

COREMEDIA CONTENT CLOUD

CoreMedia Content Cloud v11 Upgrade Guide



Copyright CoreMedia GmbH © 2025

CoreMedia GmbH

Ludwig-Erhard-Straße 18

20459 Hamburg

International

All rights reserved. No part of this manual or the corresponding program may be reproduced or copied in any form (print, photocopy or other process) without the written permission of CoreMedia GmbH.

Germany

Alle Rechte vorbehalten. CoreMedia und weitere im Text erwähnte CoreMedia Produkte sowie die entsprechenden Logos sind Marken oder eingetragene Marken der CoreMedia GmbH in Deutschland. Alle anderen Namen von Produkten sind Marken der jeweiligen Firmen.

Das Handbuch bzw. Teile hiervon sowie die dazugehörigen Programme dürfen in keiner Weise (Druck, Fotokopie oder sonstige Verfahren) ohne schriftliche Genehmigung der CoreMedia GmbH reproduziert oder vervielfältigt werden. Unberührt hiervon bleiben die gesetzlich erlaubten Nutzungsarten nach dem UrhG.

Licenses and Trademarks

All trademarks acknowledged.
January 07, 2025 (Release 2110)

1. Preface	1
1.1. Audience	2
1.2. Typographic Conventions	3
1.3. CoreMedia Services	5
1.3.1. Registration	5
1.3.2. CoreMedia Releases	6
1.3.3. Documentation	7
1.3.4. CoreMedia Training	10
1.3.5. CoreMedia Support	10
2. CoreMedia Content Cloud v11 – the new Release	13
3. Upgrade Tasks	16
3.1. Prerequisites for the Upgrade	17
3.2. General Upgrade Strategy	19
3.3. Check your Supported Environments	21
3.4. Removing Outdated Components	22
3.5. Backup Content Server Databases	23
3.6. Upgrading your Workspace with Git	24
3.7. Upgrading Custom Blueprint Code	28
3.8. Upgrading LiveContext Integration Features to Commerce Hub	29
3.9. Upgrading the Build and Deployment Process	30
4. Changes in Supported Environments	31
4.1. Added Support	32
4.1.1. SAP Commerce Cloud 2105 Support	32
4.1.2. MongoDB 5.0 Support	32
4.2. Removed Support	33
4.2.1. HCL Commerce 8.0 Support	33
4.2.2. Removed Database Support	33
4.2.3. Removed Server Platforms	33
4.2.4. Internet Explorer 11 Support for Studio	33
5. Removed Components and Features	34
6. Detailed Overview of Changes	35
6.1. Updated Third-Party Libraries	36
6.1.1. Spring Boot 2.5 and Related Third-Party Libraries	36
6.1.2. Apache Tika 2.1.0 and Related Third-Party Libraries	37
6.2. API Changes	39
6.2.1. Overview of Changed API Elements	39
6.2.2. Removal of Deprecated Legacy Rule Providers	46
6.2.3. Removal of Deprecated LdapMember#getOrganization- alUnit	46
6.2.4. Implementations of Comparable: Added Type Argu- ments	47
6.2.5. Removal of Deprecated SitemapHandler#URI_PAT- TERN	47
6.2.6. Removal of Consumer, Function, Predicate and Related Classes	47
6.2.7. Removal of Deprecated Version of XliffExporter	48
6.3. Studio Client Changes	51
6.3.1. Modern Studio Layout Always Activated	51

6.3.2. Workflow App	51
6.3.3. New Content Type Localization API	51
6.3.4. New Toast API	55
6.3.5. View Aborted Workflows in Studio	56
6.3.6. Consolidated Notifications and Running Jobs	56
6.4. Richtext Editor Changes	57
6.4.1. The Future of Richtext Editing	57
6.4.2. Deprecation of CKEditor 4 Extensions	58
6.4.3. Adapting CKEditor 4 Extensions	58
6.4.4. Enable Developer Preview (Optional)	59
6.5. Studio Server Changes	64
6.5.1. Consolidating Group IDs and Versions	64
6.5.2. Editorial Comments Cache Properties Removed	64
6.5.3. Validators by Configuration	65
6.6. Content Server Changes	66
6.6.1. Database Schema	66
6.6.2. MediaStore API	66
6.6.3. Content Server Does Not Use corem.home Anymore	66
6.6.4. Secure Blob Access	67
6.7. Workflow Server Changes	68
6.7.1. Upgrading Workflow Definitions	68
6.7.2. Final Workflow Actions	68
6.7.3. Workflow Server Does not Use corem.home Anymore	69
6.8. Content Application Engine Changes	70
6.8.1. Moved Internal Controller to Management Port	70
6.9. Feeder and Search Engine Changes	71
6.9.1. Configuration Changes and Reindexing	71
6.10. Personalization Changes	75
6.10.1. Client-Side Personalization API	75
6.11. Commerce Integration Changes	76
6.11.1. Commerce Hub API v2	76
6.11.2. Commerce Adapter Configuration Properties	76
6.11.3. Commerce Adapter Cache Configuration changes	79
6.11.4. Salesforce Commerce Cloud OCAPI Changes	79
6.11.5. LiveContext HCL and B2B Removal	80
6.11.6. Various Commerce Hub API Removals of Deprecated Code	80
6.11.7. Removal of Legacy Commerce Search Facet API and Demo Content	81
6.11.8. Updated Multi Catalog Configuration	82
6.12. Blueprint Changes	84
6.12.1. Content Issue Search Enabled by Default	84
6.12.2. Guava Optional, Function, Predicate, Supplier Have been Replaced With Their JDK Counterparts	84
6.12.3. New Blueprint Parent	84
6.12.4. Consolidating Blueprint Group IDs and Versions	85
6.12.5. Replacing dockerfile-maven-plugin with google-jib-plu- gin	85

