the interceptor also applies to instances of subtypes of the given type through the property `isInterceptingSubtypes`. Like for validators, this property defaults to `false`, meaning that interception applies only to content items of the exact type.

Each write interceptor may also introduce additional configuration options of its own.

A typical definition might look like this:

```
@Bean
MyInterceptor myInterceptor() {
  MyInterceptor myInterceptor = new MyInterceptor();
```

```
  myInterceptor.setType("CMPicture");
  myInterceptor.setImageProperty("data");
  return myInterceptor;
}
```

Example 9.103. Configuring a Write Interceptor

9.23.3 Immediate Validation

Write requests that violate hard constraints of your content type model can be aborted when a validator fails. Typical use cases include:

- Preventing a client from uploading an image that is too large.

- Making sure that a content item does not link to itself directly.

> **CAUTION**
>
> Blocking writes is not normally useful for text properties, because text values are saved continuously as the user enters data, and a write interceptor might not be able to operate appropriately during the first saves. For blobs or link lists, the impact on the user experience is typically less of a problem. In any case, you need to make sure that the user experience is not impacted negatively.

For implementing immediate validation, you can create an instance of the class `ValidatingContentWriteInterceptor` as a Spring bean and populate its `validators` property with a list of `PropertyValidator` objects. When the validators are configured to report an error issue, an offending write will not be executed (that is, the requested value will not be saved).

A configuration that limits the size of images in the `data` property of `CMPicture` content items to 1 Mbyte might look like this (class names are wrapped for layout reasons):

```
@Bean
ValidatingContentWriteInterceptor
  myValidatingContentWriteInterceptor(MaxBlobSizeValidator
myMaxBlobSizeValidator) {

  ValidatingContentWriteInterceptor validatingContentWriteInterceptor =
    new ValidatingContentWriteInterceptor();
  validatingContentWriteInterceptor.setType("CMPicture");
  validatingContentWriteInterceptor.setValidators(
    Collections.singletonList(myMaxBlobSizeValidator)));
  return validatingContentWriteInterceptor;
}

@Bean
MaxBlobSizeValidator myMaxBlobSizeValidator() {
```

```
  MaxBlobSizeValidator maxBlobSizeValidator =
    new MaxBlobSizeValidator();
  maxBlobSizeValidator.setProperty("data");
  maxBlobSizeValidator.setMaxSize(1000000);
  return maxBlobSizeValidator;
}
```

Example 9.104. Configuring Immediate Validation

Remember that the validators become active during creation, too, so that an immediate validator might validate initial values set by an earlier write interceptor.

9.23.4 Post-processing Write Requests

Write requests that have been executed by the server can be post processed by custom procedures. To this end, write post-processor objects can be configured in the Spring application context of the Studio Server.

In most cases, a write interceptor is better suited for reacting to update requests, because an interceptor can still block an update completely and because it is more efficient to make sure that the right value are written immediately. But especially during content creation it might be necessary to create links to the generated content, which would be impossible before the content has actually been created.

> **NOTE**
>
> Note that post-processors are not executed atomically with the actual write, so that the write is persisted even if a post-processor exits with an exception.

9.23.4.1 Developing Write Post-processors

In order to post process write requests as described above, create a class implementing the interface `ContentWritePostprocessor`. Alternatively, your class can also inherit from `ContentWritePostprocessorBase`, which already defines methods to configure the content type to which the write interceptor applies, and the priority at which the interceptor runs compared to other applicable interceptors.

This leaves only the method `postProcess(WriteReport<Content>)` to be implemented in custom code. The argument of the `postProcess` method provides access to all information needed for post processing the current request, which is either an update request or a create request.

The method `getEntity()` returns the content on which an update request has been executed. A write interceptor may use this method to determine the context of a write request.

The method `getOverwrittenProperties()` of the `WriteReport` object returns a map from property names to the values that have been overwritten during the write request. The new values can be retrieved as the current property value of the content returned from the method `getEntity()`.

9.23.4.2 Configuring Write Post-processors

A write post-processor is enabled by simply defining a bean in the Spring application context of the Studio web application. The interceptor framework automatically collects all post-processor beans and applies them in order whenever an update is requested. Post-processors with numerically lower priorities are executed first.

For a write post-processor implemented using the class `ContentWritePost processorBase`, the priority is configured through the `priority` property. Such post-processors also provide the property `type`, indicating that a post-processor should only run for instances of specific content types.

Priority of post-processor

Furthermore, you need to configure whether the post-processor also applies to instances of subtypes of the given type through the property `isPostpro cessingSubtypes`. Like for validators, this property defaults to `false`, meaning that post-processing applies only to content items of the exact type.

Each write post-processor may also introduce additional configuration options of its own.

9.24 Available Locales

As the `locale` property of a content item is just a plain string property, *Core-Media Studio* provides assistance with setting the locales and keeping them consistent.

For this purpose a special content item is maintained that stores a list of language tags. These tags are used to restrict the selectable locales when cloning a site or setting a content item's `locale` property. To this end a new property field called `AvailableLocalesPropertyField` is used in the Blueprint content forms, which displays the available locales as a combo box.

The locales are rendered to the user in a readable representation that is localized for the current *Studio* language. The property field can also be configured to show an empty entry that sets the field value to the empty string.

When editing the list of available locales a validator will warn you if a language tag does not match the *BCP 47* standard (http://www.rfc-edit-or.org/rfc/bcp/bcp47.txt) and it will show an error if a language tag is defined multiple times.

The content item and property storing the locales can be configured with the following two Spring configuration properties:

```
available-locales.content-path=/Settings/Options/Settings/LocaleSettings
available-locales.property-path=settings.availableLocales
```

9.25 Toasts and Notifications

9.25.1 Configure Notifications

By default, the amount of notifications requested by *Studio* is limited to 20. This value is customizable via the Spring property `notifications.limit`. The property can be overwritten in the `application.properties` file of the *Studio* application or any other Spring `properties` file that is loaded for the *Studio* context.

9.25.2 Adding Custom Notifications

On several occasions, *CoreMedia Studio* shows notifications (see also Section 2.7, "Notifications" in *Studio User Manual*). It is easily possible to add your own custom notifications to *CoreMedia Studio*. In the following the necessary steps are described.

For your server-side module where you want to create a notification, make sure you add a Maven dependency on `notification-api`. This module contains the `NotificationService` API.

Also, make sure that your application as a whole has a Maven dependency on `com.coremedia.cms:notification-elastic`. This module contains an implementation of the `NotificationService`, which again is either based on *Elastic Core* for persistence, or on an SQL persistence layer. For the Blueprint Studio application this is already taken care of by the extension module `notification-elastic-studio-lib`.

Finally, take care of declaring a `NotificationService` Spring bean, either via a component scan or an explicit declaration.

For the Studio client side, you have to add the dependency `@coremedia/studio-client.main.notification-studio-client` to the package where you want to develop new notification UIs.

9.25.3 Creating Notifications (Server Side)

To create notifications on the server side, simply inject the `NotificationService` and use it at the appropriate position (event/request handler, REST

method, task etc.) to create a new notification with the method `createNotification`:

```
Notification createNotification(@NonNull String type,
                                @NonNull Object recipient,
                                @NonNull String key,
                                @Nullable List<Object> parameters);
```

A notification always has a combination of `type` and `key`. The key is basically a subtype and will be used to determine the correct localization text key on the client side. An example of a type / key combination is "publicationWorkflow" / "offered".

A notification has a `recipient`. This parameter is typed as `Object`. For Studio notifications, it has to be a `User` object.

Additional `parameters` will be used on the client side to parametrize the no-tification's text. In advanced cases they are additionally used to configure actions and customize the notification's UI. Details are explained below.

9.25.4 Displaying Notifications (Client Side)

For displaying notifications in *CoreMedia Studio*, three levels are distinguished:

1. Simply displaying the notification in terms of a text message and an icon. For example, the notification might inform the user that a new publication workflow has arrived in its inbox.

2. The same as in 1 but with an additional click action handler. For example, clicking the publication workflow notification might open the publication workflow inbox in the Studio Control Room.

3. Completely customizing the display and controls of the notification.

Levels 1 and 2 are considered as the typical cases for displaying notifications. For these, CoreMedia offers default components. However, in certain cases it might be necessary or desired to develop a more refined notification UI.

Level 1: Simple Notification Display

For just displaying a notification in terms of an icon, and a text message, you simply have to provide an icon class property, and a text key property. These properties must match the patterns `Notification_{notificationType}_iconCls` and `Notification_{notificationType}_{notificationKey}_msg` respectively. For the example of a publication workflow notification from above, the properties look as follows:

```
Notification_publicationWorkflow_iconCls :
CollaborationIcons_properties.start_publication_workflow,
Notification_publicationWorkflow_offered_msg : "The publication workflow
\"{0}\" is new in your inbox."
```

In this example, the message property has a placeholder. By default, the `para meters` of the notification (see notification creation above) are inserted in the placeholders one after the other. Consequently, the parameters have to be of type string. However, it is also possible to compute the placeholder insertions from the notification's `parameters` (for example, if you have a complex bean as a parameter that should be the basis for all placeholder insertions). In this case your notification's Studio component (see below) has to implement the interface `TextParametersPreProcessor`.

You define your properties in your own resource bundle (`WorkflowNotific ations_properties.ts`, for instance) and have to make sure to copy it onto the resource bundle `Notifications_properties.ts` which is provided by CoreMedia:

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import resourceManager from "@jangaroo/runtime/l10n/resourceManager";
import CopyResourceBundleProperties from
"@coremedia/studio-client.main.editor-components/configuration/CopyResourceBundleProperties";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import Notifications_properties from
"@coremedia/studio-client.main.notification-studio-client/Notifications_properties";
import WorkflowNotifications_properties from
"./WorkflowNotifications_properties";

class MyStudioPlugin extends StudioPlugin {

  constructor(config:Config<StudioPlugin>){
    super(ConfigUtils.apply(Config(MyStudioPlugin, {
      rules: [
      ],

      configuration: [
        new CopyResourceBundleProperties({
          destination: resourceManager.getResourceBundle(null,
Notifications_properties),
          source: resourceManager.getResourceBundle(null,
WorkflowNotifications_properties),
        }),
      ],
    }), config));
  }
}

export default MyStudioPlugin;
```

Level 2: Simple Notification Display with Click Action

In many cases it is not enough to just display a notification. Normally, a notification is a request to the user to do something. So it should be possible to click the notification and be directed to the part of Studio where the user can do something about it.

In order to add an action click handler to your notification, you have to register your own notification component. You always register a notification component for a specific notification type:

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";

import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import PublicationWorkflowNotificationDetails from
"@coremedia/studio-client.main.control-room-editor-components/notification/components/PublicationWorkflowNotificationDetails";
import RegisterNotificationDetailsPlugin from
"@coremedia/studio-client.main.notification-studio-client/RegisterNotificationDetailsPlugin";

class MyStudioPlugin extends StudioPlugin {

  constructor(config: Config<StudioPlugin>) {
    super(ConfigUtils.apply(Config(MyStudioPlugin, {

      rules: [],

      configuration: [
        new RegisterNotificationDetailsPlugin({
          notificationType: "publicationWorkflow",
          notificationDetailsComponentConfig:
Config(PublicationWorkflowNotificationDetails),
        }),
      ],
    }), config));
  }
}

export default MyStudioPlugin;
```

You do not have to do any component developing for level 2. You can simply let your notification component extend `DefaultNotificationDetails` and add your notification action as its `baseAction`. You need to let your action extend `NotificationAction`. This yields numerous benefits like accessing the notification via the method `NotificationAction.getNotification()`. Consequently, you also have access to all the notification's parameters.

```
import { mixin } from "@jangaroo/runtime";
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import DefaultNotificationDetails from
"@coremedia/studio-client.main.notification-studio-client/components/DefaultNotificationDetails";
import TextParametersPreProcessor from
"@coremedia/studio-client.main.notification-studio-client/components/TextParametersPreProcessor";
import MyNotificationAction from "./MyNotificationAction";

class CustomNotificationDetails extends DefaultNotificationDetails implements
 TextParametersPreProcessor {

  constructor(config: Config<DefaultNotificationDetails>){
    super((()=> ConfigUtils.apply(Config(CustomNotificationDetails, {
      baseAction: new MyNotificationAction({}),
    }), config))());
  }

  preProcessTextParameters(params: Array<any>): Array<any> {
    return params;
  }
}

mixin(CustomNotificationDetails, TextParametersPreProcessor);
```

```
export default CustomNotificationDetails;
```

Level 3: Custom Notification Display

You are free to develop your own notification component that does not inherit from `DefaultNotificationDetails`. CoreMedia gives no further guidelines here but point out that your component at least has to inherit from `Notification Details`. You register your custom component just as it was described above.

9.25.5 Displaying Toasts

Toasts provide feedback, which is triggered by user interaction. Toasts always appear at the bottom left of the screen and disappear automatically after six seconds, but can be disabled in the user preferences dialog. Unlike notifications, they cannot be customized. A toast contains a title, a text and has one of the following states: INFO, SUCCESS, WARN or ERROR.

The given code example shows examples how the `ToastManager` can be used to display different types of toast messages.

```
import ToastsManager from
"@coremedia/studio-client.ext.toast-components/ToastsManager";
import ValidationState from
"@coremedia/studio-client.ext.ui-components/mixins/ValidationState";

//Example: information toast
ToastsManager.getInstance().showToastMessage("Hello", "This is a simple
message.", null);

//Example: success toast
ToastsManager.getInstance().showToastMessage("Done!", "The job finished
successfully.", ValidationState.SUCCESS);

//Example: warning toast
ToastsManager.getInstance().showToastMessage("Hint", "Maybe this will not
work.", ValidationState.WARN);

//Example: error toast
ToastsManager.getInstance().showToastMessage("Ups", "Something went wrong.",
 ValidationState.ERROR);
```

9.26 Annotated LinkLists

Every link in a list of links can be enhanced with additional settings – so called `Link Annotations`. These `Link Annotations` are stored together with the actual link in a `struct`. Link lists enhanced with `Link Annotations` are called `Annotated Link Lists`.

9.26.1 Studio Configuration

`Annotated LinkLists` are stored in a `struct` property. This is in contrast to a plain `LinkList` which is stored in a `LinkList` property. Therefore, when introducing a new `Annotated LinkList`, the doctype definition needs to have an XML property with the Struct grammar.

```
<XmlProperty Name="structList" Grammar="coremedia-struct-2008" extensions:translatable="true"/>
```

The property editor for an annotated LinkList usually is a `LinkListPropertyField` with the following configuration:

- linkListWrapper: An instance of `StructLinkListWrapper`, that wraps the annotated list
- rowWidget: An `AnnotatedLinkListWidget` that contains items which implement `IAnnotatedLinkListForm`

Existing `Annotated LinkLists` can be extended with custom forms by using the `AddItemsPlugin` on the `AnnotatedLinkListWidget`.

The custom forms need to implement the interface `IAnnotatedLinkListForm` (which basically is providing a `settingsVE` configuration) and can then start using property editors bound to sub properties of the `settingsVE` `ValueExpression` via `extendBy`.

> **NOTE**
>
> Like every property editor the annotated link list form should consider the read-only state. For this our default property editors always provide the config `forceReadOnlyValueExpression`. Either implement this manually or utilize a base component like `PropertyFieldGroup` (see example below).

The row expander of an `Annotated LinkList` changes its appearance based upon the state of its row widget(s) (see Figure 9.9, " Annotated LinkList with item with changed default value " [279], the changed parts are highlighted

with red border). That is, if there is at least one row widget instance in a row which differs from the default state, the row expander icon gets inverted. So the studio user gets a hint that at least one of the row widget instances has been changed.
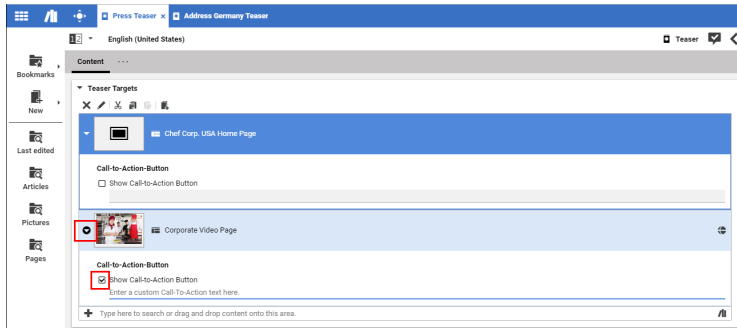
Figure 9.9. Annotated LinkList with item with changed default value

To determine if a row widget differs from its default state, every `IAnnotated LinkListForm` may provide a custom method with the following signature: `isAnnotated(annotatedLinkListProvider:IAnnotatedLink ListProvider, rowIndex:number):boolean`

These custom methods are set via the `LinkListPropertyField` config option `rowWidgetsAnnotatedPredicates`. If there are no custom methods, a default strategy is chosen to determine if the row expander has to change its appearance. If there is at least one custom method, then the default strategy is ignored.

9.26.1.1 Examples

The following example shows how to add an annotated link list form to an already existing annotated link list (for further annotations). The `ExampleAnnotated LinkListForm` is based on `PropertyFieldGroup`, implements the `IAn notatedLinkListForm` interface and shows a simple form containing an "Special Feature Enabled" `CheckBox`.

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import { mixin } from "@jangaroo/runtime";
import ValueExpression from "@coremedia/studio-client.client-core/data/ValueExpression";
import IAnnotatedLinkListForm from
"@coremedia/studio-client.ext.ui-components/components/IAnnotatedLinkListForm";
import StatefulCheckbox from "@coremedia/studio-client.ext.ui-components/components/StatefulCheckbox";
import BindPropertyPlugin from "@coremedia/studio-client.ext.ui-components/plugins/BindPropertyPlugin";
import PropertyFieldGroup from
"@coremedia/studio-client.main.editor-components/sdk/premular/PropertyFieldGroup";
```

```
interface ExampleAnnotatedLinkListFormConfig extends Config<PropertyFieldGroup>,
Partial<Pick<ExampleAnnotatedLinkListForm,
  "enabledPropertyName" |
  "settingsVE"
  >> {
}

class ExampleAnnotatedLinkListForm extends PropertyFieldGroup implements IAnnotatedLinkListForm {
  declare Config: ExampleAnnotatedLinkListFormConfig;

  /** the property of the Bean to bind in this field */
  enabledPropertyName:string;

  settingsVE:ValueExpression;

  constructor(config:Config<ExampleAnnotatedLinkListForm> = null){
    super(ConfigUtils.apply(Config(ExampleAnnotatedLinkListForm, {
      items: [
        Config(StatefulCheckbox, {
          boxLabel: "Special Feature Enabled",
          plugins: [
            Config(BindPropertyPlugin, {
              bidirectional: true,
              bindTo: config.settingsVE.extendBy(config.enabledPropertyName || "enabled"),
            }),
          ],
        }),
      ],
    }), config));
  }
}
mixin(ExampleAnnotatedLinkListForm, IAnnotatedLinkListForm);

export default ExampleAnnotatedLinkListForm;
```

The next example shows how to add this `ExampleAnnotatedLinkListForm` to an existing annotated link list. The underlying `PropertyFieldGroup` requires setting the `bindTo`, `forceReadOnlyValueExpression` and `itemId` config.

```
import Config from "@jangaroo/runtime/Config";
import LinkListPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/LinkListPropertyField";
import AnnotatedLinkListWidget from
"@coremedia/studio-client.ext.ui-components/components/AnnotatedLinkListWidget";
import CMTeaserForm from "@coremedia-blueprint/studio-client.main.blueprint-forms/forms/CMTeaserForm";
import ExampleAnnotatedLinkListForm from "./ExampleAnnotatedLinkListForm";

//...
Config(LinkListPropertyField, {
  //...
  rowWidget: Config(AnnotatedLinkListWidget, {
    itemId: CMTeaserForm.TARGET_ANNOTATION_WIDGET_ITEM_ID,
    items: [
      Config(ExampleAnnotatedLinkListForm, {
        bindTo: config.bindTo,
        forceReadOnlyValueExpression: config.forceReadOnlyValueExpression,
        itemId: "exampleAnnotation",
      }),
    ],
  }),
})
```

9.26.2 Data Migration

To convert an existing LinkList property (Doctype property: LinkListProperty) to a new Annotated LinkList property (Doctype property: XmlProperty), functionality for migration is provided and can to be adapted to migrate custom properties.

The migration is performed on demand, that is, when the Annotated LinkList is edited. This migration approach does not migrate data as a preparation step, but is performed ongoing during write requests in normal operation mode.

Data migration from a legacy linkList property `linkList` to a struct linkList property `structList` is performed as follows:

- The new XMLProperty needs to be added in the doctype definition. Then the doctype definition contains the legacy and the new property.

```
<LinkListProperty Name="linkList" Max="1" LinkType="CMLinkable"/>
<XmlProperty Name="structList" Grammar="coremedia-struct-2008" extensions:translatable="true"/>
```

- Configure a Spring bean for *CoreMedia Studio* of type `LegacyToAnnotatedLinkListAdapter` with appropriate properties.

```
@Bean
LegacyToAnnotatedLinkListAdapter customAnnotatedLinkListAdapter(ContentRepository contentRepository,
 CapConnection connection) {
    ContentType cmTeaser = contentRepository.getContentType("CMTeaser");

    LegacyToAnnotatedLinkListAdapter customAdapter = new LegacyToAnnotatedLinkListAdapter();
    customAdapter.setType(cmTeaser);
    customAdapter.setProperty("structList");
    customAdapter.setLegacyProperty("linkList");
    customAdapter.setPriority(0);
    customAdapter.setInterceptingSubtypes(true);
    customAdapter.setConnection(connection);
    return customAdapter;
}
```

- If custom properties need to be adapted, extend the `LegacyToAnnotatedLinkListAdapter` and configure this bean instead.

```
public class CustomAnnotatedLinkListAdapter extends LegacyToAnnotatedLinkListAdapter {

    @Override
    protected void populateTargetStruct(Content target, int index, StructBuilder builder, CapObject
 capObject) {
        super.populateTargetStruct(target, index, builder, capObject);
        //custom code
    }

    @Override
    protected void cleanupLegacyData(ContentWriteRequest request) {
        super.cleanupLegacyData(request);
        //custom code
    }
```

```
}
```

- Override `LegacyToAnnotatedLinkListAdapter#populateTarget Struct` to apply custom data to the struct list. For example, retrieve a setting from localSettings and apply it to the struct:

```
@Override
protected void populateTargetStruct(Content target, int index, StructBuilder builder, CapObject
capObject) {
    super.populateTargetStruct(target, index, builder, capObject);
    Struct localSettings = capObject.getStruct("localSettings");
    if (!isEmpty(localSettings)) {
        Boolean setting = getBoolean_(localSettings, CUSTOM_SETTING);
        builder.declareBoolean(ANNOTATED_LINK_STRUCT_CUSTOM_PROPERTY_NAME, setting)
    }
}
```

- If required, legacy data can be cleaned up by overriding `LegacyToAnnot atedLinkListAdapter#cleanupLegacyData`. For example, remove a setting from localSettings, that is stored in the struct now:

```
@Override
protected void cleanupLegacyData(ContentWriteRequest request) {
    super.cleanupLegacyData(request);
    Content entity = request.getEntity();
    Map<String, Object> properties = request.getProperties();
    Struct localSettings = entity.getStruct("localSettings");
    if (localSettings != null && localSettings.get(CUSTOM_SETTING) != null) {
        StructBuilder structBuilder = localSettings.builder();
        structBuilder.remove(CUSTOM_SETTING);
        properties.put("localSettings", structBuilder.build());
    }
}
```

- Then, automatic migration of the legacy property `linkList` will be done automatically on demand, that is on write access. As soon as `structList` is written (via the Studio Server), the former `linkList` and configurable properties are stored in the new struct property `structList`.

- As long as the `structList` property has not been written yet, read access on `structList` (via the Studio Server) will return the `linkList` as a Struct linkList.

- After migration, read access on `structList` will return the struct linkList directly.

- If the legacy `linkList` property is a weak link, then the `structList` property will lose this feature.

- Note: The legacy property `linkList` is still supported but it cannot be used alongside the new property `structList`.

- Note: The linklist property and the struct linklist property are different properties, the source linklist property cannot be reused with this mechanism.

9.27 Thumbnails
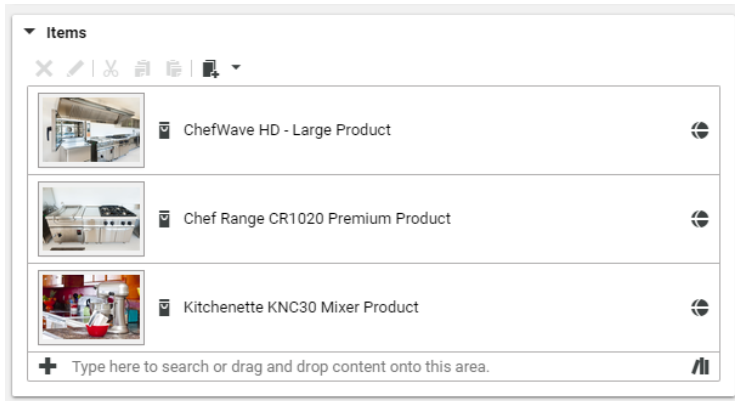
Figure 9.10. Thumbnails

Thumbnails can be found in different parts of the Studio for various items. For example content, commerce or content hub items. Components like the link list property editor have the option to show a thumbnail as a preview of the linked content. Enable the thumbnail by setting the component property `showThumbnails` to `true`.

```
    import Config from "@jangaroo/runtime/Config";
import LinkListPropertyField from
"@coremedia/studio-client.main.editor-components/sdk/premular/fields/LinkListPropertyField";

//...
Config(LinkListPropertyField, { propertyName: config.propertyName, showThumbnails: true })
```

9.27.1 Thumbnail Resolvers

> **Deprecation of Thumbnail Resolvers**
>
> `ThumbnailResolvers` have been deprecated. Instead of resolving a thumbnail for an item on the client, the calculation has been moved to the server. Find more details about this in Section 9.27.3, "Default Pictures" [286].

Thumbnail resolving is implemented in instances of `ThumbnailResolver`. `ThumbnailResolver` instances are registered with the `editorContext` and configured with a content type name and property names.

A default configuration for thumbnail resolvers can be found in the class `BlueprintFormsStudioPluginBase.ts`.

Default configuration

A `ThumbnailResolver` is selected when the configured content type matches the content type of the content in the link list. Thumbnail resolvers respect the content type inheritance. For example, the thumbnail resolver for `CMTeasable` will also be used for content of type `CMArticle`, since `CMArticle` is a subtype of `CMTeasable`.

Select thumbnail resolver

The properties are evaluated in the configured order. If one of the configured properties contains an image blob, the corresponding thumbnail URL is returned. If the property is a link list, a matching `ThumbnailResolver` is looked up for the first content of this list and the search for the thumbnail blob goes on. If no blob is found, a default icon is shown.

Property evaluation

As in the example below a notation like `localSettings.thumbnail` is supported. The example points to a link property `thumbnail` which is a child of the struct property `localSettings`.

```
        import ThumbnailResolverFactory from
"@coremedia/studio-client.ext.cap-base-components/thumbnails/ThumbnailResolverFactory";
import editorContext from "@coremedia/studio-client.main.editor-components/sdk/editorContext";

//...
editorContext._.registerThumbnailResolver(
        ThumbnailResolverFactory.create("CMCollection", "pictures", "items"));
editorContext._.registerThumbnailResolver(
        ThumbnailResolverFactory.create("CMTeasable", "pictures", "localSettings.thumbnail"));
editorContext._.registerThumbnailResolver(
        ThumbnailResolverFactory.create("CMPicture", "data"));
```

Example 9.105. Example thumbnail resolver configuration

The configuration above could be applied as follows:

Example 1: Link list contains `CMPicture` content

- When the `data` property of the `CMPicture` content contains an image, then this image is used to render the thumbnail.

Example 2: Link list contains `CMCollection` content

- When a content item inside a link list is a `CMCollection` content item, use the properties `items` and `pictures` and check the first item of these link list properties.

- When this linked item is an instance of `CMTeasable`, use the `pictures` property to look up the content that contains the thumbnail.

- Finally, when the call stack arrives at an instance of `CMPicture`, use the `data` blob property to render the thumbnail.

9.27.2 Custom Thumbnail Resolvers

In some cases the thumbnail that should be rendered for a linked content should point to an external system. For example, when you have a content type that represents an asset of another system, you can use the asset preview URL (if provided) to render the thumbnail with a custom thumbnail resolver.

A custom `ThumbnailResolver` instance can be registered to the `editor Context`:

```
editorContext._.registerThumbnailResolver(new MyCustomResolver());
```

9.27.3 Default Pictures

For numerous `RemoteBean` entities of the *CoreMedia Studio*, default pictures can be resolved in terms of their picture blob, their picture URL or both.

Default pictures are calculated on the *CoreMedia Studio* server and replace the existing `ThumbnailResolver` implementation of the client. This section describes how the resolving can be customized and accessed by the client.

Default Picture Resolving

To support a thumbnail for a *Studio* `RemoteBean` entity, the server has to provide an implementation of `DefaultPictureResolver` for it. The product already supports these resolvers for most entities of the *Studio*, including a default implementation for content items.

The interface comes with the additional default method `default int get Order()`. This allows to re-implement an existing resolver and prefer it over the default implementation.

While `DefaultPictureResolver` is the interface to implement for any kind of entity, you can also implement the more specialized interface `ContentDefaultPictureResolver` to resolve default pictures for content items. The interface includes the methods `getContentType()` and `includeSubTypes` to specify for which content items of which type the resolver applies.

Both `DefaultPictureResolver` and `ContentDefaultPictureResolver` also come in a flavour where a `DefaultPictureService` is passed into their `resolve()` method. The service can be used to resolve further pictures along the way to finally obtain the picture in question. For example, while resolving the default picture for a `Process`, one of its content items is chosen and the process picture is then resolved as the default picture of this content item.

The thumbnail resolving for the content type model can be customized in the `application.properties` of the *Studio* server. Additional entries can be added there, using the format `studio.defaultPicture.content.paths.<DOCTYPE_NAME>=<VALUE>`, where `DOCTYPE_NAME` matches the name of the content type and `VALUE` the name of the mapped content property that contains the picture blob or references another content item which contains the blob (or again another content item).

```
studio.default-picture.content.paths.CMVideo=pictures
studio.defaultPicture.content.paths.CMPicture=data
```

Example 9.106. Example content thumbnail resolver configuration

In this example the thumbnail for a CMVideo is looked up in the `pictures` link list. The list itself contains `CMPicture` content items which have their blob stored in the property `data`. Note that this path notation may also point to `Struct` properties, e.g. `localSettings.thumbnail`.

Access Thumbnails on the Studio Client

Thumbnails for the *Studio* client RemoteBeans are resolved through the `thumbnailService`. The service provides the following methods for this:

- `getThumbnailUri(model: any, operations: string = null): string` This is the common service method to resolve a thumbnail URL with optional additional image operations for the given model. The method returns the default picture URL if the entity implements `WithDefaultPicture` and the server logic is able to resolve a picture for it.

- `getThumbnail(model: any, operations: string = null): Thumbnail | string` If available, this method returns the calculated

`Thumbnail` object. In some cases, the client wants to know details about the image data, like the mime type or the blob size. If only the URL information is available, the URL `string` is returned instead.

9.28 Custom Workflows

This section describes the necessary steps to add new workflows to *Studio*. It is assumed that the corresponding new workflow definitions have already been added to the *Workflow Server* (see Workflow Manual). Certain specific require‐ ments concerning the workflow definition are covered when discussing different topics throughout the chapter.

Currently, CoreMedia offers support for publication and localization (with sub‐ types language translation and synchronization) workflows. Many topics of workflow customization concern both workflow types and are covered together in the first sections. Publication‐/translation‐specific customizations are covered in distinct sections afterward.

All customizations are done in the context of *Blueprint* extensions for your *Studio* server and client apps. For the client, both the *Main App* and the *Workflow App* need to be taken into consideration. But in general one shared customization module for both client apps is sufficient.

Examples of custom workflow configurations that apply the options described in this chapter can be found in the *CoreMedia* GitHub repositories for Additional Publication Workflows and for the Global Link Translation Workflow.

9.28.1 Fundamentals

This section describes the most basic steps to make new workflows known to *Studio*.

Studio Server and User Changes Application

For the *Studio* server, two basic customizations are possible.

Defining the Workflow Category

You have to define the process category of your workflow, either localization or publication. You have two ways to do so:

1. Let the name of your workflow definition contain either `Translation` or `Publication`.

2. Add the name of your new workflow definition to the `translationPro cessNames` or `publicationProcessNames` beans for the corresponding

workflow category. Example 9.107, "Add a new workflow with the name Studio-ThreeStepPublication to publicationProcessNames " [290] shows this for a new 3-step publication workflow.

```
@Bean
@Customize("publicationProcessNames")
List<String> addThreeStepPublicationWorkflowName() {
  return List.of("StudioThreeStepPublication");
}
```

Example 9.107. Add a new workflow with the name StudioThreeStepPublication to publicationProcessNames

Enabling Notifications for Tasks

You can switch on *Studio* notifications for tasks of your new workflow when they appear in the *Control Room* or *Workflow App* inbox. You do this via a Spring Java configuration in the application context of the Spring Boot app that acts as the *User Changes Application*. This can be the Studio server app itself (for example, in the in-memory setup), but typically it is the dedicated *User Changes Application*.

Customize the beans `notificationsForTranslationWorkflowList` or `notificationsForPublicationWorkflowList` for a translation or publication workflow, respectively. Example 9.108, "Enable notifications for new StudioThreeStepPublication workflow " [290] shows this for a new 3-step publication workflow.

```
@Bean
@Customize("notificationsForPublicationWorkflowList")
List<String> addThreeStepPublicationWorkflowNotifications() {
  return List.of("StudioThreeStepPublication");
}
```

Example 9.108. Enable notifications for new StudioThreeStepPublication workflow

Studio client

For the Studio client it is important to note that both the *Main App* and the *Workflow App* need to be taken into consideration. For the time being, workflows are still started in the *Control Room* of the *Main App*, but running workflows can only be displayed in the *Workflow App*. However, it is typically sufficient to develop one shared module for a workflow customization and add it as a dependency to both client apps (see Section 9.1, "General Remarks On Customizing (Multiple) Studio Apps" [120]).

Customizing workflows for the Studio client involves no Ext JS, so no `StudioPlu gin` or `StudioStartupPlugin` is needed. Instead, an `autoLoad` entry for the custom workflow module is the way to go (see section "Customization Entry points" [122]). In the autoloaded script, the global constant `workflowPlugins` is used to add workflow plugins. In the example below, a translation workflow is added. A corresponding method `addPublicationWorkflow` also exists.

```
workflowPlugins._.addTranslationWorkflowPlugin({
  workflowName: "MyCustomTranslation",
});
```

Example 9.109. Minimal Studio client enabling of a custom translation workflow

This is the minimal configuration needed to make a custom workflow known to the Studio client. In the example, `addTranslationWorkflowPlugin` is implicitly called with `any` for the model type parameter. A specific type will be needed once additional workflow form fields are configured (see below).

9.28.2 Workflow Steps

Most workflows have several steps to go through. They, of course, need to be configured for the process definition of a custom workflow, but in addition, some of them need to be configured for the Studio client.

In the context of process definitions, the term "workflow step" does not exist. Only tasks (possibly with entry and exit actions) can be defined. The term "workflow step" is used from a Studio client perspective in this section. Workflow steps are of course closely tied to workflow tasks, but they are not synonymous.

In a nutshell, a workflow step needs to be configured for the Studio client part of a custom workflow whenever the user is required to decide between several options of how the workflow should proceed. The result of this decision might in some cases directly be the follow-up task. In other cases the result might be a value that is set to a process variable, and the follow-up task is only determined after some additional computation.

9.28.2.1 Transitions

The following example shows the Studio client configuration for the workflow steps of the built-in *Studio Two Step Publication Workflow*.

```
workflowPlugins._.addPublicationWorkflowPlugin({

  workflowName: "StudioTwoStepPublication",
```

```
  nextStepVariable: "nextSelectedTask",

  transitions: [
    {
      task: "Approve",
      defaultNextTask: "Publish",
      nextSteps: [
        {
          name: "Compose",
          allowAlways: true,
        },
        {
          name: "Publish",
          forceCurrentPerformer: true,
        },
      ],
    },
    {
      task: "Compose",
      defaultNextTask: "Approve",
      nextSteps: [
        {
          name: "Approve",
          isAssignmentTask: true,
        },
      ],
    },
  ],
});
```

Example 9.110. Workflow steps configuration for the built-in 2-step publication workflow

First of all, you need to define a `nextStepVariable`. This denotes the process variable of the process definition into which the result of a user choice between possible next steps is written. For the *CoreMedia* publication workflows this is the variable `nextSelectedTask`. In this case the selected step directly corresponds to the follow-up task. For the *CoreMedia* translation workflows this is the `translationAction` variable. Here, some further computation happens before a follow-up task is determined.

The *transitions* configuration parameter of a `WorkflowPlugin` consists of an array of `WorkflowTransition`s. For each transition, three parameters can be configured:

task   The current task for which follow-up steps are configured.

nextSteps   A list of possible follow-up workflow steps. Each step is given as a `WorkflowStep` with the following parameters:

- *name*: The name of the step. This is the value that is written to the `nextStepVariable` process variable if this step is chosen.

- *allowAlways*: Whether the step is always allowed, no matter whether validation issues exist for example. In the example

from above, going back to `Compose` from `Approve` is always possible, even if content errors exist.

- *isAssignmentTask*: Whether the step is directly tied to a process definition task which can have assignees (see the following subsection). In the example from above, when in `Compose`, the next step `Approve` corresponds to the `Approve` task from the process definition and this task is one for which assignees may be set.

- *forceCurrentPerformer*: Whether the step is directly tied to a process definition task for which the same performer as for the current task is forced. In the example from above, when in `Approve`, the next step `Publish` corresponds to the `Publish` tasks from the process definition which has to be carried out by the same performer as for the `Approve` task.

defaultNext-Task   The default next step from the list of `nextSteps`. This parameter is mainly important for one case: If a task is accepted in the *Workflow App* and the *Next Workflow Step* dialog is opened, a validation immediately starts with this default step as a validation parameter so that the user does not need to explicitly select a next step to trigger a validation.

> **CAUTION**
>
> Currently, it is only possible to define next step configurations in the form of a `WorkflowTransition` for tasks of a running workflow. On workflow start, selecting from multiple next steps is currently not supported. So you always need one first user task from where on several follow-up steps are possible.

9.28.2.2 Assignees

Which user can accept which workflow task depends on several conditions. First of all there is an access-rights system in place where groups are granted certain rights (that is read, write, accept, complete, cancel) on a task. In addition, for publication workflows there exist *performer policies* that determine which users have the required access rights on the workflow's contents. For more details on these fundamental mechanisms, see the Workflow Manual.

The special case of assignees is covered because it also requires a Studio client configuration. If a task is an assignment task, the user of the predecessor task can specify assignees for the task. Assignees can be multiple users and groups.

Only assigned users or members of assigned groups can then carry out the task.
Currently, assignees are only supported for publication workflows out of the box.

The following example shows how the `Approve` and `Publish` tasks of the
`Studio-Two-Step-Publication` process definition are defined as assign-
ment tasks.

```
<Workflow>
  <Process name="..." startTask="...">
    <!-- ... -->

    <UserTask name="Approve"
              description="studio-three-step-publication-approve-task"
              successor="PublishOrCompose" reexecutable="true">
      <Performers
policyClass="com.coremedia.cap.workflow.plugin.AssignableResourcePermissionsPerformersPolicy"

                    assignedUsersVariable="assignedUsers_Approve"
                    assignedGroupsVariable="assignedGroups_Approve"
                    rights="approve, publish"/>
      <Rights>
        <Grant group="administratoren" rights="read, accept, cancel, delegate,
reject"/>
        <Grant group="approver-role" rights="read, accept, cancel, delegate,
reject"/>
      </Rights>
      <Assignment>
        <Reads variable="assignedUsers_Approve"/>
        <Reads variable="assignedGroups_Approve"/>
        <Writes variable="assignedUsers_Publish"/>
        <Writes variable="assignedGroups_Publish"/>
        <!-- ... -->
      </Assignment>
      <!-- ... -->
    </UserTask>

    <UserTask name="Approve"
              description="studio-three-step-publication-approve-task"
              successor="ApproveOrDoPublish" reexecutable="true">
      <Performers
policyClass="com.coremedia.cap.workflow.plugin.AssignableResourcePermissionsPerformersPolicy"

                    assignedUsersVariable="assignedUsers_Publish"
                    assignedGroupsVariable="assignedGroups_Publish"
                    rights="publish"/>
      <Rights>
        <Grant group="administratoren" rights="read, accept, cancel, delegate,
reject"/>
        <Grant group="publisher-role" rights="read, accept, cancel, delegate,
reject"/>
      </Rights>
      <Assignment>
        <Reads variable="assignedUsers_Publish"/>
        <Reads variable="assignedGroups_Publish"/>
        <!-- ... -->
      </Assignment>
      <!-- ... -->
    </UserTask>
  </Process>
</Workflow>
```

Example 9.111. Defining assignable performers policy for tasks

In order to make a task an assignment task, the `AssignableResourcePer`
`missionsPerformersPolicy` has to be set as the performers policy of the
task. In addition, this policy needs to be configured with the two parameters

assignedUsersVariable and *assignedGroupsVariable*. The values for both parameters need to be process variables, and they need to follow the exact naming pattern of `assignedUsers_{$taskName}` and `assignedGroups_{$taskName}`. In the example, one can also see that the `Approve` task reads the `assignedUsers_Approve` and `assignedGroups_Approve` variables and writes the `assignedUsers_Publish` and `assignedGroups_Publish` variables.

For the Studio client, configuring a workflow step as an assignment task is very easy as shown in Example 9.110, "Workflow steps configuration for the built-in 2-step publication workflow " [291]: In the `WorkflowTransition` definition for the `Compose` task, the `WorkflowStep` definition for `Approve` has the parameter `isAssignmentTask` set.

To conclude, the `WorkflowTransition#isAssignmentTask` configurations for the Studio client must match the `AssignableResourcePermissionsPerformersPolicy` configurations of the process definition on the workflow server.

9.28.3 Workflow Fields

CoreMedia Studio comes with predefined forms to start publication and translation workflows (*Control Room*) and to work with running publication and translation workflows (*Workflow App*). It is not possible to define custom workflow forms from scratch. Instead, the `WorkflowPlugin` API allows to extend the predefined forms.

The plugin has a form extension for both the start workflow form and the running workflow form. Both extensions define extra fields for workflow forms, and the values of all these fields stem from a view model. The view model type is the parameter `<M>` of the plugin properties *WorkflowPlugin#startWorkflowFormExtension<M extends ViewModel>* and *WorkflowPlugin#runningWorkflowFormExtension<M extends ViewModel>*. So both work with the same view model type.

As a running example, the form extensions for the Global Link Translation Workflow are used.

9.28.3.1 Start Workflow Form Extension

The *Global Link Translation Workflow* defines one extra field for the start workflow form which is a due date field where the date is given as a `Calendar`. This field's value is also taken into account for the backend validation of the workflow.

The image below shows the customized start form with a reported validation error for the field.

Figure 9.11. Start Workflow form Extension for the Global Link Translation Workflow

The following code gives the complete definition of the start workflow form extension for the *Global Link Translation Workflow*. The details are explained afterward (details on using `getLocalizer()` to localize resource bundle properties are explained in Section Section 9.28.7, "Workflow Localization" [307]).

```
        import { workflowPlugins } from
"@coremedia/studio-client.workflow-plugin-models/WorkflowPluginRegistry";
import { Binding, DateTimeField } from
"@coremedia/studio-client.workflow-plugin-models/CustomWorkflowApi";
import { getLocalizer } from "@coremedia/studio-client.i18n-models";
import Calendar from "@coremedia/studio-client.client-core/data/Calendar";
import Gcc_properties from "./Gcc_properties";
```

```
interface GccViewModel {
  globalLinkPdSubmissionIds?: string;
  globalLinkSubmissionStatus?: string;
  submissionStatusHidden?: boolean;
  globalLinkDueDate?: Date;
  globalLinkDueCalendar?: Calendar;
  globalLinkDueDateText?: string;
  completedLocales?: string;
  completedLocalesTooltip?: string;
  xliffResultDownloadNotAvailable?: boolean;
}

const getGccWorkflowPlugin = async (): Promise<TranslationWorkflowPlugin> => {
  const localizer = await getLocalizer(Gcc_properties);

  return {

    workflowName: "TranslationGlobalLink",

    nextStepVariable: "translationAction",

    startWorkflowFormExtension: StartWorkflowFormExtension<GccViewModel>({
      computeViewModel() {
        const defaultDueDate = getDefaultDueDate();
        if (!defaultDueDate) {
          return undefined;
        }

        return {
          globalLinkDueCalendar: defaultDueDate
        };
      },

      saveViewModel(viewModel: GccViewModel): Record<string, any> {
        return {
          globalLinkDueDate: viewModel.globalLinkDueCalendar,
        };
      },

      remotelyValidatedViewModelFields: ["globalLinkDueCalendar"],

      fields: [
        DateTimeField({
          label: localizer("TranslationGlobalLink_submission_dueDate_key"),
          tooltip: localizer("TranslationGlobalLink_submission_dueDate_tooltip"),
          value: Binding("globalLinkDueCalendar")
        }),
      ],
    }),
    ...
  };
};

getGccWorkflowPlugin().then((gccWorkflowPlugin) => {
  workflowPlugins._.addTranslationWorkflowPlugin(gccWorkflowPlugin);
});
```

Example 9.112. Start workflow form extension for Global Link Translation Workflow

The type of the form extension's view model is given as a simple interface. For
the start form extension, only the property globalLinkDueCalendar is

relevant, the other ones come into play in the following section where running workflow forms are considered.

To customize the start workflow form for a custom workflow, the parameter *WorkflowPlugin#startWorkflowFormExtension<M extends ViewModel>* of the `WorkflowPlugin` is used. It is created with the factory function `StartWorkflowFormExtension<M extends ViewModel>()` and has the following parameters itself:

computeViewModel(): M | undefined

A function that computes the view model for the extension's fields. As it is a start workflow form extension, there is no workflow running yet and so the function receives no parameter. It can be used to initialize the view model values with default parameters. In the example above, the view model for the start form extension only consists of one property `globalLinkDueCalendar`. Its value is computed by the function `getDefaultDueDate()`. It is important to note that the whole function `computeViewModel()` is embedded in a `FunctionValueExpression` (see Section 5.3.6, "Value Expressions" [65]). So all utility functions like `getDefaultDueDate()` can be implemented dependency-tracked and as long as they do not deliver a value, the overall result can just be `undefined`.

saveViewModel(viewModel: M): Record<string, unknown>

A function that is called once the workflow has been created and that saves the current view model state of the start form extension to the workflow. Consequently, it receives the current view model state as input parameter and delivers a record as result. For each of the record's entries it has to hold that the key corresponds to the name of a process variable and that the value matches the type of the corresponding process variable. In Example 9.112, "Start workflow form extension for Global Link Translation Workflow" [296], the `globalLinkDueCalendar` from the view model is saved to the `globalLinkDueDate` process variable of the created workflow.

fields

An array of the start form extension's fields. Fields are not defined as *Ext JS* components but in terms of a declarative API (part of `CustomWorkflowAPI`) that is independent of any UI framework. Currently, five field types are supported:
- `TextField`
- `DateField`
- `DateTimeField`
- `CheckField`
- `Button`

All fields have the following properties:

- `value`
- `label`
- `disabled`
- `hidden`
- `tooltip`
- `validationState`

These properties can either be set directly or be defined as a two-way `Binding` to one of the view model properties. In the example from above, there is only one field in the start form extension, a `DatetimeField`: Label and tooltip are directly set while the value is bound to the view model's `globalLinkDueCalendar` property. Note that a `DateTime Field's` value can only be bound to a view model property of type `Calendar`. More examples of form extension fields are covered in the next section.

remotelyValidatedViewModelFields?: (keyof M)[]

An (optional) array of view model parameters that are part of the backend workflow validation (cf. Section 9.28.5, "Workflow Validation" [304]). Changes to their values trigger a backend validation, and the values are part of the validation's parameters.

viewModelValidator?: (viewModel: M) => WorkflowSetIssues

An (optional) function that carries out a client-side validation of the view model values. The *WorkflowSetIssues* that result from this computation are merged with the issues from the backend workflow validation.

9.28.3.2 Running Workflow Form Extension

The *Global Link Translation Workflow* defines several extra fields for a running workflow form as shown in figure Figure 9.12, "Running Workflow form Extension for a Running Global Link Translation Workflow" [300] (note that the "Due Date" field is an input field here for the sake of the example, it is normally a read-only field for a running Global Link workflow).

Figure 9.12. Running Workflow form Extension for a Running Global Link Translation Workflow

The following code gives the complete definition of the running workflow form extension for the the *Global Link Translation Workflow*. The details are explained afterward.

```
        import { workflowPlugins } from
"@coremedia/studio-client.workflow-plugin-models/WorkflowPluginRegistry";
import { Binding, DateTimeField } from
"@coremedia/studio-client.workflow-plugin-models/CustomWorkflowApi";
import { getLocalizer } from "@coremedia/studio-client.i18n-models";
import Gcc_properties from "./Gcc_properties";

interface GccViewModel {
  globalLinkPdSubmissionIds?: string;
  globalLinkSubmissionStatus?: string;
  submissionStatusHidden?: boolean;
  globalLinkDueDate?: Date;
  globalLinkDueCalendar?: Calendar;
  globalLinkDueDateText?: string;
  completedLocales?: string;
  completedLocalesTooltip?: string;
  xliffResultDownloadNotAvailable?: boolean;
}

const getGccWorkflowPlugin = async (): Promise<TranslationWorkflowPlugin> => {
  const localizer = await getLocalizer(Gcc_properties);

  return {
    workflowName: "TranslationGlobalLink",

    nextStepVariable: "translationAction",

    startWorkflowFormExtension: { ... },

    runningWorkflowFormExtension: StartWorkflowFormExtension<GccViewModel>({
      computeTaskFromProcess: (process) => process.getCurrentTask(),
      computeViewModel(state: WorkflowState): GccViewModel {
        return {
```

```
               globalLinkPdSubmissionIds:
                 transformSubmissionId(state.process.getProperties()
                   .get("globalLinkPdSubmissionIds")),
               globalLinkSubmissionStatus:
                 transformSubmissionStatus(state.process.getProperties()
                   .get("globalLinkSubmissionStatus")),
               globalLinkDueDate:
                 dateToDate(state.process.getProperties().get("globalLinkDueDate")),
               completedLocales:
                 convertLocales(state.process.getProperties().get("completedLocales")),
               completedLocalesTooltip:
                 createQuickTipText(state.process.getProperties()
                   .get("completedLocales"), localesService),
               xliffResultDownloadNotAvailable:
                 downloadNotAvailable(state.task),
             };
           },

         saveViewModel(viewModel: GccViewModel) {
           return {
             globalLinkDueDate: viewModel.globalLinkDueDate,
           };
         },

         fields: [
           TextField({
             label: localizer("TranslationGlobalLink_submission_id_key"),
             value: Binding("globalLinkPdSubmissionIds"),
             readonly: true,
           }),
           TextField({
             label: localizer("TranslationGlobalLink_submission_status_key"),
             value: Binding("globalLinkSubmissionStatus"),
             readonly: true,
           }),
           DateField({
             label: localizer("TranslationGlobalLink_submission_dueDate_key"),
             value: Binding("globalLinkDueDate")
           }),
           TextField({
             label: localizer("TranslationGlobalLink_completed_Locales"),
             readonly: true,
             value: Binding("completedLocales"),
             tooltip: Binding("completedLocalesTooltip"),
           }),
           Button({
             label: localizer("translationResultXliff_Label_Button_text"),
             value: localizer("Gcc_properties.translationResultXliff_Button_text"),
             validationState: "error",
             handler: (state): GccViewModel | void => downloadXliff(state.task),
             hidden: Binding("xliffResultDownloadNotAvailable"),
           }),
         ]
       }),

     ...
   };
};

getGccWorkflowPlugin().then((gccWorkflowPlugin) => {
  workflowPlugins._.addTranslationWorkflowPlugin(gccWorkflowPlugin);
});
```

Example 9.113. Running workflow form extension for Global Link Translation Workflow

To customize the running workflow form for a custom workflow, the parameter *WorkflowPlugin#runningWorkflowFormExtension<M extends ViewModel>* is used. The view model is the same as for the `startWorkflow FormExtension`. It is created with the factory function `RunningWorkflow FormExtension<M extends ViewModel>()` which has very similar parameters to those of the previous section:

computeTaskFromProcess?: (process: Process) => Task | null | undefined;

> An optional function to compute the current task from a given process. Its purpose is described under the following point.

computeViewModel(state: WorkflowState): M

> A function that computes the view model for the extension's fields. Contrary to the `StartWorkflowFormExtension#computeViewModel()` function it receives a `WorkflowState` parameter. It has the two properties `WorkflowState#process` (a `Process` remote bean) and `WorkflowState#task` (a `Task` remote bean) which hold the currently displayed process and task respectively.

> Note that the `process` property is always given and that the bean is fully loaded. For the `task` property, things are a bit more complicated. It is set if either (1) the workflow form displays a task (e.g., opened from "Inbox") or (2) the workflow form displays a process (e.g., opened from "Running") but `RunningWorkflowFormExtension#computeTaskFromProcess()` from above is given. Otherwise `task` is set to `null`.

> Note that the function `computeTaskFromProcess` is wrapped in a dependency-tracked `FunctionValueExpression` under the hood. Thus, it may return `undefined` as long as the current task cannot be computed due to asynchronous sub-computations. The surrounding framework ensures that `RunningWorkflowFormExtension#com puteViewModel(state: WorkflowState)` will always be called with a `WorkflowState` parameter where the `process` and `task` remote bean properties are set and fully loaded (with the one exception from above where `task` is `null`).

> In the example, several view model properties are computed based on the current `WorkflowState`. While some computations are just access calls to the process variables, others require more complex computations and utilize helper functions.

saveViewModel(viewModel: M): Record<string, unknown>

Contrary to `StartWorkflowFormExtension#saveViewModel()` function, this function to save the view model changes back to the process is not only called once (upon workflow start) but whenever the view model changes.

fields

An array of the running form extension's fields. The same explanations as for `StartWorkflowFormExtension#fields` apply here. The example shows a mixture of field properties that are directly set or bound to a view model property.

remotelyValidatedViewModelFields?: (keyof M)[]

The same explanations as for `StartWorkflowFormExtension#re` `motelyValidatedViewModelFields` apply here.

viewModelValidator?: (viewModel: M) => WorkflowSetIssues

The same explanations as for `StartWorkflowFormExtension#view` `ModelValidator` apply here.

9.28.4 Additional Workflow List Actions

Workflow lists are shown in the *Main Studio Control Room* and in the overview of the *Workflow App*. Depending on the concrete list, these lists contain tasks or processes. In each case, actions can be performed on the current selection of workflow objects. These actions are either tied to toolbar buttons (Control Room) or to menu items (Workflow App). Using `WorkflowPlugin#work` `flowListActions` additional actions can be added in these places. Each `WorkflowObjectListAction` is defined in terms of the following parameters:

text

The action's text.

tooltip

The action's tooltip.

svgIcon

The action's icon given as an SVG icon. For the use of SVG icons, the same prerequisites as described in Section 9.5.1, "Localizing Types and Fields" [145] apply.

handler: (workflowObjects: Array<WorkflowObject>) => void

> The action's handler function. It receives the selected workflow objects as a parameter.

confirmMessage

> If this parameter is set, a confirmation dialog with this message is displayed upon triggering the action.

confirmTitle

> This parameter only comes into play if `confirmMessage` is set. It sets the title of the confirmation dialog. If not set, a default title is displayed.

computeActionState: (workflowObjects: Array<WorkflowObject>) => { disabled: boolean, hidden: boolean }

> This function computes the action's state in terms of its *disabled* and *hidden* status. The workflow object selection is given as a parameter.

9.28.5 Workflow Validation

This section describes the server-side customizations required for workflow validation. The client-side counterpart is very simple and was already covered in section Section 9.28.3, "Workflow Fields" [295]. It is divided into two parts: the first part (this section) covers validation for custom workflows, the second one (next section) describes how to customize validation for built-in workflows.

For the Studio server you can define or change validators that create issues for your workflow. Each validator is linked to a workflow task and optionally its state, so that you can define different validators for every stage of your process. For each set of validators you can additionally define a so-called `WorkflowValidationPreparation`. This is a step that will be executed before the Workflow validators. (For example, the dependent content is calculated in the `WorkflowValidationPreparation`)

Adding custom workflow validators

In order to add validators, or a preparation step for your workflow, you need to provide a bean of type `WorkflowValidatorsModel` within your *studio-lib* extension. In that model you need to set the `processName` according to the process that you want to add validators to. Now you need to define validators that have to implement the interface `com.coremedia.rest.cap.workflow.validation.WorkflowValidator`, and optionally an implementation of interface `WorkflowValidationPreparation`. Depending on whether you want to use the validator for the start of a workflow or for a certain

task you need to either place you validator in the `WorkflowStartValidat ors` or the `WorkflowTaskValidators`.
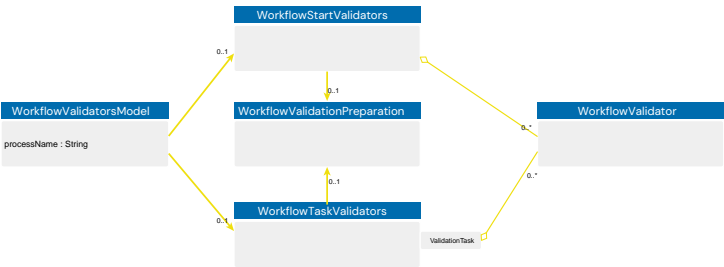
Figure 9.13. Workflow validators model class diagram

> **NOTE**
>
> If you want to add a validator to a built-in workflow, see section Section 9.28.6, "Customizing Validation of Built-In Workflows" [306].

Already existing validators

There is a set of already defined validators available, which you can use for your own validator lists. See Spring configuration classes `TranslationWork flowValidationConfiguration` `PublicationWorkflowValida tionConfiguration` for available validator beans.

Writing your own validator

If you want to define your own validator you need to implement the interface `WorkflowValidator` and create issues within the method `addIssuesIfIn valid`. The method will receive a `parameter` object that you can use to compute your issues from. Within the parameter object, the `isAbortReques tedRunnable` object is stored, that you need to check if the validation was aborted. You need to call the `isAbortRequestedRunnable` method within your validator regularly to make sure that an aborted validation does not go on longer than necessary.

Workflow Validation Settings

Despite normal configuration options provided via Spring, workflow validation behavior may also be configured via settings provided as content item. The recommended location for such settings is a settings document at `/Settings/Options/Settings/WorkflowValidation`.

A central switch, that may be configured within this settings document is the `disabled` flag, that may be used to temporarily turn off any workflow validation. If set to `true`, all server-side validation will be skipped and replaced by a warning instead, that validation is currently disabled.

Other switches to provide are up to the corresponding workflows. Typically, settings are grouped by workflow type.

More information to be found at:

-
-

9.28.6 Customizing Validation of Built-In Workflows

This section describes how to customize *start validators* for built-in workflows. Workflow start validators may need to be changed due to project requirements like, e.g., excluding specific content items from translation.

> **NOTE**
>
> It is generally not recommended to omit or change existing validators of built-in workflows. Refer to Table 3.19, "Studio Properties" in *Deployment Manual* in Deployment Manual for default start validators.

Workflow start validators for built-in workflows are defined by one property for each workflow (property name pattern `studio.workflow.validation.start-validators.*`). Refer to Table 3.19, "Studio Properties" in *Deployment Manual* in Deployment Manual for default values of these properties. See Java API documentation on `TranslationWorkflowValidationConfiguration` and `PublicationWorkflowValidationConfiguration` for details on available validator beans.

Custom validator beans implementing interface `WorkflowValidator` may be added to any of the workflows by providing a re-definition of the correspond-

ing property (using the default value) in your project code and appending custom validator bean names to the comma-separated list.

9.28.7 Workflow Localization

Workflow localization follows the same approach as content type localization described in Section 9.5.1, "Localizing Types and Fields" [145]. Localizations are added or modified using a registry. Just as for content type localization, icons are provided in terms of SVG icons, so the same prerequisites as described previously hold.

One particular aspect to take care of here is how to localize the resource bundle properties that are used for the workflow names and states. For the *Content App*, an ExtJs-based mechanism takes care of loading just the required locale-version ("en", "de", "ja") of the resource bundle. As the *Workflow App* is a React-based application, resource bundle properties localization requires a small additional effort. As the example below shows, an additional "de" or "ja" locale has to be registered via `registerLocale()` and `useLocalizer()` has to be used to localize the resource bundle properties.

The following code example shows an excerpt of the localization of the *Global Link* translation workflow.

```
import GccWorkflowLocalization_properties from "./GccWorkflowLocalization_properties";
import gccIcon from "./icons/global-link-workflow_24.svg";
import { workflowLocalizationRegistry, WorkflowLocalizationConfig, WorkflowIssuesLocalizationConfig
  } from "@coremedia/studio-client.workflow-plugin-models";
import { getLocalizer, registerLocale } from "@coremedia/studio-client.i18n-models";

registerLocale(GccWorkflowLocalization_properties, "de", async () => {
  await import("./GccWorkflowLocalization_de_properties");
});

const getGccProcessLocalization = async (): Promise<WorkflowLocalizationConfig> => {
  const localize = await getLocalizer(GccWorkflowLocalization_properties);

  return {
    displayName: localize("TranslationGlobalLink_displayName"),
    description: localize("TranslationGlobalLink_description"),
    svgIcon: gccIcon,
    states: {
      Translate:
        localize("TranslationGlobalLink_state_Translate_displayName"),
      DownloadTranslation:
        localize("TranslationGlobalLink_state_DownloadTranslation_displayName"),
      ReviewDeliveredTranslation:
        localize("TranslationGlobalLink_state_ReviewDeliveredTranslation_displayName"),
      ReviewCancelledTranslation:
        localize("TranslationGlobalLink_state_ReviewCancelledTranslation_displayName"),
      ...
    },
    tasks: {
      Prepare:
        localize("TranslationGlobalLink_task_Prepare_displayName"),
      AutoMerge:
        localize("TranslationGlobalLink_task_AutoMerge_displayName"),
      SendTranslationRequest:
```

```
        localize("TranslationGlobalLink_task_SendTranslationRequest_displayName"),
      ...
    },
  };
};

getGccProcessLocalization().then((gccProcessLocalization) => {
  workflowLocalizationRegistry._.addLocalization("TranslationGlobalLink", gccProcessLocalization);
});

const getGccIssuesLocalization = async (): Promise<WorkflowIssuesLocalizationConfig> => {
  const localize = await getLocalizer(GccWorkflowLocalization_properties);

  return {
    dateLiesInPast_globalLinkDueDate:
      localize("dateLiesInPast_globalLinkDueDate_text"),
    dateInvalid_globalLinkDueDate:
      localize("dateInvalid_globalLinkDueDate_text"),
    ...
  };
};

getGccIssuesLocalization().then((gccIssuesLocalization) => {
  workflowLocalizationRegistry._.addIssuesLocalization(gccIssuesLocalization);
});
```

Example 9.114. Workflow localization example

Display name, description and icon are defined for the workflow. As described earlier in Section 9.28.2, "Workflow Steps" [291], tasks and state are distinguished. Consequently, they are localized separately. Note that each task and state can also be localized with a separate display name and description instead of just with a string.

The example also shows that issues are localized with the `workflowLocalizationRegistry` as well.

9.28.8 Publication Workflow Specifics

This section covers publication-specific workflow customizations. These only refer to server-side customizations. For the client side, there are no publication-specific customizations beyond those covered in the previous sections.

For the server side, you need to define custom validators for your new workflow. Please refer to Section 9.28.5, "Workflow Validation" [304] for general information on how to do this. This section introduces the default publication workflow validators that cover most needs for publication workflow validation. Feel free to add further validators if needed.

> **!**
>
> **CAUTION**
>
> Before the actual validators are executed, a so called WorkflowValidationPreparation takes place, which provides the workflow's computed *change set* as well as the *dependent contents* as INFO issues.

The default publication workflow validators offer the following functionality:

PublicationContentStateValidator

> All content validators from the application context are applied to check the complete publication set for validity.

PublicationNoAssigneeValidator

> If assignees are selected, all assignees are checked for having the required (see below) content rights for the publication set as well as for having the right rights for accepting the next selected task.

PublicationSessionUserRightsWorkflowValidator

> For the workflow start, it is checked whether there are contents that need to be checked in and if the current user does not have the right to do it.

PublicationContentRightsWorkflowValidator

> Workflow validator that checks if the given members have the configured content rights on the chosen content, that a necessary to perform publication workflow. The validator can be configured.
>
> For each user task of the workflow, you can define the required content rights. For example, the `Approve` task does not need publish rights.
>
> You can also configure if assignees can accept not only the next selected task but also a number of follow-up tasks, for example if they are auto-accepted. For example, for the case of the built-in 2-step publication workflow, the user that accepts the `Approve` task also needs to be able to accept the following `Publish` task.

PublicationWorkflowUndoWithdrawValidator

> Validator that removes (undoes) toBeDeleted and toBeWithdrawn states from the given contents and their parent folders.

All the validators are defined as Spring Beans within the `PublicationWorkflowValidationConfiguration` file, where you can also find their Bean names, defined as constants, which you can import, or override.

The following code shows the validation configuration for the 3-step publication example.

```
@Configuration
public class ThreeStepPublicationWorkflowConfiguration {

  private static final String THREE_STEP_PUBLICATION_WORKFLOW_NAME = "StudioThreeStepPublication";
  public static final String THREE_STEP_PUBLICATION_VALIDATORS = "threeStepPublicationValidators";
  public static final String THREE_STEP_PUBLICATION_WORKFLOW_VALIDATORS =
"threeStepPublicationWorkflowValidators";
  public static final String DO_PUBLISH_TASK_NAME = "DoPublish";

  @Bean(THREE_STEP_PUBLICATION_WORKFLOW_VALIDATORS)
  public WorkflowValidatorsModel threeStepPublicationWorkflowValidators(
        @Qualifier(PUBLICATION_VALIDATION_PREPARATION) WorkflowValidationPreparation
publicationValidationPreparation,
        @Qualifier(THREE_STEP_PUBLICATION_VALIDATORS) List<WorkflowValidator>
threeStepPublicationValidators) {
    ValidationTask composeRunningTask = new ValidationTask(COMPOSE_TASK_NAME, TaskState.RUNNING);
    ValidationTask approveRunningTask = new ValidationTask(APPROVE_TASK_NAME, TaskState.RUNNING);
    ValidationTask publishRunningTask = new ValidationTask(PUBLISH_TASK_NAME, TaskState.RUNNING);

    final WorkflowTaskValidators taskValidators =
            new WorkflowTaskValidators(publicationValidationPreparation, Map.of(
                    composeRunningTask, threeStepPublicationValidators,
                    approveRunningTask, threeStepPublicationValidators,
                    publishRunningTask, threeStepPublicationValidators));

    return new WorkflowValidatorsModel(
      THREE_STEP_PUBLICATION_WORKFLOW_NAME,
      taskValidators,
      new WorkflowStartValidators(publicationValidationPreparation,
      threeStepPublicationValidators));
  }

  @Bean(THREE_STEP_PUBLICATION_VALIDATORS)
  public List<WorkflowValidator> threeStepPublicationValidators(
        ContentRepository contentRepository,
       @Qualifier(PUBLICATION_NO_ASSIGNEE_VALIDATOR) WorkflowValidator publicationNoAssigneeValidator,

        @Qualifier(PUBLICATION_SESSION_USER_RIGHTS_WORKFLOW_VALIDATOR) WorkflowValidator
publicationSessionUserRightsWorkflowValidator,
        @Qualifier(PUBLICATION_WORKFLOW_UNDO_WITHDRAW_VALIDATOR) WorkflowValidator
publicationWorkflowUndoWithdrawValidator,
        @Qualifier(PUBLICATION_CONTENT_ISSUES_VALIDATOR) WorkflowValidator
publicationContentIssuesValidator) {

    Map<String, Rights> requiredContentRightsForTasks = new HashMap<>();
    requiredContentRightsForTasks.put(APPROVE_TASK_NAME, Rights.valueOf("RA"));
    requiredContentRightsForTasks.put(PUBLISH_TASK_NAME, Rights.valueOf("RAP"));
    requiredContentRightsForTasks.put(DO_PUBLISH_TASK_NAME, Rights.valueOf("RAP"));
    PublicationContentRightsWorkflowValidator publicationContentRightsWorkflowValidator =
        new PublicationContentRightsWorkflowValidator(
            contentRepository,
            requiredContentRightsForTasks,
            Map.of(PUBLISH_TASK_NAME, List.of(DO_PUBLISH_TASK_NAME)));

    return List.of(publicationContentRightsWorkflowValidator,
            publicationNoAssigneeValidator,
            publicationSessionUserRightsWorkflowValidator,
            publicationWorkflowUndoWithdrawValidator,
            publicationContentIssuesValidator);
  }
}
```

Example 9.115. Workflow validation configuration for the StudioThreeStepPublication workflow

Workflow Validation Settings

The calculation of a preview of dependent contents may be configured via Spring or via settings in content. Configuring limits may be useful, when you observe too long evaluations for huge content sets to publish.

Content based configuration is done via a settings document at `/Settings/Options/Settings/WorkflowValidation` as described in section "Workflow Validation Settings" [306]. For publication workflows the following additional properties are supported:

- `publication.maxDepthToCompleteChangeSet` (Integer)

  related Spring configuration: `studio.rest.changeset-max-iterations`

- `publication.maxUpdatedLinksDepthToCompleteChangeSet` (Integer)

  related Spring configuration: `studio.rest.changeset-max-updated-links-recursion-depth`

If set to a non-empty value, these settings will override the corresponding Spring configuration properties. For details on these properties consult the corresponding description in Section 3.4, "Studio Properties" in *Deployment Manual*.

9.28.9 Translation Workflow Specifics

This section covers translation-specific workflow customizations for both the server and the client side.

Studio server

The Studio server customizations are mainly related to workflow issue computation.

Calculation of Dependent Content

Dependent content are content items not explicitly chosen for translation, but which are required to keep in-site links consistent. This is the minimal set of dependent content to be added, but an extended set may be desirable.

The easiest example are two new contents A and B where A links to B. If A is transferred to a derived site via translation, a translation of B is required as well, to mirror the relationship from master to derived.

Extension of this set may be desirable for out of date content: Content A is created and links to existing and previously translated content B. As B already exists, it is not necessarily required to be added as dependent content. But the translation of B is out-of-date. Thus, it may be desirable, adding content B nevertheless.

The behavior can be controlled by choosing a strategy for dependent content calculation. This strategy can be chosen via a checkbox below the content section.

As the calculation of dependent content may take a considerable amount of time, limits can be configured within Spring application context as well as via settings within content. For details, take a look at section "Workflow Validation Settings" [313].

If a limit will be reached, the nagbar will display a warning, which states that not all dependent content has been calculated. Per default the limit does not exist.

Compatibility prior to 2007.1

Prior to 2007.1 the dependent content calculation was different. It also contained the content items, which are required to keep in-site links consistent, but it did not include outdated dependent contents. Instead, it included existing contents, which were added as new links to some contents to translate. These additional dependent contents were meant to provide some context to the translators.

If this behavior meets your requirements, you can still switch back to this behavior. To do so, simply adapt your `DefaultStartTranslationWorkflow Form` with `hideDependentContentsStrategyChooser` set to true.

Localization Issues

The Translation Workflow displays its issues in two possible categories. These are `localization` and `content`. You can create issues with these categories in order to let your issue show up either as translation or content specific. If you write a custom translation workflow validator, you may extend the class `Local izationWorkflowValidator` which automatically creates issues with the category `localization`. Furthermore, all predefined workflow validators for translation, except the `ContentStateValidator`, produce only issues for the category `localization`.

Workflow Validation Settings

Validation of localization workflows (where localization is a superset of translation and synchronization) may be configured within Spring configuration properties. Corresponding properties are denoted in Section 3.4, "Studio Properties" in *Deployment Manual*.

Some of these settings may be overwritten via a settings document at `/Settings/Options/Settings/WorkflowValidation` as described in section "Workflow Validation Settings" [306]. For localization workflows such as translation and synchronization the following properties are supported:

- `localization.maxDepthToCompleteChangeSet` (Integer)

  related Spring configuration: `studio.translation.max-dependent-content-iterations`

- `localization.limitForDependentContentItems` (Integer)

  related Spring configuration: `studio.translation.max-dependent-contents`

If set to a non-empty value, these settings will override the corresponding Spring configuration properties. For details on these properties consult the corresponding description in Section 3.4, "Studio Properties" in *Deployment Manual*.

Studio client

The `TranslationWorkflowPlugin` has some additional configuration options compared to the `WorkflowPlugin`:

isSync

Whether the translation workflow is a synchronization workflow (defaults to `false`).

downloadXLIFFButtonVisible

Whether the default button to download the XLIFF document should be visible (defaults to `false`).

createWorkflowPerTargetSite

Whether a separate translation workflow should be created per target site on workflow start or whether one workflow for all target sites shall be created (defaults to `false`).

hideDependentContentsStrategyChooser

> Whether the dependent content strategy chooser should be visible for the start workflow form (defaults to `true`).

allowSelfAssignedPullTranslation

> Whether pull translation leads to automatic self-assignment of the first workflow task (defaults to `true`).

9.28.10 Synchronization Workflow Specifics

For the Synchronization Workflow, a custom merge strategy can be added to the merge strategy chooser of the Start Synchronization Workflow Panel.

This customization requires a change for the Studio client and the Workflow Server. The change for Studio client is described in this section, for customization of the workflow-server refer to Section " AutoMergeSyncAction " in *Blueprint Developer Manual* .

Adding a merge strategy for a synchronization workflow on the Studio client side is done via the `AddMergeStrategyPlugin` which is part of the package `@coremedia/studio-client.main.control-room-editor-components`.

```
import { workflowPlugins } from
"@coremedia/studio-client.workflow-plugin-models";

workflowPlugins._.registerMergeStrategy("CustomSyncWorkflow",
"newMergeStrategy");
```

Example 9.116. Adding a New Merge Strategy

In order to add localization for a new merge strategy you can utilize the `WorkflowLocalizationRegistry` of the package `@coremedia/studio-client.workflow-plugin-models`.

```
import CustomSyncWorkflow_properties from "./CustomSyncWorkflow_properties";
import { workflowLocalizationRegistry } from
"@coremedia/studio-client.workflow-plugin-models";

workflowLocalizationRegistry._.addMergeStrategyLocalization(
  "CustomSyncWorkflow",
  "newMergeStrategy",
  {
    displayName: CustomSyncWorkflow_properties.newMergeStrategy_displayName,
```

```
    description: CustomSyncWorkflow_properties.newMergeStrategy_description
});
```

Example 9.117. Adding a New Merge Strategy Localization

Workflow Validation Settings

Just as for translation workflows, a preview of dependent contents is calculated when starting a synchronization workflow. As this may take a considerable amount of time to evaluate for huge changesets, limits can be configured within Spring configuration properties as well as within content settings.

Here, synchronization shares the same settings as for the translation workflow (both summarized as localization workflows). For available configuration options take a look at section "Workflow Validation Settings" [313].

9.29 Content Hub

The *CoreMedia Content Hub* allows integrating various external and asset management systems into the *Studio* library. It allows you to integrate just about any external system or platform into your CoreMedia system.

> **NOTE**
>
> Every integration of an external system for the *Content Hub* or any other *CoreMedia* hub is called *adapter*.

This section describes the functionality of the *Content Hub* and the required steps to implement a custom adapter for it. Adapters are usually implemented as extensions for *Studio*, using the extensions point `studio-lib` (and `studio` if UI customizations are required).

9.29.1 Basic Setup

The basic functionality of an adapter is to enable the user to browse through the content of an external system in the *Studio* library. You have to implement the following interfaces:

`<YOUR_ADAPTER_NAME>Settings`

Settings interfaces are used to map adapter specific connection parameters (like a connection URL) to Java code. You only have to declare the Settings interface according to the data your adapter needs. Implementations are generated automatically, backed with the data of your configuration. The `getter` methods of these interfaces must match the corresponding fields of the `settings` struct as described in . The name of the interface is arbitrary, the `Settings` suffix is just a convention.

`com.coremedia.contenthub.api.ContentHubAdapterFactory`

An implementation of `ContentHubAdapterFactory` declares the type (a `ContentHubAdapterType`) of an adapter. While a factory can create multiple adapter instances (for example multiple RSS connections), the type defines attributes that are common for all adapter instances of the factory. The factory implements the factory method `createAdapter` to create an adapter instance. `createAdapter` has one argument, the binding, which in particular provides

a settings property of your Settings interface. The method `getId` identifies the factory and is used when an adapter configuration is read from the content.

`com.coremedia.contenthub.api.ContentHubAdapter`

A `ContentHubAdapter` implementation resolves the tree structure of entities of an external system. It returns `Folder` and `Item` instances. Concrete examples and more documentation about `ContentHubAdapters` can be found in the *Blueprint* and in the Javadoc of the interface `com.coremedia.contenthub.api.ContentHubAdapter`

`com.coremedia.contenthub.api.Item`

The `Item` interface extends the `ContentHubObject` interface which describes their common attributes such as the name and the ID of the entity. Items have a type described by an instance of `com.coremedia.contenthub.api.ContentHubType`. A `com.coremedia.contenthub.api.ContentHubType` consists of a name and a parent type. The type hierarchy determines the icons the items are shown with in *Studio*.

If the items in your external system have names like file names, with extensions suitable to determine a MIME type from (for instance `myimage.jpg`), you can start with the `com.coremedia.contenthub.api.BaseFileSystemItem`, which derives the ContentHubType from the MIME type. Otherwise, you must implement `getContentHubType()`.

`com.coremedia.contenthub.api.Folder`

The `Folder` interface extends the `ContentHubObject` interface which describes their common attributes such as the name and the id of the entity. Folders have the default `com.coremedia.contenthub.api.ContentHubType folder` that may be overridden if you want to use more specific icons in *Studio*.

`com.coremedia.contenthub.api.search.ContentHubSearchService`

If your external system allows for searching, you can propagate this to your Content Hub adapter by implementing a `ContentHubSearchService` and returning it in your `ContentHubAdapter#searchService()` implementation. You must implement at least the actual `search` method. The search capabilities of particular external systems differ. Therefore, the `ContentHubSearchService` has some feature flags that you can activate if you can

support them via the external system. For details see the Javadoc of `Con tentHubSearchService.`

> **NOTE**
>
> Adapter implementations should be stateless objects to ensure that pressing the *Reload* Button in *Studio* will invalidate the backend data as well. For example the RSS adapter does not keep the root folder as an `Object` variable. The adapter recreates the root folder with its feed items when the node is re-requested / when the user presses the reload button.

> **NOTE**
>
> Each adapter decides if and how to paginate the request of children. The *Content Hub* always requests all children until the specified page. It might be necessary to cache each page to reduce requests. Be aware that ContentHubAdapter.invalidate() will be called when the author explicitly wants to refresh a folder. In this case the cached data has to be invalidated.
>
> The pagination will be triggered by scrolling in the client. Should a user scroll to the last element of a folder (library tree, or library list view) that supports pagination, the next page will be requested automatically.

9.29.2 Adapter Configuration

Once the implementation of an adapter has been created, an additional configuration must be available to tell *Studio* which concrete instances to display. These instances are configured in settings content items in a folder named `Connections`. The `Connections` folder should contain only *Content Hub* connections content items, otherwise you will encounter some warnings in the logging. Each content item contains a `Struct List connections`. Every `connection` sub-struct defines the following properties:

Name	Type	Required	Description
connectionId	String	x	The identifier of the connection. For technical reasons, it must not contain '/' characters.
factoryId	String	x	The identifier of the implementing factory class.
settings	Struct	x	A struct that defines the connection attributes.

Name	Type	Re-quired	Description
enabled	Boolean		Allows disabling a connection.
itemTypes	Link		Links to a settings content item that contains the item type mapping. Alternatively, you can override `getItemTypes()` in your `ContentHubAdapterType` and implement this mapping hard coded.
contentTypeMap-ping	Link		Links to a settings content item that contains the mapping from *Content Hub* types to content types. Alternatively, you can override `getContentTypeMapping()` in your `ContentHubAdapterType` and implement this mapping hard coded.

Table 9.9. Connection Struct Properties

Every connection struct must contain a sub-struct `settings`. Properties of this struct will automatically be mapped to the settings interface that you have created for the adapter. For example, if the settings interface contains the method `String getConnectionUrl()`, then the struct must provide the `String` property `connectionUrl`.

> **CAUTION** ⚠
>
> Please take care of security protection. The `settings` should not contain secrets like passwords or API tokens. For example, better store them in a dedicated secrets manager and only pass them through to the external system in your custom adapter implementation during runtime.
>
> The following rules of thumb provide additional protection for sensitive data:
>
> - Restrict access to the `Connections` folder to the people that actually configure the adapters.
>
> - Do not publish adapter configuration. The adapters are only accessed in the *CoreMedia Studio*. As such they are not relevant on the live side.
>
> - Ensure that there are not links to Settings content. The adapter configuration is identified by means of their location. Links are not required. A link would risk that the Settings content is accidentally published if for example its referring content is published.
>
> - Exclude the content and folder from website search by checking the corresponding option.
>
> - Prevent access to arbitrary content from the *Headless Server* and from other client applications. See Section 3.5, "Security" in *Headless Server Manual* for more details.

9.29.2.1 Global, User and Site Specific Connections

The `Connections` folder can be located in the following folders:

- `/Settings/Options/Settings/Content Hub/`: These connections are available for all users. Please note that the connections are read with admin privileges. So even if users don't have the permission to read this folder, the global *Content Hub* connections will be available for them nevertheless.

- `<SITE_ROOT>/Options/Settings/Content Hub/`: Site specific connections are only available when the corresponding site is selected as preferred site in *Studio*.

- `<USER_HOME>/`: connections located in home folders are only available for the corresponding user.

The base folders for the global or site specific lookup can be customized via Spring properties. To customize the location, override the following default property values:

- `contenthub.studio.globalConfigurationPath`: `/Settings/Options/Settings/Content Hub`

- `contenthub.studio.siteConfigurationPath`: `/Options/Settings/Content Hub`

9.29.2.2 Content Type Mapping

Content Hub defines a ContentHubType for each Content Hub Object. The ContentHubType defines how a Content Hub Object is displayed within Studio. The ContentHubType is also used by the Content Hub framework to map a ContentHub Item to a CoreMedia ContentType during the import of the item. Each ContentHubObject has a ContentHubType (also folders). This means it is necessary to provide two mappings:

- External item to ContentHubType mapping

  The first mapping is needed to display the items of an external system in CoreMedia Studio. This mapping is from external Item to ContentHubType. Therefore, you need to implement the method `ContentHubObject#getContentHubType`. You can also create a hierarchical relationship between the ContentHubTypes which enables you to provide for example icons for a more general type. Per default the ContentHub offers the abstract classes `BaseFileSystemHubObject` and `BaseFileSystemItem` that you can use if you implement a file based system. It will analyze the MimeType of an external item and create ContentHubTypes from it. The Content Hub's framework already offers the localization and Icons for MimeType's to ensure a fast Setup.

- ContentHubType to CoreMedia ContentType Mapping

  The second mapping is needed to import the external items into CoreMedia. Therefore, you need to provide a mapping from ContentHubType to CoreMedia ContentType. Therefore, you need to implement the method `ContentHubItem#getCoreMediaContentType`. Per default the `BaseFileSystemItem` provides a functionality for this mapping. You need to provide a `Map<ContentHubType, String>` to the Item, and it will recursively map the ContentHubType to a CoreMedia ContentType which is represented as String here.

9.29.3 Content Hub Content Creation

With Content Hub it is possible to create *CoreMedia* content from *Content Hub* Items. Therefore, you need to implement the interface `ContentHubTransformer`. Pressing the "Create Content" button in the library's toolbar, or dragging and dropping a selection of Content Hub items or folders to the Studio library will trigger a content import from the selected Content Hub Objects to *CoreMedia* content.

> **WARNING**
>
> When importing content from an external system to CoreMedia, it is the responsibility of the `ContentHubTransformer` to deliver a valid `ContentModel`. Also, a `ContentHubTransformer` should check the content that is about to be imported for security issues!

A `ContentHubAdapter` must implement the `transformer()` method, which returns a `ContentHubTransformer` suitable for the adapter's items. A `ContentHubTransformer` returns a `ContentModel`. `ContentModel`s are used as placeholders for contents to be created. A `ContentHubTransformer` should never create content on its own but always use `ContentModel`s. This ensures that all existing `ContentWriteInterceptor`s of *Studio* are executed for the newly created content as well. The following example shows a Transformer implementation for RSS:

```
public ContentModel transform(Item item,
                              ContentHubAdapter contentHubAdapter,
                              ContentHubContext contentHubContext) {
  RSSItem rssItem = (RSSItem) item;
  ContentModel contentModel =
ContentModel.createContentModel(item.getRssEntry().getTitle(), item.getId(),
 item.getCoreMediaContentType());
  contentModel.put("title", item.getName());
  String description = extractDescription(item);
  if (description != null) {
    contentModel.put("detailText",
ContentCreationUtil.convertStringToRichtext(description));
  }

  SyndEntry rssEntry = item.getRssEntry();
  List<String> imageUrls = feedImageExtractor.extractImageUrls(rssEntry);
  List<ContentModelReference> refs = new ArrayList<>();
  for (String imageUrl : imageUrls) {
    ContentModelReference contentModelRef =
ContentModelReference.create(contentModel, "CMPicture", imageUrl);
    refs.add(contentModelRef);
  }
  contentModel.put("pictures", refs);
```

```
   return contentModel;
}
```

Example 9.118. Implementing a ContentHubTransformer (1)

The example method can be separated into two steps:

- Setting the default content properties for the target content via the Content Model.

  Since the ContentModel is a Content representation, it is possible to add properties, just like for regular content. These properties will be used for the actual content creation by the *Content Hub*.

- Collecting additional references

  Some adapters for the *Content Hub* may want to create additional content for a single transform call, maybe even recursively. An RSS feed for example can contain text and images. Therefore, a CMArticle should be created for the text content, but also CMPicture content items for the images of it. ContentHubTransformers support this by ContentModelReferences. They allow developers to create contents incrementally.

The example below shows the usage of ContentModelReferences for the RSS *Content Hub* adapter:

```
 public ContentModel resolveReference(ContentHubObject owner,
                                      ContentModelReference reference,
                                      ContentHubAdapter contentHubAdapter,
                                      ContentHubContext contentHubContext)
{
  String imageUrl = (String) reference.getData();
    String imageName = ContentCreationUtil.extractNameFromUrl(imageUrl);
    if (imageName == null) {
      return null;
    }
   ContentModel referenceModel = ContentModel.createReferenceModel(imageName,
 reference.getCoreMediaContentType());
     referenceModel.put("data", new UrlBlobBuilder(owner,
 "rssPicture").withUrl(imageUrl).withEtag().build());
     referenceModel.put("title", "Image " + imageName);

     return referenceModel;
}
```

Example 9.119. Implementing a ContentHubTransformer (2)

For every ContentModelReference that has been created within the transform method, the resolveReference method is called. Since the reference data is an image URL, create a new ContentModel of type CMPicture and put the image blob into it.

> **NOTE**
>
> `ContentModelReference`s are resolved recursively. That means if the `ContentModel` that is returned by the `resolveReference` method contains a `ContentModelReference` again, the `resolveReference` method will be called again.

9.29.4 Content Hub Object Preview

Content Hub offers a preview for ContentHub Items and Folders in the library. Selecting a Content Hub Object will show this customizable Preview.

A preview is structured into so called *sections*. A section will be displayed as a `Collapsible Panel` within Studio. A section has a header and can be filled with so called *elements* that consist of key value pairs. An element can display data in form of blob (Picture), Calendar and String. The data is shown in key value pairs. The keys can be localized, or marked as non localizable (for example, if the Content Hub Object name should appear as a label for a preview picture).

If no preview is defined, the ContentHubType Icon, and the Content Object's name will be shown as default preview.

The last section is called *Located In* and shows a list of CoreMedia Content, that was imported from the selected Content Hub Item. This list is not configurable and will only be shown for Content Hub Items, as folders cannot be imported.

Thumbnail Preview

Additionally to the customizeable Preview, every implementation of Content Hub Objects can override the method `getThumbnailBlob`. This URL will be used to render a preview thumbnail of each item inside the Studio library's thumbnail view.

In order to create a custom Content Hub Preview, you must implement the `ContentHubObject` method `getDetails`, which returns a list of `DetailsSection`s. A section can be configured to be non collapsible, also the header can be hidden. Within that Section a list of `DetailsElement` need to be defined. You can put a value of type Calendar, String or `ContentHubBlob` into this element. Set the boolean value `html` of the `DetailsElement` to `true` if the given String should be rendered as HTML. This allows to embed `iframes` into the Content Hub Preview, for example YouTube videos. The inter-

face `ContentHubObject` offers a special constant `SHOW_TYPE_ICON` that can be used to show the Content Hub Object's type icon instead of a picture.

> **WARNING**
>
> Embedding `iframe` from other sources is a security risk. To embed a preview in this way, ensure that the CSP settings of the Studio's `application.prop erties` contains an exception for the URL that is loaded by the `iframe`.

In order to provide preview pictures in form of `ContentHubBlob` you need to provide an object of that type as value for your `DetailsElement`. This blob does not need to contain an InputStream yet, but only metadata and a so called `classifier`. In a second call the method `getBlob` will be called by the framework with the `classifier` in order to resolve the blob. This method must return a `ContentHubBlob` with an InputStream containing the picture's data.

> **WARNING**
>
> Pictures sent as preview to the client should not exceed the limit of 10Mbyte. If that is the case, the framework will not display the picture in the client.

The Labels of Sections and Elements can be localized in your Content Hub Client extension. Therefore, you need to provide a localization key that consists of the `label` and postfix _*sectionItemKey* for Element labels or _*sectionName* for Section Headers.

9.29.5 Content Hub Error Handling

As The Content Hub Framework communicates with an external third-party system, communication errors could occur. These errors are visualized as so called error objects that appear in the Studio library. When clicking on an error object, an error message will appear with a button that offers the possibility to reload the object.

Within the Adapter implementation, the implementer can throw a `Con tentHubException`. This Exception can contain a `ContentHubException ErrorCode` that will result in a custom error message on the client side. Any other Exception will lead to a general error message at the client. The error message can be localized using the following scheme: Concrete classname of the implementation of `ContentHubExceptionErrorCode` combined with the Code. (for example, "ContentHubExceptionRssErrorCode_CUSTOM_ERROR").

Using the postfix "_icon" or "_title" you can also set a title or an Icon from the CoreMedia CoreIcons.

Together with the `ContentHubExceptionErrorCode` a String List of arguments can be passed in the `ContentHubException`. These arguments can be included with the placeholder *{1}* within the localization. Note that the first argument will always be the *connectionId* of the Adapter and can be included in the error message with the placeholder {0}. The arguments passed to the Exception can be shown starting with *{1}*.

9.29.6 Studio Customization

Developing new adapters can require client-side Studio customization too. These customizations are easily done by overriding the `_properties.ts` files of the *Content Hub* Studio plugin. Examples for overriding property files can be found in documentation or the file `BlueprintFormsStudioPlugin.ts` inside the *CoreMedia Blueprint*.

> **NOTE**
>
> *CoreMedia Content Hub Adapters and Studio Customization*
>
> Since the *Content Hub* supports connecting to *CoreMedia* systems as well, the rendering of the labels, type icons and names of *Content Hub* entities can be handled the same way as they are for content entities. When the ContentHub-Types of an adapter have the same values as the CoreMedia ContentTypes (for example, "CMArticle") *Studio* will try to display the entities using the existing content type icons and labels out of the box.

9.29.6.1 Customizing Labels and Icons

The properties file `ContentHub_properties.ts` contains the label and icon values for adapter folders and items. New entries can simply be added by overriding this file. The *Content Hub* will always try to lookup an existing icon or type name mapping in the resource bundles first. If no match is found, the technical name or a default name will be displayed, depending on the `ContentHubType`.

The file expects entries with the following format:

Adapters

```
adapter_type_<ADAPTER_FACTORY_ID>_name : "<TYPE_LABEL>",
adapter_type_<ADAPTER_FACTORY_ID>_icon : CoreIcons_properties.<KEY_FOR_ICON>,
```

For icon values, *CoreMedia* recommends to use the existing `CoreIcons` resource. If a `null` value is returned for the `getName()` method of the `ContentHubAdapterType` interface, this `name` property will be used instead. If no such property has been defined, the `factoryId` will be used as tree node name instead.

Folders

```
folder_type_<ADAPTER_FOLDER_TYPE>_name : "<TYPE_LABEL>",
folder_type_<CONTENTHUB_TYPE>_icon : CoreIcons_properties.<KEY_FOR_ICON>,
```

Folders have their own `type` attribute, which allows modifying the icon and label for folders. The label `Folder` and the folder icon are the default values for *Content Hub* folders.

Items

```
item_type_<CONTENTHUB_TYPE>_name : "<TYPE_LABEL>",
item_type_<CONTENTHUB_TYPE>_icon : CoreIcons_properties.<KEY_FOR_ICON>,
```

If no icon is found for the given item type (and the item type is not a *CoreMedia* content type), the *Content Hub* will try to use the entity's name suffix (file suffix) to resolve a matching icon. If no match is found, the property `Item_icon` will be used as fallback.

9.29.6.2 Custom Columns

The *Content Hub* allows adding custom columns to the *Studio* library by implementing the interface `ColumnProvider`. The "Type" is always displayed as first column, regardless of any ColumnProvider.

Implementing ColumnProviders

By default, the *Content Hub* displays the columns "Type" and "Name". The "Name" Column is provided by a default ColumnModelProvider. In order to display custom columns you can add a `ColumnModelProvider` for your Adapter. This enables you to add Columns that show data in form of String, Date or Icon (with or without

Text). The following code shows an example implementation for the *CoreMedia* adapter:

```
public class CoreMediaColumnModelProvider implements ColumnModelProvider {
  @Override
  public Boolean isApplicable(String factoryId) {
    return "coremedia".equals(factoryId);
  }

  @Override
  public List<Column> getColumns(Folder folder) {
    List<Column> columns = new ArrayList<>();
    //we should set at least one column to flex, so the collection view's
width will be filled.
    columns.add(new DefaultAdapterColumn("name", "nameValue", 100, -1, false,
 false, false, true, false));
    columns.add(new DefaultAdapterColumn("name", "name", 150, -1));
    columns.add(new DefaultAdapterColumn("status", "status", 100, -1));
    return columns;
  }

  @Override
  public List<ColumnValue> getColumnValues(ContentHubObject hubObject) {
    //note that a folder or item may be passed here
    CoreMediaContentHubObject coreMediaEntity = (CoreMediaContentHubObject)
 hubObject;
    //the backing entity is content, so we don't have to care about the
concrete type
    Content content = coreMediaEntity.getContent();

    List<ColumnValue> columnValues = new ArrayList<>();
    columnValues.add(new DefaultAdapterColumnValue("name",
hubObject.getDisplayName(), null, null));
    columnValues.add(new DefaultAdapterColumnValue("status",
getLifecycle(hubObject), null, null));

    return columnValues;
  }
}
```

Example 9.120. Defining a Custom ColumnModelProvider

The example adds two columns `name` and `status`, using the `DefaultAd apterColumn` class. The index for the column is set to '1' and '2' which ensures that the "name" column is located before the "status" column. It is also possible to set a width. However, there should be at least one column that has a flexValue set. This will ensure the columns fill the width of the library.

The header of the column model can be localized through the properties file `ContentHub_properties.ts`:

```
<COLUMN_TITLE>_header : "<COLUMN_LABEL>"
```

If no matching label was found, the original title value will be used as fallback.

9.30 Feedback Hub

The *CoreMedia Feedback Hub* offers the possibility to provide feedback for CoreMedia content. It is possible to connect external systems to Feedback Hub in order to collect feedback.

This section describes the integration of a feedback providing system into the *CoreMedia Feedback Hub* These integrations are implemented as extensions for *CoreMedia*, using the extension point `studio-lib` (and `studio` if localization of custom error messages is required).

Currently, the Feedback Hub supports Validators, *Editorial Comments* and the Keywords Integration. Conceptually, it is designed to support arbitrary flavors of feedback, though, and future versions of *CMCC* may introduce more Feedback Hub features. The Feedback Hub API already reflects the possibility of other feedback flavors, so that for now some concepts like Adapters may appear unnecessarily generic with respect to the Keywords Integration.

9.30.1 Basic Setup

The following part explains which interfaces have to be implemented in your `studio-lib` extension. The Error handling and its localization for the client is explained in Section 9.30.4, "Error handling" [334]. How to configure your Adapter bindings is explained in Section 9.30.2, "Adapter Configuration" [331].

`<YOUR_ADAPTER_NAME>Settings`

First it is necessary to provide a Settings interface, which has getters for the configurable data of your Adapter. Settings interfaces are used to map Adapter specific connection parameters (like credentials) to Java code. You only have to declare the Settings interface according to the data your Adapter needs. Implementations are generated automatically, backed with the data of your configuration. The `getter` methods of these interfaces must match the corresponding fields of the `settings` struct as described in Section 9.30.2, "Adapter Configuration" [331]. The name of the interface is arbitrary, the `Settings` suffix is just a convention.

`com.coremedia.feedbackhub.adapter.FeedbackHubAdapterFactory`

An implementation of a `FeedbackHubAdapterFactory` delivers instances of a `FeedbackHubAdapter`, that is used for the actual connection to an ex-

ternal system. The id that is returned within the `getId` method, has to match the `factoryId` value from your Section 9.30.2, "Adapter Configuration" [331]. In the `create` method you must return an instance of the specific `Feedback HubAdapter`. The method receives the adapter specific settings as parameter. Any errors that occur during the creation of a `FeedbackHubAdapter` can be thrown as `FeedbackHubException` (described in Section 9.30.4, "Error handling" [334]) if they should result into a specific error message at the client. Any other exceptions result in a general error message at the client.

The factory needs to be available as a Spring bean within the Spring context of the `studio-server`. Therefore, you also must provide a Spring configuration file that instantiates the FeedbackHubAdapterFactory as Spring Bean (@Bean), so that it can be collected by the Feedback Hub framework.

`com.coremedia.feedbackhub.adapter.keywords.BlobKeywordsFeedbackHubAdapter`

A `BlobKeywordsFeedbackHubAdapter` is a predefined adapter which delivers a list of KeyWords for a given blob and a locale. The result type of the `getKeywords` method is `java.util.concurrent.CompletionStage`. This enables you to implement long running operations, like round trips to external systems, with the stage's asynchronous execution facilities so that no threads are blocked. `getKeywords` is called, when a user requests keywords for CoreMedia content. It will receive a blob from the content (which is configured in the Section 9.30.2, "Adapter Configuration" [331]) and the locale from the content. Any errors that occur during the calculation of keywords can be thrown as `FeedbackHubException` (described in Section 9.30.4, "Error handling" [334]) if they should result into a specific error message at the client. Any other exceptions will result in a general error message at the client.

The following table shows the settings that are configurable for the `Blob KeywordsContentFeedbackProviderSettings` interface.

Name	Type	Description
sourceBlobProp-erty	String	The name of the content blob property to analyze and compute keywords for.

Table 9.10. Settings Struct Properties

`com.coremedia.feedbackhub.adapter.text.TextContentFeedbackProvider`

A `TextContentFeedbackProvider` is a predefined adapter which delivers a list of `FeedbackItem`s for a given list of property values. The result type of the `analyzeText` method is `java.util.concurrent.Completion`

`Stage`. This enables you to implement long running operations, like round trips to external systems, with the stage's asynchronous execution facilities so that no threads are blocked. Any errors that occur during the calculation of keywords can be thrown as `FeedbackHubException` (described in Section 9.30.4, "Error handling" [334]) if they should result into a specific error message at the client. Any other exceptions will result in a general error message at the client.

The following table shows the settings that are configurable for the `TextCon tentFeedbackProviderSettings` interface.

Name	Type	Description
sourceProperties	String	A comma separated list of property names that are used to extract text from. If the text is a markup field, the markup will be converted to plain-text automatically. Invalid properties will be ignored since it is possible to configure the adapter for multiple content types.

Table 9.11. Settings Struct Properties

`com.coremedia.feedbackhub.adapter.FeedbackHubErrorCode`

The `FeedbackHubErrorCode` is part of the error handling, described in Section 9.30.4, "Error handling" [334] and needs to be implemented by an `enum`. The `enum` stores all error codes for errors that can occur within the specific adapter implementation or its factory. In case of an error the code is transferred to the client where it is shown as an error message in Feedback Hub's window nagbar.

9.30.2 Adapter Configuration

The configuration for a Feedback Hub Adapter can be provided via a settings content item. For Feedback Hub every adapter needs its own content item. Here it is possible to configure for which content type and also in which tab of the Feedback Hub window an adapter appears. Adapters can be grouped into panels via the `groupId`. If two adapters share the same group ID, they appear in the same panel. For every new `groupId` a new panel is shown in the Feedback Hub window. How to configure the item see Section 9.30.3, "Localization" [333]. You can define a Feedback Hub configuration globally by placing the content item in the folder structure `/Settings/Options/Settings/Feedback Hub`. If you want to define a site specific Feedback Hub configuration, you need to place the content item in the folder structure `SITE_ROOT_FOLDER/Op`

`tions/Settings/Feedback Hub`. The name of the content item itself is not relevant. The settings content item must hold a struct with at the following entries:

Name	Type	Description
factoryId	String	The identifier of the implementing factory class. The value must match the return value of the method `getId`.
targetProperty	String	The name of the property, where the feedback refers to
groupId	String	The `groupId` configures in which tab of the Feedback Hub window the Adapter is shown. If several adapters share the same `groupId`, they are shown in the same tab.
contentTypes	String	A list of comma separated values for which content types the adapter is shown.
enabled	Boolean	Only if set to `true`, the adapter appears.
reloadMode	String	Can be set to `manual`, `auto` or `none`. If set to `auto`, the adapter feedback will reload the feedback automatically when the corresponding `observedProperties` are configured and changed. If set to `manual`, the user will have to reload the feedback manually. If set to `none`, no reload attempt will be triggerd.
observedProper-ties	String	A comma separated list of property names that the Feedback Hub will listen to if `reloadMode` is set to `manual` or `auto`. Unknown properties will be ignored silently, because the adapter may have been configured for different content types which don't share the same property names.
settings	Struct	A struct that defines the attributes that will be passed to the specific adapters. The attributes set in your settings struct have to match the Settings-Interface, mentioned in Section 9.30.1, "Basic Setup" [329]. The values of the struct will be passed to your Adapter

Table 9.12. Connection Struct Properties

> **CAUTION**
>
> Please take care of security protection. The `settings` should not contain secrets like passwords or API tokens. For example, better store them in a dedicated secrets manager and only pass them through to the external system in your custom adapter implementation during runtime.
>
> The following rules of thumb provide additional protection for sensitive data:
>
> - Restrict access to the `Feedback Hub` folder to people that actually configure the adapters.
> - Do not publish adapter configuration. The adapters are only accessed in the *CoreMedia Studio*. As such they are not relevant on the live side.
> - Ensure that there are not links to Settings content. The adapter configuration is identified by means of their location. Links are not required. A link would risk that the Settings content is accidentally published if for example its referring content is published.
> - Exclude the content and folder from website search by checking the corresponding option.
> - Prevent access to arbitrary content from the *Headless Server* and from other client applications. See Section 3.5, "Security" in *Headless Server Manual* for more details.

9.30.3 Localization

In *CoreMedia Feedback Hub* you may provide localization for the UI elements in the `FeedbackHubPanel` of your adapter. The localization for Feedback Hub needs to be provided in an extra Feedback Hub extension for `studio-client`. Within that extension you need to provide a `StudioPlugin` that holds the configuration which copies the resource bundle of your adapter to the `Feed backHub_properties.ts`.

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import resourceManager from "@jangaroo/runtime/l10n/resourceManager";
import CopyResourceBundleProperties from
"@coremedia/studio-client.main.editor-components/configuration/CopyResourceBundleProperties";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import FeedbackHub_properties from
"@coremedia/studio-client.main.feedback-hub-editor-components/FeedbackHub_properties";
import FeedbackHubCustom_properties from "./FeedbackHubCustom_properties";
```

```
class CustomFeedbackHubStudioPlugin extends StudioPlugin {
  constructor(config: Config<StudioPlugin>) {
    super(ConfigUtils.apply(Config(CustomFeedbackHubStudioPlugin, {

      rules: [],

      configuration: [
        new CopyResourceBundleProperties({
          destination: resourceManager.getResourceBundle(null,
FeedbackHub_properties),
          source: resourceManager.getResourceBundle(null,
FeedbackHubCustom_properties),
        }),
      ],
    }), config));
  }
}

export default CustomFeedbackHubStudioPlugin;
```

Example 9.121.

In your `FeedbackHubCustom_properties` you can provide a localization for the following items. The <factoryId> value needs to match the `factoryId` of your configuration, `collection` the value of the `collection` field of your `FeedbackItem`:

Item	naming pattern	Description
Main Tab Icon	<factoryId>_icon-Cls	The tab icon that is shown for the tab of your adapter's groupID
Main Tab Title	<factoryId>_title	Title that is shown for the panel of you Adapter
Main Tab Tooltip	<factoryId>_tooltip	Tooltip that is shown for the panel of you Adapter
Sub-Tab Title	<factoryId>_<collection>_tab_title	The title of a collection tab
Sub-Tab Tooltip	<factoryId>_<collection>_tab_tooltip	The tooltip used for a collection tab

Table 9.13. Localization for Custom Feedback Hub Adapter

9.30.4 Error handling

Feedback Hub offers the possibility to send error codes and arguments to the client, where they can be localized to error messages (please see Section 9.30.3,

"Localization" [333]). The arguments are of Type String and provide the possibility to create dynamic error messages. In the implementations of the `Content FeedbackProviderFactory` and the `ContentFeedbackProvider`, errors that should result in a specific error message to the client need to be wrapped into a `FeedbackHubException`, with a specific `FeedbackHubErrorCode` and an optional list of arguments. This exception will be caught by the framework and the code will be passed to the client.

> **NOTE**
>
> If errors occur which result in an exception not of Type `FeedbackHubException` they will be caught by the framework and delivered to the client with a general error message.

In your custom Feedback Hub adapter, you need to use the following naming pattern in order to localize the error messages: `<groupId>_error_<Error Code_of_CustomFeedbackHubErrorCode_Implementation>`. If you have for example a `CustomFeedbackHubErrorCode` Enum which is implementing `FeedbackHubErrorCode` with the value `ERROR_CUSTOM` and the `groupId myAdapter`, the localization would be: `myAdapter_error_ERROR_CUSTOM`

Within the localization value you can use placeholders like {0}, {1} etc. that are filled with the arguments that were passed to the `FeedbackHubException`. The arguments occur in the same order as they were passed to the exception.

> **NOTE**
>
> The first argument is always the ID of the binding. It is set by the framework and can be referenced with `{0}`!

An error appears in the Feedback Hub window in a red nagbar at the bottom of the Window. (Error Appearance is shown in Section 4.7.6, "Getting Keyword Recommendations" in *Studio User Manual*)

9.30.5 FeedbackItem Rendering

The `FeedbackItem`s that are rendered by the Studio are automatically sorted, depending on their attributes. Every `FeedbackItem` can override the method `getCollection()` in order to render it to a specific panel of a sub-tab panel inside the feedback's tab. By default, the value of the collection is `null`, which means that the `FeedbackItem` is rendered directly on the feedback

tab panel. The given picture shows an example how this rendering is used within the "Imagga" integration:
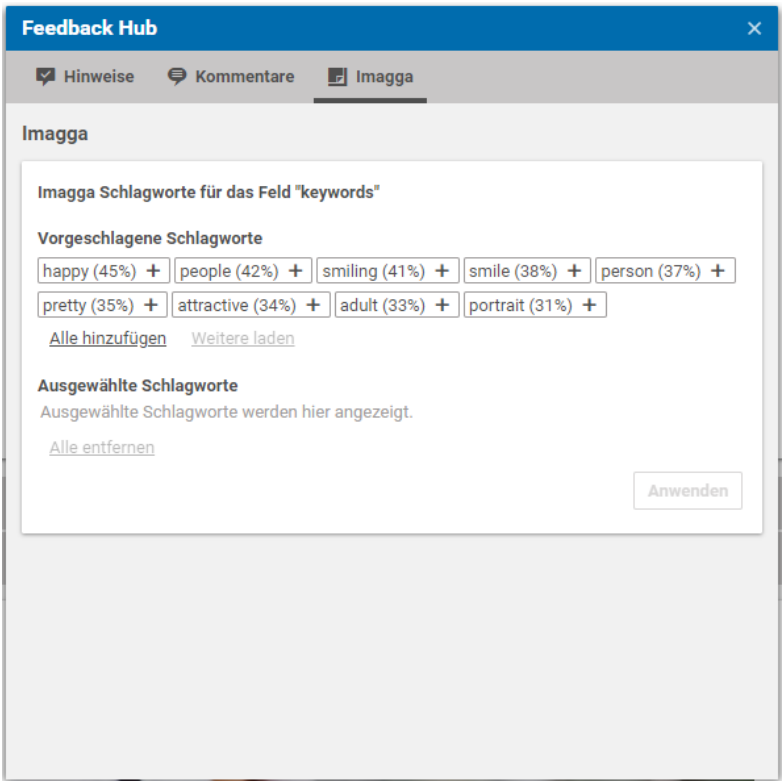
Figure 9.14. Default Rendering of FeedbackItems used for the CoreMedia Labs project "Imagga"

For more complex feedback, the feedback tab supports some predefined areas and `FeedbackItem` types. These special types are described in section Section 9.30.6, "Predefined FeedbackItems" [338]. The different areas are highlighted in the picture below.
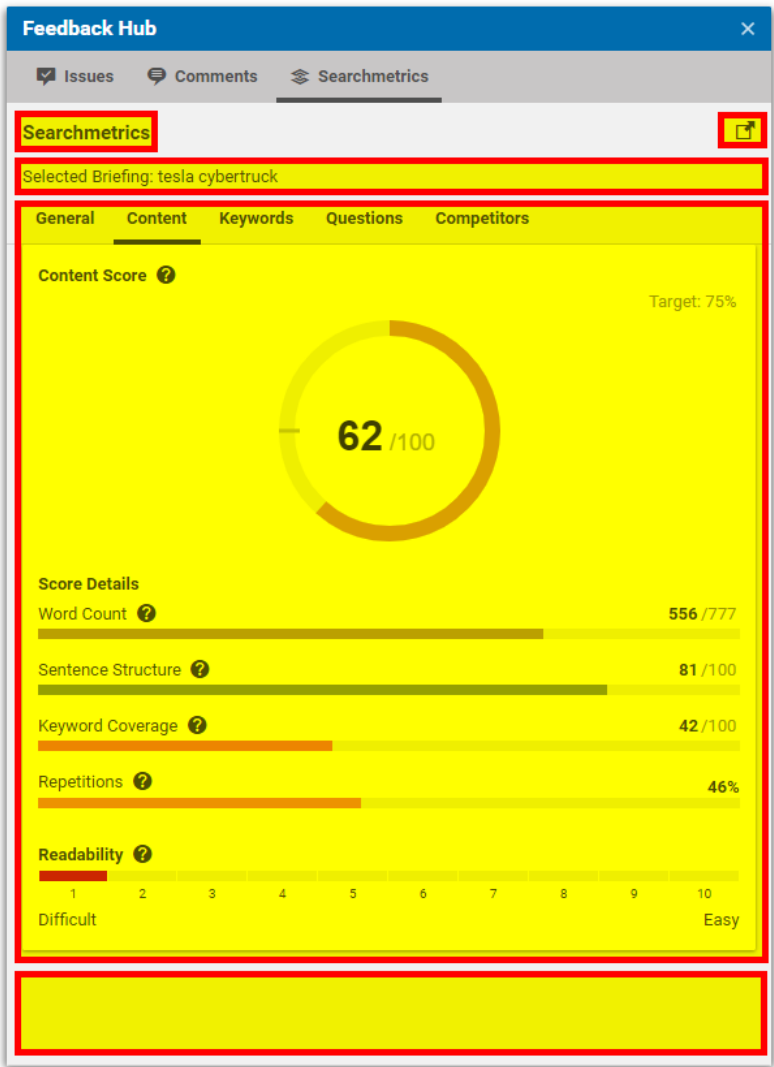
Figure 9.15. Tabbed Rendering of FeedbackItems used for the CoreMedia Labs project "Searchmetrics"

At the top the name of the integration is rendered. If available, an additional link button is rendered which usually points to the external system that is integrated.

Above and below of the tab panel in the middle of the window, the header and footer items are displayed. If your `FeedbackItem` instance returns `header` or `footer` for `getCollection`, the item will be rendered into the corresponding area. Therefore, these collection names are reserved words. For all other collections a separate tab will be created and the `FeedbackItems` will be rendered onto these tabs. For example, the given picture shows the rendering of 6x `FeedbackItems` (1x gauge + 5 score bars) which all return the collection name `content` and therefore rendered onto the "Content" tab (for localization see Section 9.30.3, "Localization" [333]).

9.30.6 Predefined FeedbackItems

The Feedback Hub comes with a list of predefined `FeedbackItems`, which means that you only have to implement some Java code in order to render feedback graphs and widgets in the Studio. In this section you find the list of available `FeedbackItems`, their layout and configuration parameters.

Please note that, depending on the type of the `FeedbackItem`, new instances are created through a corresponding builder, or through the factory class `FeedbackItemFactory`.

9.30.6.1 Score Bar

A `ScoreBarFeedbackItem` renders a colored score bar with the actual and maximum value. They are created trough method `ScoreBarFeedback Item.builder`.

Main Scores
Content Score ❓   **78** /100

Figure 9.16. Example of a ScoreBarFeedbackItem

Property	Type	Default Value	Description
collection	String	null	The sub tab the panel should be rendered too.
title	String	null	The title of the panel.
help	String	null	Help text that, if set, will be rendered as a help icon next to the title.

Property	Type	Default Value	Description
label	String	`null`	The label that is shown above the bar.
value	float	`0`	The value of the score.
maxValue	float	`0`	The maximum value of the bar.
color	String	`null`	If set, the whole bar will be rendered with this color. The `reverseColors` attribute is ignored in this case.
reverseColors	boolean	`false`	If true, the color of the bar be reversed (green to red).
decimalPlaces	float	`0`	The number of decimal places to be rendered for the value.

Table 9.14. `FeedbackItem ScoreBarFeedbackItem`

9.30.6.2 Rating Score Bar

A `RatingBarFeedbackItem` renders a colored and segmented score bar with additional labels. They are created trough method `RatingBarFeedback Item.builder`.

Figure 9.17. Example of a RatingBarFeedbackItem

Property	Type	Default Value	Description
collection	String	`null`	The sub tab the panel should be rendered too.
title	String	`null`	The title of the panel.
help	String	`null`	Help text that, if set, will be rendered as a help icon next to the title.

Property	Type	Default Value	Description
label	String	null	The label that is shown above the bar.
value	float	0	The value of the score.
maxValue	float	0	The maximum value of the bar.
color	String	null	If set, the whole bar will be rendered with this color. The reverseColors attribute is ignored in this case.
reverseColors	boolean	false	If true, the color of the bar be reversed (green to red).
minLabel	String	null	The label to render at the beginning of the score bar.
maxLabel	String	null	The label to render at the end of the score bar.

Table 9.15. `FeedbackItem RatingBarFeedbackItem`

9.30.6.3 Percentage Score Bar

A `PercentageBarFeedbackItem` renders a colored and score bar with a percentage value. They are created trough method `PercentageBarFeed backItem.builder`.
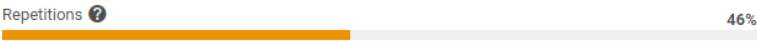
Figure 9.18. Example of a PercentageBarFeedbackItem

Property	Type	Default Value	Description
collection	String	null	The sub tab the panel should be rendered too.
title	String	null	The title of the panel.

Property	Type	Default Value	Description
help	String	null	Help text that, if set, will be rendered as a help icon next to the title.
label	String	null	The label that is shown above the bar.
value	float	0	The value of the score.
color	String	null	If set, the whole bar will be rendered with this color. The reverseColors attribute is ignored in this case.
reverseColors	boolean	false	If true, the color of the bar be reversed (green to red).
decimalPlaces	float	0	The number of decimal places to be rendered for the value.

Table 9.16. `FeedbackItem PercentageBarFeedbackItem`

9.30.6.4 Gauge Bar

A `GaugeFeedbackItem` renders a colored percentage gauge graph with additional labels and links. They are created trough method `GaugeFeedbackItem.builder`.
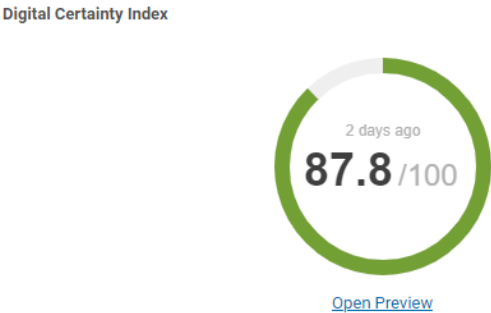
Figure 9.19. Example of a GaugeFeedbackItem

Property	Type	Default Value	Description
collection	String	null	The sub tab the panel should be rendered too.
title	String	null	The title of the panel.
help	String	null	Help text that, if set, will be rendered as a help icon next to the title.
value	float	0	The value of the score.
decimalPlaces	int	0	The number of decimal places for the value.
targetValue	float	0	The target percentage to achieve. If set, a target indicator will be rendered on the gauge.
gaugeTitle	String	null	The title that is shown below the gauge.
url	String	null	If set, a link button will be rendered below the gauge.
linkText	String	null	The text used for the gauge link button.
age	long	0	The last time the data of this gauge was fetched, milliseconds from 1970.

Property	Type	Default Value	Description
color	String	`null`	If set, the whole bar will be rendered with this color. The `reverseColors` attribute is ignored in this case.
reverseColors	boolean	`false`	If true, the color of the bar be reversed (green to red).

Table 9.17. `FeedbackItem GaugeFeedbackItem`

9.30.6.5 Keyword Selector

A `KeywordFeedbackItem` renders are keyword selector with tag suggestions based on an external system. They are created trough method `KeywordFeedbackItem.builder`.
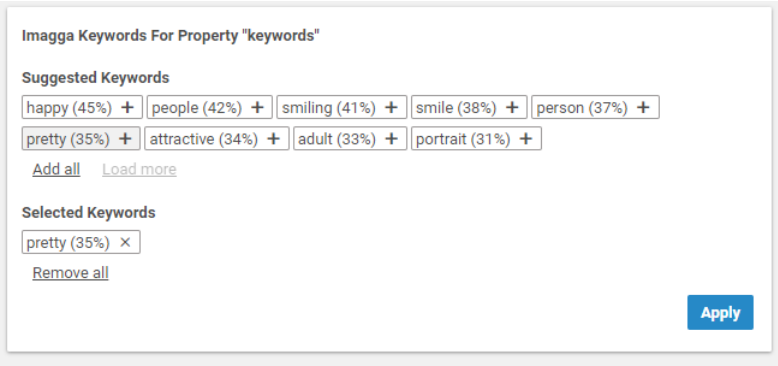
Figure 9.20. Example of a KeywordFeedbackItem with service "Imagga".

Property	Type	Default Value	Description
collection	String	`null`	The sub tab the panel should be rendered too.
title	String	`null`	The title of the panel.
help	String	`null`	Help text that, if set, will be rendered as a help icon next to the title.

Property	Type	Default Value	Description
property	String	null	The name of the property the keywords are applied to.
keywords	List<Keyword>		The keywords that should be shown as suggestions.

Table 9.18. `FeedbackItem KeywordFeedbackItem`

9.30.6.6 Comparing Score Bar

A `ComparingScoreBarFeedbackItem` renders two score bars for direct comparison. They are created trough method `ComparingScoreBarFeedbackItem.builder`.

Figure 9.21. Example of a ComparingScoreBarFeedbackItem

Property	Type	Default Value	Description
collection	String	null	The sub tab the panel should be rendered too.
title	String	null	The title of the panel.
help	String	null	Help text that, if set, will be rendered as a help icon next to the title.
label	String	null	The label that is shown above the bar.
value1	float	0	The value of the first score.
value2	float	0	The value of the second score.
maxValue1	float	0	The maximum value of the first bar.

Property	Type	Default Value	Description
maxValue2	float	0	The maximum value of the second bar.
targetValue1	float	0	The target value of the first bar.
targetValue2	float	0	The target value of the second bar.
color1	String	null	If set, the whole bar will be rendered with this color. The reverseColors attribute is ignored in this case.
color2	String	null	If set, the whole bar will be rendered with this color. The reverseColors attribute is ignored in this case.
reverseColors	boolean	false	If true, the color of the bar be reversed (green to red).
reverse	boolean	false	If true, the arrow direction is reversed.
unitLabel	String	null	The unit string the difference is measured in.
unitTitle	String	null	The bold title of the right column.
barTitle	String	null	The bold title of the left column.

Table 9.19. `FeedbackItem ComparingScoreBarFeedbackItem`

9.30.6.7 Label

A `LabelFeedbackItem` is simply used to render text. They are created trough method `LabelFeedbackItem.builder`.

Quality Assurance Issues (10)

Figure 9.22. Example of a bold LabelFeedbackItem

Property	Type	Default Value	Description
collection	String	`null`	The sub tab the panel should be rendered too.
bold	boolean	`false`	Set to true to render a bold label.
text	String	`null`	The text to render.
args	Array	`null`	Optional params to parameterize the label.

Table 9.20. `FeedbackItem LabelFeedbackItem`

9.30.6.8 External Link

An `ExternalLinkFeedbackItem` renders an external link. They are created trough factory class `FeedbackItemFactory`.

Accessibility Issues (1)
Click here to see all issues in Siteimprove

Figure 9.23. Example of a ExternalLinkFeedbackItem used inside a "Siteimprove" integration

Property	Type	Default Value	Description
collection	String	`null`	The sub tab the panel should be rendered too.
title	String	`null`	The title of the panel.
url	String	`null`	The URL of the link.
linkText	String	`null`	The link label.

Table 9.21. `FeedbackItem ExternalLinkFeedbackItem`

9.30.6.9 Empty

An `EmptyFeedbackItem` is used just to create a sub tab with empty content. This might be useful when a tab's feedback is not yet available, but the tab should still be visible. They are created trough factory class `FeedbackItemFactory`.

Property	Type	Default Value	Description
collection	String	null	The sub tab the panel should be rendered too.

Table 9.22. `FeedbackItem EmptyFeedbackItem`

9.30.6.10 Feedback Link

A `FeedbackLinkFeedbackItem` is a special external link button that is rendered in the upper right corner of a feedback tab and usually points to the external system that is integrated by the tab. Therefore it does not belong to any collection. `FeedbackLinkFeedbackItem`s are created trough factory class `FeedbackItemFactory`.

Property	Type	Default Value	Description
url	String	null	The URL the external link button should point to.

Table 9.23. `FeedbackItem FeedbackLinkFeedbackItem`

9.30.6.11 Error Feedback

A `ErrorFeedbackItem` is used to render an error message for the Feedback Hub. `ErrorFeedbackItem`s are created trough factory class `FeedbackItemFactory`. For more details see also Section 9.30.4, "Error handling" [334].
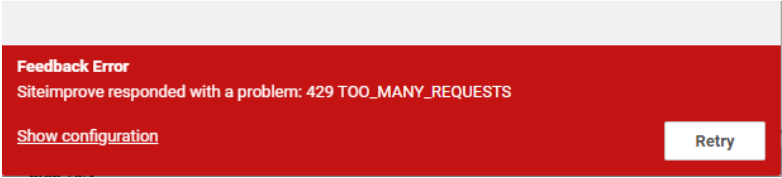
Figure 9.24. Example of a ErrorFeedbackItem inside an integration of "Siteimprove"

Property	Type	Default Value	Description
errorCode	Feedback-HubError-Code	List	Additional error details.

Table 9.24. `FeedbackItem ErrorFeedbackItem`

9.30.7 Custom Adapters for Feedback Hub

As mentioned before, the Feedback Hub comes with some predefined `Feed backAdapter` implementations. If you want implement your own feedback adapter and the existing ones are not suitable for you, custom `FeedbackPro vider` implementations can be used to return any kind of feedback.

A detailed example for a custom Feedback Hub adapter implementation can be found here: https://github.com/CoreMedia/feedback-hub-adapter-tutorial

The tutorial describes how to implement feedback based on a CoreMedia content repository. It shows how predefined `FeedbackItems` can be used to display graphs and charts in the Studio. The available `FeedbackItems` are described in the section Section 9.30.6, "Predefined FeedbackItems" [338].

9.30.8 Editorial Comments for Feedback Hub

Editors have the possibility to write comments for certain property fields in a content form and get an overview of existing comments in the *CoreMedia Feedback Hub*. While comments are enabled for many property fields per default,

you might want to enable them for custom property fields or content forms, or disable some of them entirely. This section describes how to do that.