6.12.6. Content Hub and Feedback Hub Adapters as Plugins	87
6.12.7. Commerce "Candy Shop" Developer Setup Removed	88
6.12.8. Using Standard Spring Boot Logging Configuration	88
6.12.9. Removed Optimizely Integration	90
6.12.10. Removed Chef Deployment	90
6.12.11. Removed Watchdog/Probedog	90
6.12.12. Removed Dynamic Packages Proxy App	91
6.12.13. blueprint-doctypes-xmlrepo.xml moved	91
6.13. Frontend Workspace Changes	93
6.13.1. Using pnpm Instead of Yarn	93
6.13.2. Using NodeJS 16 (LTS)	94
6.13.3. Removal of Deprecated FreemarkerFacade Methods	94
6.13.4. Optimized Rendering of Embedded Content in Rich-text	94
6.14. Command Line Tool Changes	96
6.14.1. Deleted Deprecated Options of some Commandline Tools	96
7. Studio Client to TypeScript Migration	97
7.1. Studio Client in TypeScript/npm	98
7.2. Upgrading Studio Client to <i>CoreMedia Content Cloud</i> v11	100
7.2.1. Converting Studio Client to TypeScript	100
7.2.2. Completing Studio Client v11 Upgrade	105
7.2.3. Type Error Fixing Hints	107
7.3. Changes Between ActionScript and TypeScript for Developers	111
7.3.1. ActionScript → TypeScript	111
7.3.2. MXML → TypeScript	131

List of Figures

3.1. Update your *CoreMedia Content Cloud* v10 workspace to v11 24

List of Tables

1.1. Typographic conventions 3

1.2. Pictographs 4

1.3. CoreMedia manuals 7

6.1. Changed elements in com.coremedia.blueprint.base.links.impl package 39

6.2. Changed elements in com.coremedia.blueprint.cae package 39

6.3. Changed elements in com.coremedia.blueprint.themeimporter package 40

6.4. Changed elements in com.coremedia.cap.common.infos package 40

6.5. Changed elements in com.coremedia.cap.multisite package 40

6.6. Changed elements in com.coremedia.cap.transform package 41

6.7. Changed elements in com.coremedia.cmdline package 42

6.8. Changed elements in com.coremedia.common.util package 42

6.9. Changed elements in com.coremedia.ldap package 43

6.10. Changed elements in com.coremedia.objectserver.web package 43

6.11. Changed elements in com.coremedia.translate package 44

6.12. Changed elements in com.coremedia.cap.transform package 45

6.13. Changed elements in com.coremedia.xml package 45

6.14. Changed elements in hox.corem.server.media package 45

6.15. Replacements for Studio plugins 63

6.16. Changed properties for editorial.comments 64

6.17. Changed properties 67

6.18. pnpm commands comparison 93

7.1. Basic ActionScript-TypeScript example comparison 112

7.2. Interfaces ActionScript-TypeScript example comparison 113

7.3. Imports ActionScript-TypeScript example comparison 115

List of Examples

6.1. Typical Modern XliffExporter Usage	49
6.2. Using the ToastService	55
6.3. Create editor	61
6.4. Use CKEditor in field	61
6.5. Healthcheck	86
6.6. Commerce Adapter health check	87
7.1. Removing redundant prefix configuration	115
7.2. TypeScript example class	116
7.3. Mixin in ActionScript example	117
7.4. Mixins in TypeScript example	119
7.5. Ext config example	121
7.6. Bindable configs	121
7.7. Declaring Ext configs in ActionScript	122
7.8. Type-cast object code into class	124
7.9. Populate Config object in ActionScript	124
7.10. Different ways to instantiate class	124
7.11. Config properties in class	125
7.12. Declaring a virtual class member	126
7.13. Extending superclass Config type	126
7.14. TypeScript detecting type errors for existing properties	126
7.15. Preventing use of untyped properties	127
7.16. Create Ext Config objects with Config function	128
7.17. Instantiate object from Config object	129
7.18. Inline ad-hoc Config object	129
7.19. Typical work of constructor done in TypeScript	130
7.20. Using ConfigUtils utility class	130
7.21. Component with utility class in client	130
7.22. Example MXML file	131

1. Preface

This manual describes the upgrade from a *CoreMedia CMCC v10 Blueprint* system to a *CoreMedia CMCC v11 Blueprint* system.

- [Chapter 2, *CoreMedia Content Cloud v11 - the new Release* \[13\]](#) gives a short overview of the main changes in *CoreMedia Content Cloud* v11. It describes some of the benefits you get from your upgrade tasks.
- [Chapter 3, *Upgrade Tasks* \[16\]](#) describes the actual upgrade tasks in general. The following chapters then go into more detail per application and per feature.
- [Chapter 4, *Changes in Supported Environments* \[31\]](#) describes changes in the supported environments, such as databases.
- [Chapter 5, *Removed Components and Features* \[34\]](#) lists the removed components and features and links to more information.
- [Chapter 6, *Detailed Overview of Changes* \[35\]](#) gives a high-level overview of all changes, except for the Studio Client to Typescript switch. This includes new, deprecated and removed components and third-party software; new operation and configuration schemes, and much more.
- [Chapter 7, *Studio Client to TypeScript Migration* \[97\]](#) describes details of the largest change in *CoreMedia Content Cloud* v11, the switch from ActionScript/MXML to TypeScript for Studio Client.