In order to enable comments on a property field, the following prerequisites have to be met:

1. The property field has to be registered by the `editorialCommentProp` `ertyRegistryService`.
2. The property field needs to implement the `SideButtonMixin` to be able to display a comment button.
3. The field needs to have the `PropertyFieldPlugin` as one of its plugins.

9.30.8.1 Register PropertyFields for Editorial Comments

As mentioned in the above list, property fields have to be registered in a global `editorialCommentPropertyRegistryService`. The Service lets you register `PropertyRegistryModel`s based on any combination of a property name, a component's xtype or the CapType of a content in a content form. Please note, that *Editorial Comments* have been enabled per default for a variety of property field, such as text areas, text fields and link lists. The `xtypes` of these components are already registered in the `editorialCommentPropertyRe` `gistryService`.

Just like the registration of property fields, the registry service can also exclude property fields from the *Editorial Comments* feature. See the example below to understand how registration and exclusion is done:

```
import Config from "@jangaroo/runtime/Config";
import session from "@coremedia/studio-client.cap-rest-client/common/session";
import CapType from "@coremedia/studio-client.cap-rest-client/common/CapType";
import StudioPlugin from
"@coremedia/studio-client.main.editor-components/configuration/StudioPlugin";
import IEditorContext from
"@coremedia/studio-client.main.editor-components/sdk/IEditorContext";
import editorialCommentPropertyRegistryService from
"@coremedia/studio-client.main.feedback-hub-editor-components/components/comments/service/propertyregistry/editorialCommentPropertyRegistryService";
import PropertyRegistryModel from
"@coremedia/studio-client.main.feedback-hub-editor-components/components/comments/service/propertyregistry/PropertyRegistryModel";
import CKEditor5RichTextArea from
"@coremedia/studio-client.ext.ckeditor5-components/CKEditor5RichTextArea";

class EditorialCommentsStudioPluginBase extends StudioPlugin {
  constructor(config: Config<StudioPlugin> = null) {
    super(config);
  }

  override init(editorContext: IEditorContext): void {
    super.init(editorContext);

    // enable comments for title property in article content forms
    editorialCommentPropertyRegistryService.register(
     new PropertyRegistryModel(null, "title", this.#getCapType("CMArticle"))
```

```
    );

    // enable comments for all richtext areas
    editorialCommentPropertyRegistryService.register(
      new PropertyRegistryModel(CKEditor5RichTextArea.xtype, null, null)
    );

    // disable comments for detailText richtext property in pages
    editorialCommentPropertyRegistryService.exclude(
      new PropertyRegistryModel(null, "detailText",
this.#getCapType("CMChannel"))
    );
  }

  #getCapType(contentName: String): CapType {
    return session._.getConnection().getContentRepository()
      .getContentType(contentName);
  }
}

export default EditorialCommentsStudioPluginBase;
```

As you can see, the `propertyName` must be passed without its `"proper`
`ties."` prefix. You can use the `editorialCommentPropertyRe`
`gistryService` in your own Studio plugins to customize the default config‐
uration.

> **WARNING**
>
> Please note, that the order of registrations and exclusions is important, since
> excluded property fields might be registered again by a following registration.

9.30.8.2 Enable Editorial Comments for Custom PropertyFields

In order to use the *Editorial Comments* feature for custom property fields, the
property field needs to implement the `SideButtonMixin` and call `init`
`SideButtonMixin()` in its constructor. This mixin enables the component
to render a floating button in its top right corner. The property field also needs
to make use of the `PropertyFieldPlugin` as shown in the example below:

```
import Config from "@jangaroo/runtime/Config";
import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
import Container from "@jangaroo/ext-ts/container/Container";
import PropertyFieldPlugin from
"@coremedia/studio-client.main.editor-components/sdk/premular/PropertyFieldPlugin";

class CustomField extends Container {

  static override readonly xtype: string = "my.customField";

  constructor(config: Config<Container>) {
    super(ConfigUtils.apply(Config(CustomField, {
      sideButtonVerticalAdjustment: 10,
```

```
        sideButtonHorizontalAdjustment: 20,
        sideButtonRenderToFunction: host => host["bodyEl"],
        plugins: [
          Config(PropertyFieldPlugin, {
            propertyName: "description",
          }),
        ],
      }), config));
    }
}

export default CustomField;
```

As shown in the example, `CustomField` implements the `SideButtonMixin` and now provides a variety of properties to render the button correctly. You can define offsets to change the position or provide a function that specifies where the button should be rendered in the component hierarchy. You could also disable the button by setting `sideButtonDisabled` to true;

NOTE

The title of a comment thread in the FeedbackHub depends on the property name, set in the corresponding `PropertyFieldPlugin`. These titles are localized just like in the content form. This means, you will have to provide a localization in the form <CapType>_<PropertyName>_text in `BluePrintDoc umentTypes_properties.ts`. (e.g. CMChannel_title_text)

If no localization is given, the key will be displayed as a fallback. Also, if the property field label is hidden by a surrounding `PropertyFieldGroup`, the comment thread will automatically use the title of the `PropertyFieldGroup` as the label of the comment thread.

9.30.8.3 Notification for Editorial Comments

Whenever a new comment has been created, users who participated in the content or the comments will get notifications. As a studio developer you can fine tune which user will get a notification by disable a user lookup strategy via spring property. For more details have a look at Section 3.4.10, "Editorial Comments Configuration" in *Deployment Manual*.

9.30.9 Keywords Integration for Feedback Hub

Feedback Hub offers an API to connect external systems to CoreMedia that provide keywords for selected content. The connection from CoreMedia to the external system is called Adapter.

If an Adapter is implemented and configured as shown below, it appears in the configured tab of Feedback Hub window (how to configure tabs can be found here: Section 9.30.2, "Adapter Configuration" [331]). In the tab the user has the possibility to trigger a request for keywords. The user guide for Feedback Hub can be found here: Section 4.7.6, "Getting Keyword Recommendations" in *Studio User Manual*.

In order to connect CoreMedia to an external system, it is necessary to implement the interfaces `FeedbackHubAdapterFactory`, `BlobKeywordsFeedbackHubAdapter` and `FeedbackHubErrorCode` in a `studio-lib` extension. Furthermore, a settings content item must be provided that configures where (for which ContentType) and what keyword feedback (which external system) is shown (see Section 9.30.2, "Adapter Configuration" [331]).

9.31 User Manager

CoreMedia Studio comes with an integrated user manager UI that allows admin-istrating the groups and users that are allowed to access the different *CoreMedia* components and content. It also includes a rights management where access types for content are configurable for groups and users. This section describes the overall configuration properties of the *CoreMedia Studio* User Manager.

The following table describes the available Spring properties that you can con-figure for the User Manager.

studio.usermanager.protectInternalDomain	
Type	Boolean
Default	false
Description	If set to `true`, the members of the internal domain can't be edited, created or deleted. In fact such internal members are hidden from the user manger. Only the rights management for external members is enabled then.

studio.usermanager.enableContentLiveGroups	
Type	Boolean
Default	false
Description	If set to `true`, the content server group and live server group properties of a group can be edited.

studio.usermanager.minSearchCharacters	
Type	Integer
Default	3
Description	Sets the maximum number of search characters to input before the external member search is triggered. The default value is `3`. If set to `0`, all members will be lazy loaded once the corresponding view in the user manager is opened. If your underlying user provider contains a great amount of users we recommend not to increase this value so that only concrete search requests are executed against the user provider. The flag is ignored for members of the internal domain.

studio.usermanager.protectedGroupNames	
Type	CSV
Default	administratoren
Description	Protected groups can't be renamed and their rights can't be changed. The `ad ministratoren` group always is protected. Additional values can be added using comma separated values. For external domains use the format `name@domain`.

studio.usermanager.managerUsers	
Type	String
Default	`null`
Description	This optional list of comma separated values allows configuring specific administrative users that are allowed to access the *CoreMedia Studio*'s user manager. For external domains use the format `name@domain`.

studio.usermanager.managerGroups	
Type	String
Default	`null`
Description	This optional list of comma separated values allows configuring specific groups of administrative users that are allowed to access the *CoreMedia Studio*'s user manager. The direct subgroups of the specified groups are allowed as well. For external domains use the format `name@domain`. If this list and the list of `studio.usermanager.managerUsers` are empty all administrative users are allowed.

studio.usermanager.offerDeleteHomeFolder	
Type	Boolean
Default	`true`

Description   If set to `false`, there is no option to delete user's home folder when deleting a user.

Table 9.25. User Manager Spring Properties

9.32 User Properties

The user management of the *CoreMedia Studio* comes with the additional properties `displayName` and `email` available for every user. In order to load the default values for these fields from an existing user provider, the corresponding Spring properties have to be configured.

If no user provider is connected to the Content Server, the fields `displayName` and `email` have to be inputted manually. These custom inputs will override the defaults that are read from the user provider.

The following table describes the available Spring properties that can be configured to map user provider user properties to a *Studio* user.

Spring Property Name	Maps to User Property	Description
`cap.user.properties.displayname`	displayName	The display name of the user. If set, this name will be used inside Studio instead of the regular login name.
cap.user.properties.email	email	The email address of the user.
`cap.user.properties.names`		A comma separated list of values that should be read from the UAPI user object and stored as user properties for the Studio Client API. *Note that these properties will be visible for all other users that have a Studio login*. This field can be used to show additional user information in Studio.

Table 9.26. User Provider Property Mapping

> **WARNING**
>
> The values for this field are stored in an `EditorProfile` content inside the users `Home` folder. Since this content contains personalized data, CoreMedia strongly recommends prohibiting the write access to these contents to all users that are not administrators. You can do this by setting only read access to the `Home` folder for the content type `EditorProfile` or by applying this rule using the User Manager or by adding and importing the following rule in your `users.xml` file:
>
> ```
> <rule content="/Home" type="EditorProfile" rights="R"/>
> ```

9.33 Adding Entity Controllers

This section shows how to implement custom REST controllers for Studio and invoke them using remote beans.

It covers the basic concepts behind `RemoteBeans`, `EntityControllers` and REST linking in Studio.

9.33.1 Prerequisites

The section assumes that you are familiar with Section 4.1.5, "Project Extensions" in *Blueprint Developer Manual* .

9.33.2 Implementing the Java Backend

Let's start with implementing a so called `EntityController` class. An instance of `EntityController` is created for every remote bean that is created in Studio via the following call:

```
beanFactory.getRemoteBean(...)
```

`EntityControllers` are used when you have multiple elements of the same type in Studio, `Content` instances in Studio, for example, are created through `EntityControllers`. The same applies for messages of the notification API or CMS users and user groups.

In this example, you will create entities that represent notes created by users. The user should be able to create, update and delete notes.

The note model would look like this:

```
public class Note {
  private String description;
  private String owner;
  private String noteId;

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  public String getOwner() {
    return owner;
  }
```

```
    public void setOwner(String owner) {
      this.owner = owner;
    }

    public String getNoteId() {
      return noteId;
    }

    public void setNoteId(String noteId) {
      this.noteId = noteId;
    }
}
```

Example 9.122. Note model

You also need a representation class for this model (the reason for this will be explained later).

```
public class NoteRepresentation {
  private String description;
  private String owner;
  private String noteId;

  NoteRepresentation(Note note) {
    this.description = note.getDescription();
    this.owner = note.getOwner();
    this.noteId = note.getNoteId();
  }

  public String getDescription() {
    return description;
  }

  public String getOwner() {
    return owner;
  }

  public String getNoteId() {
    return noteId;
  }
}
```

Example 9.123. Representation class for note model

You also have a service which handles the notes:

```
@DefaultAnnotation(NonNull.class)
public class NotesService {

  private final List<Note> list;

  public NotesService() {
    list = new ArrayList<>();
    Note dummy1 = new Note();
    dummy1.setOwner("me");
    dummy1.setNoteId("1");
    dummy1.setDescription("I have to write a real storage for this!");
    Note dummy2 = new Note();
    dummy2.setOwner("me");
    dummy2.setNoteId("2");
    dummy2.setDescription("And a lot of other stuff too!");
    list.add(dummy1);
    list.add(dummy2);
  }
```

```
  @Nullable
  public Note getNote(String id) {
    return list.stream().filter(note ->
id.equals(note.getNoteId())).findFirst().orElse(null);
  }

  public boolean deleteNote(String id) {
    Note noteToDelete = getNote(id);
    if (noteToDelete == null) {
      return false;
    }
    list.retainAll(
            list.stream().filter(note -> note !=
noteToDelete).collect(Collectors.toList())
    );
    return true;
  }

  @Nullable
  public Note updateNote(String id, String description) {
    Note note = getNote(id);
    if (note == null) {
      return null;
    }
    note.setDescription(description);
    return note;
  }

  public List<Note> getNotes() {
    return Collections.unmodifiableList(list);
  }

  public void setNotes(List<Note> notes) {
    list.clear();
    list.addAll(notes);
  }
}
```

Example 9.124. Service for note handling

So you have a note with a description, an owner and an ID and a service you can query for notes. You now have to create the `EntityController` class that wraps the REST operations around it:

```
@RestController
@RequestMapping(value = "notes/note/{" + NoteEntityController.PATH_PARAM_ID
 + "}", produces = MediaType.APPLICATION_JSON_VALUE)
public class NoteEntityController implements EntityController<Note> {
  public static final String PATH_PARAM_ID = "id";

  private final NotesService notesService;

  public NoteEntityController(NotesService notesService) {
    this.notesService = notesService;
  }

  @Override
  public Note getEntity(@NonNull Map<String, String> params) {
    return notesService.getNote(params.get(PATH_PARAM_ID));
  }

  @NonNull
  @Override
  public Map<String, String> getPathVariables(@NonNull Note entity) {
    HashMap<String, String> map = new HashMap<>();
    map.put(PATH_PARAM_ID, entity.getNoteId());
```

```
      return map;
  }

  @GetMapping
  public ResponseEntity<NoteRepresentation> getRepresentation(@PathVariable
Map<String, String> params) {
      Note note = getEntity(params);
      if (note == null) {
        return ResponseEntity.notFound().build();
      }
    return ResponseEntity.ok().contentType(MediaType.APPLICATION_JSON).body(new
NoteRepresentation(note));
  }

  @DeleteMapping
  public boolean delete(@PathVariable Map<String, String> params) {
      Note note = getEntity(params);
      if (note == null) {
        return false;
      }
      return notesService.deleteNote(note.getNoteId());
  }

  @PutMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
  public ResponseEntity<NoteRepresentation> setProperties(@PathVariable final
Map<String, String> params,
                                                          @RequestBody final
Map<String, Object> json) {
      String description = (String) json.get("description");
      Note note = getEntity(params);
      if (note == null) {
        return ResponseEntity.notFound().build();
      }
    Note updatedNote = notesService.updateNote(note.getNoteId(), description);

      if (updatedNote == null) {
        return ResponseEntity.badRequest().build();
      }
    return ResponseEntity.ok().contentType(MediaType.APPLICATION_JSON).body(new
NoteRepresentation(updatedNote));
  }
}
```

Example 9.125. Entity Controller class for TEST operations

Have a look on the class `NoteEntityController` in detail:

```
@RestController
@RequestMapping(value = "notes/note/{" + NoteEntityController.PATH_PARAM_ID
 + "}", produces = MediaType.APPLICATION_JSON_VALUE)
```

Example 9.126. Annotation for bean creation

The first two annotations are used to tell Spring what kind of bean you are cre-
ating. The first annotation states that the class represents a REST controller. The
`@RequestMapping` annotation provides the REST configuration. The `pro
duces` value defines that all requests expecting the JSON format will be accep-
ted and the `value` property tells Spring under which URL the entity can be in-
voked. Note that the URL can have multiple path parameters. This example shows
the most simple form with only one `Id` parameter.

The class has one REST GET method:

```
@GetMapping
public ResponseEntity<NoteRepresentation> getRepresentation(@PathVariable
Map<String, String> params) {
  Note note = getEntity(params);
  if (note == null) {
    return ResponseEntity.notFound().build();
  }
  return ResponseEntity.ok().contentType(MediaType.APPLICATION_JSON).body(new
 NoteRepresentation(note));
}
```

Example 9.127. REST GET method of NoteEntityController

So when a GET is executed on this controller, the note NoteRepresentation is returned and serialized to JSON. If the return format should differ from the originating model, you can freely customize the representation class. Because of the automatic REST linking, it is important that you don't return the same class here that has been defined as type of the EntityController! You can put models of other EntityControllers inside your representation as well. These entities will be converted to references during serialization. By this, different EntityControllers can be linked to each other. So you always have to create a representation class for the model that is bound for the EntityController. You just have to make sure that this representation contains the fields that should be supported by the RemoteBean you will implement.

> **NOTE**
>
> Note that in this example, it is not covered how and where these notes are stored. The methods in NotesService have to be implemented properly to support a real data access layer.

Next, add support for deletion by adding the following method:

```
@DeleteMapping
public boolean delete(@PathVariable Map<String, String> params) {
  Note note = getEntity(params);
  if (note == null) {
    return false;
  }
  return notesService.deleteNote(note.getNoteId());
}
```

Example 9.128. Deletion of note in NoteEntityController

The method is pretty simple: if a DELETE request is executed in the controller, the corresponding helper is invoked and the note is deleted.

The same applies for updates:

```
@PutMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<NoteRepresentation> setProperties(@PathVariable final
 Map<String, String> params,
                                                        @RequestBody final
Map<String, Object> json) {
  String description = (String) json.get("description");
  Note note = getEntity(params);
  if (note == null) {
    return ResponseEntity.notFound().build();
  }
  Note updatedNote = notesService.updateNote(note.getNoteId(), description);

  if (updatedNote == null) {
    return ResponseEntity.badRequest().build();
  }
  return ResponseEntity.ok().contentType(MediaType.APPLICATION_JSON).body(new
 NoteRepresentation(updatedNote));
}
```

Example 9.129. Update of note in NoteEntityController

You have finished the Java part now. Finally, you have to declare the entity as bean in the Spring configuration:

```
@Bean
public NotesService notesService() {
  return new NotesService();
}

@Bean
public NoteEntityController noteEntityController(NotesService notesService)
 {
  return new NoteEntityController(notesService);
}
```

Example 9.130. Declare NoteEntityController as bean

You can rebuild the module and restart Studio now. The next steps can be implemented using the incremental Studio build that doesn't require a Studio restart.

9.33.3 Implementing Studio Remote Beans

You can now create custom remote beans which are linked to the corresponding `EntityControllers`.

Every remote bean consist of an interface and an implementing class. For the note model, the files `Note.ts` and `NoteImpl.ts` would look like:

```
import RemoteBean from "@coremedia/studio-client.client-core/data/RemoteBean";

abstract class Note extends RemoteBean {

  abstract getDescription():string;
```

```
  abstract getUser():string;

  abstract getNoteId():string;
}

export default Note;
```

Example 9.131. Abstract class of Note remote bean

with the implementing class

```
import { mixin } from "@jangaroo/runtime";
import RemoteBeanImpl from
"@coremedia/studio-client.client-core/data/impl/RemoteBeanImpl";
import Note from "./Note";

class NoteImpl extends RemoteBeanImpl implements Note {
  static readonly REST_RESOURCE_URI_TEMPLATE: string = "notes/note/{id:[^/]+}";

  constructor(uri:string) {
    super(uri);
  }

  getDescription():string {
    return this.get("description");
  }

  getUser():string {
    return this.get("user");
  }

  getNoteId():string {
    return this.get("noteId");
  }
}
mixin(NoteImpl, Note);

export default NoteImpl;
```

Example 9.132. Implementing class of Note remote bean

When implementing remote beans, you have to make sure that the URI path of the remote bean described in the constant REST_RESOURCE_URI_TEMPLATE.

```
[static readonly REST_RESOURCE_URI_TEMPLATE: string = "notes/note/{id:[^/]+}";]
```

Example 9.133. Remote Bean URI path

matches the REST URL of the Java controller entity class.

In the last step, Studio has to register this class as a RemoteBean. Studio comes with a plugin for that, so simply add the following line in the init section of your Studio plugin or the init.ts file or your plugin module:

```
beanFactory._.registerRemoteBeanClasses(NoteImpl)
```

Example 9.134. Register class as remote bean

You can now use your custom remote bean within components to render a note's description.

9.33.4 Using the EntityController

Before using the newly created remote Bean inside a component, let's see if the REST request is actually working. You can test this by logging into Studio, open a new tab and invoking the following URL: **http://local-host:43080/rest/api/notes/note/1** (the path may differ depending on your setup)

The result should look like this:

```
{
  "description": "I have to find a real storage for this!",
  "owner": "me",
  "noteId": "1"
}
```

Example 9.135. Result of Note

The URL segment api/ is configured for all Studio REST controllers and ensures that all REST request are located under one unique segment.

So your `EntityController` is working and you have declared a `Remote Bean` for it. Now, invoke it from TypeScript.

You can simply use the base class of your Studio plugin rules (if available) or any other component base class that is created just to quickly test your code.

```
const note = as(beanFactory.getRemoteBean('notes/note/1'), Note);
note.load((loadedNote):void => {
  console.log(`My note says: ${loadedNote.getDescription()}`);
});
```

Example 9.136. Invoke class from TypeScript

Note that the invocation of the remote bean is done without the api segment. Remote beans have to be loaded manually or via `ValueExpressions`. Compile your workspace with this code and reload Studio. You should see the following message on your browser console:

```
My note says: I have to write a real storage for this!
```

Example 9.137. Output from remote bean

Next, use the remote bean inside a component:

```
Config(DisplayField, {
            plugins: [
              Config(BindPropertyPlugin, {
                componentProperty: "value",
                bindTo: ValueExpressionFactory.create('description',
                  beanFactory.getRemoteBean('notes/note/1')),
              }),
            ],
          }),
```

Example 9.138. Remote bean used inside a component

This example creates a label which contains the description of your note. Usually `RemoteBeans` are always accessed through a `ValueExpression`. The `ValueExpression` is then responsible for loading the value out of the `Re moteBean`.

9.33.5 REST Linking (Java Backend)

The note example has shown how to create a custom remote bean. However, in the real world you usually have to deal with a list of remote beans, so let's improve the example by adding another `EntityController` that is responsible for loading a list of notes.

First of all, you have to create the required Java classes for this. The model could look like this:

```
public class NoteList {
  private List<Note> notes = new ArrayList<>();

  public List<Note> getNotes() {
    return notes;
  }

  public void setNotes(List<Note> notes) {
    this.notes = notes;
  }
}
```

Example 9.139. Java class for notes list

Next, the matching representation which looks the same again:

```
public class NotesRepresentation {
  private List<Note> notes;
```

```
  public NotesRepresentation(NoteList noteList) {
    this.notes = noteList.getNotes();
  }

  public List<Note> getNotes() {
    return notes;
  }
}
```

Example 9.140. Notes list representation

So you can create the `EntityController` from it:

```
@RestController
@RequestMapping(value = "notes", produces = MediaType.APPLICATION_JSON_VALUE)
public class NotesEntityController implements EntityController<NoteList> {

  private final NotesService notesService;

  public NotesEntityController(NotesService notesService) {
    this.notesService = notesService;
  }

  @Override
  @NonNull
  public NoteList getEntity(@NonNull Map<String, String> pathVariables) {
    NoteList noteList = new NoteList();
    noteList.setNotes(notesService.getNotes());
    return noteList;
  }

  @GetMapping
  public NotesRepresentation getRepresentation(@NonNull Map<String, String>
 pathVariables) {
    NoteList entity = getEntity(pathVariables);
    return new NotesRepresentation(entity);
  }

  @PutMapping
  public ResponseEntity<NotesRepresentation> setRepresentation(@NonNull
Map<String, String> pathVariables,
                                                               @RequestBody
 final Map<String, Object> json) {
    //noinspection unchecked
    notesService.setNotes((List<Note>) json.get("notes"));
    NoteList entity = getEntity(pathVariables);
    return ResponseEntity.ok().contentType(MediaType.APPLICATION_JSON).body(new
 NotesRepresentation(entity));
  }
}
```

Example 9.141. NotesEntityController for notes list

The example returns a list of all notes of the `NotesService`. In addition to
this, you can change the list. Have a look at the code of the put mapping:

```
@PutMapping
public ResponseEntity<NotesRepresentation> setRepresentation(@NonNull
Map<String, String> pathVariables,
                                                             @RequestBody
final Map<String, Object> json) {
  //noinspection unchecked
  notesService.setNotes((List<Note>) json.get("notes"));
```

```
   NoteList entity = getEntity(pathVariables);
   return ResponseEntity.ok().contentType(MediaType.APPLICATION_JSON).body(new
 NotesRepresentation(entity));
}
```

Example 9.142. Put mapping for notes list

The changed properties of the NotesList will be passed via the `RequestBody`. You can expect that its structure is the same as in the NotesRepresentation, so the property `notes` will contain the changed list of notes.

As a last step, you have to add the Spring bean to your Spring configuration:

```
@Bean
public NotesEntityController notesEntityController(NotesService notesService)
 {
  return new NotesEntityController(notesService);
}
```

Example 9.143. Adding a Spring bean to Spring configuration

Again, the Java part is finished and you can rebuild the extension and restart Studio.

9.33.6 REST Linking (Studio RemoteBeans)

Since you have created another `EntityController`, you have to declare the matching remote beans the same way you already did for the Note remote bean. That means, you have to declare the interface

```
import RemoteBean from "@coremedia/studio-client.client-core/data/RemoteBean";
import Note from "./Note";

abstract class Notes extends RemoteBean {

  abstract getNotes():Note[];
}

export default Notes;
```

Example 9.144. Interface for remote bean for notes list

and the implementing class

```
import { mixin } from "@jangaroo/runtime";
import RemoteBeanImpl from
"@coremedia/studio-client.client-core/data/impl/RemoteBeanImpl";
import Note from "./Note";
import Notes from "./Notes";
```

```
class NotesImpl extends RemoteBeanImpl implements Notes {
  static readonly REST_RESOURCE_URI_TEMPLATE:string = "notes";

  constructor(uri:string) {
    super(uri);
  }

  getNotes():Note[] {
    return this.get("notes");
  }
}
mixin(NotesImpl, Notes);

export default NotesImpl;
```

Example 9.145. Implementing class for remote bean for notes list

and finally tell Studio that a new remote bean type is there:

```
beanFactory._.registerRemoteBeanClasses(NoteImpl, NotesImpl)
```

Example 9.146. Register remote bean with Studio

Rebuild and reload Studio. Once you are logged in, test the new REST controller manually by invoking the following URL in another browser tab: `http://loc alhost:43080/rest/api/notes/` (the path may differ depending on your setup). As a result, you should see the following:

```
{
  notes: [
    {
      $Ref: "notes/note/1"
    },
    {
      $Ref: "notes/note/2"
    }
  ]
}
```

Example 9.147. Test result of remote bean

Note that not the plain JSON of the entities is serialized, but the references to them instead. For every class that is part of a representation a lookup is made if there is a corresponding `EntityController` declared for it. If true, the link to this controller is serialized instead of the linked entity.

> **WARNING**
>
> The serialization and deserialization of entities consumed or produced by the `EntityController` is never handled by the controller itself. Please do not make any assumptions on how serialization and deserialization is implemented in your code, as this is not part of the Public API.

Invoke this inside TypeScript:

```
const notes = as(beanFactory.getRemoteBean("notes"), Notes);
notes.load((loadedNotes):void => {
  console.log('I have ${loadedNotes.getNotes().length} notes`);
});
```

Example 9.148. Invoke notes in TypeScript

The code looks similar to the previous example. The matching remote bean is created and loaded and the status of the bean is logged to the console. Note that only the Notes bean has been loaded through this code. The child elements must be loaded separately, so to display everything you can do something like this:

```
const notes = as(beanFactory.getRemoteBean('notes'), Notes);
notes.load((loadedNotes:Notes) => {
  console.log(`I have ${loadedNotes.getNotes().length} notes`);

  loadedNotes.getNotes()[0].load(note1 => console.log(note1.getDescription()));

  loadedNotes.getNotes()[1].load(note2 => console.log(note2.getDescription()));
});
```

Example 9.149. Display child elements of notes list

The output will look like this:

```
I have 2 notes
I have to write a real storage for this!
And a lot of other stuff too!
```

Example 9.150. Output of notes list

You can also change this list. Reverse, for example, the order of the notes in the list:

```
const notes = as(beanFactory.getRemoteBean("notes"), Notes);
notes.set("notes", notes.getNotes().slice().reverse());
```

Example 9.151. Reverse order of notes list

> **WARNING**
>
> Please mind that `slice()` is called before the array is reversed. You should not directly change the result of `getNotes()` as this has unintended side effects.

Now, inspect the request header of the resulting PUT request:

```
{
  "notes": [
    {
      "$Ref": "notes/note/2"
    },
    {
      "$Ref":"notes/note/1"
    }
  ]
}
```

Example 9.152. Request header of PUT request

As you can see the entities are once again serialized by only using the references to the single notes handled by the `NoteEntityController`. When receiving the new list in the `NotesEntityController`'s `@PutMapping` the references are already resolved and you do not need to take care of that.

9.34 Multiple Previews Configuration

Starting with version 2007.1, CoreMedia supports multiple previews, using a content based configuration. The content based configuration style takes the requirements of a cloud style deployment into account, where a configuration file or environment variable based configuration requires a new deployment and/or a server restart. The content based configuration enables the Studio user to add or remove a preview at any time. While this approach is very convenient, there is one drawback. Using a replicated content repository on other installations, for example, in a stage/live deployment scenario, may become difficult, as some of the configuration values may not fit in a different scenario.

Multiple previews can be enabled and configured using one or more CMSettings content items in well known folders. If none of these content items exist or all contained previews are disabled, the standard single preview is used as the default preview, thus maintaining downward compatability.

The default location of a CMSettings content item to configure one or more global previews is this repository path:

```
All Content/Settings/Options/Settings/Multi Preview
```

Additionally, it is also possible to restrict one or more previews to a single site. In this case, another CMSettings content item is expected below a sites folder at the relative path

```
Options/Settings/Multi Preview
```

The names of the CMSettings content items are freely choosable. Among others, the aforementioned, well known paths and the content type for the settings are configurable via application.properties at deployment level. For a complete list of all deployment level configuration options for the multi preview, please refer to the deployment manual.

9.34.1 Configuration of a preview

Studio supports two types of preview services.

• Preview URL Provider: The preview provider delivers a ready to use preview URL, which will be displayed directly in the preview frame.

- Preview URL Service Provider: The preview provider delivers the URL to a (potentially external) preview URL service, which in turn delivers the real preview URL.

To configure one or more preview, the structure of the CMSettings content items requires a structure similar to this:

Figure 9.25. Settings Document with two configured previews

previews	An array of structs, where each entry defines exactly one preview.
id	The ID of a preview (mandatory). The ID of a preview must be unique for all globally and site locally configured previews! The ID is also used as a localization key, if the displayName is missing.
providerId	The ID of an existing preview provider (mandatory). The provider ID is the bean name of a server side preview provider implementation, provided by means of Spring Boot. CoreMedia comes with several preview providers 'out of

	the box', covering already many requirements of a preview. For details about these providers, please refer to the sections below.
displayName	The name for the preview, to be displayed in the preview selection menu. Though the name is neither mandatory nor must be unique, he should be choosen carefully. To give an example, previews, restricted to a site, may use an abreviated site name, while global previews may use a more common preview name.
enabled	Boolean flag to en- or disable a preview. Defaults to false, if missing!
userGroupAllowList	An array of strings, containing usergroup names who are permitted to use this preview. If empty or missing, all Studio users are eligable to use this preview.
previewUrlAllowList	An array of strings, containing endorsed URLs for the preview additionally to those, delivered automatically by the preview providers. This is list is merged with all other endorsed URLs of all configured previews, preview providers and of application properties, in order to control the URLs in the preview frame and prevent CSRF.
connectSrcAllowList	An array of strings, containing endorsed connect sources for Studio additionally to those, delivered automatically by the preview providers. This list is merged with all other endorsed connect sources of all configured previews, preview providers and of application properties.
urlTransformationsDisabled	Boolean flag to en- or disable the transformation of the preview URL by the Studio client (in most cases the addition of further query parameters for preview date, selected persona etc.). Note that these transformations are always enabled for a preview service URL (see section Section 9.34.7, "Generic Preview URL Service Provider" [378] below). This flag decides on the enablement of transformations for the final preview URL.

config	A struct containing preview provider specific configuration values.

9.34.2 CAE Preview Provider

The CAE preview provider is meant to replace the standard single preview. By default, it uses the same deployment level configuration from application.properties. In order to enable the CAE preview provider, please create a CMSettings content item, add a struct array named 'previews' and add a struct with these keys:

id	Freely choosable, unique preview ID, 'caePreview', for example
providerId	caePreviewProvider
displayName	Non localized display name for the preview selection, 'CAE Preview', for example
enabled	true

9.34.2.1 Provider specific config keys

previewHost	By default, the CAE preview provider uses the deployment level configuration value `studio.previewUrlPrefix` of the application properties file. Using `previewHost`, that value can be overwritten by the settings content item.

9.34.3 Headless Preview Provider

The Headless preview provider offers a preview on the JSON encoded content, delivered by the headless server. In order to enable the Headless preview provider, add a struct to the 'previews' array with these keys:

id	Freely choosable, unique preview ID, 'headlessPreview', for example
providerId	headlessPreviewProvider
displayName	Non localized display name for the preview selection, 'JSON Preview', for example
enabled	true

9.34.3.1 Provider specific config keys

previewHost   Headless preview provider uses the deployment level configuration value `studio.multipreview.headlessPreviewHost`. Using `previewHost`, that value can be overwritten by the settings content item.

9.34.4 Commerce Headless Preview Provider

The Commerce Headless Preview Provider offers a preview on the JSON encoded commerce objects and augmenting content, delivered by the headless server. In order to enable the Commerce Headless Preview Provider, add a struct to the 'previews' array with these keys:

id   Freely choosable, a preview ID, 'commerceHeadlessPreview', for example

providerId   commerceHeadlessPreviewProvider

displayName   Non localized display name for the preview selection, 'JSON Preview', for example

enabled   true

9.34.4.1 Provider specific config keys

previewHost   Headless preview provider uses the deployment level configuration value 'studio.multipreview.headlessPreviewHost'. Using 'previewHost', that value can be overwritten by the settings content item.

9.34.5 Studio URI-Template Preview Provider

The Studio URI template preview provider offers the possibility to define a URI template, which points to any desired preview endpoint. The template uses predefined template variables (as described below) to calculate most any desired preview URL, for example, a URL to a restful preview endpoint for a progressive

web application (pwa) or a single page application. In order to enable the URI template preview provider, add a struct to the 'previews' array with these keys:

id	Freely choosable, unique preview ID, 'myPWAPreview', for example
providerId	genericStudioPreviewProvider
displayName	Non localized display name for the preview selection, 'PWA Preview', for example
enabled	true

9.34.5.1 Provider specific config keys

uriTemplate	The URI template (Spring Boot style) to calculate/evaluate the preview URL.
	Example: https://my-pwa-host.de/preview/{numericContentId}/{contentType}/{previewDateRFC1123}

These template variables are available to the URI template:

|	
contentId	Contains the schemed content ID, like 'coremedia:///cap/content/550'
numericContentId	Contains only the numeric part of the content ID.
contentType	The Type of the content object, CMChannel, for example.
fqdn	The value of the environment variable ENVIRONMENT_FQDN (fully qualified domain name).
previewDate	The preview date as used by studio client, formatted as: 'dd-MM-yyyy HH:mm VV'
previewDateRFC1123	The preview date, formatted accordingly to RFC 1123, which is commonly used for date HTTP headers: 'EEE, dd MMM yyyy HH:mm:ss zzz'
rootSegment	The homepage root segment of the preview content object.
siteId	The site id of the preview content object.
view	The type of the request preview, fragmentPreview, for example.
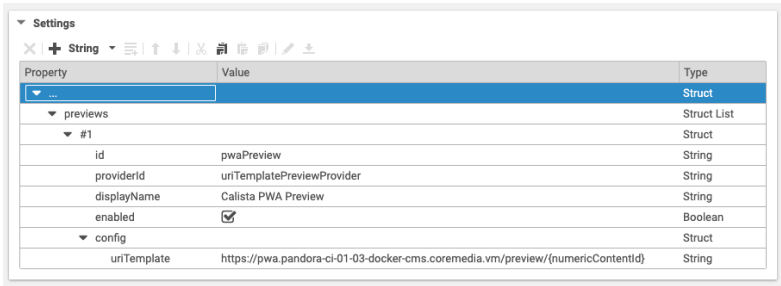
Figure 9.26. Example configuration of the Generic URI–Template Preview Provider

9.34.6 Common URI–Template Preview Provider

The common URI–Template Preview Provider is very similar to the previous pre-view provider. In contrast to the Studio URI–Template Preview Provider, this more common alternative is not bound to a Studio controller endpoint, which is used to prettify the querystring of the runtime parameters of Studio. These are the supported configuration keys:

id Freely choosable, unique preview ID, 'myPWAPreview', for ex-ample

providerId uriTemplatePreviewProvider

displayName Non localized display name for the preview selection, 'PWA Preview', for example

enabled true

9.34.7 Generic Preview URL Service Provider

In contrast to the generic URI template preview provider, the generic preview URL service provider does not provide the URL to the previewable content. In-stead, the URL to an external preview URL service is provided, which is responsible to deliver the effective preview URL for the content.

The URL to the preview service is configured very similar to the generic URI template preview provider. The provided URL of the preview URL service is ex-tended by client side runtime query parameters for example like this:

```
https://my-preview-url-service.de/previewurl/550?contentType=CMChannel&previewDate=XXXX&

view=XXX&userVariant=&userVariantTS=1592902149681&p13n_test=true&contentTimestamp=48634
```

In this example, the full querystring was appended by the Studio client to the URL of the basic preview URL service ('https://my-preview-url-service.de/pre-viewURL/550').

The appended querystring is added at runtime and cannot be altered. An external preview URL service has to implement this 'contract'. This means, if the service has to support, for example, the preview date, it has to use the predefined date format and the query parameter name 'previewDate'.

MULTIPLE PREVIEWS IN STUDIO
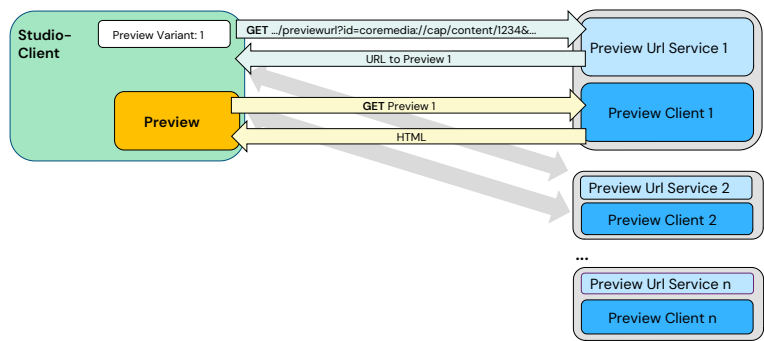
Figure 9.27. Studio with multiple Previews

In order to enable the generic preview URL service provider, add a struct to the 'previews' array with these keys:

id   Freely choosable, unique preview ID, 'myPWAPreviewUrlService', for example

providerId   previewUrlServicePreviewProvider

displayName   Non localized display name for the preview selection, 'PWA Preview', for example

enabled   true

9.34.7.1 Provider specific config keys

uriTemplate   The URI template (Spring Boot style), to calculate/evaluate the URL to the preview URL service.

Example: https://my-pwa-host.de/previewurl/{numericContentId}

These template variables are available to the URI template:

contentId	Contains the schemed content ID, like 'coremedia:///cap/content/550'
numericContentId	Contains only the numeric part of the content ID.
contentType	The type of the content object, `CMChannel`, for example.
fqdn	The value of the environment variable ENVIRONMENT_FQDN (fully qualified domain name).
rootSegment	The homepage root segment of the preview content object.
siteId	The site ID of the preview content object.

These request query parameters are appended automatically by Studio Client.

previewDate	The preview date as used by Studio Client, formatted as: 'dd-MM-yyyy HH:mm VV'
view	The type of the request preview, fragmentPreview, for example.

9.34.8 Public API of the Preview URL Service

While the Studio server already uses the preview URL service, it is possible to extend the service by providing additional preview providers or by integrating the preview URL service in a different environment than the Studio server, for example, as a microservice to an external preview.

9.34.8.1 Developing a custom PreviewProvider

Whenever the delivered preview providers don't meet the requirements for a special preview, it is possible to implement your own preview provider. Imple-

mentations may use the base implementation `AbstractContentPreview Provider`, which already implements some more common aspects, like checking the preview settings for restrictions to a site or certain content types. (see the Javadocs for details).

```
public class CustomPreviewProvider extends AbstractContentPreviewProvider
{
  public CustomPreviewProvider(SitesService sitesService) {
    super(sitesService);
  }
  ...
}
```

A very basic example of an implementation has to implement these additional methods:

```
@Override
public Optional<Preview> getPreviewUrl(
    Content entity,
    PreviewSettings previewSettings,
    Map<String, Object> parameters
) {
  return Optional.of(
    Preview.of(
      previewSettings,
      "https://mypreviewservice.com/path/to/service/" + entity.getId(),
      isPreviewUrlService())
    );
}
@Override
public boolean isPreviewUrlService() {
  return false;
}
@Override
public boolean validate(PreviewSettings previewSettings) {
  return true;
}
```

The example above 'calculates' direct preview URLs, so `isPreviewUrlSer vice()` has to return 'false'.

The calculation in this example is very static, eliminating the need to validate any configuration of the preview settings. More sophisticated implementations may validate the values of the given 'previewSettings'. The validate method is invoked by the preview URL service, whenever the configuration of a preview is changed, added or removed. If the validation fails (returning `false`), the preview will not become available through the preview URL service.