1.1 Audience

This manual is intended for developers, architects, and project managers who plan to upgrade from *CoreMedia Content Cloud* v10 to *CoreMedia Content Cloud* v11.

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	Courier new	cm systeminfo start
Command line entries		
Parameter and values		
Class and method names		
Packages and modules		
Menu names and entries	Bold, linked with	Open the menu entry Format Normal
Field names	Italic	Enter in the field <i>Heading</i>
CoreMedia Components		The <i>CoreMedia Component</i>
Applications		Use <i>Chef</i>
Entries	In quotation marks	Enter "On"
(Simultaneously) pressed keys	Bracketed in "<>", linked with "+"	Press the keys <Ctrl>+<A>
Emphasis	Italic	It is <i>not</i> 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-11>

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://npm.coremedia.io>

Your pnpm client first needs to be logged in 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
Adaptive Personalization Manual	Developers, architects, administrators	This manual describes the configuration of and development with <i>Adaptive Personalization</i> , the CoreMedia module for personalized websites. You will learn how to configure the GUI used in <i>CoreMedia Studio</i> , how to use predefined contexts and how to develop your own extensions.
Analytics Connectors Manual	Developers, architects, administrators	This manual describes how you can connect your CoreMedia website with external analytic services, such as Google Analytics.
Blueprint Developer Manual	Developers, architects, administrators	<p>This manual gives an overview over the structure and features of <i>CoreMedia Content Cloud</i>. It describes the content type model, the <i>Studio</i> extensions, folder and user rights concept and many more details. It also describes administrative tasks for the features.</p> <p>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.</p>
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 <i>Content Application Engine [CAE]</i> . You will learn how to write JSP or Freemarker templates that access the other CoreMedia modules and use the sophisticated caching mechanisms of the CAE.

Manual	Audience	Content
Content Server Manual	Developers, architects, administrators	This manual describes the concepts and administration of the main CoreMedia component, the <i>Content Server</i> . You will learn about the 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 <i>Elastic Social</i> 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 <i>Headless Server</i> . You will learn how to deploy the Headless Server and how to use its endpoints for your sites.
Importer Manual	Developers, architects	This manual describes the structure of the internal CoreMedia XML format used for storing data, how you set up an <i>Importer</i> application and how you define the transformations that convert your content into CoreMedia content.
Multi-Site Manual	Developers, Multi-Site Administrators, Editors	This manual describes different options to design your site hierarchy with several languages. It also gives guidance to avoid common pitfalls during your work with the multi-site feature.

Manual	Audience	Content
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 <i>CoreMedia Search Engine</i> and the two feeder applications: the <i>Content Feeder</i> and the <i>CAE Feeder</i> .
Site Manager Developer Manual	Developers, architects, administrators	<p>This manual describes the configuration and customization of <i>Site Manager</i>, the Java based stand-alone application for administrative tasks. You will learn how to configure the <i>Site Manager</i> with property files and XML files and how to develop your own extensions using the <i>Site Manager API</i>.</p> <p>The Site Manager is deprecated for editorial work.</p>
Studio Developer Manual	Developers, architects	This manual describes the concepts and extension of <i>CoreMedia Studio</i> . You will learn about the underlying concepts, how to use the development environment and how to customize <i>Studio</i> to your needs.
Studio User Manual	Editors	This manual describes the usage of <i>CoreMedia Studio</i> for editorial and administrative work. It also describes the usage of the <i>Adaptive Personalization</i> and <i>Elastic Social</i> GUI that are integrated into <i>Studio</i> .
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 <i>CoreMedia Unified API</i> , which is the recommended API for most applications. This includes access to the content repository, the workflow repository and the user repository.

Manual	Audience	Content
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 <i>Workflow Server</i> . 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 CoreMedia 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:

<http://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:

<http://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 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 breakdown, you should be logging at debug level. See [Section 4.7, “Logging”](#) in *Operations Basics* for details.

Log files

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>
```

2. CoreMedia Content Cloud v11 – the new Release

This guide is intended to assist you during your upgrade. Familiarize yourself with the changes and actual tasks when upgrading from *CoreMedia Content Cloud* v10 to v11. Use this introduction to learn about the major enhancements and to understand the big picture.

Use the following chapters to review all the information about the changes that concern you – your individual upgrade – in detail.

There's no need to read this guide from A to Z as the chapters can be read individually. Simply use the search function to navigate quickly to the topics in this guide that are especially interesting for you.

New Features - The Highlights

We have been working on our Studio's interface, releasing various improvements that have the goal to significantly support the users in their daily work and to make it more enjoyable.

Among them:

- The Workflow App, which opens fullscreen in a new browser tab, and in the new Studio design.

With the Workflow App, tasks such as localization and publication have both more room and a cleaner work area. This new user interface also gives us enough space for adding new features in the near future which will translate into greatly improved UX and productivity.

- New, exciting visual changes, illustrations, and animations, for example a login screen that comes in different variations based on your individual location and preferences.
- The new Preferred Site Selector, which enables an improved and intuitive locale and name sorting.
- Redesigned Toasts and Notifications, that minimize distractions and help focusing on important messages.
- The Content Issue Search, to tidy up your content repository, for example by increasing issue visibility through custom dashboard widgets in Studio. A healthy content repos-

itory ensures flawless processing of editorial workflows. But most importantly, it improves the browsing and shopping experience of your customers.

- CKEditor 5 as the future foundation for Rich Text editing, to enable new features such as *Search and Replace* and *Source Editing* and improve the UX and DX with a modern technical foundation. The CKEditor 5 is currently available as a developer preview.

Technical Changes - Overview

Besides these new features, CoreMedia has done a lot of work behind the scenes. Updated libraries, an improved developer experience, removal of outdated functionalities, and more. While these improvements will ease your life in the long run, they require immediate upgrade tasks.

The following list shows the most important changes in terms of the upgrade effort, while [Chapter 6, Detailed Overview of Changes \[35\]](#) lists all changes in detail. See the <http://bit.ly/cmcc-11-supported-environments> document for the list of the supported databases, operating systems, and browsers of this release.

- Studio client is implemented in TypeScript instead of ActionScript/MXML

TypeScript is a very popular programming language with an active community. There are many experienced developers that will be able to support you with developing your Studio extensions in the future. You first have to convert your existing extensions into TypeScript. CoreMedia provides the necessary tools for this upgrade.

- Introduction of new Commerce Hub/Adapter API v2

A more capable and cleaned up API for Commerce adapters is introduced and required. If you have a custom Commerce adapter or have customized one of our default adapters, you have to migrate your existing code to the v2 API. Users of our out-of-the-box adapters need to make sure to use a v2 adapter and update the adapter configuration.

- Upgrade to Spring Boot 2.5

In order to benefit from the improvements and new features of Spring Boot 2.5 you have to do some changes in your custom code, especially if you have implemented your own Spring Web MVC handlers.

- Removal of the CoreMedia Watchdog/Probedog component

The Watchdog/Probedog component made by CoreMedia is outdated and can be replaced by modern concepts like JMX and Spring Boot actuators.

- Removal of LiveContext HCL legacy extension

The LiveContext HCL Blueprint extension has been removed. Existing customizations should be replaced on basis of the Commerce Hub architecture.

- Removal of the Chef deployment

In times of container technologies and container orchestration, Chef is an outdated technology and providing an idempotent deployment process with Chef is far more

complex and error prone, than using state of the art container technologies. CoreMedia container images are proven to function well in any container runtime environment.

- A new container image build process

Container images are now build using the Google Jib Maven plugin. This plugin is designed to build container images without the use of Docker, providing a more secure way to build container images. It also provides reproducible builds with identical image digests on consequent builds on the same source state.

3. Upgrade Tasks

This chapter contains detailed information about the necessary upgrade steps for each component.

- [Section 3.1, “Prerequisites for the Upgrade” \[17\]](#) provides an overview of the software you need to install in order to build the new workspace.
- [Section 3.2, “General Upgrade Strategy” \[19\]](#) describes the general steps required to update your v10 workspace. The upgrade details are described in the other subsections.
- [Section 3.3, “Check your Supported Environments” \[21\]](#) provides information on changed supported environments, such as databases.
- [Section 3.4, “Removing Outdated Components” \[22\]](#) describes how you can remove components that are no longer supported in *CoreMedia Content Cloud* v11. Therefore, reducing the conflicts when upgrading the workspace via Git.
- [Section 3.5, “Backup Content Server Databases” \[23\]](#) references schema changes of the Content Server databases.
- [Section 3.6, “Upgrading your Workspace with Git” \[24\]](#) describes how you can merge the 11.2110 workspace into your v10 Blueprint workspace using Git.
- [Section 3.7, “Upgrading Custom Blueprint Code” \[28\]](#) describes how you can merge the 11.2110 workspace into your v10 Blueprint workspace using Git.
- [Section 3.8, “Upgrading LiveContext Integration Features to Commerce Hub” \[29\]](#) describes how you migrate “old” LiveContext features to the new Commerce Hub architecture.
- [Section 3.9, “Upgrading the Build and Deployment Process” \[30\]](#) describes the migration of an old build and deployment process to the *CoreMedia Content Cloud* v11 build and deployment.

3.1 Prerequisites for the Upgrade

First of all, in order to upgrade to *CoreMedia Content Cloud* v11, you should start from the latest *CoreMedia Content Cloud* v10 version.

The list below gives a detailed overview of the software you must have installed prior to the upgrade.

- **Git >= 2.25.0**

The guide uses the Git command **restore** which is only available since Git version 2.25.0.

- **NodeJS 16.x**

In order to compile the Studio Client workspace and all frontend related components, you need to install NodeJS in the latest version 16.x.

- **pnpm 6.x**

In order to compile the Studio Client workspace and all frontend related components, you need to install pnpm as the packet manager. Simply call `npm install -g pnpm@6` on the commandline.

- **Configuring Access to CoreMedia npm Repository**

In *CoreMedia Content Cloud* v11 CoreMedia provides an npm registry to provide CoreMedia npm packages. To access this registry, you need a GitHub user token and need to set the credentials. The GitHub user needs to be a member of the coremedia-contributions organisation in GitHub. If in doubt, contact CoreMedia support to validate your permissions.

1. Creating a GitHub Token

Create a GitHub token as described in <https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>. with the following parameters:

- read:packages
- read:org
- read:user

2. Configuring your npm Repositories

- a. Configure the npm repositories with the following pnpm calls:

```
pnpm config set @coremedia:registry https://npm.coremedia.io
pnpm config set @jangaroo:registry https://npm.coremedia.io
```

b.