9.34.8.2 Adding a custom Preview Provider to the PreviewUrlService

Any preview provider must be created as a Spring bean. The most convenient way is, to create an additional Spring Boot configuration class and provide a factory method for all additional providers. The preview URL service 'sees' all beans of the type `PreviewProvider` and registers them. To use them, you

have to use the providers bean name (in this example 'myCustomPreviewPro-vider') in the corresponding configuration content item.

```
@Configuration(proxyBeanMethods = false)
public class CustomPreviewUrlServiceConfig {
  @Bean
  public PreviewProvider<Content> myCustomPreviewProvider(
      SitesService sitesService
  ) {
    return new CustomPreviewProvider(sitesService);
  }
}
```

9.34.8.3 Obtaining the PreviewUrlService in Studio Server

The preview URL service in the Studio server can be obtained simply by referencing it by its interface. Let's say, you want to implement a new REST service and want to use the preview URL service. The basic approach using plain Spring Boot would be:

```
@RestController
public class CustomRestController {
  private final PreviewUrlService previewUrlService;
  public CustomRestController(PreviewUrlService previewUrlService) {
    this.previewUrlService = previewUrlService;
  }
  ...
}
```

9.34.8.4 Obtaining the PreviewUrlService independently from Studio Server

By adding the following Maven dependency to your extension, the preview URL service will be automatically instantiated as a Spring bean by the means of Spring Boot. The bean is visible under the name 'contentPreviewUrlService'.

```
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>preview-url-service</artifactId>
</dependency>
```

To use the preview URL service, the service needs to be configured by offering one or more preview providers. The providers must be created as described above, using a Spring Boot configuration class.

Please read the Javadocs for detailed information about the `PreviewUrlService` and `PreviewProvider`.

9.35 Quick Search Configuration

Starting with version 2207.2, CoreMedia comes with a quick search dialog, which supports the parallel search in different parts of the product, such as content and commerce entities.

The quick search dialog can be customized using the `ConfigureQuickSearchPlugin`. The given example shows the standard configuration that is used in the `BlueprintFormsStudioPlugin.ts`:

```
    new ConfigureQuickSearchPlugin({
keepResults: true,
imageDocTypes: [
  "CMPicture"
],
actions: [
  ...
],
types: [
  new AppQuickSearchType({
    title: QuickSearch_properties.QuickSearch_search_type_apps,
  }),
  new ContentQuickSearchType({
    id: "allContent",
    title: QuickSearch_properties.QuickSearch_search_type_content,
    limit: 5,
  }),
]
})
```

Example 9.153. Quick Search Default Configuration

The given list explains the configuration in detail:

- `keepResults`: If set to `true`, the last result of the quick search is stored and shown again when the dialog is opened the next time. Otherwise the search is shown with cleared search results, everytime when the dialog is opened. If only one usage of the `ConfigureQuickSearchPlugin` sets this flag to `true`, it will be kept this way.

- `imageDocTypes`: The field configures the list of content types that should be used for the image preview. The list is only extendable and will not be overwritten by different usages of the `ConfigureQuickSearchPlugin`.

- `actions`: This field allows to configure custom actions that are included in the app search. More details about this are shown in the section below.

- `types`: Configures the list of `QuickSearchType`s. Every `QuickSearchType` is searched when the user inputs a search term. If a result is found, a search result section is rendered for the corresponding quick search type.

9.35.1 Quick Search Types

Classes extending the abstract class `QuickSearchType` implement the actual search that is executed once the user inputs a search term. Additional search types can be implemented and added with an entry to the `types` section of the `ConfigureQuickSearchPlugin`.

A `QuickSearchType` returns a list of `RemoteBean`s. Note that currently only the rendering of the following `RemoteBean`s are supported:

- `Content`
- `CatalogObject`
- `ContentHubObject`

9.35.2 Search for Custom Actions

In some situation, it comes in handy to search for a feature instead instead of items. For that reason, the quick search dialog support to register custom actions with keywords and make them searchable. The `BlueprintFormsStudioPlugin.ts` for example defines the following action:

```
      new ConfigureQuickSearchPlugin({
  ...
  actions: [
    ...
    new QuickSearchActionConfiguration({
      svgIcon: preferences,
      label: FrameComponents_properties.PreferenceWindow_title,
      additionalKeywords:
[FrameComponents_properties.PreferenceWindow_shortcuts_text,
        FrameComponents_properties.PreferenceWindow_language_text,
        FrameComponents_properties.PreferenceWindow_SiteSelector_title,
        FrameComponents_properties.PreferenceWindow_dialogs_title,
        FrameComponents_properties.PreferenceWindow_shortcuts_text,
      ],
      action: (): void => {
        new OpenDialogAction({
          dialogDefaults: Config(StudioPreferenceWindow, {}),
        }).execute();
      },
    }),
    ...
  ],
  ...
})
```

Example 9.154. Quick Search Default Configuration

Here, an `OpenDialogAction` is added to the quick search dialog which should open the preferences dialog. The configuration defines an icon and label that is shown if the search term matches one of the `additionalKeywords` or the label of the action. The `action` field configures the function that is called if the user (single) clicks on the search result.

9.36 Quick Create

This section describes different ways to customize the content creation through the "Quick Create" dialog.

9.36.1 Default Folders

CoreMedia Blueprint comes with different possibilities to customize the location where new content should be created with the "Quick Create" dialog and the component `FolderChooserListView`. This section describes the available target folder calculation and customization options of this component.

The target folder is determined in the following order:

1. FolderChooserListView Lookup Methods

2. Last used folder

3. Custom "folderPathsExpression" Expression

4. Content Creation Settings

5. EditorPreferences

6. Folder Bookmarks

FolderChooserListView Lookup Methods

The component for the folder selection `FolderChooserListView` supports lookup methods that you can register, so that the target folder can be calculated dynamically. Find an example in the class `BlueprintFormsStudioPlugin`. This default implementation uses the selected folder in the library (if opened) for creating a new content.

```
      Config(EditorMainView, {
  plugins: [
    Config(AddDefaultFolderChooserEntry, { lookup: (content: Content,
contentType:ContentType): any => {
       if(contentType.isSubtypeOf("CMArticle")) {
         return "/Sites/MySite/English/Articles/";
       }

       //not applicable? let's continue with other lookups methods and
variants
       return null;
     } }),
  ],
}),
```

Example 9.155. Adding a FolderChooserListView lookup method

Note that the method is not called in a "dependency-tracked" context, which means it can't return `undefined` in order to be evaluated again when content is loaded inside it. When `null` is returned, the target folder evaluation is continued with the next lookup method or with the next strategy described here.

Last Used Folders

When a user creates content through the "New Content" dialog, the location is stored inside a history in the `EditorPreferences` content of this user. This logic is part of the `FolderChooserListView` component and can not be customized.

Custom "folderPathsExpression" Expression

If you use the `FolderChooserListView` in a custom component or dialog, you can configure the parameter `folderPathsExpression` for it. This expression can contain a fixed path or can be a `FunctionValueExpression` which calculates the target folder. The return value of the expression can be a string array with different paths or `undefined`, if the calculation is not finished yet.

Content Creation Settings

The easiest way to customize the target location for new content is to configure the settings content `Content Creation`. You can create it inside the global settings folder `/Settings/Options/Settings/` or site specific settings folders `Options/Settings/`.

The settings content must contain a link list with name `paths` which contains a mapping from the content type to the desired target folders. You can find examples in the *CoreMedia Blueprint* default content.

EditorPreferences

The `EditorPreference` settings content is used as another fallback for looking up content creation mappings. These settings are used the same way as `Content Creation` settings, but the name of the `Struct` property is `contentCreationPaths` instead.

Folder Bookmarks

If none of the above strategies returns a folder, the user's bookmarks are checked for folder bookmarks and suggested for the content creation.

9.36.2 Quick Create Post-Processing

CoreMedia Studio provides two ways to post-process newly created content. You can use the post-processing to initialize the content with default values or to trigger other actions.

Content Initializer

Content initializers are explained in detail in Section 9.5.7, "Client-side initialization of new content items" [161]

Quick Create Success Handler

Quick create does support a "success" hook where methods are executed once the content has been created successfully. You can register these methods for the class `QuickCreate`:

```
QuickCreate#addSuccessHandler(contentType: string, onSuccess: AnyFunction):
 void
```

Example 9.156. Quick Create Success Handler Registration

9.37 Locale Switcher Configuration

The Locale Switcher is a combo box that is available on the toolbar of every content form. It allows to switch through the various localized variants of the current content. The Locale Switcher can be configured using the flags described in this section.

Visibility

The Locale Switcher can be configured to be visible or hidden by default. The following example shows the configuration flag that is used to customize the visibility.

```
DocumentFormToolbarPlugin.LOCALE_SWITCHER_DISABLED = true | false;
```

Example 9.157. Locale Switcher Visibility Configuration

Document Hierarchy Resolving

To calculate the correct translation/synchronization relation between two con‐tents, the correct way is resolving a contents' referrers and find the master content this way. Since this calculation can be expensive, the Locale Switcher simply resolves this relation using the site hierarchy instead. There are some setups though where especially the master content might be outside the site hierarchy, for example, when using a shared master content. For these setups, you can set the ASSUME_MASTER_IN_SITE_HIERARCHY flag to false to force the Locale Switcher to resolve the relation using the referrers.

```
ContentSiteUtil.ASSUME_STRICT_SITE_HIERARCHY = true | false;
```

Example 9.158. Locale Switcher Strict Hierarchy Flag

9.38 Developing Studio Apps

This section describes how to develop custom *Studio Apps* and integrate them with the *CoreMedia Studio Apps* environment, both in terms of workspace integration and application integration.

9.38.1 Overview

While in previous times, the *CoreMedia Studio* only consisted of exactly one app, the Studio has evolved into a multi-app environment by now. By default, the *Studio* consists of the *Content App* and the *Workflow App*. The *Campaign App* can be also be added for cloud customers. All of these apps are independent of each other in the sense that they run in separate browser windows and fulfil their separate functionality. The apps can also be implemented with different frontend technologies. As it stands, the *Content App* and the *Workflow App* are based on ExtJS while the *Campaign App* is based on ReactJs. However, all the apps are part of the surrounding Studio environment: They know of each other and can interact with each other. For example, when working with the *Control Room* in the *Content App*, you can open a workflow in the *Workflow App*. If the *Workflow App* is not yet opened, it will automatically be opened in a new browser window. Similarly, when working with the *Workflow App* you can open the content items of a workflow in the *Content App*.

Figure gives an overview of the Studio Apps environment displaying the above-mentioned apps.
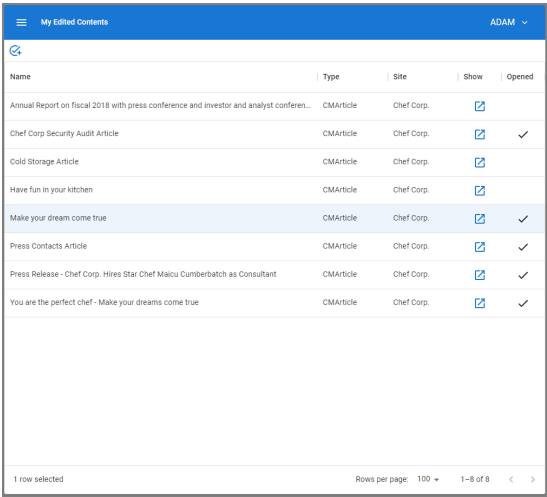
Figure 9.28. Different Studio Apps Connected Via Service Layer

To allow independent apps based on different frontend technologies that still can interact with each other, two loose-coupling mechanisms are utilized.

- *App Manifests:* Each app comes with an app manifest that holds information about the app. Each app loads the manifests of all other apps and can use the information. For more details, see Section Section 9.38.4, "App Manifest and Apps Menu Entries" [394] below.
- *Service layer:* Apps can offer services that other apps can use. For this purpose, the apps communicate via the browser's *BroadcastChannel* API. To avoid that the developer needs to concern oneself with the intricacies of message sending, parsing and so on, a higher level API is provided on top of that, the *Service Agent API*. Each app is equipped with its own service agent instance that communicates with the service agents of other apps. For more details, see Section Section 9.38.5, "App Services" [398] below.

In the remainder of the section, a demo *Studio App* will be used as a running example. It is an app that displays the edited contents of a user (similar to the *My Edited Contents* part of the *Control Room*) but with additional functionality. It is shown in the following Figure Figure 9.29, "The My-Edited-Contents Demo App" [392]. The app is written in ReactJs and the source code can be obtained from the *CoreMedia Help Center* under this link: My-Edited-Contents Demo App Source Code.

Figure 9.29. The My-Edited-Contents Demo App

9.38.2 Workspace Integration

Workspace integration of a custom app follows the same rules that hold for other extensions / plugins. Basically two ways are distinguished, namely as (1) part of the Blueprint and (2) as an application plugin.

Blueprint Integration

The Blueprint integration follows the integration of the built-in apps.

- A new custom app is simply placed under `WORKSPACE_ROOT/apps/studio-client/apps` just as the *Content App* and the *Workflow App* are.
- The app package is added to the `pnpm-workspace.yaml`.
- The app is added as a dependency to the package `@coremedia-blueprint/studio-client.studio` located in `WORKSPACE_ROOT/apps/studio-client/global/studio`.
- The app is added to the `appPaths` property of `WORKSPACE_ROOT/apps/studio-client/global/studio/jangaroo.config.js` alongside the other apps. App path, name and build directory are specified here.

```
appPaths: {
  "@coremedia-blueprint/studio-client.main.app": "",
  "@coremedia-blueprint/studio-client.workflow.app": "apps/workflow-app",

  "@coremedia-internal/edited-contents-app": {
    name: "studio-client.my-edited-contents",
    path: "apps/mec-app",
    buildDirectory: "dist"
  }
},
```

Instead of specifying the app path and name like this, they can also be put in the `jangaroo` entry of the app's `package.json`.

```
"jangaroo": {
  "appName": "studio-client.my-edited-contents",
  "appPath": "apps/mec-app"
}
```

Integration As An Application Plugin

In Section *Blueprint Workspace For Developers > Concepts And Architecture > Application Plugins > Plugins For Studio Client* of the *Blueprint Developer Manual*, it is described how Studio extensions can be added as so-called *application plugins*. These plugins do not need to be part of the Blueprint build but can be added dynamically later on.

A custom app as a whole can also be added as such an application plugin. In this case, the extension point to which the plugin is added is simply `studio-client` (instead of `studio-client.main` or `studio-client.work flow` for the *Content App* and *Workflow App*).

For the development phase of an app application plugin, the easiest way is to modify `WORKSPACE_ROOT/apps/studio-client/global/stu dio/jangaroo.config.js` as follows:

- Add the directory root path of your app to the `additionalPackagesDirs` property.
- Add the app to the `appPaths` property as described above.

9.38.3 Accessing the Studio Apps Context

The package `@coremedia/studio-client.app-context-models` provides the central utility to access the Studio Apps context. All public API types and methods are exported via the package's `index.ts` and can con-

sequently be imported from the package top-level. In the following sections, many parts of this package will be covered in more detail.

The most central type of the package is `StudioAppsContext` which provides a wide range of utility methods to interact with the Studio Apps environment. It can be accessed via the global constant `studioAppsContext`:

```
import { studioAppsContext } from
"@coremedia/studio-client.app-context-models";
```

It allows to access information about other apps and their offered services, to run or focus apps, to access the initial startup parameters of the current app and several additional utilities. The most important method is `studioAppsContext._.initAppServices()`. It initializes the whole apps context for the current app. It is recommended to call this method as early as possible in the app's lifecycle.

9.38.4 App Manifest and Apps Menu Entries

Section Section 9.2, "Customizing Entries to the Apps Menu" [123] covers how to add *App Menu* entries from the perspective of the *Content App* and the *Workflow App*. This section covers all relevant aspects of app manifests and how they are used for custom *Studio Apps*.

App manifests are the basic mechanism to loosely couple the different *Studio Apps.* All apps that are integrated into the workspace as described above in Section Section 9.38.2, "Workspace Integration" [392] know each other: Each app is equipped with a manifest that holds information about the app and each app loads the manifests of all other apps to use that information. This takes place during the above-mentioned app bootstrap method `studioAppsContext._.initAppServices()`.

One of the immediate consequences of having an app being equipped with a manifest is that the app automatically appears the the *Apps Menu* of the built-in *CoreMedia* apps as shown in the following figure for the *My-Edited-Contents App* entries.
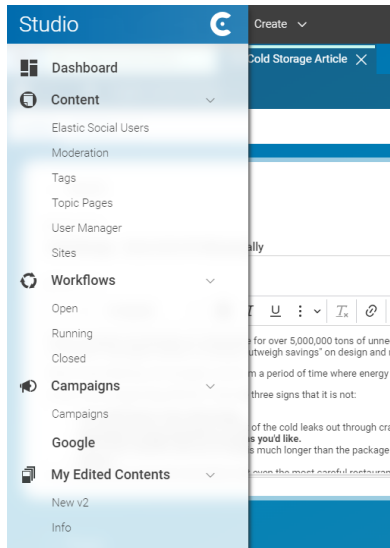
Figure 9.30. Apps Menu With My-Edited-Contents App

Currently, *CoreMedia* does not provide a reusable apps menu component for a custom app that is not implemented with ExtJs. However, the `studioAppsContext` utility already provides the model for it. Calling `studioAppsContext._.observeRunAppSections()` returns so-called `RunAppEntries / RunAppSections` that can be used to build a custom apps menu from scratch. Each entry / section has a `run()` method which automatically starts the corresponding app if it is not already running and focuses an app (brings it into the foreground) if it is already running. For ReactJs, the demo *My-Edited-Contents App* already contains an apps menu implementation that looks similar to the one from the built-in *CoreMedia* apps.

For app manifests, *CoreMedia* follows the Web standard Web App Manifests. The manifest of a custom app can include all the properties from the standard, but in addition, some *CoreMedia*-specific properties can be included. The latter are prefixed with `cm`.

Among the various properties supported by the standard, the following basic ones are mandatory for *CoreMedia Studio* apps:

```
"icons": [
  {
    "src": "icons/edited-contents-24.svg",
    "type": "image/svg",
    "sizes": "24x24"
  },
```

```
    ...
]
```

- *cmKey:* The unique app key.
- *name:* The app name.
- *short_name:* The app short name.
- *icons:* App icons in different sizes (24x24, 192x192, 512x512) where each icon is configured in terms of an object with properties *"src", "type" (the MIME type, e.g. "image/png" or "image/svg") and "sizes"*.

In addition to these basic manifest properties, there are some advanced properties that are of particular interest for *CoreMedia Studio* apps.

- *categories:* This is a string list of categories of the app. If none are given, the app's short name is considered as its sole category. The first category of this list is important for the result of `studioAppsContext._.observeRun AppSections()` for the *Apps Menu*: The first category of an app makes up its own `RunAppSection`. These are e.g. "Content", Workflows", "Campaigns" and "My Edited Contents" in Figure Figure 9.30, "Apps Menu With My-Edited-Contents App" [395].
- *shortcuts:* For an app, so-called shortcuts can be specified. These are sub-paths/URLs into the app. In the *Apps Menu*, these shortcuts are by default displayed as sub-entries (of type `RunAppEntry`) of the app's `RunAppSec tion`. For example, the "New v2" shortcut of the "My Edited Contents" section is defined as follows:

```
"shortcuts": [
  {
    "cmKey": "mec-v2",
    "name": "New v2",
    "url": "under-construction"
  }
],
```

For this to work, the app needs to offer a `RouterService` (for more details, see Section 9.38.5.2, "Built-In Services And Utilities" [405]). This service is called via `RouterService#setPath()` once the shortcut entry is clicked. The service has to take care of setting the specified URL of the shortcut. For example, in the *My-Edited-Contents App*, a *ReactJs* `HashRouter` is in place and the service's implementation uses the `useNavigate()` hook to set the url.

It is also possible for a shortcut to appear outside its app's `RunAppSection`. For a shortcut, a custom `cmCategory` can be set. If this category matches any app's `RunAppSection`, the shortcut is placed there as a sub-`RunAp pEntry`. If no such section exists, the shortcut makes up its own top-level

section. For example, the "Dashboard" section in Figure Figure Figure 9.30, "Apps Menu With My-Edited-Contents App" [395] is actually a shortcut of the *Content App*. But the app's category is "Content" while the shortcut's `cmCat egory` is "Dashboard".

- *cmServiceShortcuts:* While the ordinary shortcuts from above specify a sub-path/URL into the app, so-called *service shortcuts* are shortcuts where an app's service is called when the shortcut is run / clicked. For more details about services, see Section Section 9.38.5, "App Services" [398] below. For now, it is sufficient to say that a service shortcut specification complies to the `ServiceShortcut` type where the service to call is given via the property `cmService` of type `ServiceMethodDescriptorWithProps`. But it is also possible to leave out the property. In that case, an under-the-hood default service is used, and it is sufficient to register a *ShortcutRunner* for the shortcut's `cmKey`. For example, for the *My-Edited-Contents App*, this service shortcut is specified in the manifest.

```
"cmServiceShortcuts": [
  {
    "cmKey": "mec-info",
    "name": "Info",
  }
],
```

For this to work without any service specification, the app also registers a runner for this shortcut.

```
useEffect(() => {
  studioAppsContext._.getShortcutRunnerRegistry()
    .registerShortcutRunner(
      "mec-info",
      () => alert("This is the My Edited Contents app!")
    );
}, []);
```

- *cmServices:* These are the "guaranteed" services an app offers (actually, so called *service descriptors* are used here, see Section Section 9.38.5.1, "Service Agent API" [399] below.). An app might or might not offer additional services, but these services that are specified in the manifest are required to be offered by the app. So the developer must make sure to register these services early on in the app's lifecycle. One important thing that the *Studio Apps* framework offers is that for these services, a *service runner* is automatically set up. This means that although an app is not yet running, other apps already know the app's services with state "runnable". If such a service is requested via `Ser viceHandler#fetch()` and the app is not yet running, it is automatically opened in a browser tab. For more details on services and service runners, see Section Section 9.38.5, "App Services" [398].

For example, the *My-Edited-Contents App* specifies one service in its manifest:

```
"cmServices": [
  {
      "name": "myEditedContentsService"
    }
  ],
```

The required service is then registered early on in the app's lifecycle right after the app's basic app initialization.

```
studioAppsContext._.initAppServices().then(() => {
  myEditedContentsServiceDesc = createMyEditedContentsServiceDescriptor({
    providingApp: studioAppsContext._.getMyAppDescriptor().cmKey
  });
  const myEditedContentsService = new MyEditedContentsServiceImpl(setSortModel);
  getServiceAgent().registerService(myEditedContentsService, myEditedContentsServiceDesc);
});
```

The code fragment shows an important aspect. All of the `cmServices` from the manifest are required to include `WithProvidingApp` in their descriptor properties. Upon registration, it is mandatory to set the property `With ProvidingApp#providingApp`. The result from `studioAppsContext._.getMyAppDescriptor().cmKey` is identical with the value of `cmKey` from the manifest but it is only available after `studioAppsContext._.initAppServices()`.

9.38.5 App Services

The app manifests of the previous section provide the basis for integrating apps with each other. For more advanced interactions, an app needs the ability to trigger an action in another app. For example, when working with the *Workflow App*, this app cannot display individual content items. But of course the *Content App* can. So clicking on a content item in the *Workflow App* should open the content item in the *Content App*. The basic mechanism is sketched in Figure .

To allow such interactions between apps running on different frontend technologies and in different browser windows, the *Studio Apps* framework comes with a service layer: Apps offer services to other apps and use services from other apps. The communication to allow these services is based on the browser's *BroadcastChannel* API. However, the developer does not need to concern itself with that. Instead, a higher-level API is provided on top, the *Service Agent* API. It is described in this section. The *Service Agent* is accessible via the NPM package `@coremedia/service-agent`. Its public API is exposed via its `index.ts` file and all types can be imported from the packages top-level.

The top-level method `getServiceAgent()` of `@coremedia/service-agent` provides global access to a *Service Agent* singleton instance. Typically, it is sufficient for an app to use this instance without the need to set up an own instance.

9.38.5.1 Service Agent API

In this section the basics of the *Service Agent* API are described. Built-in services of *CoreMedia Studio Apps* and other pre-defined service types are covered later on.

What is a Service?

The requirements for a service are quite simple and stem from the fact that a service can only offer asynchronous methods due to the message communication layer that is used underneath. A service must comply to the type constraint `Service<T>` (from the `@coremedia/service-agent` module) where `T` is the service's interface. This means in particular:

- A service method can return an *rxjs*`Observable`. In that case, the method name must start with the `observe_` prefix. Covering the library *rxjs* for reactive streams is out of scope for this documentation. For more details, see the various online materials, e.g. RxJS Overview or Learn RxJS.

- If a service method does not return an `Observable` then is must return a `Promise`.

- No other methods are allowed.

- The service may have additional non-function properties.

How the `Promises` and `Observables` are handled across the *Broadcast-Channel* bweteen the different apps is hidden from the developer and taken over by the framework.

There is one additional requirement for services that is not covered by the type constraint `Service<T>`. As all service handling is carried out via the browser's *BroadcastChannel*, the arguments and return values of all service methods must be serializable for the *BroadcastChannel*. Data sent to the channel is serialized using the structured clone algorithm. That means you can send a broad variety of data objects safely without having to serialize them yourself.

Registering Services

As shown in Figure Figure 9.28, "Different Studio Apps Connected Via Service Layer" [391], each app uses its own instance of the *Service Agent*. When an app registers a service with its *Service Agent*, this agent takes care of broadcasting the service to the *Service Agents* of other apps. The registration of a service is

done via the `ServiceAgent#registerService()` method. The following code fragment shows the example where the *Content App* registers the `ContentFormService` whch allows to open content items in a form tab.

```
getServiceAgent().registerService(
  new ContentFormServiceImpl(),
  createContentFormServiceDescriptor({
    providingApp: studioAppsContext._.getMyAppDescriptor().key
  }),
);
```

A service is always registered together with a so-called *service descriptor*. The descriptor describes the service in terms of specific properties where the `name` property is the only madatory one. Upon querying for services, another service descriptor is given and the *Service Agent* matches it against the descriptors of registered services. More details follow below.

As stated above, the registered service needs to comply to the type constraint `Service<T>`. However, this only applies to the service interface. In the case above for example, `ContentFormServiceImpl` has methods that do not comply to the `Service<T>` constraint. But the class implements the interface `ContentFormService` which does comply to the constraint. There are two possibilities to register `ContentFormServiceImpl` with the *Service Agent*. Either it needs to be cast / assigned to its interface `ContentFormService` or the *service descriptor* is used like in the example. The service descriptor for the `ContentFormService` is defined like this:

```
import { serviceDescriptorFactory, ServiceDescriptorWithProps }
  from "@coremedia/service-agent";
import { WithProvidingApp } from "@coremedia/studio-client.app-context-models";

export interface ContentFormServiceProps extends WithProvidingApp {}

export function createContentFormServiceDescriptor(
  props: ContentFormServiceProps = {},
): ServiceDescriptorWithProps<ContentFormService, ContentFormServiceProps> {
  return serviceDescriptorFactory<ContentFormService, ContentFormServiceProps>({
    ...props,
    name: "contentFormService",
  });
}
```

To define service descriptors, it is recommended to use the `serviceDescriptorFactory` function from the `@coremedia/service-agent` module. It returns a `ServiceDescriptorWithProps` which is typed for specific services and descriptor properties. In this case, the descriptor is for services of type `ContentFormService` and their additional properties of type `ContentFormServiceProps` (which currently just extends `WithProvidingApp`). So using this service descriptor for `ServiceAgent#registerService()` fulfils two purposes: (1) It defines the describing properties of the registered service and (2) it enforces that the registered service is of type `ContentFormService` which complies to the `Service<T>`.

Registering Service Runners

Besides registering service it is also possible to register a *service runner*. A service runner is a function that is called when a service is requested but no such service is running at the moment. So it can be used to only launch a service once it is requested. It is done via the `ServiceAgent#registerServiceRunner()` method. It also takes a *service descriptor* as its first argument and the runner function as its second argument.

An important example was already mentioned in Section Section 9.38.4, "App Manifest and Apps Menu Entries" [394]. The services of an app manifest's `cmSer` `vices` are required to be set up upon app initialization. The *Studio Apps* framework automatically sets up a service runner for each of these services. In this case, the runner simply starts the app if it is not yet running. For example, if the *Content App* is not yet running in a browser window, there is no running service `ContentFormService` in place. But because the `ContentForm` `Service` (more specifically, its descriptor) is part of the *Content App*'s `cmSer` `vices` manifest entry, a runner for this service is already in place.

Un-Registering Services and Service Runners

To un-register a service or a service runner, the methods `ServiceAgent#un` `registerServices()` and `ServiceAgent#unregisterServiceRun` `ners()` are used. Both methods take a `ServiceDescriptorWithProps` as their argument. All services whose descriptor matches the given descriptor are un-registered. A service descriptor match is carried out based on *lodash*'s `isMatch()` method, see lodash documentation. The descriptor argument of the unregister method is taken as the second argument of `isMatch()` and the descriptor of the registered services / runners are taken as the first argument. Consequently, the call `ServiceAgent#unregisterServices({name:` `"someService", prop: "myValue"})` would unregister all services with the "name" value "someService" and the "prop" value "myValue", no matter what other properties their descriptors have.

Querying / Requesting Services

The previous paragraphs cover the topic of registering services and service runners and using service descriptors for this purpose. Now the topic of querying / requesting services is covered.

There are two ways to request services.

- `ServiceAgent#getServices()`: Returns all services that are currently known.
- `ServiceAgent#observeServices()`: Returns an *rxjs* `Observable` of all known services. Once the known services change, the `Observable` emits a new value. The `Observable` also always emits the current value on subscribe.

In both cases, a singular version exists (`ServiceAgent#getService()` and `ServiceAgent#observeService()`)) where the first matching service is returned.

The first argument that both methods take is a service descriptor. The descriptor is used to match against the descriptors of the registered services. The matching is done based on *lodash*'s `isMatch()` method. The descriptor argument of the request method is taken as the second argument of `isMatch()` and the descriptors of the registered services are taken as the first argument. Consequently, the call `ServiceAgent#getServices({name: "someService", prop: "myValue"})` would return all services with the "name" value "someService" and the "prop" value "myValue", no matter what other properties their descriptors have. Just as for registering services, it is also recommended for requesting services to use `ServiceDescriptorsWithProps` as arguments. For example, the code for registering the `ContentFormService` in the *Content App* was shown above. The *My-Edited-Contents App* uses this service to open a content item from the edited contents list when clicked. The code looks as follows (the `fetch()` of the code fragment is explained below).

```
<IconButton color={"primary"} onClick={async () => {
  const contentFormService = await getServiceAgent()
    .getService(createContentFormServiceDescriptor())?
    .fetch();
  contentFormService && await
contentFormService.openContentForm(params.row.id);
}}>
```

Because the `ServiceDescriptorWithProps<ContentFormService, ContentFormServiceProps>` returned from `createContentFormServiceDescriptor()` is used, it is possible to continue in a type-safe way: Typescript knows that the returned service is of type `ContentFormService`.

In addition to the service descriptor, both service request methods take an optional second argument of type `ServiceRetrievalOptions`. Currently, there is only one option, namely `ServiceRetrievalOptions#state`. It allows to specify whether only running services shall be retrieved or also runnable services. The latter are services that are not yet running but a service runner exists, see above. By default, all services are retrieved, running and runnable ones.

Using Services

The fact that both running and runnable services can be retrieved is also the reason that no services are directly returned by the two request methods. Instead, a `ServiceHandler` is returned for each service in the result. A handler provides the descriptor for its service via `ServiceHandler#descriptor` and the state of its service ("running" or "runnable") via `ServiceHandler#state`. An important property of a `ServiceHandler#descriptor` is `provider` which holds the `ServiceAgent#getOrigin()` of the service agent where the service was initially registered.

Above all, a service handler offers the `ServiceHandler#fetch()` method. This method returns a `Promise` for the service that only resolves once the service is running. For a running service, this is straightforward. But for a runnable service, the service runner needs to be called first and then it is waited until the service is actually running.

One catch with the `ServiceHandler#fetch()` method is that it always tries to immediately provide a service. Either it is already running or a service runner is attempted. If one simply wants to wait for a service to become available, this can be achieved in multiple ways. For example, in the *My-Edited-Contents App*, the "Opened" column from Figure Figure 9.29, "The My-Edited-Contents Demo App" [392] shows which content items are opened in the *Content App*. To determine this, once again the `ContentFormService` is used. But in this case, it is just waited for the service to become available. If it is not running, the column is simply hidden. The code looks as follows:

```
import {filter, firstValueFrom } from "rxjs";

const serviceHandler = await firstValueFrom(
  getServiceAgent()
    .observeService(createContentFormServiceDescriptor(), {
      state: "running"
    })
    .pipe(filter(Boolean))
);
const contentFormService = await serviceHandler.fetch();
contentFormService.observe_openedContents().subscribe({
  ...
});
```

The combination of `ServiceAgent#observeService()`, state "running" and `firstValueFrom` lets the code wait for a running service and only then continue. For *ReactJs* apps like the *My-Edited-Contents App*, it is helpful to use a generic custom hook. The following `useService()` hook is part of the demo app code and provides the requested service or `null` if no such service is running.

```
import { switchMap } from "rxjs";
import {getServiceAgent, ServiceDescriptorWithProps, Service}
  from "@coremedia/service-agent";

export function useService<T extends Service<T>>(
  serviceDesc: ServiceDescriptorWithProps<T>
): T | null {
  const [service, setService] = useState<T | null>(null);

  useEffect(() => {
    const subscription = getServiceAgent()
      .observeService(serviceDesc, {state: "running"}).pipe(
        switchMap((services) => services.length > 0
          ? services[0].fetch()
          : Promise.resolve(null))
      )
      .subscribe({
        next: (service) =>
          setService((currentService: T | null) => {
            if (currentService && !service) {
              return null;
            }
            if (!currentService && service) {
              return service;
            }
            return currentService;
          }),
        error: () => setService(null),
        complete: () => setService(null),
      });
    return () => subscription.unsubscribe();
  }, [serviceDesc]);

  return service;
}
```

A further possibility to just wait for a service is using `ServiceAgent#ex ecuteServiceMethod()`, see below.

As an advanced option, the `ServiceHandler#fetch()` method takes an optional argument of type `ServiceRunningOptions`. The option `Service RunningOptions#reconnect` allows to specify how to proceed if the service becomes unavailable while still in use. This is especially interesting for `Observ able` service methods. If the value is set to "off", the service is simply no longer usable. Subscriptions to `Observable`s from service methods are automatically terminated with an error. If the value is set to "wait" (which is the default), it is waited for a service to become available which has a service descriptor matching the descriptor of the initial service. If the value is set to "launch", it is attempted to launch a service (via a service runner) with a descriptor matching the descriptor of the initial service if none is already available.

Shorthand Utility executeServiceMethod()

Instead of first retrieving `ServiceHandlers`, then calling `fetch()` on them and finally executing a service method, the `ServiceAgent` provides a short-hand utility. The `ServiceAgent#executeServiceMethod()` allows to

execute a service method without the need to retrieve a `ServiceHandler` beforehand. The service method is identified by a given `ServiceMethod DescriptorWithProps` (which includes a `ServiceDescriptorWith Props` and additional options to identify the specific method). `Ser viceAgent#executeServiceMethod()` is only successful if a service matching the descriptor is running or can be run. The option `ServiceMethod DescriptorWithProps#serviceRunningOpts#reconnect` does not only decide how to proceed in case of a connection loss but also how to initially proceed if a matching service is not available. In case of value "launch", a service is launched if missing (via a service runner). In case of value "wait", the method waits for a service to become available. In case of value "off", an error is thrown. The default value is "launch". As an example, the following code once again shows how to wait for the `ContentFormService` to become available and then observe content items opened in form tabs.

```
getServiceAgent().executeServiceMethod({
  serviceDescriptor: createContentFormServiceDescriptor(),
  method: "observe_openedContents",
  serviceRunningOpts: {reconnect: "wait"}
}).subscribe(...);
```

9.38.5.2 Built-In Services And Utilities

In the previous section, the basics of the *Service Agent* API were covered. This section covers the built-in services and related utility that are ready to use in custom apps. In addition, the services that are already offered by the *Content App* and the *Workflow App* are covered.

Framework Services

Several service types and their descriptors are already defined in the *Studio Apps* framework. The following services are defined in the `@coremedia/stu dio-client.app-context-models` module and can be used as top-level imports.

- *StudioAppService*: This service represents a *Studio App* itself. For each app with a proper manifest according to the guidelines from Section Section 9.38.4, "App Manifest and Apps Menu Entries" [394], a `StudioAppService` is automatically registered where the most important feature is the app's descriptor with properties of type `StudioAppServiceProps`. They mirror the contents of the manifest and provide this information at run-time.

- *RouterService*: This service allows to set the sub-URL/path of an app. As described in Section Section 9.38.4, "App Manifest and Apps Menu Entries" [394],

setting up a `RouterService` is mandatory for each app that offers URL shortcuts in its manifest. Otherwise the `RunAppEntry#run()` methods returned from `studioAppsContext._.observeRunAppSections()` do not work. For the `RouterService` only the interface is provided by the framework. A sub-path is set via the method `RouterService#setPath()`. How this path is actually manifested in the app is up to the app's implementation. For example, the *My-Edited-Contents App* uses a `HashRouter` and the `useNavigate()` hook is used for the implementation of the `RouterService#setPath()`.

- *AutoLogoutService*: In the *CoreMedia Studio*, the user is automatically logged out after a certain period of inactivity. For a smooth user experience it is important that all apps log out jointly. For this purpose, a custom app needs to register its own `AutoLogoutService`. Note that the service does not do the actual log-out itself. Instead, it is responsible for tracking user inactivity and communicate this with the other `AutoLogoutService`s of other apps. So typically, the service is embedded in some form of a wider login context. For example, in the *My-Edited-Contents App*, the service is set up as follows:

```
import {
  AutoLogoutService,
  createAutoLogoutService,
  createAutoLogoutServiceDescriptor
} from "@coremedia/studio-client.app-context-models";
import InputActivityTracker
  from
"@coremedia/studio-client.app-context-models/activitytracker/InputActivityTracker";
import FetchActivityTracker
  from
"@coremedia/studio-client.app-context-models/activitytracker/FetchActivityTracker";
...

async #setupAutoLogoutService(): Promise<void> {
  this.#autoLogoutService = createAutoLogoutService({
    autoLogoutDelay: 1800000,
    activityTrackers: [
      new InputActivityTracker(document.body),
      new FetchActivityTracker(),
    ],
    serviceAgent: getServiceAgent(),
  });
  getServiceAgent().registerService(
    this.#autoLogoutService,
    createAutoLogoutServiceDescriptor()
  );
  this.#autoLogoutSubscription = this.#autoLogoutService
    .observe_status()
    .subscribe(async (status) => {
      if (status === "loggedOut" || status === "autoLoggedOut") {
        await this.#doLogout();
        await this.tearDownAutoLogoutService();
      }
    });
}
```

The service is created with a delay of 30 minutes and two activity trackers, then registered with the *Service Agent* and finally an observer for the service's

status is set up. If the status switches to "loggedOut" or "autoLoggedOut", then the actual logout is performed via `doLogout()` which is not covered here and is not part of the `AutoLogoutService`. In addition, when the user explicitly logs out in the current app for example via a button then `AutoLogoutService#forceLogout()` must be called to trigger the logout.

Just as all apps should log out jointly, logging into one app should also reload the other apps that are currently opened in a browser window but do not have a valid session. The above mentioned `StudioAppService` offers a method for that. In the *My-Edited-Contents App*, this code is executed upon login:

```
const appServices = getServiceAgent().getServices(
  createStudioAppServiceDesc(), {state: "running"}
).filter(handler => handler.descriptor.provider !==
getServiceAgent().getOrigin());
appServices.forEach(
  async (appServiceHandler) =>
    (await appServiceHandler.fetch()).reloadWithoutSession()
);
```

Studio Apps Utilities

As already mentioned in Section Section 9.38.3, "Accessing the Studio Apps Context" [393], the `studioAppsContext` utility is also available as a top-level import of the `@coremedia/studio-client.app-context-models` module. It provides the following utility functions:

- *initAppServices()*: This method has already been covered in Section Section 9.38.4, "App Manifest and Apps Menu Entries" [394].

- *observeRunSections()*: This method has already been covered in Section Section 9.38.4, "App Manifest and Apps Menu Entries" [394].

- *getShortcutRunnerRegistry()*: This method has already been covered in Section Section 9.38.4, "App Manifest and Apps Menu Entries" [394].

- *runApp()*: This method takes a `ServiceDescriptorWithProps<Studi oAppService, StudioAppServiceProps>` argument to run the app that this descriptor denotes. If the app is already running in a browser window, it is brought into the foreground.

- *getMyAppDescriptor()*: Returns the `ServiceDescriptorWithProps<Stu dioAppService, StudioAppServiceProps>` of the current app. It mirrors the contents of the app's manifest.

- *focusMe()*: Focuses the current app and brings its browser window / tab into the foreground. This method is typically called inside service methods where an action is triggered that must be immediately visible to the user, for example `ContentFormService#openContentForm()`.

- *getAppStartupParameters()*: Sometimes an app is not just run in a browser window but also with additional parameters. Currently, this is only the case when the app is run via a service runner for one of the app's `cmServices` from their manifest, see Section Section 9.38.4, "App Manifest and Apps Menu Entries" [394]. The startup parameters are then available via this method. In this case, it provides the app with the information which service was initially requested to trigger the app's start-up. An example to use this method is given in Section Section 9.38.6, "Multi-Instance Apps" [414].

- *setWindowOpenHandler()*: Apps are opened in browser windows / tabs. This method allows to override the handler that is called when an app is run or brought into the foreground. In the future this might be a lever to allow for more advanced PWA features. For now, it is mainly to deal with browser security restrictions. A popup blocker might prevent the focussing of an app's window / tab. This goes unchecked by the default window handler of the *Studio Apps* framework. But the module `@coremedia/studio-client.app-context-models` provides a handler that display a warning for that. It is currently used by all built-in apps and also by the *My-Edited-Contents App*.

```
import { openOrFocusApp, studioAppsContext }
  from "@coremedia/studio-client.app-context-models";

studioAppsContext._.setWindowOpenHandler(openOrFocusApp);
```

- *addWindowValidityObservable()*: This is another tool to deal with browser security restrictions. For app windows to be able to interact properly with each other (especially to focus each other), they need to be in the same so-called "window group" of the browser. This is only the case, if all apps were started beginning with one app and further apps are always opened from one of the already running ones. Once a browser window is opened isolated from this chain and an app run in it, this app is not connected to the other apps. This method allows to add an `Observable` to track whether the current app window is a connected one. By default, no such `Observable` is in place but the module `@coremedia/studio-client.app-context-models` provides one. It is currently used by all built-in apps and also for the *My-Edited-Contents App*.

```
import { observeWindowStudioAppsConnection, studioAppsContext }
  from "@coremedia/studio-client.app-context-models";

    studioAppsContext._.addWindowValidityObservable(
```

```
        observeWindowStudioAppsConnection()
    );
```

In addition, the *My-Edited-Contents App* also reacts if the window connection cannot be established.

```
studioAppsContext._.observeWindowValidity().subscribe((valid) => {
  if (!valid) {
    alert("This browser tab has no connection to the other Studio browser
 tabs. Please close it and continue in another Studio browser tab.");
  }
});
```

Data Transfer Services

The module `@coremedia/studio-client.interaction-services-api` contains some base services for data transfers between apps.

- *DragDropService*: The built-in *CoreMedia Studio Apps*, namely the *Content App* and the *Workflow App*, offer to drag content items between apps and thus between browser windows. For this, HTML5 drag/drop is used (also cf. Section Section 9.15, "HTML5 Drag And Drop" [201]). In order to allow such drag operations to result in a drop in a custom app, it is sufficient to just register a "drop" event handler. However, the `DragDropService` also offers information about the dragged items during the drag operation and not only once the drop happens. The service is only running during a drag operation and unregistered once this is finished.

  The property `DragDropService#dataTransferItems` mirrors the drag data of an HTML5 drag / drop event (`DragEvent.dataTransfer.items`). No matter whether the service data is used or the data from the drop event, they have the same structure. For example, for a typical drag operation of content items, the drag data looks like this:

```
{
  "cm-studio-rest/uri-list": "[\"content/3350\",\"content/3356\",\"content/3370\",\"content/3360\"]",

  "cm-content/uuid-list":
"[\"14a565c-170a-436-681-e930202040\",\"ta061ca-682-400-a72-3f1751f9t6\",\"t8f5468-28b-41cb-9lcd-1elc85ael7\",\"6c1t819-1142-480-96a-0adtf0a37ca\"]",

  "cm-member/uuid-list": "[]",
}
```

  Here it is visible that different flavors / types of drag data exist. The first two types are just different representations of the same content items. But it is visible that also member items (users and groups) might be dragged. The currently supported types are enumerated in `DataTransferTypes` of the module `@coremedia/studio-client.interaction-services-api`.

The following shows the drop handler of the *My-Edited-Contents App* to receive content items via a drag drop operation from the *Content App*.

```
onDrop={event => {
  const uriListData = event.dataTransfer.getData(DataTransferTypes.CM_STUDIO_REST_URI_LIST);
  if (!uriListData) {
    event.preventDefault();
    return;
  }
  const parsedUriList = JSON.parse(uriListData);
  if (!parsedUriList || !Array.isArray(parsedUriList)) {
    event.preventDefault();
    return;
  }
  const contentUriRestTemplate = new URITemplate(ContentImpl.REST_RESOURCE_URI_TEMPLATE);
  const contents = parsedUriList.filter((uri) =>
    contentUriRestTemplate.matches(uri)).map(beanFactory._.getRemoteBean);
  contents && contents.length > 0
    && session._.getConnection().getCapListRepository()
    .getEditedContents().addItems(contents);
}}
```

While custom apps can receive drops of content items from the built-in apps, there currently exists no further support to set up a drag operation from a custom app back to the built-in apps. However, it is possible if HTML5 drag / drop is used and the data is set up according to the above structure.

- *Clipboard*: Similar to drag / drop, the clipboard allows to transfer content or member items between apps. The `Clipboard` of the module `@core media/studio-client.interaction-services-api` implements the Clipboard Web API and offers methods for reading and writing clipboard data. To access the clipboard, the global constant `clipboard` from `@coremedia/studio-client.interaction-services-api` can be used.

The clipboard reads and writes data in the form of `ClipboardItem`s. These items support different flavors / types, analogous to the drag / drop data from above. For example, the *My-Edited-Contents App* has a global key handler to paste from the clipboard.

```
useEffect(() => {
  const onPaste = async (ev: KeyboardEvent) => {
    if (ev.key === "v" && (ev.ctrlKey || ev.metaKey)) {
      const clipboardItems = await clipboard._.read();
      const clipboardItem = clipboardItems.find((clipboardItem)
        => clipboardItem.types.includes(DataTransferTypes.CM_STUDIO_REST_URI_LIST));
      if (!clipboardItem) {
        return;
      }
      const restUrisItem = await clipboardItem
        .getType(DataTransferTypes.CM_STUDIO_REST_URI_LIST);
      const restUrisItemString = restUrisItem === "string"
        ? restUrisItem
        : await (restUrisItem as Blob).text();
      try {
        const restUris: Array<string> = JSON.parse(restUrisItemString);
        if (restUris) {
          const contents = restUris.map(beanFactory._
            .getRemoteBean).filter((bean) => is(bean, Content));

          contents
            && contents.length > 0
            && session._.getConnection().getCapListRepository()
            .getEditedContents().addItems(contents);
        }
      } catch (e) {
        // ignore
      }
    }
  };

  window.addEventListener("keyup", onPaste, false);
```

```
  return () => {
    window.removeEventListener("keyup", onPaste, false);
  };
}, []);
```

The *My-Edited-Contents App* also has a similar global key handler to copy into the clipboard, using the selection of the edited contents list.

```
const [selection, setSelection] = useState<GridSelectionModel>([]);

...

useEffect(() => {
  if (!selection || selection.length === 0) {
    return;
  }
  const onCopy = async (ev: KeyboardEvent) => {

    if (ev.key === "c" && (ev.ctrlKey || ev.metaKey)) {

      try {
        const data: Record<string, any> = {};
        const remoteBeans = selection
          .map(selected => beanFactory._.getRemoteBean(selected.toString()))
          .filter(Boolean);

        if (!remoteBeans.every(RemoteBeanUtil.isAccessible)) {
          return;
        }

        const restUriBlob = new Blob(
          [
            JSON.stringify(remoteBeans.map((remoteBean) => remoteBean.getUriPath())),
          ],
          {type: DataTransferTypes.CM_STUDIO_REST_URI_LIST},
        );
        data[restUriBlob.type] = restUriBlob;

        await clipboard._.write([new ClipboardItemImpl(data)]);

      } catch (e) {
        // ignore
      }
    }
  };

  window.addEventListener("keyup", onCopy, false);

  return () => {
    window.removeEventListener("keyup", onCopy, false);
  };
}, [selection]);
```

Feature Services Of Content And Workflow App

The *Content App* and the *Workflow App* offer some feature services out of the box. All of them are also listed under the `cmServices` property of the app manifests, so there exist automatic service runners for them that launch the apps if needed, cf. Section Section 9.38.4, "App Manifest and Apps Menu Entries" [394].

- *ContentFormService*: This service and its descriptor factory are exported by the module `@coremedia/studio-client.content-service-api`. The service is offered by the *Content App* and allows to open content items in form tabs, track which content items are currently opened and which content item is the active one.

- *ProjectFormService*: This service and its descriptor factory are exported by the module `@coremedia/studio-client.project-services-api`. The service is offered by the *Content App* and allows to open project items in form tabs, track which project items are currently opened and which project item is the active one.

- *CollectionViewService*: This service and its descriptor factory are exported by the module `@coremedia/studio-client.collection-view-services-api`. The service is offered by the *Content App* and allows to display content items in the collection view (library) and to open the collection view in a specific content search state.

- *WorkflowFormService*: This service and its descriptor factory are exported by the module `@coremedia/studio-client.workflow-services-api`. The service is offered by the *Workflow App* and allows to open workflow objects (processes and tasks) in forms, track which workflow objects are currently opened and which workflow object is the active one. In the case of the *Workflow App*, there is at most one workflow object opened but the API was intentionally kept similar to the above-mentioned form service APIs.

9.38.5.3 Adding Custom Services

The previous sections cover the built-in services and utilities that are available for custom apps. This section covers how to add custom services to the *Studio Apps* framework.

Services of Custom App

Adding services that a custom app offers is straightforward and a typical pattern for *ReactJs* apps is as follows, taken from the *My-Edited-Contents App*.

```
const [sortModel, setSortModel] = React.useState<GridSortItem>({
  field: "displayName",
  sort: undefined,
});

...

useEffect(() => {
  const myEditedContentsServiceDesc = createMyEditedContentsServiceDescriptor({
    providingApp: studioAppsContext._.getMyAppDescriptor().cmKey
  });

  const myEditedContentsService = new MyEditedContentsServiceImpl(setSortModel);
  getServiceAgent().registerService(myEditedContentsService, myEditedContentsServiceDesc);

  return () => myEditedContentsServiceDesc
              && getServiceAgent().unregisterServices(myEditedContentsServiceDesc);
}, [])
```

The service is registered in the context of a `useEffect()` hook. The service gets passed a state setter function of a *ReactJs* component, so that it can have

effect on the app. Finally, the service is unregistered when the component is un-mounted.

In this particular case, the `MyEditedContentsService` is one that is also listed in the `cmServices` list of the app manifest. So it its `MyEditedCon tentsServiceProps` are required to extend `WithProvidingApp` and the providing app must be specified.

Adding Services to The Content and Workflow App

Another use case is when a custom app requires a service from a built-in *Core-Media* app that does not exist out of the box. However, in this case, such a service can be added via a classical `StudioPlugin` or `StudioStartupPlugin`, cf. Section 9.3, "Studio Plugins" [130]. For example, the *My-Edited-Contents App* has a button to show a `StartWorkflowWindow` in the *Content App*. The service for this is not built-in. But part of the *My-Edited-Contents App* workspace is the `MyEditedContentsAppStartupPlugin` for the *Content App* with the following service registration code.

```
override async init(): Promise<void> {
  super.init();

  const availablePublicationProcesses = await this.waitForPublicationWorkflowAccess();
  const publicationWindowService =
    new ShowStartPublicationWindowServiceImpl(availablePublicationProcesses);
  const serviceDesc = createShowStartPublicationWindowServiceDescriptor({
    providingApp: studioAppsContext._.getMyAppDescriptor().cmKey,
  });
  getServiceAgent().registerService(publicationWindowService, serviceDesc);
}
```

The service implementation `ShowStartPublicationWindowServiceImpl` looks like this.

```
class ShowStartPublicationWindowServiceImpl implements ShowStartPublicationWindowService {

  #availableProcessDefinitions: String[];

  constructor(availableProcessDefinitions) {
    this.#availableProcessDefinitions = availableProcessDefinitions;
  }

  async showStartPublicationWindow(contentUris: Array<string>): Promise<void> {
    const config = Config(StartWorkflowWindow);
    config.title = ControlRoom_properties.StartWorkflowWindow_publication_title;
    config.defaultWorkflowName = "Publish Edited Contents";
    config.availableProcessDefinitionNames = this.#availableProcessDefinitions;
    config.initialContents = contentUris.map(beanFactory._.getRemoteBean);
    config.getWorkflowIssuesWindowFunction = WorkflowUtils.getWorkflowIssuesWindow;
    const window = new StartWorkflowWindow(config);
    window.show();

    await studioAppsContext._.focusMe();
  }
}
```

Both service implementation and the registration code take place in the traditional *ExtJs CoreMedia Studio* context. It is even possible to add this new service as a `cmService` entry to the *Content App*'s manifest by adding this fragment to the `jangaroo.config.js` of the plugin module.

```
appManifests: {
  en: {
    cmServices: [
      { name: "showStartPublicationWindowService" },
    ],
  },
}
```

Now the service can be used in the *My-Edited-Contents App*.

```
<IconButton color={"primary"} onClick={async () => {
  const contentUris = contentDetails.map((contentDetail) => contentDetail.id);
  getServiceAgent().executeServiceMethod({
    serviceDescriptor: createShowStartPublicationWindowServiceDescriptor(),
    method: "showStartPublicationWindow",
  }, contentUris);
}}>
```

9.38.6 Multi-Instance Apps

All of the *Studio Apps* provided by *CoreMedia* (the *Content App*, *Workflow App* and *Campaign App*) are singleton apps. There is always only one instance of an app running in a browser window. However, the framework also provides support for multi-instance apps where an app can run in multiple instances in multiple browser windows in parallel.

A straightforward utility are so called *launch handlers* for apps and their shortcuts. The mechanism follows the Launch Handler API. A launch handler controls how an app is launched, for example if it uses an existing window or creates a new one. The default is the singleton mode that the built-in apps use: Only if an app is not yet running, a new window is opened. Otherwise, the existing window is focused. But it is possible to add a launch handler for the app itself or for one of its shortcuts in the app manifest. These launch handlers are currently taken into account for the `studioAppsContext._.runApp()` method and the `RunAppEntry#run()` methods for entries returned from `studioAppsContext._.observeRunAppSections()`.

For example, this entry at the top level of the *My-Edited-Contents App* manifest would open a new instance of the app each time the app is run with one of the two methods.

```
launch_handler: {
  client_mode: "navigate-new"
}
```

A more sophisticated utility to support multi-instance apps is using entries in the `cmServices` list of the app manifest with regular expressions. Suppose a multi-instance app is to offer a `MultiInstanceService` but that service is different from instance to instance, in this example depending on an `id` parameter of the service descriptor. In that case, the `cmService` entry can look like this:

```
{
  name: "multiInstanceService",
  id: {
    type: "RegExp",
    value: ".*",
  },
}
```

The `id` parameter is dynamic. Because the *Service Agent API* can also deal with regular expressions in service descriptors, the *Studio Apps* framework can set up a dynamic service runner for this service. So for example, another app requests this service in the usual way:

```
getServiceAgent().getService({name: "multiInstanceService", id: 123}).fetch()
```

Initially no app is running that offers this service with this id. But because of the manifest entry from above, it is a "runnable" service and the concrete id "123" matches the regular expression ".*". Consequently, the app is launched in a new browser window. This app can now set up the service with this exact id because it can access the requested service via `studioAppsContext._.get AppStartupParameters()` and `StartupParameters#requested ServiceDescriptors`.

If the same service request from above (with id "123") is made again, the app is already running and can just be focused. But if the service is requested with a different id, a new instance of the app is launched.

10. Rich Text Editing

In this section you will get to know about the rich text editing components integrated into *CoreMedia Studio*. Rich text editing is powered by *CKEditor* as provided by *CKSource sp. z o.o. sp.k.*. As such, you will learn how *CKEditor* is integrated into *CoreMedia Studio*, how you may customize *CKEditor* and how to integrate with other features of *CoreMedia Studio*.

Update from CKEditor 4

Starting with CMCC 12 (2401) the previous *CKEditor 4* integration is not supported anymore. If you are updating from an adapted *CKEditor 4* integration, please consult CMCC 11 documentation for upgrade hints, which can be summarized as: For any custom plugin you wrote, take its specification and rewrite it based on the *CKEditor 5* architecture. You will benefit from a better architecture and more flexibility when it comes to integrating native *CKEditor 5* plugins for use in *CoreMedia Studio*.

10.1 CKEditor 5 Concepts

In this section you will get rough sketches of some design concepts of *CKEditor 5* and of concepts for support editing of *CoreMedia Rich Text 1.0* and *CoreMedia Studio* integration.

For more details on *CKEditor 5* consult the corresponding CKEditor 5 documentation.

10.1.1 Glance at CKEditor 5 Architecture

This section will provide a rough glance at the architecture of *CKEditor 5* with focus on the "data layer", which is important to understand for *CoreMedia Rich Text 1.0* integration. You will find much more details on the architecture at Introduction to CKEditor 5 architecture. And within that, you will find a detailed documentation of the various layers in *CKEditor 5* at Editing engine.

CKEditor 5 Editing Layers
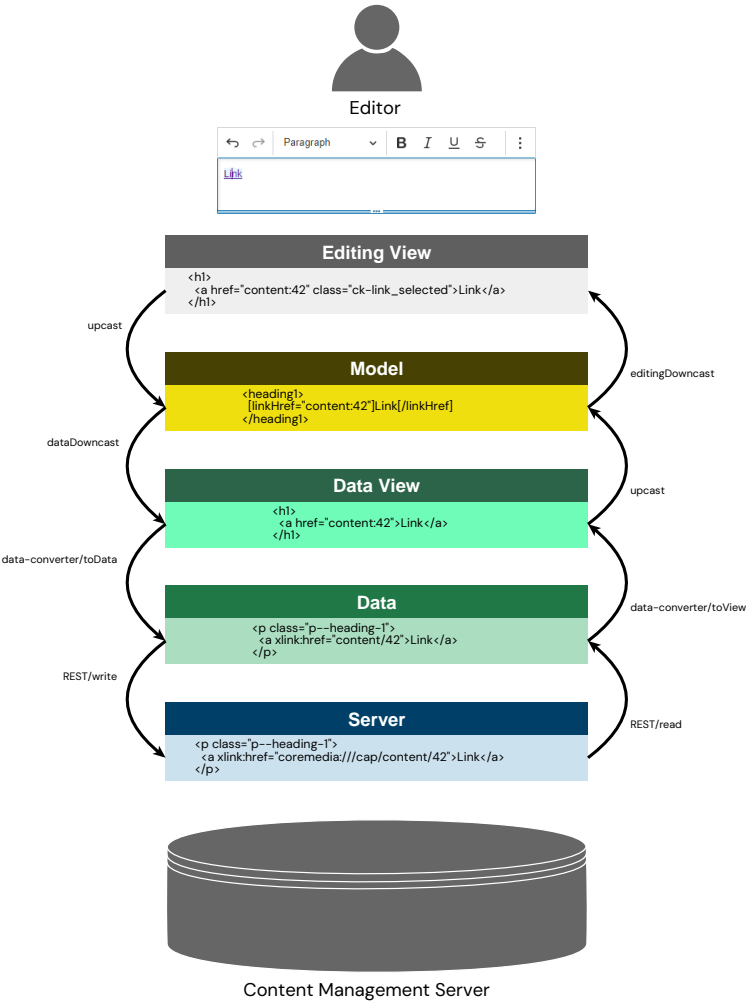
Figure 10.1. CKEditor 5 Editing Layers

Figure 10.1, "CKEditor 5 Editing Layers" [418] shows how different layers are involved dealing with *CoreMedia Rich Text 1.0*. Starting at reading *CoreMedia Rich Text 1.0* from server, the processing steps are as follows:

1. **Initial Data on Server:**

As you see, *CoreMedia Rich Text 1.0* provides only a small subset of what HTML offers. Element `<h1>` is unknown, and instead represented as `<p class="p--heading-1">`. Anchor attribute `href` is also unknown and instead *CoreMedia Rich Text 1.0* relies on attributes as defined by XLink schema such as `xlink:href`. The value of `xlink:href` refers to a content with ID 42 here.

2. **REST/read:**

   Studio REST backend transforms content-URIs to a shortened format: `content/42` instead of `coremedia:///cap/content/42`.

3. **data-converter/toView:**

   The `RichTextDataProcessor` applies the following transformations before *the data enter the world of CKEditor 5* (starting with the data view):

   • `<p class="p--heading-1">` is transformed to `<h1>`. This follows the design principle as described in Section 10.1.2, "Design Principle: HTML First" [420].

   • The link `content/42` received from *Studio* server is rewritten to `content:42` and set as `href` of the anchor rather than the corresponding XLink attribute. This eases *CKEditor 5* plugins to accept the reference as normal URL with a custom link scheme `content:` and reduces the efforts to hook deeply into corresponding plugins.

4. **upcast:**

   This is *CKEditor 5* terminology: As the model layer takes the lead, *CKEditor 5* leverages the incoming data view to its model representation. The model knows about cursor positions, selection ranges and represents inline styles as *attributed texts* rather than elements around text (shown in graph with square brackets).

5. **editingDowncast:**

   Now *CKEditor 5* prepares the editing view for the editors. Classes are applied to highlight selections, possibly empty block elements get some so called "filler" elements, which ensure, that the cursor can be placed inside that element, and more.

   For *Studio* integration, for example, BLOB references are resolved here, so that some image or placeholder symbol is shown.

The way back from edited text to the date stored on server is similar. For data-processing it is important to respect: Mappings have to be *bijective*: As `<p class="p--heading-1">` mapped to `<h1>` in `toView` mapping, it now needs to be transformed back from `<h1>` to `<p class="p--heading-1">`. This also means, that when creating the data view representation, you must

add enough information, so that, when returning from editing view, the original state can be restored.

Relevant Change Detection

As you may guess from the previous transformation process, data set at *CKEditor 5* may contain subtle differences when returning from processing through the various layers. Examples are ignorable whitespace possibly stripped or class-attribute values, which got re-ordered.

Auto-Checkout

Such *semantically equal* data must not trigger, for example, auto-checkout in *CoreMedia Studio*, as it would trigger content items to be checked out by the current editor. Without being aware of this, an editor would check out a content item just by opening it in *CoreMedia Studio*. If this succeeds, other editors will be blocked from editing — if it fails due to missing permissions the corresponding editor would receive an alert message instead.

For the integration of *CKEditor 5* into *CoreMedia Studio* this is prevented by validating the model state of *CKEditor 5* (see Document and corresponding property `version`). Only if this validation signals a change from previously set data, auto-checkout is triggered.

For details take a look at CoreMedia CKEditor 5 Plugins regarding the Autosave feature. Understanding these may be important if you want to integrate *CKEditor 5* into custom components. Also, the Data Facade Plugin [425] provides support to deal with this aspect.

10.1.2 Design Principle: HTML First

An important design decision for data-processing — like for transformation of *CoreMedia Rich Text 1.0* — is to stick to the design principle *HTML first*.

Sticking to this principle lowers the barriers (or even removes them) for plugins as provided by *CKEditor 5* to handle the received data. So, CoreMedia may have decided to stick to represent links as `xlink:href` attribute in data view. Then we would be forced to define an upcast from data view to model, which follows the same rules as defined in Link plugin provided by *CKEditor 5* regarding the `src` attribute. This again would have increased maintenance costs when applying *CKEditor 5* updates.

Thus, if you introduce similar mappings, it is recommended to find the best representation in the data view, which can be handled by standard *CKEditor 5* plugins.

The concept is also important to understand, when integrating plugins provided by *CKEditor 5*. They do not know of the received data (here: *CoreMedia Rich Text 1.0*) but only of the representation in data view.

Examples:

- **General HTML Support:** To register known elements or attributes, which are not necessarily supported by explicit plugins, you may want to register them via General HTML Support. This ensures, they are not removed when loaded from the server as they are considered *unknown*. The elements and attributes to register here, are those from data view. So, if you add an attribute to headings, you will register this for `<h1>` rather than `<p>` (its representation in *CoreMedia Rich Text 1.0*).

- **Styles:** Same applies to the Styles feature. If you register styles to be applied to headings, you will register them for their representations in data view rather than in *CoreMedia Rich Text 1.0* data.

What is second?

If HTML is *first*, the obvious question is: What is second? Some answers to this question may help you to design your customized processing of data retrieved from the server.

Second is data-consistency

Thus, if any data from the server cannot be retained, there must be another way to represent them in the various layers. See some examples:

- **Embedded Media:** For images backed by content BLOB properties we need the `src` attribute of the `<img>` element to load the BLOB data from *Studio* server. In this case, we have to remember the original value of `xlink:href` which denotes the content and property to read the BLOB from. The question is, how to handle attributes like `xlink:show` and `xlink:role` available for images as well as for anchors.

- **Augmented Data for Differencing:** Augmenting elements and attributes exist for difference highlighting retrieved from the server (see: Section 10.2.5, "Differencing Plugin" [425]).

Possible Solutions

There are various design approaches you may choose from. Here is a short summary of the approaches used in context of the Section 10.2.10, "Rich Text Plugin" [427] which may help you to design similar approaches:

- For `xlink:href` of the `<img>` element, the original value is stored as `data-xlink-href` on data-processing, thus for data view. For editing downcast the attribute is ignored, thus it is only kept in the model layer. Instead, it controls filling the `src` attribute with a corresponding BLOB value.

  Thus, HTML data attributes are used to retain data and possibly even strip them in editing downcast.

- For `xlink:role` and `xlink:show` the data attribute solution for the `<img>` element is used, as there is no alternative attribute in HTML. Thus, images will have `data-xlink-role` and `data-xlink-show` as attributes.

  Different to that, `xlink:role` and `xlink:show` map to the `target` attribute for `<a>` elements, as they are slightly related. A good mapping guarantees, that pasted HTML from external resources is kept at best effort. So, `target="_blank"` is mapped to `xlink:show="new"`.

  For a complete overview of mapping approaches, see Section 10.2.9, "Link Plugins" [427] and the contained `LinkTarget` plugin.

- Sometimes, like in differencing augmentation, the corresponding elements and attributes are just forwarded with `xdiff:` namespace directly from data over data view and model up to the editing view.

  In addition, an artificial element `xdiff:br` was introduced to help on CSS styling to highlight added or removed newlines.

As you see, designing a good mapping requires, among other things, to respect data consistency as well as editing experience and good compatibility to pasted HTML data.

10.1.3 Studio Integration: Service Agent

CoreMedia Studio integrates *CKEditor 5* instances of ClassicEditor. These instances are loosely coupled with *CoreMedia Studio* via the `serviceAgent` as provided by the `@coremedia/service-agent` npm pacakge.

The services are registered in `@coremedia-blueprint/studio-client.main.ckeditor5-plugin` and contain, for example:

- **`StudioContentDisplayService`:** Responsible for resolving content-references to their names or corresponding type icons.
- **`StudioBlobDisplayService`:** Responsible for resolving BLOB property references to either images to render inline in *CKEditor 5* or placeholder icons for audio BLOBs, for example.

Thus, there is no direct communication from *CoreMedia Studio* to *CKEditor 5* and vice versa. If you require information from *Studio*, we recommend a similar approach.

10.1.4 Studio Integration: CKEditor 5 Configurations

CoreMedia Studio integrates *CKEditor 5* instances of ClassicEditor. These instances are configured in the package `@coremedia-blueprint/studio-client.ckeditor5`.

Each flavor of *CKEditor 5* you want to integrate into *CoreMedia Studio* has to be provided as an extra configuration similar to the default `ckeditorDefault.ts`, which is the main instance used in most content forms of *CoreMedia Studio*.

All available instances must be exposed as factory methods in `ckeditor.ts`, registered in `@coremedia-blueprint/studio-client.main.ckeditor5-plugin` and may then be referenced in configurations of `CKEditor5RichTextArea` via `editorType`.

For details, customizing existing *CKEditor 5* configurations or providing and using new ones, see Section 10.3, "CKEditor 5 Customization" [430].

10.2 CKEditor 5 CoreMedia Plugins

While *CKEditor 5* comes with a rich set of plugins, some additional plugins are required especially for integration into *CoreMedia Studio*. This section provides a reference to these plugins, which are described in detail at CoreMedia CKEditor 5 Plugins.

10.2.1 BBCode Plugin

This plugin provides support editing BBCode data. It provides a data-processor for mapping BBCode data to HTML in data view as well as for mapping HTML in data view back to BBCode data.

The plugin is bundled in the npm package `@coremedia/ckeditor5-bbcode`. For more details regarding this plugin consult CoreMedia CKEditor 5 Plugin: BBCode.

Package Information

10.2.2 Blocklist Plugin

This plugin provides support for reviewers to identify forbidden or discouraged terms within the displayed text. Blocked words are stored and retrieved via a `BlocklistService`, so that a all reviewers share a common list of blocked words.

The plugin is bundled in the npm package `@coremedia/ckeditor5-coremedia-blocklist`. For more details regarding this plugin consult CoreMedia CKEditor 5 Plugin: Blocklist.

Package Information

10.2.3 Content Clipboard Plugin

This plugin provides support for drag and drop operations from *CoreMedia Studio* to the integrated *CKEditor 5* instance.

The plugin is bundled in the npm package `@coremedia/ckeditor5-coremedia-content-clipboard`. For more details regarding this plugin consult CoreMedia CKEditor 5 Plugin: Content Clipboard.

Package Information

10.2.4 Data Facade Plugin

This plugin can be perceived as proxy on the *CKEditor 5* data layer. Setting data as well as retrieving data should be performed via this plugin and its provided controllers, if you want to ensure that changed data are only provided on editorial actions.

Data access without this proxy may result that a given API access flow like setting and immediately afterward getting the data may result in different data, as *CKEditor 5* may have applied some normalization (like reordering attributes).

Thus, if you need to ensure, that modifications are not propagated if they are only a result of normalization (like, to prevent *auto-checkout* of content items just by opening them within *CoreMedia Studio*), this plugin will help you to prevent this via an internally maintained cache of previously set data, that gets invalidated as soon as editorial actions are applied in *CKEditor 5*.

The plugin is bundled in the npm package `@coremedia/ckeditor5-data-facade`. For more details regarding this plugin consult CoreMedia CKEditor 5 Plugin: Data Facade.

Package Information

10.2.5 Differencing Plugin

This plugin is meant to ensure, that differencing data as generated in *CoreMedia Studio* are forwarded to editing view, so that CSS rules can be applied to it, to highlight changes, additions and deletions.

The plugin is bundled in the npm package `@coremedia/ckeditor5-coremedia-differencing`. For more details regarding this plugin consult CoreMedia CKEditor 5 Plugin: Differencing.

Package Information

10.2.6 Font Mapper Plugin

When pasting rich text from external sources into *CKEditor 5*, some characters of the pasted text might originate from a font that is not typically available for all platforms. This plugin allows mapping such characters to their named entities or Unicode equivalents, for example.

By default, this plugin already contains a mapping table for the Microsoft Word `Symbol` font, and automatically converts input content (for instance while pasting from Microsoft Word) accordingly.

The plugin is bundled in the npm package `@coremedia/ckeditor5-font-mapper`. For more details regarding this plugin consult CoreMedia CKEditor 5 Plugin: Font Mapper.

10.2.7 General Rich Text Support Plugin

The plugin *General Rich Text Support* ("GRS") ensures that any valid *CoreMedia Rich Text 1.0* text, especially attributes, can be loaded into *CKEditor 5*. It does not provide any editing features, but only registers elements, attributes and attribute values, which are not yet supported by corresponding editing and/or data-processing features. Having this, yet unknown elements and attributes are not removed when loaded from server. For stricter behavior, just allowing elements and attributes, which you can edit in *CKEditor 5* you may want to skip installing this plugin.

> ### When To Remove The Plugin
>
> If you want to ensure, that, for example, on copy & paste between property editors all elements are removed, which cannot be created in the target editor, you should remove this plugin (or replace integration of the Studio Essentials Plugin [429] by only the Rich Text Plugin [427]).
>
> Removing the plugin will disallow any of these unknown elements to appear in *CKEditor 5* and subsequently in stored *CoreMedia Rich Text 1.0* property value.
>
> Note though, that this will also trigger removing such unknown elements when they are read from server. For example, because they got created by API calls or in other client applications. Such cleanup will be applied as soon as editors start changing the text.

GRS is based on General HTML Support ("GHS") provided by *CKEditor 5*.

The plugin is bundled in the npm package `@coremedia/ckeditor5-coremedia-richtext-support`. For more details regarding this plugin consult CoreMedia CKEditor 5 Plugin: General Rich Text Support.

The plugin is part of Studio Essentials Plugin [429].

10.2.8 Images Plugin

The plugin is responsible for showing and editing images in *CoreMedia Rich Text 1.0* that are stored as BLOB properties in corresponding contents.

The plugin is bundled in npm package `@coremedia/ckeditor5-core-media-images`. For more details regarding this plugin consult CoreMedia CKEditor 5 Plugin: Images.

10.2.9 Link Plugins

These plugins integrate with the Link feature provided by *CKEditor 5*. They provide support for content links (also sometimes referred to as "internal links") and adapt the target behavior for links such as, if to open in new tab, in named tab.

Regarding links in *CKEditor 5* there is also an assistive plugin `LinkAttributes` available, explained in more detail in Section 10.3.10, "Link Editing" [445]. It is recommended having a look at that section if you wan to provide custom attributes for link editing.

The plugins are bundled in npm package `@coremedia/ckeditor5-core-media-link`. For more details regarding the plugins consult CoreMedia CKEditor 5 Plugin: Link.

10.2.10 Rich Text Plugin

This plugin provides a data processor for *CKEditor 5*, so that you can load *CoreMedia Rich Text 1.0* into *CKEditor 5*. Understanding this plugin and its configuration is crucial for integrating most of the plugins provided by *CKEditor 5* as well as, when you want to provide support for additional attributes and elements as they are known by HTML5, for example.

The data-processor also defines some *reserved classes*, which are applied as `class` attribute values, which are important to understand, when designing the delivery of contents on your web page. These classes include, for example, `p--heading-1`, to denote a paragraph as to be rendered as `<h1>` or more sophisticated classes such as `tr--header`, `td--header` to be rendered as `<thead>` and `<th>` respectively.

The plugin is bundled in the npm package `@coremedia/ckeditor5-coremedia-richtext`. For more details regarding this plugin and its configuration consult CoreMedia CKEditor 5 Plugin: Rich Text.

The plugin is part of the Studio Essentials Plugin [429] plugin.

Note on Strictness

Apart from the element and attribute mappings, the Rich Text plugin also ships with a sanitation layer. This layer is responsible *to store data on server at best effort*. *Best effort* means, that the resulting data after data processing must

represent valid *CoreMedia Rich Text 1.0*. Sanitation ensures that, for example, any invalid elements are removed at last processing stage, so that all other data are kept.

Unknown Element Example: You added the Highlight introducing the `<mark>` element to *CKEditor 5*. As initially no mapping exists for that element, the purpose of sanitation is to remove the element prior storing it on server, so that at least the data including the text wrapped by the `<mark>` element are stored on server.

The sanitation itself provides a configuration regarding the level of sanitation as can be seen in Example 10.1, "Strictness Configuration" [428]. Four different values are possible, where `LOOSE` is the default and the others are `STRICT`, `LEGACY` and `NONE`.

Strictness.strict, .loose, .legacy and .none

```
// Strictness also exported in CoreMediaStudioEssentials for lean
// dependency management.
import CoreMediaStudioEssentials, { Strictness } from
  "@coremedia/ckeditor5-coremedia-studio-essentials";

// Alternative import location:
// import { Strictness } from
//    "@coremedia/ckeditor5-coremedia-richtext";

ClassicEditor.create(domElement, {
  plugins: [
    // Typical dependency used to also integrate Rich Text Plugin.
    CoreMediaStudioEssentials,
    // ...
  ],
  "coremedia:richtext": {
    // The default strictness level.
    strictness: Strictness.LOOSE,
  },
});
```

Example 10.1. Strictness Configuration

Available Strictness levels:

Loose (Default)   By default, the Rich Text plugin ships with a strictness level *loose*, which is recommended for best robustness and enough for the given purpose. This will validate the data after data processing short before sending it to server. In contrast to simple validation, it tries to repair a possibly invalid state. This includes removing unknown elements or attributes as well as adding possibly missing required attributes.

Such invalid states should not occur in production, but may be a result in development processes, like when you enabled a plugin for a new HTML element, but did not adapt the data processing yet. The sanitation ensures, that the

data, despite this new element, can still be stored on server.

Legacy   This level is similar to the *CKEditor 4* behavior available until CMCC 11. It also validates the element structure and known and required attribute names, but it skips validating attribute values. The result may not represent valid *Core-Media Rich Text 1.0* and should be used with care. It just exists for best compatibility towards *CKEditor 4*.

One example attribute, which is handled different to *loose* mode is `dir`: Only values `ltr` and `rtl` are supported. *loose* mode will remove the `dir` attribute on any different value. *legacy* mode instead will keep it.

Strict   This is the highest sanitation level. It does not only check for valid DTD, but regarding attribute values, also checks, what attribute values are meant to be. Thus, attributes like `width` and `height` for `<img>` elements may, according to DTD, contain strings. In *strict* mode, they are enforced to be numeric.

None   This disables any sanitation. It is like taking away a safety net. In general, this is not recommended, as it requires perfectly shaped data processing rules, which never result in possibly invalid *CoreMedia Rich Text 1.0*. It may help setting this level while debugging (see Section 10.4, "Debugging CKEditor 5" [458]) and if you experience any performance issues during sanitation process.

10.2.11 Studio Essentials Plugin

This plugin is an aggregator for essential plugins when you use *CKEditor 5* in context of *CoreMedia Studio* to edit *CoreMedia Rich Text 1.0*. These plugins guarantee, that any valid *CoreMedia Rich Text 1.0* stored on server are loaded into *CKEditor 5* without corrupting the data, such as removing elements or attributes not handled by corresponding plugins.

The contained plugins are: *Package Information*

- Rich Text Plugin [427]
- General Rich Text Support Plugin [426]

The plugin is bundled the in npm package `@coremedia/ckeditor5-coremedia-studio-essentials`. For more details regarding this plugin consult CoreMedia CKEditor 5 Plugin: Studio Essentials.

10.3 CKEditor 5 Customization

In this section you will learn how to customize existing *CKEditor 5* instances and how to provide and use custom configurations of *CKEditor 5*. As described in Section 10.1.4, "Studio Integration: CKEditor 5 Configurations" [423] it is important to understand that each different flavor of *CKEditor 5* — adjusted toolbars, adapted plugins, for instance — requires an extra configuration of ClassicEditor. In Section 10.3.6, "Adapting Existing Configurations" [437] you will learn, where to locate and how to adapt these configurations to your needs. In Section 10.3.7, "Providing New Configurations" [438] you will see how to apply and use custom configurations and what requirements you should be aware of for smooth integration into *CoreMedia Studio*.

Prior to reading those sections, you should read Quick start for *CKEditor 5*.

10.3.1 Best Practice: ckeditorDefault.ts

It is recommended, to have a look into `ckeditorDefault.ts` in the package `@coremedia-blueprint/studio-client.ckeditor5` as *best practice* for providing additional custom configurations.

It contains examples such as how to localize *CKEditor 5* and its plugins at configuration time (see Section 10.3.2, "Localizing CKEditor 5" [430]) as well as recommended plugin configurations to provide the best compatibility to *CoreMedia Rich Text 1.0* edited with previous versions of *CoreMedia Content Cloud* (see, for example, section "Alignment Configuration" [435] and section "Image Styles Configuration" [436]).

To change the default, for example, to add plugins provided by *CKEditor 5*, just adapt `ckeditorDefault.ts` according to your needs. For a possible approach see Section 10.3.11, "Customizing ckeditorDefault.ts By Example" [449].

To start with a new configuration to be used in dedicated contexts, take `ckeditorDefault.ts` as boilerplate to copy and adapt and eventually register and use. For a possible approach see Section 10.3.7, "Providing New Configurations" [438].

10.3.2 Localizing CKEditor 5

If customizing *CKEditor 5*, you most likely also want to apply corresponding localized labels. The approach differs, depending on whether you want to apply

localization to custom *CKEditor 5* plugins or to apply localization as part of the configuration of *CKEditor 5* instances within *CoreMedia Studio*.

Set Locale at CKEditor 5 Instantiation:

CKEditor 5 has a configuration option `language`. Upon instantiation, you have to forward the locale to *CKEditor 5* as retrieved via `LocaleUtil.getLocale()` similar to the example in Example 10.2, "CKEditor 5 Instance Localization" [431].

```
import LocaleUtil from
  "@coremedia/studio-client.cap-base-models/locale/LocaleUtil";

const language = LocaleUtil.getLocale();

ClassicEditor.create(domElement, {
  language,
});
```

Example 10.2. CKEditor 5 Instance Localization

Plugin Localization

Localizing UI elements within custom plugins is best done via the `translation-service` as described in the Localization guide for *CKEditor 5*. This localization will then automatically apply the locale as configured at instantiation of the *CKEditor 5* instance.

Configuration Localization

As can be seen in the example included in section "Image Styles Configuration" [436], it may be required to apply localization while configuring *CKEditor 5* plugins, for example. The best practice to apply can be seen in `ckeditorDefault.ts` (see Section 10.3.1, "Best Practice: ckeditorDefault.ts" [430]). It boils down to using the `localization` object and `localize` function provided by `LocalizationUtils`. In Example 10.3, "Using LocalizationUtils" [431] you can see a rough sketch, on how to apply these localizations.

```
import { localization, localize } from
  "@coremedia-blueprint/studio-client.ckeditor5/lang/LocalizationUtils";
import LocaleUtil from
  "@coremedia/studio-client.cap-base-models/locale/LocaleUtil";

localization.add({
  "de": {
    "Type your text here...": "Text hier eingeben..."
  }
});

const language = LocaleUtil.getLocale();

ClassicEditor.create(domElement, {
  placeholder:
    localize("Type your text here...", language),
```

```
  language,
});
```

Example 10.3. Using LocalizationUtils

10.3.3 Custom Assets in CKEditor 5 Package

Custom assets, such as images, cannot be used out of the box in the `@core-media-blueprint/studio-client.ckeditor5` package. This section describes how to adjust the webpack configuration by showing the common usecase of using images in the editor.

> **NOTE**
>
> First, its important to understand why it is not sufficient to simply use a webpack file-loader: When added to the *CoreMedia Studio* bundle, the `@coremedia-blueprint/studio-client.ckeditor5` package is not processed as other Blueprint packages, containing a jangaroo config file. Static resources will simply not be included in the final bundle. Even though, using a file-loader would work just fine in this package, we would not be able to access the resources in a running *Studio*. Inlined assets can help to solve this issue by keeping the assets directly inside the JavaScript code.

Inlined Assets

You can enable inlined assets for common image formats in the webpack config as follows.

```
module: {
    rules: [
        {
          test: /\.(png|jpg|gif)$/,
          type: 'asset/inline'
        },
        ...
    ]
}
```

Example 10.4. Webpack config with inlined assets

With inlined assets enabled, images will be included into the generated JavaScript bundle and can then be displayed in the browser without having to be added as separate resources. Please have a look into the webpack documentation to learn more about Inlining Assets in Asset Modules.