```
pnpm login --registry=https://npm.coremedia.io
```

You will be asked for user and password. Enter your GitHub username (all lowercase) and the created GitHub token as the password.

3.2 General Upgrade Strategy

This section describes the general steps for an upgrade. The details will be described in the following sections.

The major technical change between v10 and v11 is that *Studio Client* has been migrated from ActionScript/MXML and Maven to TypeScript and npm. While the details of this change are discussed in [Section 7.1, “Studio Client in TypeScript/npm” \[98\]](#), there is an impact on the overall upgrade strategy which is described in the following.

1. Upgrade your v10 workspace to 2107.3 (or a newer AMP, when available) by merging in the corresponding branch of CoreMedia's Blueprint workspace and follow the release notes as usual. 2107.3+ is the mandatory starting point, prior releases will not work.
2. Apply the *Studio Client* workspace conversion tool to your workspace as described in [Section 7.2.1, “Converting Studio Client to TypeScript” \[100\]](#). This replaces almost all files under `apps/studio-client` by their TypeScript/npm counterpart. Build this state of the workspace (see [Build the Converted Workspace \[104\]](#)). In most cases, this should succeed without fatal errors (non-fatal type errors are expected). As a smoke test and checkpoint, try to start *Studio Client* with your customizations (see [Start Converted Studio Client](#)). Note that you are still on v10 2107, only that all *Studio Client* code is now in TypeScript and it is built by `pnpm`.
3. Before you can upgrade to v11 [2110], you must ignore-merge CoreMedia Blueprint's branch `2107.3-ts`. This tells Git that your workspace has already been converted and ignores all changes that result from converting an unchanged CoreMedia Blueprint workspace. For details see [Section 3.6, “Upgrading your Workspace with Git” \[24\]](#).

In the Pre-Release, CoreMedia does not provide a Blueprint GitHub branch and thus also no `2107.3-ts` branch. [Section 3.6, “Upgrading your Workspace with Git” \[24\]](#) contains instructions how to create such a branch for the Pre-Release.

4. Remove all components (for example, Optimizely, Aurora B2B) from your workspace, that are no longer supported in *CoreMedia Content Cloud* v11. This way, you will reduce the number of conflicts when you merge the CMCC v11 workspace. See [Section 3.4, “Removing Outdated Components” \[22\]](#) for details.
5. Now, merge in CoreMedia Blueprint Branch 2110.1. This may result in merge conflicts and issues where the new major release introduced breaking changes. To resolve these issues, read this document and all applicable release notes for *Studio Client* carefully. In the workspace, you find documentation of the new command lines to build and watch *Studio Client* (`apps/studio-client/README.md`).

When all your customizations work again, you most likely still experience type errors when building the *Studio Client* workspace, and your TypeScript-capable IDE should also report such errors. You can ignore these non-fatal errors for now, but we recom-

mend fixing them to improve code quality, find bugs and ease maintenance. Section 4 [106] discusses in detail how to fix type errors.

3.3 Check your Supported Environments

CoreMedia has removed support for some environments and added support for others. Check [Chapter 4, *Changes in Supported Environments* \[31\]](#) for the changes. If necessary, update your environmental software.

3.4 Removing Outdated Components

Some components, such as the Optimizely integration, are no longer supported in *CoreMedia Content Cloud* v11. Remove these components from your v10 workspace in order to reduce conflicts with the v11 workspace.

Components to Remove

- Optimizely integration
- Aurora B2B

3.5 Backup Content Server Databases

The database schema of the Content Server databases has changed, see [Section 6.6.1, “Database Schema” \[66\]](#) for details.

It is not possible to start a Content Server of a previous release on the database after the new Content Server has adapted the database tables. CoreMedia strongly advises to make a backup of the Content Server database before upgrading.

3.6 Upgrading your Workspace with Git

In principle, you have two ways to migrate your workspace to *CoreMedia Content Cloud* v11. You can either install the new workspace and move all your customizations into the workspace or you merge the new workspace into your v10 workspace. The first approach is a lot of manual work and only recommended if you have few customizations in v10, or if most of your customizations became a product feature in v11. Especially for *Studio Client* customizations, manual conversion to v11 is cumbersome, since ActionScript/MXML has been replaced by TypeScript and Maven by npm. Thus, we recommend the second approach, which is described in the following.

Migration Steps

Figure 3.1, “ Update your *CoreMedia Content Cloud* v10 workspace to v11 ” [24] shows how you migrate your workspace to *CoreMedia Content Cloud* v11 with as little conflicts as possible.

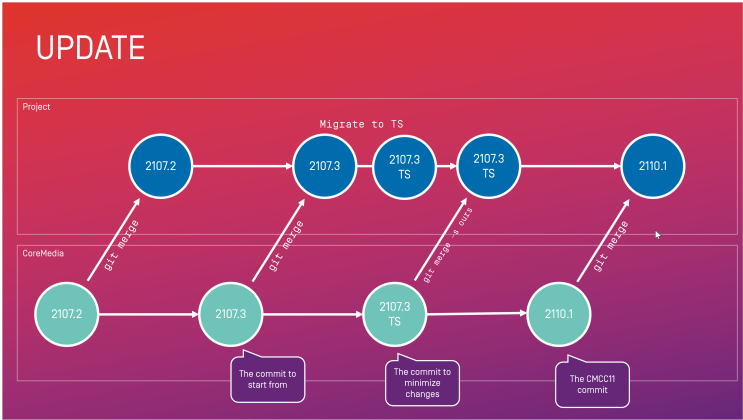


Figure 3.1. Update your *CoreMedia Content Cloud* v10 workspace to v11

The following step-by-step instructions use a Linux shell and Git as VCS, but the process works as well under Windows and with any other VCS that supports branching and merging. Using another VCS works analogously, you only need to find out the corresponding commands to create and change branches, check-in files, and merge branches.

CAUTION

Under Windows, for all given command lines in this guide, forward slashes in file names must be replaced by backslashes.

When using PowerShell, additional quoting may be necessary, so using the standard command prompt is recommended.



1. Preparing Your Workspace

First, upgrade to AEP 2107, minimum AMP 2107.3, following the usual upgrade process. In the following, 2107.3 is used as the version number you are starting from. Please replace it accordingly if you are already on a later AMP version (2107.4, 2107.5, ...). As the target version, you can keep 2110.1, but you can also try the last 2110 AMP, 2110.3. Since AMP release only contain few, non-breaking changes and this is only an intermediate step to reach the latest v11 release, it does not make such a big difference which 2110 AMP you use, but in rare cases, using another AMP may reduce merge conflicts.

Make sure your workspace is in a clean VCS state:

```
$ git status
nothing to commit, working tree clean
```

Create a new upgrade branch, called `upgrade-to-cmcc11` here:

```
$ git checkout -b upgrade-to-cmcc11
```

2. Convert Studio Client to TypeScript

Follow the detailed instructions to convert *Studio Client* to TypeScript and *npm* described in [Section 7.2.1, "Converting Studio Client to TypeScript" \[100\]](#).

3. Creating a v11 CoreMedia Vendor Branch

To create the optimal version history for the three-way-merge of your Blueprint customizations and CoreMedia's changes, the best approach is to create a "vendor branch" in your Git repository. Fortunately, this is not too complicated.

Create the vendor branch named `coremedia-2110` off the 2107.3 commit:

```
$ git checkout -b coremedia-2110 cmcc-10-2107.3
```

Then, convert the original 2107.3 Blueprint workspace as you just converted your customized workspace, as described in [Section 7.2.1, "Converting Studio Client to TypeScript" \[100\]](#). Since all modules already contain an `extNamespace`, the first task ([Add extNamespace Property](#)) can be left out, and running Studio Client ([Start](#)

Converted Studio Client] is optional, as there are no errors to expect. It can still be useful as a checkpoint to prove everything went well. The important part is to commit the resulting state and tag it with `cmcc-10-2107.3-ts`:

```
$ git tag cmcc-10-2107.3-ts
```

Download version 2110.1 of the Blueprint workspace from the CoreMedia Website:

<https://releases.coremedia.com/cmcc-11/artifacts/2110.1/cmcc-11-blueprint-workspace.zip>

Download the archive, then replace the whole Blueprint workspace by the unzipped contents of that archive:

```
$ git rm -r .
unzip $DOWNLOAD/cmcc-11-blueprint-workspace.zip
git add -A
git commit -m "CoreMedia Blueprint 2110.1"
```

Finally, switch back to your upgrade branch:

```
$ git checkout upgrade-to-cmcc11
```

4. Ignore-Merge Converted CoreMedia Blueprint Workspace

After your customized v10 workspace has successfully been converted to TypeScript, you can continue with the actual upgrade to v11. Before merging with the v11.2110.1 workspace provided by CoreMedia, however, you must ignore-merge the original state of 2107.3, converted to TypeScript, to allow the VCS to distinguish your custom changes from changes caused by the TypeScript conversion.

Ignore-merge the vendor branch into your upgrade branch:

```
$ git merge -s ours cmcc-10-2107.3-ts
```

5. Remove Abandoned Components

Remove all components (for example, Optimizely, Aurora B2B) from your workspace, that are no longer supported in *CoreMedia Content Cloud* v11. This way, you will reduce the number of conflicts when you merge the CMCC v11 workspace. See [Section 3.4, "Removing Outdated Components" \[22\]](#) for details.

6. Merge with v11.2110.1 Workspace

Merge the vendor branch into your upgrade branch:

```
$ git merge coremedia-2110
```

Depending on how many and which files were changed in your workspace and on the vendor branch, there will be a number of merge conflicts that need to be resolved.

7. Build the Merged Workspace

When merging is finished successfully, continue by building the Maven workspace:

```
$ mvn clean install -DskipTests -Pdefault-image
```

You may encounter further errors resulting from incompatibilities between your customizations and v11 (breaking) changes. Consult all sections of this Upgrade Guide as well as all relevant release notes to resolve these issues.

Again, *Studio Client* needs special attention. While conversion to TypeScript is mostly automated, there are some tasks to perform after the merge. Please continue in [Section 7.2.2, “ Completing Studio Client v11 Upgrade ” \[105\]](#).

3.7 Upgrading Custom Blueprint Code

Most of the changes in *CoreMedia Content Cloud* v11 are usually handled by the steps described in the previous chapters. There will however be custom code or even custom applications in your workspace that you will have to upgrade manually. To identify the changes that affect your custom extension, all of them are listed in detail in [Chapter 6, Detailed Overview of Changes \[35\]](#).

3.8 Upgrading LiveContext Integration Features to Commerce Hub

In case you have commerce extensions based on legacy LiveContext APIs, you should upgrade them and make sure everything fits into the Commerce Hub architecture.

The CoreMedia Commerce Hub controls the communication of CoreMedia apps with commerce systems by defining a vendor agnostic API covering the most common eCommerce features and providing a default client-server implementation of this API.

The client part of the CoreMedia Commerce Hub is named generic client. The server part is named adapter service. Adapter services are vendor-specific extensions of the base adapter which itself defines the Commerce Hub API and serves as a runtime environment controlling the communication between generic client and commerce system.

There is no general How-to Guide or manual about how to migrate your extension code, because it depends a lot on your customization itself.

For example, if you are running a blueprint extension which adds commerce-specific links to your CAE, you should consider to move the commerce-related code to the commerce adapter. The CAE extension itself should never communicate with the commerce system directly. Instead, the generic client library runs in the CAE and reads all commerce link templates from the commerce adapter again and provides them to the CAE linkschemes.

To learn more about the Commerce Hub architecture in general, have a look at the reference manual [Custom Commerce Adapter Developer Manual](#)

3.9 Upgrading the Build and Deployment Process

CoreMedia Content Cloud v11 has some changes that affect the build and deployment process. Adapt your processes accordingly.

- The *dockerfile-maven-plugin* has been replaced with *google-jib-plugin*.. See [Section 6.12.5, “Replacing dockerfile-maven-plugin with google-jib-plugin” \[85\]](#) for details.
- *npm* has been replaced with *pnpm*. See [Section 6.13.1, “Using pnpm Instead of Yarn” \[93\]](#) for details.
- The *Watchdog/Probedog* tools have been removed. See [Section 6.12.11, “Removed Watchdog/Probedog” \[90\]](#) for details.
- The Chef deployment has been removed. See [Section 6.12.10, “Removed Chef Deployment” \[90\]](#) for details.
- Studio Client and Frontend have to be build separately. They will not be build with the Maven call.
- The Blueprint has got a new POM parent, see [Section 6.12.3, “New Blueprint Parent” \[84\]](#) for details.
- Some Blueprint group IDs and versions have been changed, see [Section 6.12.4, “Consolidating Blueprint Group IDs and Versions” \[85\]](#) for details.

4. Changes in Supported Environments

This chapter lists third-party components that are no longer supported by *CoreMedia Content Cloud* or which have been added.

4.1 Added Support

4.1.1 SAP Commerce Cloud 2105 Support

With *CoreMedia Content Cloud* v11, CoreMedia supports SAP Commerce Cloud 2105 by default. If you need support for an earlier SAP Commerce version, please contact us.

4.1.2 MongoDB 5.0 Support

MongoDB has been updated to version 5.0.2 for the Docker Deployment.

4.2 Removed Support

4.2.1 HCL Commerce 8.0 Support

HCL Commerce 8.0 is no longer supported with *CoreMedia Content Cloud* v11 by default. Only HCL Commerce 9.0 (Aurora B2C based) and 9.1 (Emerald/Headless with Elastic Search) are supported in *CoreMedia Content Cloud* v11 out-of-the-box. If project approval is required for the upgrade, please contact CoreMedia.

4.2.2 Removed Database Support

Support for the following databases has been removed:

- IBM DB2 in all versions is not supported anymore.
- MySQL 5.7
- PostgreSQL 9.6
- MongoDB 3.6

4.2.3 Removed Server Platforms

Support for Microsoft Windows Server 2016 Standard has been removed.

4.2.4 Internet Explorer 11 Support for Studio

Support for Internet Explorer 11 for *CoreMedia Studio* has been removed.

5. Removed Components and Features

This section lists components and features which have been removed in *CoreMedia Content Cloud*. You will find detailed information in the other sections.

- Some API components have been removed. See [Section 6.2, "API Changes" \[39\]](#) for a list of API changes.
- Some supported environments have been removed, see [Chapter 4, *Changes in Supported Environments* \[31\]](#) for details.
- CoreMedia Watchdog/Probedog, see [Section 6.12.11, "Removed Watchdog/Probedog" \[90\]](#) for more details.
- LiveContext HCL and B2B with Aurora B2B site, see [Section 6.11.5, "LiveContext HCL and B2B Removal" \[80\]](#) for more details.