You can now add images to the package and reference them in your CSS files.

```
.example {
  background: url("../img/image.png");
}
```

Example 10.5. Inlined asset usage in CSS files

Or use them in your TypeScript files.

```
import customImage from "../img/image.png";
```

Example 10.6. Inlined asset usage in TypeScript files

> **NOTE**
>
> Please note that importing images into TypeScript modules may require to de-
> clare a module for the imported file ending in a separate `.d.ts` file. Additionally
> `esModuleInterop` will have to be enabled in the `tsconfig.json`.

10.3.4 Embedded Media in CKEditor 5

CKEditor 5 can handle embedded media like images, which are backed by BLOB
properties stored in content. A first step to support this is integrating the cor-
responding plugin as described in Section 10.2.8, "Images Plugin" [426].

Next, you need to configure content types in *CoreMedia Studio*, which can be
referenced as embedded media by default, for example, on drag and drop from
library.

To do so, two possible approaches exist, where the latter one is recommended:

editorContext.registerRichTextEmbeddableType Function

This will register a content type name to be regarded (by default) as embeddable
media object. It requires to set a property to refer to for accessing the corres-
ponding BLOB data, thus, it should refer to a BLOB property.

ConfigureDocumentTypes Class

As part of the general configuration of content types within *CoreMedia Studio*
you may define a `richTextImageBlobProperty` which refers to a BLOB
property of the corresponding content type. Setting this automatically marks
the content type and its child types as being regarded as embeddable media.

Example 10.7, "Configuration of Embeddable Media in *CKEditor 5*" [434] shows a typical usage of this configuration option.

To apply this configuration, adapt the `BlueprintFormsStudioPlugin` in *CoreMedia Blueprint* accordingly.

```
new ConfigureDocumentTypes({
  names: "CMPicture,CMImage",
  richTextImageBlobProperty: "data",
  /* ... */
}),
```

Example 10.7. Configuration of Embeddable Media in CKEditor 5

Linking to Embeddable Media Contents

If you want to link to the contents defined as *embeddable*, you have to apply an alternative approach to plain drag and drop. Instead, use the Link feature, that is, drag and drop to the link dialog. Find details regarding the Link feature for contents at Section 10.2.9, "Link Plugins" [427].

10.3.5 Basic Configuration of CKEditor 5

At least for *CoreMedia Rich Text 1.0* editing support within *CKEditor 5* and compatibility to previous releases of *CoreMedia Content Cloud*, some configuration options are considered mandatory for any flavor of *CKEditor 5* configured in *CoreMedia Blueprint*. This section will tell you about those mandatory aspects.

In general, it is recommended using `ckeditorDefault.ts` as best-practice approach, which will also apply the following recommendations. For details see Section 10.3.1, "Best Practice: ckeditorDefault.ts" [430].

Autosave Plugin

One of the mandatory *CKEditor 5* plugins to add to any configuration is the Autosave plugin. The required configuration for the plugin is provided from parameter of type `CKEditorPluginConfig` (package `@coremedia/ studio-client.ckeditor-common`). It ensures, that entered data are eventually written back to the server.

Essential Plugins

Just as *CKEditor 5* recommends integrating their Essentials plugin, CoreMedia recommends applying the CoreMedia Essentials plugin as described in Section 10.2.11, "Studio Essentials Plugin" [429]. It ensures that any valid *CoreMedia Rich*

Text 1.0 can be loaded and edited within *CKEditor 5* without losing any previously set formatting options.

Link Configuration

The protocol used for external links in *CoreMedia Studio*, can be set directly in the editor configuration in the *CoreMedia Blueprint*. The default protocol is set to `'https://'` as shown in Example 10.8, "Link Configuration in ckeditorDefault.ts" [435].

```
link: {
  defaultProtocol: 'https://'
},
```

Example 10.8. Link Configuration in ckeditorDefault.ts

For details have a look at the corresponding feature documentation of *CKEditor 5*: Link.

Alignment Configuration

Text alignment in *CoreMedia Rich Text 1.0* is applied as *class* attribute values, which are:

- `align--left`
- `align--right`
- `align--center`
- `align--justify`

For best compatibility with previous versions of *CoreMedia Content Cloud* it is recommended to configure the same alignment options for the Text alignment feature that ships with *CKEditor 5*. To do so, apply the configuration to the editor instance as shown in Example 10.9, "Text Alignment Configuration" [435].

```
alignment: {
  options: [
    {
      name: "left",
      className: "align--left",
    },
    {
      name: "right",
      className: "align--right",
    },
    {
      name: "center",
      className: "align--center",
    },
    {
      name: "justify",
      className: "align--justify",
    },
  ],
},
```

Example 10.9. Text Alignment Configuration

Image Styles Configuration

Ever since, image (or media object) alignment in *CoreMedia Rich Text 1.0* is applied as class attribute value to `<img>` elements, which are:

- `float--left`
- `float--right`
- `float--none`

A fourth available option, which is an unset floating class, often was referred to as "Page default" to emphasize, that alignment behavior is inherited from parent element hierarchy.

For best compatibility with previous versions of *CoreMedia Content Cloud* it is recommended to configure the same alignment options for the Image styles feature that ships with *CKEditor 5*. To do so, apply the configuration to the editor instance as shown in Example 10.10, "Image Alignment Configuration" [436].

```
image: {
  styles: {
    options: [
      {
        name: 'float-left',
        icon: alignLeftIcon,
        title: localize('Left-aligned', language),
        className: 'float--left',
        modelElements: [ 'imageInline' ]
      },
      {
        name: 'float-right',
        icon: alignRightIcon,
        title: localize('Right-aligned', language),
        className: 'float--right',
        modelElements: [ 'imageInline' ]
      },
      {
        name: 'float-none',
        icon: withinTextIcon,
        title: localize('Within Text', language),
        className: 'float--none',
        modelElements: [ 'imageInline' ]
      },
      {
        name: 'inline',
        title: localize('Page default', language),
        icon: pageDefaultIcon,
      }
    ]
  },
  toolbar: [
    'imageStyle:float-left',
    'imageStyle:float-right',
    'imageStyle:float-none',
    'imageStyle:inline',
    // ... any more toolbar entries, you would like to add
  ]
},
```

Example 10.10. Image Alignment Configuration

For details regarding localizing the corresponding labels, have a look at Section 10.3.2, "Localizing CKEditor 5" [430].

10.3.6 Adapting Existing Configurations

CoreMedia Blueprint ships with predefined configurations of *CKEditor 5*, namely instances of ClassicEditor. These configurations are provided in *CoreMedia Blueprint* package `@coremedia-blueprint/studio-client.ckeditor5`.

In the following you will get a rough sketch, on how to adapt these configurations. For a detailed walkthrough and much more details, have a look at Section 10.3.11, "Customizing ckeditorDefault.ts By Example" [449].

> ### *CoreMedia Content Cloud* Upgrade Considerations
>
> In the following you will adapt the file `ckeditorDefault.ts` that ships with *CoreMedia Blueprint*. As usual, you the file might being updated when upgrading *CoreMedia Content Cloud*, which again may cause merge conflicts. Yet, you immediately take benefit from upgrades adding new features, for example.
>
> Having this, you may want to ensure to untangle your customizations a little from the existing configuration. Like, declaring extra toolbar entries in an extra variable, even imported from another file. Choose those options, whichever suit you best.

The *CKEditor 5* instance, which is almost used anywhere in *CoreMedia Blueprint* for editing rich text properties is configured in `ckeditorDefault.ts`, which is part of package `@coremedia-blueprint/studio-client.ckeditor5`.

Adapting this instance is nearly the same as described in CKEditor 5 documentation such as Quick start. Only remarkable difference: Instead of creating the *CKEditor 5* instance directly, a factory method is exposed that is used in rich text property fields to create the desired instance.

And of course, there are subtle requirements such as plugins to install, which are required for editing *CoreMedia Rich Text 1.0*. You will find an overview of these plugins at Section 10.2, "CKEditor 5 CoreMedia Plugins" [424].

Thus, to add any plugin, just extend the `plugins` configuration, possibly adapt the toolbar and, if required, provide some configuration for your added plugin.

10.3.7 Providing New Configurations

In Section 10.3.6, "Adapting Existing Configurations" [437] you learned how to customize configurations that ship with *CoreMedia Blueprint*. Another option to take is providing new configurations, which then can be used in custom content forms. In this section you will learn how to add and use them.

You will get a rough sketch here, how to add a new configuration. For a detailed walkthrough, have a look at Section 10.3.12, "Providing New CKEditor 5 Configuration By Example" [455]. In contrast to the example, which starts next to `cked itorDefault.ts` and propagating it up to the content forms, you proceed in reverse, starting with the goal you want to achieve:

In the end, you want to be able to reference your new custom editor by a string key in a configuration property `editorType` of `RichTextPropertyField` within one of the forms available in package `@coremedia-blueprint/ studio-client.main.blueprint-forms`, for example.

To declare this custom editor type, you need to adapt `init.ts` in `@core- media-blueprint/studio-client.main.ckeditor5-plugin` and register your new *CKEditor 5* instance at `editorTypeMap` as key-value pair. Key is a descriptor that is then used for reference when configuring the `editor- Type`. The value is a factory method of type `CreateCKEditorFunction` (package `@coremedia/studio-client.ckeditor-common`).

This factory method gets a reference to the DOM element, which should be replaced by the to-be-created *CKEditor 5* instance and a `CKEditorPlugin Config` (again, package `@coremedia/studio-client.ckeditor- common`). Applying this is crucial, as it, for example, provides the bridge from `Autosave` plugin of *CKEditor 5* to storing the data in the server. For details see Section 10.3.5, "Basic Configuration of CKEditor 5" [434].

You can freely choose where to define this factory method. For consistency, it is recommended adding this parallel to `ckeditorDefault.ts` in `@core- media-blueprint/studio-client.ckeditor5` as sketched in Section 10.3.12, "Providing New CKEditor 5 Configuration By Example" [455], where you may want to continue reading to get more details on adding a new configuration.

Respecting Feature Flags

In Section 10.3.8, "Using Configuration Feature Flags" [439] you will see, how to use *feature flags* as part of your configuration. Typically, flags are registered globally by *CoreMedia Studio* plugins, such as `CKEditor5StudioPlugin` similar to Example 10.11, "CoreMedia Rich Text 1.0 in *CoreMedia Studio*" [439].

```
rules: [
  Config(CKEditor5RichTextArea, {
    plugins: [
      Config(OnlyIf, {
        isAdministrator: true,
        then: Config(
          CKEditor5FeatureFlagPlugin,
          { featureFlags: ["administrative"] }
        ),
      }),
    ],
  }),
],
```

Example 10.11. CoreMedia Rich Text 1.0 in CoreMedia Studio

You may want to respect such globally available feature flags in your configuration, like, for example, the `administrative` flag, which is forwarded to all configurations via the example above.

For details regarding usage of feature flags and default flags that ship with *CoreMedia Blueprint* you may want to respect, see Section 10.3.8, "Using Configuration Feature Flags" [439].

10.3.8 Using Configuration Feature Flags

In Section 10.3.7, "Providing New Configurations" [438] you learned how to add new *CKEditor 5* configurations to *CoreMedia Blueprint*. While this is the recommended way for adding different configurations of *CKEditor 5*, there may be reasons for a more lightweight approach for only minor adaptations. This section will tell you about an alternative way.

We assume, that you have read and understand Section 10.3.6, "Adapting Existing Configurations" [437], because we are now going into another detail of adapting existing configurations. We call it **CKEditor 5 Feature Flags**.

What is a CKEditor 5 Feature Flag?

Different to fully-fledged *CKEditor 5* configurations, *feature flags* change a minor aspect of the configuration. In *CoreMedia Blueprint*, for example, `ckeditorDefault.ts` ships with a minor configuration adaptation for administrators. It enables the so-called Source editing feature. The feature provides an additional toolbar button to administrators in *CKEditor 5*, that switches to the XML representation of *CoreMedia Rich Text 1.0* as can be seen in Figure 10.2, "CKEditor 5 Source Editing Feature" [440].

Figure 10.2. CKEditor 5 Source Editing Feature

This minor configuration adaptation does not require an extra configuration to be provided nor to register it at some place, as it would have been required for adding new configurations. Having this, it may be tempting to provide any adaptation as feature flag. There are reasons, though, to choose either the one or the other. Find more details later in section "When to prefer CKEditor 5 Feature Flags?" [442].

Adding Feature Flags to Configurations

Just as stated in Section 10.3.1, "Best Practice: ckeditorDefault.ts" [430], `ckeditorDefault.ts` is the best practice approach, you may want to have a look at when learning about feature flags. It contains a configuration similar to the one shown in Example 10.12, "Feature Flag in ckeditorDefault.ts" [440].

```
export const administrative = "administrative";

export const createDefaultCKEditor: CreateCKEditorFunction = (
  domElement:(string | HTMLElement),
  pluginConfig: CKEditorPluginConfig):
  Promise<ClassicEditor> => {

    const defaultToolbarItems = [ /* ... */ ];

    if (pluginConfig.featureFlags?.includes(administrative)) {
      defaultToolbarItems.push("|", "sourceEditing");
    }

    /* ... */

  };
```

Example 10.12. Feature Flag in ckeditorDefault.ts

The example shows, how a feature flag called `administrative` is validated, if this is included in the feature flags handed over to the factory method within

CKEditorPluginConfig. If it is, the default toolbar is extended by an additional entry for the source editing button.

It may be obvious, that similarly you could modify almost anything within the *CKEditor 5* instance creation. You just need to define corresponding identifiers for feature flags, which can then be handed over to the factory method. This is described in section "Using Feature Flagged CKEditor 5 Instances" [441]. Note, though, that some pitfalls exist, and that it is generally not advisable removing or adding plugins via such flags. You will get to know more details in section "Possible Pitfalls Using CKEditor 5 Feature Flag?" [443]. While the limitation does not necessarily apply to the source editing feature, we stick to this rule in ckeditorDefault.ts also for this plugin.

Using Feature Flagged CKEditor 5 Instances

Feature flags are typically registered globally, thus, forwarded to all factory methods for *CKEditor 5*. One example is the administrative flag. This is propagated via CKEditor5StudioPlugin similar to the configuration as shown in Example 10.13, "CoreMedia Rich Text 1.0 in *CoreMedia Studio*" [441].

```
rules: [
  Config(CKEditor5RichTextArea, {
    plugins: [
      Config(OnlyIf, {
        isAdministrator: true,
        then: Config(
          CKEditor5FeatureFlagPlugin,
          { featureFlags: ["administrative"] }
        ),
      }),
    ],
  }),
],
```

Example 10.13. CoreMedia Rich Text 1.0 in CoreMedia Studio

The configuration uses the OnlyIf plugin, to conditionally apply the feature flag administrative to all *CKEditor 5* factory methods via the CKEditor5FeatureFlagPlugin. Thus, also your custom factory methods may now add administrative behaviors to *CKEditor 5*.

Applying feature flags based on editor type: Using the condition property of the OnlyIf plugin, and the component handed over to the corresponding predicate, you could conditionally apply feature flags based on the editorType configured for the given component. As the editorType directly maps to a corresponding factory method you may select only those factory methods, which are supporting the corresponding feature flag.

Predefined Feature Flags

CoreMedia Blueprint ships with one predefined feature flag forwarded to all *CKEditor 5* factory methods. It is the administrative flag. It is set as soon

as an administrative user is logged in to *CoreMedia Studio*. Thus, you may want to respect this flag in all your provided factory methods.

When to prefer CKEditor 5 Feature Flags?

As described in section "What is a CKEditor 5 Feature Flag?" [439] the concept of feature flags overlaps with providing additional factory methods for *CKEditor 5*. While you are free to choose any option depending on which suits you better, this section provides some ideas on when to prefer which option.

The ideas can be summarized as: If you want contextual minor adaptations of *CKEditor 5*, like based on the editor's role, you may be better off using feature flags. If you want different configurations based on rich text property to edit, you may be better off using custom factory methods.

Via feature flag: Source Editing Feature: Let us first have a look at the source editing feature, where *CoreMedia Blueprint* ships with a feature flag, to enable this only for administrators. In this case, the adaptation is minimal regarding the behavior of *CKEditor 5*: It just adds a toolbar button. All other configuration should stay the same and duplicating it as extra factory method just for administrators raises the risk of having diverged configurations in the end.

Via feature flag: Disallow creating links for restricted users: Assume, based on user roles, you want to limit some editors, so that they cannot create links (either external or content links). All other editing options should stay the same. Similar to the source editing feature, you can, as first step, just remove the corresponding toolbar entry in configuration based on a feature flag. Note though, that editors are still able using keyboard shortcuts or edit existing links. Thus, the recommended additional action as part of the factory method is to disable all commands related to link editing. You should not remove the corresponding plugins, though, as this may corrupt your data or provide inconvenience when editing texts containing links, like not being able to click on content links with appropriate action. Find details for this in section "Possible Pitfalls Using CKEditor 5 Feature Flag?" [443].

Via extra configuration: Disallow links in teaser texts: Similar to the above, assume, you want to disallow creating links in teaser text properties. While it may be tempting just reusing the feature flagged configuration from above, it is recommended to instead provide an extra configuration, thus factory method. In this case you may just want to skip adding the link and corresponding plugins like the link target plugin. The benefit of this is, that there is no need to take care of the commands or even commands added later via a *CKEditor 5* upgrade.

Just to complete this idea of removing links in teaser texts, there is something to take care of regarding copy and paste: If you copy and paste links, for example, from article text to teaser text, they will not be removed by default. You may want to remove the *General Rich Text Support*, too. For details, see Section 10.2.7, "General Rich Text Support Plugin" [426].

Possible Pitfalls Using CKEditor 5 Feature Flag?

Short Summary: In the following you will get some detailed hints regarding possible pitfalls using feature flags. To summarize this in advance: If you only adapt toolbar configurations, you are fine. If you want some more fine-grained tuning like removing or adding plugins, or like adding or removing options to choose from as editor, you should carefully read and understand the following.

In section "Adding Feature Flags to Configurations" [440] you learned, how to use feature flags within the *CKEditor 5* factory methods. We have shown how to enable or disable the source editing feature based on such a flag. We also stated, that we may have also removed (or added) the corresponding plugin from the `plugins` configuration part, and told that in general you should not do that. We are now going to explain why, so that in the end, you may also know why it would be possible to apply an exception here for the source editing feature.

Section 10.2.11, "Studio Essentials Plugin" [429] roughly sketches the reason: By default, any data read from server that is not supported to be created by corresponding commands in *CKEditor 5* may be removed by *CKEditor 5* as it is considered *unknown*. Without knowing more details this would mean: If you disable a plugin such as for bold text, all `<strong>` elements will be removed automatically when loaded into *CKEditor 5*. Having this, not to corrupt data provided by others, you may (safely) remove the toolbar button for bold text, but you should keep the corresponding plugin.

Going a little more into details, there is a safety net actually, described in Section 10.2.11, "Studio Essentials Plugin" [429], which is called *General Rich Text Support*. This will prevent such valid *CoreMedia Rich Text 1.0* to be removed automatically. If this plugin is missing or misconfigured, though, you may experience such a data loss.

You may now know why you could have removed the Source Editing plugin in non-administrative mode: It does not change allowed elements or attributes. Thus, it is a pure user interface feature.

There are more possible pitfalls, except from removing or adding plugins by feature flags. We will describe another possible pitfall next. Prior to that, or to skip the next pitfall description, just ensure, that you carefully check for possible side-effects if removing or adding configuration.

One other possible pitfall, just as last example, is the text alignment feature. As described in section "Alignment Configuration" [435] it is configured having four possible values, including, for example, `align--left` and `align--right`. If, by feature flag, you remove any of these options, it will cause editors not to toggle, but to add additional classes.

As example, let us assume, you removed `align--right` for restricted editors, which are not allowed to set right alignment option. Now data is read from

server containing `align--right` set by more privileged editors. If the restrict-ted editors now set the alignment to the available option `align--left`, the text alignment feature will not know about toggling the other applied class. You will end up with an element having both classes set: `align--left` and `align--right`.

So, anytime you use feature flags, ensure that you carefully review, what this means to existing data or to data created by those having the feature turned on and those having the feature turned off.

Notes on Source Editing Feature

There are some caveats, why source editing should not be enabled for casual editors. Summarized in short, when editing raw *CoreMedia Rich Text 1.0* you need to take care providing valid *CoreMedia Rich Text 1.0*, as otherwise data may be lost. If in doubt, ensure to check in contents prior to editing them. Also *Undo* may be an option to get back to a previously valid result. See Section 3.7, "Editing Rich Text Source Code" in *Studio User Manual* for details on rich text source editing.

10.3.9 Creating Custom Plugins

Besides adapting existing configurations (Section 10.3.6, "Adapting Existing Configurations" [437]) or adding new configurations (Section 10.3.7, "Providing New Configurations" [438]), providing your own custom plugins to *CKEditor 5* is an important task.

The guide Creating a basic plugin for *CKEditor 5* will tell you first basic details, which you should read and understand before you continue reading.

Workspace Setup

It is recommended developing custom plugins for *CKEditor 5* in an extra work-space, independent of *CoreMedia Blueprint* just as CoreMedia does for the plugins within the repository CoreMedia CKEditor 5 Plugins. As *CKEditor 5* is only loosely coupled to *CoreMedia Studio* (see Section 10.1.3, "Studio Integration: Service Agent" [422]) there is no need to integrate with Ext JS tooling or UI. Having this, you can easily use the tool-chain available for *CKEditor 5* and follow correspond-ing guides like in Creating a basic plugin.

Studio Integration

Integrating these plugins to *CoreMedia Studio* most often just requires to register them as described in Section 10.3.6, "Adapting Existing Configurations" [437] and Section 10.3.7, "Providing New Configurations" [438].

Some more effort is required for communication from your *CKEditor 5* plugin to *CoreMedia Studio* and vice versa. It is recommended using the `serviceAgent`

as described in Section 10.1.3, "Studio Integration: Service Agent" [422] to set up corresponding services.

10.3.10 Link Editing

In this section you will get some more details on customizing *CKEditor 5* regarding required customizations for link editing behavior based on the Link feature of *CKEditor 5*. For general information, like, how to configure `https://` as default protocol for links, please have a look at the corresponding feature documentation.

This section is recommended to be read, if you are about to implement support for editing custom attributes, that are bound to links.

The section "Custom Link Attributes" [445] describes an assistive plugin `LinkAttributes` that helps to manage link-related attributes. The plugin is bundled in npm package `@coremedia/ckeditor5-link-common`. For more details regarding the plugins consult CoreMedia CKEditor 5 Plugin: LinkAttributes.

The section "Handle Artificial xlink:role" [447] describes how to deal with so-called *artificial* `xlink:role` attributes. Short: How to use `xlink:role` to store information, that shall not be represented within the `target` attribute within *CKEditor 5* model and view layers.

Default Plugins

The default *CKEditor 5* configuration `ckeditorDefault.ts` (`@coremedia-blueprint/studio-client.ckeditor5`) contains these main plugins provided by CoreMedia:

- `ContentLinks`
- `LinkTarget`

They are responsible to enable linking to content items as well as to provide support for `target` attribute editing. For details see Section 10.2.9, "Link Plugins" [427].

Custom Link Attributes

Along with the plugins listed in section "Default Plugins" [445] `ckeditorDefault.ts` also refers to a plugin called `LinkAttributes`. This plugin may be important, when it is about adding support for additional attributes bound to links.

This plugin can be summarized as to integrate into typical link attribute editing and cleanup behavior as implemented by *CKEditor 5* Link feature. This includes the so-called *two-step-caret-movement* (see TwoStepCaretMovement) as well as removing all link-related attributes on removing links.

> ### Prefer CKEditor 5 API, if available
>
> If *CKSource* provides an API for *CKEditor 5* to register such link-related attributes, it is preferred to use that one, as it is assumed to cover more use-cases.

The plugin provides two configuration layers:

- *CKEditor 5* instance configuration
- configuration API suitable for use in custom plugins

CKEditor 5 Instance Configuration: `ckeditorDefault.ts` ships with a configuration for attributes, that are part of *CoreMedia Rich Text 1.0*, but not yet covered by corresponding editing features.

```
import { LinkAttributesConfig } from "@coremedia/ckeditor5-link-common";

export const linkAttributesConfig: LinkAttributesConfig = {
  attributes: [
    {
      view: "title",
      model: "linkTitle",
    },
    {
      view: "data-xlink-actuate",
      model: "linkActuate",
    },
  ],
};
```

Example 10.14. LinkAttributes Configuration

Example 10.14, "LinkAttributes Configuration" [446] shows a configuration for attributes `title` and `data-xlink-actuate`, that are a result of the default data-processing for *CoreMedia Rich Text 1.0* (see Section 10.2.10, "Rich Text Plugin" [427]) of attributes `xlink:title` and `xlink:actuate`.

```
import { Link } from 'ckeditor5';
import { LinkAttributes, linkAttributesConfig } from
  "@coremedia/ckeditor5-link-common";

return ClassicEditor.create(domElement, {
  /* ... */
  plugins: [
    /* ... */
    Link,
    LinkAttributes,
    /* ... */
  ],
  link: {
    defaultProtocol: 'https://',
    ...linkAttributesConfig,
  },
});
```

```
/* ... */
```

Example 10.15. LinkAttributes Configuration Usage

Example 10.15, "LinkAttributes Configuration Usage" [446] demonstrates the integration into `ckeditorDefault.ts` merging the configuration with the configuration of the *CKEditor 5* Link feature. Plugin `LinkAttributes` will parse this configuration and trigger corresponding configuration for two-step-caret-movement and link-attribute cleanup.

```
import { Plugin } from "ckeditor5";
import { getLinkAttributes, LinkAttributes } from
  "@coremedia/ckeditor5-link-common";

export class MyLinkTitleEditing extends Plugin {
  static readonly pluginName: string = "MyLinkTitleEditing";

  static readonly requires = [LinkAttributes, /* ... */];

  init(): void {
    const { editor } = this;

    getLinkAttributes(editor)?
      .registerAttribute({ view: "title", model: "linkTitle" });
  }
}
```

Example 10.16. LinkAttributes at Plugin Initialization

Example 10.16, "LinkAttributes at Plugin Initialization" [447] is a typical usage from within plugins. Here, a plugin provides capabilities for link title editing. It is recommended to move the configuration for this attribute to the plugin initialization then.

Thus, if you have any custom attribute to edit, that is only valid in context of links, we recommend using `LinkAttributes` to register this attribute, or, as alternative, carefully review and adapt behaviors as can be found in sources of the *CKEditor 5* Link plugin.

Handle Artificial xlink:role

In this section you will learn how to deal with so-called *artificial* `xlink:role` attributes, that should not be represented as `target` in the *CKEditor 5* model and view layers. You will learn, how to override this behavior to store the value in any other attribute.

> ### Artificial xlink:role Attribute
>
> We call an `xlink:role` attribute *artificial* when it is non-empty for any other value of `xlink:show` than `"other"`.
>
> It is *artificial* in that sense that the typical transformation applied to *CoreMedia Rich Text 1.0* will use the value of `xlink:role` to render the HTML `target` when `xlink:show` is set to `"other"`. In other cases the value of `xlink:role` is typically ignored, its use is not clearly defined. And we call this usage *artificial*.

To deal with *artificial* `xlink:role` states, you may add a data-processing rule with at least `"high"` priority, that processes the `xlink:role` attribute before the default processing. For `toData` processing this is best done in `prepare` step and for `toView` processing in `imported` step. For details find corresponding references in Section 10.2.10, "Rich Text Plugin" [427].

Convenience API: For convenience, if not even recommended, as we ensure correct processing order, you may use a factory method for a data-processing rule called `mapArtificialXLinkRole` that ships with `@coremedia/ckeditor5-coremedia-richtext`.

A simple example, assuming usage from within a custom plugin, is shown in Example 10.17, "Example Usage of mapArtificialXLinkRole" [448]. If adding the generated rule, exactly as shown in the example, it will activate a mode that can be described as: *Remove any artificial `xlink:role` attribute*. In any other case than having `xlink:show` set to `"other"`, it will just strip `xlink:role` from the data and consequently will not add when transforming the view back to the data.

```
import { Plugin } from "ckeditor5";
import { mapArtificialXLinkRole }
  from "@coremedia/ckeditor5-coremedia-richtext";

export class ArtificialRoleToClass extends Plugin {
  static readonly pluginName: string = "ArtificialRoleToClass";

  init(): void {
    const { editor: {data: { processor } } } = this;

    if (isRichTextDataProcessor(processor)) {
      processor.addRules([
        mapArtificialXLinkRole(/* ... config ... */);
      ]);
    }
  }
}
```

Example 10.17. Example Usage of mapArtificialXLinkRole

If you want to store it in some other attribute instead, your configuration of `mapArtificialXLinkRole` may be similar as shown in Example 10.18, "Example Configuration of mapArtificialXLinkRole" [449]. Here, the artificial `xlink:role` is stored as additional `class` attribute value within CKEditor 5 model and view layers and later restored from `class` attribute in `toData` processing.

```
{
  toView: (element, role) => {
    const sanitizedRole = role.replaceAll(/\s/g, "_");
    element.classList.add(`role_${sanitizedRole}`);
  },
  toData: (element) => {
    const matcher = /^role_(\S*)$/;
    const matchedClasses: string[] = [];
    let role: string | undefined;
    for (const cls of element.classList) {
      const match = cls.match(matcher);
      if (match) {
        const [matchedCls, matchedRole] = match;
        role = matchedRole;
        matchedClasses.push(matchedCls);
      }
    }
    // Cleanup any matched classes and possibly left-over `class=""`.
    element.classList.remove(...matchedClasses);
    if (element.classList.length === 0) {
      element.removeAttribute("class");
    }
    return role;
  },
}
```

Example 10.18. Example Configuration of mapArtificialXLinkRole

If instead, you store the state in some different attribute (e.g., within the anchor's `download` attribute), ensure to register the corresponding attribute as belonging to the link. For details, see section "Custom Link Attributes" [445].

10.3.11 Customizing ckeditorDefault.ts By Example

While in Section 10.3.6, "Adapting Existing Configurations" [437] you only got a rough sketch of how to make adjustments to existing configurations, you will now go step-by-step through a possible use case as an example: You will add the Highlight plugin available for *CKEditor 5* to the default configuration that ships with *CoreMedia Blueprint*. To do so, you need to:

- Add the Plugin's Dependencies [450]
- Add the Plugin [451]
- Configure Data-Processing [451]

Rich Text Editing | Customizing ckeditorDefault.ts By Example

- Adapt CSS Styling [452]
- Adapt Delivery [452]

For older releases of *CoreMedia CKEditor 5 Plugins* you will find a section at the end called Compatibility [455].

See Also

In Section 10.3.12, "Providing New CKEditor 5 Configuration By Example" [455] you will see, how to apply this customization only to one single dedicated content property.

Add the Plugin's Dependencies

The Highlight plugin available for *CKEditor 5* is part of the `@ckeditor/ckeditor5-highlight` package. Thus, within `@coremedia-blueprint/studio-client.ckeditor5` we add the corresponding dependency:

```
pnpm add --save-dev "@ckeditor/ckeditor5-highlight@41.1.0"
```

Use Fixed Versions for CKEditor 5

In contrast to other dependencies, it is important always only using fixed versions like `@41.1.0` for *CKEditor 5* dependencies. *CKEditor 5* blocks any use of mixed package versions at runtime.

You will notice that when opening a content form in *CoreMedia Studio* by a missing *CKEditor 5* instance and a console error of an uncaught `CKEditorError` with error code ckeditor-duplicated-modules.

TypeScript Typings

Starting from v35.0.0 first *CKEditor 5* packages were developed in TypeScript. Prior to that, typings were only available via DefinitelyTyped. As long as corresponding packages are not migrated to TypeScript, you may want to install the corresponding typings from *DefinitelyTyped* as `@types/ckeditor__ckeditor5-highlight` in this case.

Note, that typings at *DefinitelyTyped* are not always up-to-date. In these cases one possible option is ignoring these errors with TypeScript `@ts-expect-error` annotation. This will also automatically tell you at compile time, when typings got updated accordingly.

Add the Plugin

Next, you add the plugin to `ckeditorDefault.ts` in package `@coremedia-blueprint/studio-client.ckeditor5` and extend the toolbar accordingly:

```
import { Highlight } from "ckeditor5";

/* ... */

const defaultToolbarItems = [
  /* ... */
  "superscript",
  "highlight",
  "removeFormat",
  /* ... */
];

return ClassicEditor.create(domElement, {
  /* ... */
  plugins: [
    /* ... */
    Highlight,
    /* ... */
  ],
});

/* ... */
```

Of course, you may also adapt the configuration of the plugin within `ckeditorDefault.ts`. For this example we stick to the default configuration (which is applying no extra configuration).

Configure Data-Processing

As documented for the Highlight plugin, it uses inline `<mark>` element. As described in Section 10.1.2, "Design Principle: HTML First" [420] you will not change that, neither for editing view nor for data view (see Section 10.1.1, "Glance at CKEditor 5 Architecture" [417] to get to know about the various layers). Instead, you have to adapt your data-processing, as *CoreMedia Rich Text 1.0* does not support the `<mark>` element.

Extend the data-processing as follows:

```
import { replaceElementByElementAndClass } from
  "@coremedia/ckeditor5-coremedia-richtext";

/* ... */

return ClassicEditor.create(domElement, {
  /* ... */
  plugins: [
    /* ... */
    Highlight,
    /* ... */
  ],
  "coremedia:richtext": {
    rules: [
      // Highlight plugin support.
      replaceElementByElementAndClass({
        viewLocalName: "mark",
        dataLocalName: "span",
        // "mark" is the default here, derived from `viewLocalName`. Thus,
        // we may skip it here.
```

```
        dataReservedClass: "mark",
      }),
    ],
  },
});
/* ... */
```

As you see it uses a method `replaceElementByElementAndClass` available from `@coremedia/ckeditor5-coremedia-richtext`. This method covers a typical use case when it comes to identifying a representation of an HTML element in *CoreMedia Rich Text 1.0*: It maps the `<mark>` element to a corresponding representation as `<span>` with identifying class attribute value and vice versa.

Thus, if you highlighted some text with green background:

```
<mark class="marker-green">Highlight me</mark>
```

It will be transformed in *CoreMedia Rich Text 1.0* data via data-processing to:

```
<span class="mark marker-green">Highlight me</span>
```

Note, that `replaceElementByElementAndClass` is only a shorthand function for a slightly more complex, but also more versatile, mapping you could apply. To get to know more about possible configuration options of the data-processing for *CoreMedia Rich Text 1.0*, have a look at Section 10.2.10, "Rich Text Plugin" [427].

Adapt CSS Styling

Luckily CSS styles provided by *CKEditor 5* and its plugins also directly apply to *CKEditor 5* instances in *CoreMedia Studio*. Thus, to support, for example, the class `marker-green` as sketched in the previous step, there is nothing which needs to be done, to make the default style work in *CoreMedia Studio*. But perhaps, the Green is not Green enough? Just apply this SCSS snippet to your *CoreMedia Studio* styling:

```
.ck-content {

  .marker-green {
    background-color: #00ff00;
  }

  /* ... */
}
```

For details, have a look at Section 9.17.1, "Blueprint Studio Theme" [211].

Adapt Delivery

Of course, adapting *CoreMedia Studio* and the *CKEditor 5* configuration is only half of the way to take. Depending on the technology you use for delivery, you now have to apply most likely the same mapping from:

```
<span class="mark marker-green">Highlight me</span>
```

to:

```
<mark class="marker-green">Highlight me</mark>
```

unless you are satisfied applying just some additional CSS-rules for the `<span>` element.

Thus, you may want to have a look at one of these manuals:

- **Section 4.3.4.1, "Rendering Markup" in** *Content Application Developer Manual*

  More specifically, in *CoreMedia Blueprint*, you may want to adapt `Blueprin tRichtextFiltersConfiguration` providing a corresponding instance of `XMLFilter`. For this use-case you may extend the configuration of bean `reservedClassToElementFilter` as shown in Example 10.19, "Adapting Bean reservedClassToElementFilter" [453].

```
@Bean
ReservedClassToElementFilter reservedClassToElementFilter() {
  return new ReservedClassToElementFilter(List.of(
    // <span class="mark"> -> <mark>
    ReservedClassToElementConfig.of("span", "mark", "mark"),
    /* … */
  ));
}
```

Example 10.19. Adapting Bean reservedClassToElementFilter

- **Frontend Developer Manual**

  More specifically, in *CoreMedia Blueprint*, you may want to adapt `vari ables/_coremedia-richtext-1.0.scss` and `partials/_core media-richtext-1.0.scss` in `@coremedia/brick-utils` as shown in Example 10.20, "Adapting variables/_coremedia-richtext-1.0.scss" [453] and Example 10.21, "Adapting partials/_coremedia-richtext-1.0.scss" [453].

```
/* Same Colors as CKEditor 5 Highlight Plugin by default */
$cm-richtext-mark-marker-yellow: hsl(60, 97%, 73%) !default;
$cm-richtext-mark-marker-green: hsl(120, 93%, 68%) !default;
$cm-richtext-mark-marker-pink: hsl(345, 96%, 73%) !default;
$cm-richtext-mark-marker-blue: hsl(201, 97%, 72%) !default;
$cm-richtext-mark-pen-red: hsl(0, 85%, 49%) !default;
$cm-richtext-mark-pen-green: hsl(112, 100%, 27%) !default;
```

Example 10.20. Adapting variables/_coremedia-richtext-1.0.scss

```
mark {
  &.marker-yellow {
```

```
    background-color: $cm-richtext-mark-marker-yellow;
    color: inherit;
  }

  &.marker-green {
    background-color: $cm-richtext-mark-marker-green;
    color: inherit;
  }

  &.marker-pink {
    background-color: $cm-richtext-mark-marker-pink;
    color: inherit;
  }

  &.marker-blue {
    background-color: $cm-richtext-mark-marker-blue;
    color: inherit;
  }

  &.pen-red {
    background-color: transparent;
    color: $cm-richtext-mark-pen-red;
  }

  &.pen-green {
    background-color: transparent;
    color: $cm-richtext-mark-pen-green;
  }
}
```

Example 10.21. Adapting partials/_coremedia-richtext-1.0.scss

- **Chapter 5,** in *Headless Server Manual*

  More specifically, in *CoreMedia Blueprint*, you may want to adapt the existing
  default view configuration in headless-server-base, first by declaring
  the reserved class mark as shown in Example 10.22, "Adapting richtext/in-
  cludes/classes.yml" [454] and then by defining the mapping as shown in Ex-
  ample 10.23, "Adapting richtext/default.yml" [454].

```
classes:
  - &inline_styles !!java.util.ArrayList
    # ...
    - &mark_style      mark
```

Example 10.22. Adapting richtext/includes/classes.yml

```
handlerSets:
  text: &text_handlers
    - !Handler
      eventMatcher: !Matcher {qname: *span, classes: *inline_styles}
      outputHandler:
        !ElementWriter
        # ...
        elementTransformer:
          !ElementFromClass
          mapping:
            # ...
            *mark_style: mark
```

Example 10.23. Adapting richtext/default.yml

Compatibility

Configuration of `coremedia:richtext` provides compatibility modes for rule parsing. The default compatibility mode is `latest`. Find below enumerated compatibility modes and the corresponding configuration.

v10 Version 11 of *CoreMedia CKEditor 5 Plugins* introduced a new way to configure data-processing. While originally using object-style definitions in version 10, with rather limited DOM manipulation support, version 11 comes with array-style definition and rich DOM manipulation support.

You may switch back to the old behavior by setting `compatibility` to `v10` in configuration. The example shown in section "Configure Data-Processing" [451] will look like given in the following example then:

```
import { replaceByElementAndClassBackAndForth } from

"@coremedia/ckeditor5-coremedia-richtext/src/compatibility/v10/rules/ReplaceBy";

/* ... */

return ClassicEditor.create(domElement, {
  /* ... */
  plugins: [
    /* ... */
    Highlight,
    /* ... */
  ],
  "coremedia:richtext": {
    compatibility: "v10",
    rules: {
      elements: {
        // Highlight Plugin Support
        mark: replaceByElementAndClassBackAndForth("mark", "span",
"mark"),
      },
    },
  },
});

/* ... */
```

10.3.12 Providing New CKEditor 5 Configuration By Example

In this example, you will learn, how to introduce a new configuration and use it for an extended editing feature for a given rich text property.

The example is based on Section 10.3.11, "Customizing ckeditorDefault.ts By Example" [449]: Here, you want to apply the same plugin but only make it available to a dedicated content property.

Proceed as follows:

1. 1. Copy And Adapt the Configuration [456]

1. Copy And Adapt the Configuration

You start with copying `ckeditorDefault.ts` to `ckeditorCustom.ts`. For consistency, it is recommended to also adapt the name of the factory method `createDefaultCKEditor`. Rename it to `createCustomCKEditor`.

After that, you apply the same adaptations to the *CKEditor 5* instance as described in Section 10.3.11, "Customizing ckeditorDefault.ts By Example" [449].

So, now you have a custom *CKEditor 5* configuration with the Highlight plugin enabled.

2. Propagate the Configuration

At the same path as `ckeditorDefault.ts` you will find a file `ckeditor.ts`. We expose the factory method in here as shown in Example 10.24, "Adapting ckeditor.ts" [456].

```
/* ... */
export { createDefaultCKEditor } from "./ckeditorDefault";
export { createCustomCKEditor } from "./ckeditorCustom";
/* ... */
```

Example 10.24. Adapting ckeditor.ts

Now, for referencing the new *CKEditor 5* instance within Ext JS components such as rich text property fields, you need to expose it in `init.ts` of `@coremedia-blueprint/studio-client.main.ckeditor5-plugin` as can be seen in Example 10.25, "Adapting init.ts" [456].

```
import { createDefaultCKEditor, createCustomCKEditor, /* ... */ } from
  "@coremedia-blueprint/studio-client.ckeditor5/";

/* ... */

richTextAreaRegistry.registerRichTextArea(
  RichTextAreaConstants.CKE5_EDITOR,
  Config(CKEditor5RichTextArea, {
    editorTypeMap: new Map([
      [
        CKEditorTypes.DEFAULT_EDITOR_TYPE,
        createDefaultCKEditor
      ],
      [
        "custom",
        createCustomCKEditor
      ],
      /* ... */
    ]),
```

```
  })
);
```

Example 10.25. Adapting init.ts

3. Use the Configuration

You want to apply the Highlight feature when editing the `detailText` of content-type `CMTeasable`. Thus, you need to adapt `DetailsDocumentForm` of package `@coremedia-blueprint/studio-client.main.blueprint-forms`.

The change is as simple as the previous steps. In the `items` of the `Details DocumentForm` you will find a `RichTextPropertyField`. As this does not explicitly set an `editorType` it uses the default *CKEditor 5* instance instead.

Now simply add a reference to your new `editorType` as registered in `init.ts` mentioned above. Thus, the result will be similar as shown in Example 10.26, "Adapting DetailsDocumentForm" [457].

```
/* ... */
class DetailsDocumentForm extends PropertyFieldGroup {
  /* ... */

  constructor(config: Config<DetailsDocumentForm> = null) {
    super(ConfigUtils.apply(Config(DetailsDocumentForm, {
      /* ... */
      items: [
        /* ... */
        Config(RichTextPropertyField, {
          bindTo: config.bindTo,
          itemId: "detailText",
          propertyName: "detailText",
          editorType: "custom", // Enable Highlight Plugin
          initialHeight: 200,
        }),
      ],
      /* ... */
    }), config));
  }
}

/* ... */
```

Example 10.26. Adapting DetailsDocumentForm

Now you are ready applying highlights to your text.

10.4 Debugging CKEditor 5

When adapting *CKEditor 5* as described in Section 10.3, "CKEditor 5 Customization" [430] or for contacting the CoreMedia support team, it may sometimes be important to understand the details of interaction between *CoreMedia Studio* and *CKEditor 5*. In this section, you will get a short glimpse on how to do that.

The (or one) key to success is the `ckdebug` hash-parameter you may add to the *CoreMedia Studio* URL, like `http://localhost:3000/#ckdebug=verbose`, where the parameter value denotes the verbosity of the output. Without a parameter value, it defaults to log level `INFO`.

ckdebug

Regarding data-processing of *CoreMedia Rich Text 1.0* provided by Rich Text Plugin [427] the most important keywords to look for in the browser console are `HtmlDomConverter` and `CoreMediaRichText`.

While `HtmlDomConverter` will tell step by step, which transformation steps have been applied either from *CoreMedia Rich Text 1.0* to *CKEditor 5* data view or vice versa, `CoreMediaRichText` summarizes in- and output, provides details on the sanitation process (responsible for ensuring valid CoreMedia Rich Text 1.0 even on corrupted data-processing) and details on the configured rule sets.

Rules Overview

Data-processing rules are provided as plain array of mapping rules, similar to firewall or mail-filter rules. If you want to validate, that your rule is active and processed in expected order (if order matters), you may find debugging output as sketched in Example 10.27, "CoreMediaRichText: Rules Overview" [458].

```
CoreMediaRichText: 16 rule configurations added.
toData Rules (16):
    toData-transform-xlink-attributes
    replace-i-by-em
    replace-b-by-strong
    replace-s-by-span.strike
    replace-del-by-span.strike
    replace-strike-by-span.strike
    replace-u-by-span.underline
    ...
toView Rules (12):
    toView-transform-xlink-attributes
    replace-em-by-i
    replace-span.strike-by-s
    replace-span.underline-by-u
    replace-span.code-by-code
    replace-p.p--div-by-div
    replace-p.p--heading-#-by-headings
...
```

Example 10.27. CoreMediaRichText: Rules Overview

Note, that especially for debugging purpose, it is recommended adding a descriptive ID to the rule. You will see a generic ID otherwise in output. Utility methods for mapping that ship with *CoreMedia CKEditor 5 Plugins* generate these IDs automatically.

Rules Execution

At first glance, if you see something missing in the editing view or later in the data stored on the server, you may want to compare the corresponding values. See the corresponding example outputs in Example 10.28, "CoreMediaRichText: From Data to Data View" [459] and Example 10.29, "CoreMediaRichText: From Data View to Data" [459].

```
CoreMediaRichText: Transformed RichText to HTML within 2.1 ms:
{
  in: '<?xml version="1.0" encoding="utf-8"?>
    <div xmlns="http://www.coremedia.com/2003/richtext-1.0"
        xmlns:xlink="http://www.w3.org/1999/xlink">
      <p>Lorem</p>
    </div>'
  out: '<p>Lorem</p>'
}
```

Example 10.28. CoreMediaRichText: From Data to Data View

```
CoreMediaRichText: Transformed HTML to RichText within 1.6 ms:
{
  in: '<p>Lorem</p>'
  out: '<?xml version="1.0" encoding="utf-8"?>
    <div xmlns="http://www.coremedia.com/2003/richtext-1.0">
      <p>Lorem</p>
    </div>'
}
```

Example 10.29. CoreMediaRichText: From Data View to Data

For more details, have a look at output of `HtmlDomConverter`, with entries similar to those shown in Example 10.30, "To Data Processing: Processing Stages" [459]. They will tell you, in what stage of processing and in which order rules are applied. In the given example you see, that the transformation of `<h1>` to the well-known representation in *CoreMedia Rich Text 1.0* as `<p class="p--heading-1">` happens in a stage called *imported*. This stage information may help you to analyze your data-processing if it provides unexpected results. For details on these stages, have a look at Section 10.2.10, "Rich Text Plugin" [427] and the referenced documentation in *CoreMedia CKEditor 5 Plugins* workspace.

```
HtmlDomConverter:
  convert(#document-fragment) {
    input: '#document-fragment: type: 11, contents: <h1>Hello World!</h1>'
  }
...
convert(H1) {
```

```
  input: '<h1>: ...'
}

convert(H1); Stage: prepared {
  input: '<h1>: ...',
  prepared: '<h1>: ...'
}

convert(H1); Stage: imported {
  input: '<h1>: ...',
  imported: '<p>: ...'}

... children being processed ...

convert(H1); Stage: importedWithChildren {
  input: '<h1>: ...',
  importedWithChildren: '<p>: ...'
}

...

HtmlDomConverter: convert(H1); Stage: Done. {
  input: '#document-fragment: type: 11, contents: <h1>Hello World!</h1>',
  output: '#document-fragment: type: 11,
    contents:
      <p xmlns="http://www.coremedia.com/2003/richtext-1.0"
         class="p--heading-1">
        Hello World!
      </p>'}
...
```

Example 10.30. To Data Processing: Processing Stages

For details regarding data-processing and transformation within the layers of *CKEditor 5* you may want to have a look at Section 10.1.1, "Glance at CKEditor 5 Architecture" [417].

Sanitation

The *Sanitation* is responsible for handling any XML elements, attributes or structure that would not validate when send to the server. Thus, it is a kind of firewall, to ensure, that the data provided by the editor are stored on the server at best effort. For details see Section 10.2.10, "Rich Text Plugin" [427] and especially section "Note on Strictness" [427].

Typically, you will see debugging output similar as shown in Example 10.31, "Sanitation: Default Operation" [461]. It signals standard operation with no applied modifications. If you have an element in data view, where a corresponding mapping is missing, you will instead get a warning (with preceding debug information) as shown in Example 10.32, "Sanitation: Warnings" [461]. Such warnings typically signal, that you should adjust your data-processing. The example output tells you about a `<mark>` element left at parent element `<p>`, that is not supported by *CoreMedia Rich Text 1.0*. In the given case, sanitation will remove the `<mark>` element and replace it by its children within the given parent node.

To fix such issues, you should adapt your data processing configuration as sketched in section "Configure Data-Processing" [451].

```
[DEBUG] CoreMediaRichText:
  Sanitation done:
    Visited Elements: 5;
    Removed: 0 severe of 0 total;
    Removed Invalid Attributes: 0,
    Maximum Element Depth: 1;
    Duration (ms): 0.10000002384185791
```

Example 10.31. Sanitation: Default Operation

```
[DEBUG] CoreMediaRichText: Removing mark (type: 1, parent: p): invalidAtParent

[WARN] CoreMediaRichText:
  Sanitation done with issues (turn on debug logging for details):
    Visited Elements: 3;
    Removed: 1 severe of 1 total;
    Removed Invalid Attributes: 0,
    Maximum Element Depth: 2;
    Duration (ms): 0.5
```

Example 10.32. Sanitation: Warnings

11. Security

In this chapter you will get to know about security mechanisms in *CoreMedia Studio*. This chapter does not cover general deployment aspects but focuses on application level security topics.

11.1 Preview Integration

It is recommended to serve the preview application and *CoreMedia Studio* application from different origins (the origin includes protocol, host, port), as described in Section 3.3, "Basic Preview Configuration" [21]. By separating the application origins, the browser ensures that both applications run independently in their own environment without direct access to each other (see Same-origin policy). Potential vulnerabilities in the preview application can not automatically propagate into the Studio application and vice versa.

It is highly recommended serving both, *CoreMedia Studio* and the embedded preview over HTTPS. The unencrypted HTTP protocol should only be used in a well separated development environment. Due to several browser constraints regarding mixed content it is highly discouraged to serve *CoreMedia Studio* and the embedded preview over different protocols.

11.2 Content Security Policy

Cross-site scripting (XSS) vulnerabilities are a severe threat for all high profile web applications like *CoreMedia Studio*. While conscientious output escaping always has to be the first choice in order to avoid cross-site scripting attacks, most modern web browsers offer a new standard called Content Security Policy (CSP) as a second line of defense (see http://www.w3.org/TR/CSP/).

Default Policy

The standard Blueprint *CoreMedia Studio* enables Content Security Policy by default. It configures at least the following default CSP directives in the browser.

```
default-src 'none';
style-src 'self' 'unsafe-inline';
script-src 'self' 'unsafe-eval';
img-src 'self';
connect-src 'self';
object-src 'self';
font-src 'self';
media-src 'self';
manifest-src 'self';
frame-src <YOUR_PREVIEW_ORIGIN> 'self';
```

The configuration represents the minimum set of directives to comply with the Studio's and its third-party library requirements. Both, the `unsafe-inline` value of the style-src directive and the `unsafe-eval` value of the script-src directive are required by Ext JS.

Customize Policy

Each of the CSP directives that are included in the default configuration plus the report-uri directive can be easily customized.

> **CAUTION**
>
> Note, that weakening the policy settings can have severe effects on the application's security. Especially re-enabling inline script execution is considered harmful as it thwarts all efforts to prevent XSS.

Customization is done via a set of `studio.security.csp.*` properties in the `WEB-INF/application.properties` property file of the *Studio Server* web application. Each property is responsible for one Content Security Policy directive.

> **NOTE**
>
> Please note that for legacy reasons the configuration needs to be done in the *Studio Server*. The *Studio Client* will reuse the CSP directives by sending a request to the *Studio Server* and dynamically creating a meta HTML element which adds the directives before the actual *Studio Application* is bootstrapped.

- `studio.security.csp.scriptSrc`: Takes a list of values for the script-src policy directive. Default values are `'self','unsafe-eval'`.
- `studio.security.csp.styleSrc`: Takes a list of values for the style-src policy directive. Default values are `'self','unsafe-inline'`.
- `studio.security.csp.frameSrc`: Takes a list of values for the frame-src policy directive. The following values are appended if applicable.
  - `studio.previewUrlWhitelist` values if specified OR Schema and authority of `studio.previewUrlPrefix` if specified.
  - `'self'`, if frameSrc does not contain 'none'
- `studio.security.csp.connectSrc`: Takes a list of values for the connect-src policy directive. Default value is `'self'`.
- `studio.security.csp.fontSrc`: Takes a list of values for the font-src policy directive. Default value is `'self'`.
- `studio.security.csp.imgSrc`: Takes a list of values for the img-src policy directive. Default value is `'self'`.
- `studio.security.csp.mediaSrc`: Takes a list of values for the media-src policy directive. Default value is `'self'`.
- `studio.security.csp.objectSrc`: Takes a list of values for the object-src policy directive. Default value is `'self'`.
- `studio.security.csp.reportUri`: Takes a list of values for the report-uri policy directive. If no custom list is provided, the directive is not included in the CSP directives.

Here is an example how an adapted property would look like.

```
studio.security.csp.objectSrc='self',www.exampleDomain.com
```

Using frame-ancestors directive

The frame-ancestors directive is used to defend clickjacking attacks. Due to the way the *Studio Client* defines its CSP directives, it cannot be configured via `WEB-INF/application.properties`. This is because the CSP standard does not support setting this directive in meta HTML elements.

In order to configure the directive you need to adjust the configuration of the web server so it provides a corresponding CSP HTTP header. Our default docker deployment will already set the frame-ancestors to `'self'`.

Please note that the frame-ancestors directive is part of the Content Security Policy Level 2 standard which is not yet supported by all the browsers that support Content Security Policy Level 1. If required, similar functionality can be achieved for 'legacy' browsers by setting an appropriate `X-Frame-Options` header in the web server delivering the *Studio Client*.

Write CSP Compliant Code

According to the default policy, inline JavaScript will not be executed. This restriction bans both inline script blocks and inline event handlers (for example `onclick="..."`). The first restriction wipes out a huge class of cross-site scripting attacks by making it impossible to accidentally execute scripts provided by a malicious third-party. It does, however, require a clean separation between content and behavior (which is good practice anyway). The required code changes for inline JavaScript code can be summarized as follows:

- Inline script blocks needs to move into external JavaScript files.
- Inline event handler definitions must be rewritten in terms of `addEventListener` and extracted into component code.

CSP violations can be easily discovered by monitoring the browser console. All violations are logged as errors including further details about the violation type and culprit.

Customize CSP Mode

CoreMedia Studio can run in one of four supported CSP modes.

- `ENFORCE`: Full CSP protection is enabled. All directives are enforced and reported.
- `ENFORCE_ALLOW_DISABLE`: Enable full CSP protection unless the `disableCsp` query parameter is set to 'true'. This mode is not recommended for a production environment.
- `REPORT`: CSP protection is enabled in report only mode. All violations are reported using the `report-uri` directives configured in `studio.security.csp.reportUri` but the directives are not enforced. This mode is not recommended for a production environment.
- `DISABLE`: CSP protection is disabled. This setting is not recommended.

The configuration is done via the `studio.security.csp.cspMode` key of the `WEB-INF/application.properties` property file of the *Studio Server* web application.

11.3 Single Sign On Integration

The default *CoreMedia Studio* authentication process is implemented based on Spring Security. Due to this open standard it is possible to replace the standard authentication mechanism with a common redirect based SSO solution like *OAuth2*.

While the authentication process can be replaced, the *CoreMedia Content Server* still needs to have a matching user provider configured in order to perform a fine grained authorization. Please refer to the Content Server Manual for further details about user providers.

This documentation does not replace the SSO manufacturers manual about how to integrate with Spring Security. This section only covers *CoreMedia Studio* specific adjustments that need to be made to a generic integration.

> **WARNING**
>
> Do not modify the authentication process and the Spring Security filter chain unless you know what you are doing. An improperly configured security context can cause severe security issues.

11.3.1 Disable EditingRestSecurityAutoConfiguration

The entrypoint to *Studio*'s Spring Security configuration is the AutoConfiguration class `com.coremedia.springframework.component.security.EditingRestSecurityAutoConfiguration` in module `editing-rest-security`, where a custom `SecurityFilterChain` bean is created.

To implement your custom authentication mechanism you have to first disable the built-in AutoConfiguration class, which can be done by setting the following Spring property:

```
spring.autoconfigure.exclude=
com.coremedia.springframework.component.security.EditingRestSecurityAutoConfiguration
```

11.3.2 Create your own AutoConfiguration

We recommend to create a new Maven module with a new `AutoConfigura tion` class. The module needs a compile-dependency on `editing-rest-security-component` and of course has to be added to the application by adding a dependency on it, e.g. in the `studio-blueprint-component` module.

The AutoConfiguration class should then look like this:

```
@AutoConfiguration
@EnableWebSecurity
@Import(com.coremedia.rest.security.config.EditingRestSecurityBaseConfiguration.class)
public class MySsoAutoConfiguration {
    ...
}
```

`@EnableWebSecurity` is necessary for customizing the Spring Security configuration and the import of `EditingRestSecurityBaseConfigura tion` adds some beans that are used in the authentication process and some that are used by your custom `SecurityFilterChain`.

Do not forget to expose your new class as AutoConfiguration by adding a Spring imports file at `src/main/resources/META-INF/spring/org.springframework.boot.autoconfigure.Auto Configuration.imports` containing the fully qualified name of your Auto-Configuration class.

In the next sections we will go through the beans that you will create in this Configuration class: The `securityFilterChain` and the `springSecur ityCapUserFinder`.

11.3.3 Create your own SecurityFilterChain

The concrete configuration of the `SecurityFilterChain` will of course depend heavily on your SSO provider, but there are also some mandatory and recommended Studio-specific settings. The following example has been created for an OAuth2 provider, and we will go through it step-by-step.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http,
                                               RequestMatcher
unauthenticatedRequestMatcher,
                                               RequestMatcher
authenticatedRequestMatcher,
SessionFixationProtectionStrategy sessionFixationProtectionStrategy,
                                               RequestMatcher
csrfIgnoringRequestMatcher,
                                               RequestMatcher
logoutRequestMatcher,
                                               CapLogoutHandler
capLogoutHandler,
                                               LogoutSuccessHandler
logoutSuccessHandler,
                                               AccessDeniedHandler
accessDeniedHandler) throws Exception {
  return http
          .oauth2Login(Customizer.withDefaults())
          .authorizeHttpRequests(authorize -> {
authorize.requestMatchers(unauthenticatedRequestMatcher).permitAll();

authorize.requestMatchers(authenticatedRequestMatcher).authenticated();
          })
          .sessionManagement(session ->
session.sessionAuthenticationStrategy(sessionFixationProtectionStrategy))
          .csrf(csrf ->
csrf.ignoringRequestMatchers(csrfIgnoringRequestMatcher))
          .exceptionHandling(exception ->
exception.accessDeniedHandler(accessDeniedHandler))
        .logout(logout -> logout.logoutRequestMatcher(logoutRequestMatcher)

                  .addLogoutHandler(capLogoutHandler)
                  .logoutSuccessHandler(logoutSuccessHandler))
          .headers(Customizer.withDefaults())
          .build();
}
```

There are different styles of writing such a configuration. In this example we create several `Configurers` on the HttpSecurity bean like e.g. `logout()`, customize their behavior by calling methods on them and then get back to the HttpSecurity bean with `and()` to continue with the next `Configurer`.

The first configurer `oauth2Login()` adds support for authentication using OAuth2 and is just an example. Your SSO provider might require different config‐uration.

`authorizeHttpRequests()` and the following `requestMatchers` con‐figure which requests require authentication in the first place. Generally only the requests to `/api/**` are protected, but there are also some paths below that need to be accessible without authentication. To this end you can use the predefined RequestMatchers `unauthenticatedRequestMatcher` and `authenticatedRequestMatcher`.

`sessionManagement()` and `csrf()` are used to configure protection against session fixation attacks and CSRF with predefined strategies for *Studio*.

`exceptionHandling()`: The predefined `accessDeniedHandler` is a `com.coremedia.rest.security.config.SimpleLogoutAccess`

`DeniedHandler` takes care of correct redirection if the request is a request to the logout url. Other requests receive a 403 response.

`logout()`: The predefined `capLogoutHandler` and `logoutSuccessHand ler` take care of closing a user's `CapSession` and correct redirection on logout. The predefined `logoutRequestMatcher` is configured for path `/logout` and method `POST` and is also used to configure the `accessDenied Handler`.

`headers()` adds some recommended security headers to the response.

11.3.4 Create your own SpringSecurityCapUserFinder

With the `SecurityFilterChain` you now have configured the process to authenticate a user against your SSO provider and provide access to the Studio api. After authentication the user details are usually represented by a SSO specific details object linked to the Spring Security `Authentication` object.

Now *CoreMedia Studio* needs to know the matching `com.core media.cap.user.User` for the current SSO specific user details. Each individual Unified API operation has to be performed in the name of the currently authenticated `User` in order to be able to perform a fine grained authorization in the *CoreMedia Content Server*. To create the mapping between SSO specific user details and a `User` for the chosen SSO system, you have to implement a `SpringSecurityUserFinder`.

The `SpringSecurityCapUserFinder` interface consists of only one method that finds a `User` for a given `Authentication` object. In order to write a finder for the chosen SSO system you can extend the `AbstractSpring SecurityCapUserFinder`.

```java
public class XYZSpringSecurityCapUserFinder
        extends AbstractSpringSecurityCapUserFinder
        implements SpringSecurityCapUserFinder {

  @Override
  public User findCapUser(Authentication authentication) {
    Object principal = authentication.getPrincipal();
    if (principal instanceof XYZ) {
      String username = GET_USER_NAME_FROM_USER_DETAILS;
      return getCapConnection().getUserRepository()
              .getUserByName(username, DOMAIN);
    }
    return null;
```

```
   }
}
```

The custom user finder is automatically picked up when it is defined as a bean with the name `springSecurityCapUserFinder` in the Spring context like this:

```
@Bean
SpringSecurityCapUserFinder springSecurityCapUserFinder(CapConnection
capConnection) {
  XYZSpringSecurityCapUserFinder xyzSpringSecurityCapUserFinder = new
XYZSpringSecurityCapUserFinder();
  xyzSpringSecurityCapUserFinder.setCapConnection(capConnection);
  return xyzSpringSecurityCapUserFinder;
}
```

11.3.5 Studio Login Url

By default, the *Studio* client shows a local login page if it detects that no user is logged in. Because this behavior is not appropriate in an SSO setting, you should set the *Studio* backend property `studio.loginUrl` to the SSO login page. The *Studio* frontend will then forward the user to the login page, if no current session can be found.

In our OAuth2 example, we set the `studio.loginUrl` to `/rest/login` to use the built-in Spring Security login page. The prefix `rest` is necessary because *Studio* client and server are deployed separately but are accessed through a common proxy and everything below `/rest` is forwarded to the server.

11.3.6 Proxy settings

As mentioned in the previous section, *Studio* client and server are deployed separately but are accessed through a common proxy. Depending on the SSO system, it might be necessary to add some proxy rules.

For the OAuth2 example, we had to add rules to forward requests to `/oauth2` and `/login/oauth2` to the *Studio* server.

11.4 Auto Logout

CoreMedia Studio provides two complementing mechanisms for automatically logging out inactive users: server-side session management and client-side activity tracking.

Jointly, these two algorithms keep the number of active sessions to a minimum, reducing the opportunity for an attacker to hijack a *Studio* session. The session timeouts for these algorithms can be configured separately. You should strive for a balance between security and user convenience.

Server-Side Session Management

A login to *CoreMedia Studio* is supported by a servlet session that is established with the web application container. If the client application in the browser does not contact the web application for a certain time, the servlet session will be closed by the container.

When the servlet session dies and the *Studio* client contact the server again, the condition will be detected and an appropriate error message is shown. The user will need to log in again.

Note that this timeout appears typically when the browser is closed or when the client machine is suspended or shut down. As long as *Studio* is open in a running browser, it continually fetches events from the server using HTTP requests. These requests keep the session alive.

You can configure the timeout via Spring Boot property `server.servlet.session.timeout`. (For WAR deployment use `web.xml` file of the *Studio* web application). Most containers set a default value of 30 minutes. Because the *Studio* client contacts the server at least every 20 seconds, you may opt to reduce the timeout significantly. You should not reduce it to less than a couple of minutes, though, so that temporary network problems do not cause *Studio* to disconnect.

Client-Side Activity Tracking

In order to detect that the user is not interacting with a running *CoreMedia Studio*, a client-side process continually detects mouse movements and write requests, which provide a good indication of use activity.

When the user is inactive for too long, the *CoreMedia Studio* session is closed and the login screen is shown again. This timeout can be configured using the

application property `studio.auto-logout.delay`. By default, the timeout is set to 30 minutes.

11.5 Logging

In order to support the detection of attacks and analysis of incidents, authentication failures as well as successful authentication events are logged by *CoreMedia Studio*. Example 11.1, "Example Output" [475] shows some typical log entries.

```
2025-02-27 13:43:30 [WARN] (http-bio-8080-exec-8)
  Http401AuthenticationFailureHandler [] -
  Failed Authentication - User: Rick,
  IP: 127.0.0.1
2025-02-27 13:51:11 [INFO] (http-bio-8080-exec-6)
  Http200AuthenticationSuccessHandler [] -
  Successful Authentication - User: Rick (coremedia:///cap/user/8),
  IP: 127.0.0.1
```

Example 11.1. Example Output

Marker Hierarchy

To get a better overview of security events you might want to duplicate or even redirect such events to extra access logs. To do so *CoreMedia Studio* uses a SLF4j Marker hierarchy

- **coremedia** – root marker
  - **security** – security related entries
    - **authentication** – for example login or logout events
    - **authorization** – events such as missing rights for certain actions

Example 11.2. Marker Hierarchy

Filtering

Filtering log entries is described in Logback's Online Documentation, Chapter 7: Filters.

> **NOTE**
> The `JaninoEventEvaluator` has been removed without a replacement. To migrate existing configurations, please follow the steps described in Logback's Online Documentation, Chapter 7: Filters –JaninoEventEvaluator–.

```
    <appender name="access"
        class="ch.qos.logback.core.FileAppender">
```

```
  <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
    <evaluator class="com.acme.CustomAuthenticationExpressionEvaluator"/>
    <OnMismatch>DENY</OnMismatch>
    <OnMatch>ACCEPT</OnMatch>
  </filter>
  <encoder><pattern>${log.pattern}</pattern></encoder>
  <file>access.log</file>
</appender>
```

Example 11.3. Configure Access Log

```java
      package com.acme;

import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.core.boolex.EventEvaluatorBase;

/**
 * Example of a custom expression evaluator
 * that checks if the event contains a marker with the string "authentication".

 */
public class CustomAuthenticationExpressionEvaluator extends
EventEvaluatorBase<ILoggingEvent> {

    public boolean evaluate(ILoggingEvent event) {
        return event.getMarkerList() != null &&
event.getMarkerList().contains("authentication");
    }
}
```

Example 11.4. Custom Expression Evaluator

Example 11.3, "Configure Access Log" [475] shows an example of how to log authen-
tication events to a file named access.log. Only authentication events will
be logged here.

```
  <appender name="security"
        class="ch.qos.logback.core.FileAppender">
  <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
    <evaluator class="com.acme.CustomSecurityExpressionEvaluator"/>
    <OnMismatch>DENY</OnMismatch>
    <OnMatch>ACCEPT</OnMatch>
  </filter>
  <encoder><pattern>${log.pattern}</pattern></encoder>
  <file>security.log</file>
</appender>
```

Example 11.5. Configure Security Log

```java
      package com.acme;
```

```java
import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.core.boolex.EventEvaluatorBase;

/**
 * Example of a custom expression evaluator
 * that checks if the event contains a marker with the string "security".
 */
public class CustomSecurityExpressionEvaluator extends
EventEvaluatorBase<ILoggingEvent> {

    public boolean evaluate(ILoggingEvent event) {
        return event.getMarkerList() != null &&
event.getMarkerList().contains("security");
    }
}
```

Example 11.6. Custom Expression Evaluator

Example 11.5, "Configure Security Log" [476] shows an example how to log any security related events to a file named `security.log`. As `security` contains `authentication`, also authentication log entries will go here.

```xml
    <appender name="default"
            class="ch.qos.logback.core.FileAppender">
  <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
    <evaluator class="com.acme.CustomSecurityExpressionEvaluator"/>
    <OnMismatch>NEUTRAL</OnMismatch>
    <OnMatch>DENY</OnMatch>
  </filter>
  <encoder><pattern>${log.pattern}</pattern></encoder>
  <file>default.log</file>
</appender>
```

Example 11.7. Configure Default Log

Example 11.7, "Configure Default Log" [477] shows an example for an appender which ignores any security related log entries. You might want to use this approach to hide login/logout entries from unauthorized personal.

```xml
    <logger name="com.coremedia"
        additivity="false"
        level="info">
  <appender-ref ref="security"/>
  <appender-ref ref="access"/>
  <appender-ref ref="default"/>
</logger>
```

Example 11.8. Configure Logger

Example 11.7, "Configure Default Log" [477] eventually binds all appenders to the given logger.

```
    <turboFilter class="ch.qos.logback.classic.turbo.MarkerFilter">
  <Marker>security</Marker>
  <OnMatch>DENY</OnMatch>
</turboFilter>
```

Example 11.9. Suppress Security Logging

Example 11.9, "Suppress Security Logging" [478] is just another example in case you completely want to suppress security log entries using so called turbo filters. See more about turbo filters in Logback's Online Documentation, Chapter 7: Filters –TurboFilters–.

12. Configuration Reference

Different aspects of *CoreMedia Studio* can be configured with different proper‐ ties. All configuration properties are bundled in the Deployment Manual (Chapter 3, *CoreMedia Properties Overview* in *Deployment Manual*). The following links contain the properties that are relevant for *Studio*:

- Section 3.4.1, "Studio Configuration" in *Deployment Manual* contains properties for the general configuration of *Studio*.
- Section 3.4.2, "Available Locales Configuration" in *Deployment Manual* contains properties for the available locales in *Studio*.
- Section 3.4.4, "Content Configuration" in *Deployment Manual* contains prop‐ erties for the content repository paths with special meaning in *Studio*.
- Section 3.4.5, "Navigation Validator Configuration" in *Deployment Manual* contains properties for validating the navigation structure in *Studio*.
- Section 3.4.6, "Preview URL Service Properties" in *Deployment Manual* contains properties for the Multi Preview Menu in *Studio*.
- Section 3.4.7, "Content Security Policy Configuration" in *Deployment Manual* contains properties for the configuration of the Content Security Policy (CSP) in *Studio*.
- Section 3.4.8, "Content Hub Configuration" in *Deployment Manual* contains properties for the configuration of for the *CoreMedia Content Hub*.
- Section 3.4.9, "Feedback Hub Configuration" in *Deployment Manual* contains properties for the configuration of for the *CoreMedia Feedback Hub*.
- Section 3.4.10, "Editorial Comments Configuration" in *Deployment Manual* contains properties for the configuration of for the *CoreMedia Editorial Comments* feature, which establishes a connection to the relational database.

Glossary

Blob	Binary Large Object or short blob, a property type for binary objects, such as graphics.
CaaS	Content as a Service or short caas, a synonym for the CoreMedia Headless Server.
CAE Feeder	Content applications often require search functionality not only for single content items but for content beans. The *CAE Feeder* makes content beans searchable by sending their data to the *Search Engine*, which adds it to the index.
Content Application Engine (CAE)	The *Content Application Engine* (*CAE*) is a framework for developing content applications with *CoreMedia CMS*.
	While it focuses on web applications, the core frameworks remain usable in other environments such as standalone clients, portal containers or web service implementations.
	The CAE uses the Spring Framework for application setup and web request processing.
Content Bean	A content bean defines a business oriented access layer to the content, that is managed in *CoreMedia CMS* and third-party systems. Technically, a content bean is a Java object that encapsulates access to any content, either to CoreMedia CMS content items or to any other kind of third-party systems. Various CoreMedia components like the CAE Feeder or the data view cache are built on this layer. For these components the content beans act as a facade that hides the underlying technology.
Content Delivery Environment	The *Content Delivery Environment* is the environment in which the content is delivered to the end-user.
	It may contain any of the following modules:
	• *CoreMedia Master Live Server* • *CoreMedia Replication Live Server* • *CoreMedia Content Application Engine* • *CoreMedia Search Engine* • *Elastic Social* • *CoreMedia Native Personalization*

Content Feeder	The *Content Feeder* is a separate web application that feeds content items of the CoreMedia repository into the *CoreMedia Search Engine*. Editors can use the *Search Engine* to make a full text search for these fed items.
Content item	In *CoreMedia CMS*, content is stored as self-defined content items. Content items are specified by their properties or fields. Typical content properties are, for example, title, author, image and text content.
Content Management Environment	The *Content Management Environment* is the environment for editors. The content is not visible to the end user. It may consist of the following modules:

- *CoreMedia Content Management Server*
- *CoreMedia Workflow Server*
- *CoreMedia Studio*
- *CoreMedia Search Engine*
- *CoreMedia Native Personalization*
- *CoreMedia Preview CAE*

Content Management Server	Server on which the content is edited. Edited content is published to the Master Live Server.
Content Repository	*CoreMedia CMS* manages content in the Content Repository. Using the Content Server or the UAPI you can access this content. Physically, the content is stored in a relational database.
Content Server	*Content Server* is the umbrella term for all servers that directly access the CoreMedia repository:
	Content Servers are web applications running in a servlet container.

- *Content Management Server*
- *Master Live Server*
- *Replication Live Server*

Content type	A content type describes the properties of a certain type of content. Such properties are for example title, text content, author, ...
Contributions	Contributions are tools or extensions that can be used to improve the work with *CoreMedia CMS*. They are written by CoreMedia developers – be it clients, partners or CoreMedia employees. CoreMedia contributions are hosted on Github at https://github.com/coremedia-contributions.
Control Room	*Control Room* is a *Studio* plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other *Studio* users.
CORBA (Common Object Request Broker Architecture)	The term *CORBA* refers to a language- and platform-independent distributed object standard which enables interoperation between heterogenous applications over a network. It was created and is currently controlled by

the Object Management Group (OMG), a standards consortium for distributed object-oriented systems.

CORBA programs communicate using the standard IIOP protocol.

CoreMedia Studio	*CoreMedia Studio* is the working environment for business specialists. Its functionality covers all the stages in a web-based editing process, from content creation and management to preview, test and publication. As a modern web application, *CoreMedia Studio* is based on the latest standards like Ajax and is therefore as easy to use as a normal desktop application.
Dead Link	A link, whose target does not exist.
Derived Site	A derived site is a site, which receives localizations from its master site. A derived site might itself take the role of a master site for other derived sites.
DTD	A Document Type Definition is a formal context-free grammar for describing the structure of XML entities. The particular DTD of a given Entity can be deduced by looking at the document prolog:

```
<!DOCTYPE    coremedia    SYSTEM    "http://www.core
media.com/dtd/coremedia.dtd"
```

There're two ways to indicate the DTD: Either by Public or by System Identifier. The System Identifier is just that: a URL to the DTD. The Public Identifier is an SGML Legacy Concept.

Elastic Social	*CoreMedia Elastic Social* is a component of *CoreMedia CMS* that lets users engage with your website. It supports features like comments, rating, likings on your website. *Elastic Social* is integrated into *CoreMedia Studio* so editors can moderate user generated content from their common workplace. *Elastic Social* bases on NoSQL technology and offers nearly unlimited scalability.
EXML	EXML is an XML dialect used in former CoreMedia Studio version for the declarative development of complex Ext JS components. EXML is Jangaroo 2's equivalent to Apache Flex (formerly Adobe Flex) MXML and compiles down to ActionScript. Starting with release 1701 / Jangaroo 4, standard MXML syntax is used instead of EXML.
Folder	A folder is a resource in the CoreMedia system which can contain other resources. Conceptually, a folder corresponds to a directory in a file system.
FTL	FTL (FreeMarker Template Language) is a Java-based template technology for generating dynamic HTML pages.

gRPC	gRPC is an open source high performance Remote Procedure Call (RPC) framework.
Headless Server	CoreMedia Headless Server is a CoreMedia component introduced with CoreMedia Content Cloud which allows access to CoreMedia content as JSON through a GraphQL endpoint. The generic API allows customers to use CoreMedia CMS for headless use cases, for example delivery of pure content to Native Mobile Applications, Smartwatches/Wearable Devices, Out-of-Home or In-Store Displays or Internet-of-Things use cases.
Home Page	The main entry point for all visitors of a site. Technically it is often referred to as root document and also serves as provider of the default layout for all subpages.
IETF BCP 47	Document series of *Best current practice* (BCP) defined by the Internet Engineering Task Force (IETF). It includes the definition of IETF language tags, which are an abbreviated language code such as en for English, pt-BR for Brazilian Portuguese, or nan-Hant-TW for Min Nan Chinese as spoken in Taiwan using traditional Han characters.
IOR (Interoperable Object Reference)	A CORBA term, *Interoperable Object Reference* refers to the name with which a CORBA object can be referenced.
Jangaroo	*Jangaroo* is a JavaScript framework developed by CoreMedia that supports TypeScript (formerly MXML/ActionScript) as an input language which is compiled down to JavaScript compatible with Ext JS. You will find detailed descriptions on the Jangaroo webpage http://www.jangaroo.net. Jangaroo 4 is the ActionScript/MXML/Maven based version for CMCC 10. Since CMCC 11 (2110), Jangaroo uses TypeScript and is implemented as a *Node.js* and *npm* based set of tools.
Java Management Extensions (JMX)	The Java Management Extensions is an API for managing and monitoring applications and services in a Java environment. It is a standard, developed through the Java Community Process as JSR-3. Parts of the specification are already integrated with Java 5. JMX provides a tiered architecture with the instrumentation level, the agent level and the manager level. On the instrumentation level, MBeans are used as managed resources.
Locale	Locale is a combination of country and language. Thus, it refers to translation as well as to localization. Locales used in translation processes are typically represented as IETF BCP 47 language tags.
Master Live Server	The *Master Live Server* is the heart of the *Content Delivery Environment*. It receives the published content from the *Content Management Server* and makes it available to the *CAE*. If you are using the *CoreMedia Multi-Master Management Extension* you may use multiple *Master Live Server* in a CoreMedia system.

Master Site	A master site is a site other localized sites are derived from. A localized site might itself take the role of a master site for other derived sites.
MIME	With Multipurpose Internet Mail Extensions (MIME), the format of multi-part, multimedia emails and of web documents is standardised.
MXML	MXML is an XML dialect used by Apache Flex (formerly Adobe Flex) for the declarative specification of UI components and other objects. Up to CMCC 10 (2107), CoreMedia Studio used the Open Source compiler Jangaroo 4 to translate MXML and ActionScript sources to JavaScript that is compatible with Ext JS 7. Starting with CMCC 11 (2110), a new, Node.js and npm based version of Jangaroo is used that supports standard TypeScript syntax instead of MXML/ActionScript, still compiling to Ext JS 7 JavaScript.
OCI (Open Container Initiative)	The Open Container Initiative (OCI) is a Linux Foundation project that defines open industry standards for container formats and runtimes. OCI specifications ensure compatibility and interoperability between container tools, engines, and orchestration platforms like Docker and Kubernetes.
ORAS (OCI Registry As Storage)	ORAS (OCI Registry As Storage) is a tool and specification that extends OCI registries to store and distribute OCI artifacts beyond container images. It provides a standardized way for developers to push and pull arbitrary content types to and from container registries, enabling these registries to function as general artifact stores.
Personalisation	On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.
Projects	With projects you can group content and manage and edit it collaboratively, setting due dates and defining to-dos. Projects are created in the Control Room and managed in project tabs.
Property	In relation to CoreMedia, properties have two different meanings:
	In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content item depends on the content type.
	In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.
Replication Live Server	The aim of the *Replication Live Server* is to distribute load on different servers and to improve the robustness of the *Content Delivery Environment*. The *Replication Live Server* is a complete Content Server installation. Its content is an replicated image of the content of a *Master Live Server*. The *Replication Live Server* updates its database due to change events from the *Master Live Server*. You can connect an arbitrary number of *Replication Live Servers* to the *Master Live Server*.
Resource	A folder or a content item in the CoreMedia system.

ResourceURI	A ResourceUri uniquely identifies a page which has been or will be created by the *Active Delivery Server*. The ResourceUri consists of five components: Resource ID, Template ID, Version number, Property names and a number of key/value pairs as additional parameters.
Responsive Design	Responsive design is an approach to design a website that provides an optimal viewing experience on different devices, such as PC, tablet, mobile phone.
Site	A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In *CoreMedia CMS* a site especially consists of a site folder, a site indicator and a home page for a site. A typical site also has a master site it is derived from.
Site Folder	All contents of a site are bundled in one dedicated folder. The most prominent document in a site folder is the site indicator, which describes details of a site.
Site Indicator	A site indicator is the central configuration object for a site. It is an instance of a special content type, most likely `CMSite`.
Site Manager Group	Members of a site manager group are typically responsible for one localized site. Responsible means that they take care of the contents of that site and that they accept translation tasks for that site.
Template	In CoreMedia, FreeMarker templates used for displaying content are known as Templates. OR In *Blueprint* a template is a predeveloped content structure for pages. Defined by typically an administrative user a content editor can use this template to quickly create a complete new page including, for example, navigation, predefined layout and even predefined content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes Application	The *User Changes Application* is a *Content Repository* listener, which collects all content, modified by *Studio* users. This content can then be managed in the *Control Room*, as a part of projects and workflows.
Variants	The set of all content items in a multi-site hierarchy related to each other via master references. This includes the top-level master content items themselves.
Version history	A newly created content item receives the version number 1. New versions are created when the content item is checked in; these are numbered in chronological order.

Weak Links	In general *CoreMedia CMS* always guarantees link consistency. But links can be declared with the *weak* attribute, so that they are not checked during publication or withdrawal.
	Caution! Weak links may cause dead links in the live environment.
Workflow	A workflow is the defined series of tasks within an organization to produce a final outcome. Sophisticated applications allow you to define different workflows for different types of jobs. So, for example, in a publishing setting, a document might be automatically routed from writer to editor to proofreader to production. At each stage in the workflow, one individual or group is responsible for a specific task. Once the task is complete, the workflow software ensures that the individuals responsible for the next task are notified and receive the data they need to execute their stage of the process.
Workflow Server	The *CoreMedia Workflow Server* is part of the Content Management Environment. It comes with predefined workflows for publication but also executes freely definable workflows.
XLIFF	XLIFF is an XML-based format, standardized by OASIS for the exchange of localizable data. An XLIFF file contains not only the text to be translated but also metadata about the text. For example, the source and target language. *CoreMedia Studio* allows you to export content items in the XLIFF format and to import the files again after translation.

Index