- Chef deployment, see [Section 6.12.10, "Removed Chef Deployment" \[90\]](#) for details.
- Commerce "Candy Shop" developer setup, see [Section 6.12.7, "Commerce "Candy Shop" Developer Setup Removed" \[88\]](#) for more details.
- Content Hub and Feedback Hub adapters have been removed as extensions and added as plugins, see [Section 6.12.6, "Content Hub and Feedback Hub Adapters as Plugins" \[87\]](#) for more details.
- Optimizely integration, see [Section 6.12.9, "Removed Optimizely Integration" \[90\]](#) for more details.

6. Detailed Overview of Changes

This chapter provides an overview of the improvements in *CoreMedia Content Cloud* v11 over *CoreMedia Content Cloud* v10.

- **Section 6.1, “Updated Third-Party Libraries” [36]** lists updated third-party libraries.
- **Section 6.2, “API Changes” [39]** lists all API changes in tables and describes some other API changes of *CoreMedia Content Cloud* v11 in separate sections.
- **Section 6.3, “Studio Client Changes” [51]** describes changes in the Studio Client.
- **Section 6.4, “Richtext Editor Changes” [57]** describes changes to the richtext editor in the Studio Client.
- **Section 6.5, “Studio Server Changes” [64]** describes changes in the Studio Server.
- **Section 6.6, “Content Server Changes” [66]** describes changes in the Content Servers.
- **Section 6.7, “Workflow Server Changes” [68]** describes changes in the Workflow Server.
- **Section 6.8, “Content Application Engine Changes” [70]** describes changes in the Content Application Engine.
- **Section 6.9, “Feeder and Search Engine Changes” [71]** describes changes in the Feeder and the Search Engine.
- **Section 6.10, “Personalization Changes” [75]** describes changes in the *CoreMedia Content Cloud* v11 personalization features.
- **Section 6.11, “Commerce Integration Changes” [76]** describes changes in the different CoreMedia commerce integrations.
- **Section 6.12, “Blueprint Changes” [84]** describes changes in the *CoreMedia Blueprint*.
- **Section 6.13, “Frontend Workspace Changes” [93]** describes changes in the Frontend workspace. This also includes frontend related topics in the *CoreMedia Blueprint*.
- **Section 6.14, “Command Line Tool Changes” [96]** describes changes in the command line tools of *CoreMedia Content Cloud*.

A complete list of changes is available on the <https://documentation.coremedia.com/cm-cc-11/release-notes> pages.

6.1 Updated Third-Party Libraries

This section lists all updated third-party libraries in *CoreMedia Content Cloud* v11.

6.1.1 Spring Boot 2.5 and Related Third-Party Libraries

In order to benefit from the improvements of the latest version of the Spring framework and to prepare for making use of the new features, various 3rd party libraries have been updated.

- Spring Boot 2.5.6
- Spring Data 2021.0.6
- Spring Framework 5.3.12
- Spring Security 5.5.3
- AssertJ 3.19.0
- AspectJ 1.9.7
- Caffeine 2.9.2
- Classmate 1.5.1
- Commons dbcp2 to 2.8.0
- Esapi 2.2.3.1
- Gson 2.8.8
- Hibernate 5.4.32.Final
- Hibernate Validators 6.2.0.Final
- Jackson 2.12.5
- Janino 3.1.6
- Logback 1.2.6
- Micrometer 1.7.5
- MySQL 8.0.27
- Netty 4.1.69.Final
- Postgres 42.2.24
- Slf4j 1.7.32
- Tomcat 9.0.54

- XMLUnit 2.8.3

For more details on the changes in the Spring Boot 2.5 and 2.4 releases, see the following locations:

- [Spring Boot 2.5 release notes](#)
- [Spring Boot 2.4 release notes](#)
- [Spring 5.3 GA announcement](#)
- [Spring Security 5.5 GA announcement](#)

Spring Web MVC now treats `null` arguments as missing, see <https://github.com/spring-projects/spring-framework/wiki/Upgrading-to-Spring-Framework-5.x#web-applications> and adapt your handlers accordingly.

Due to changes in the `esapi` library some unused methods of its `SecurityConfiguration` class had to be removed. You will most likely not recognize this change because the usage of this methods was not supported.

6.1.2 Apache Tika 2.1.0 and Related Third-Party Libraries

Apache Tika has been updated to new major version 2.1.0. As part of this change, the following transitive dependencies of Apache Tika have also been updated to match versions used by Tika:

- `com.adobe.xmp:xmpcore`: 6.1.11
- `com.drewnoakes:metadata-extractor`: 2.16.0
- `com.rometools:rome`: 1.16.0
- `commons-io:commons-io`: 2.10.0
- `org.apache.james:apache-mime4j-core`: 0.8.4
- `org.apache.james:apache-mime4j-dom`: 0.8.4
- `org.bouncycastle:bcmail-jdk15on`: 1.69
- `org.bouncycastle:bcprov-jdk15on`: 1.69
- `org.ow2.asm:asm`: 9.2

If you use these libraries in project code, please check their respective release notes for changes and upgrade information. No changes were necessary in the CoreMedia Blueprint for these updates.

Apache Tika restructured its Maven artifacts and split up the `tika-parsers` artifact from previous versions into separate artifacts. If you use Tika in project code, you may have to adapt Maven dependencies accordingly.

Apache Tika also removed some deprecated metadata keys [TIKA-1974]. If you have configured the *Content Feeder* or *CAE Feeder* to extract metadata values with configuration properties `feeder.tika.append-metadata` or `feeder.tika.copy-metadata`, then make sure that the configured metadata keys are still supported by Tika. You can find metadata keys supported by Tika in its API documentation, for example in interface `org.apache.tika.metadata.TikaCoreProperties`.

6.2 API Changes

This section contains changes in the CoreMedia APIs. You will find more API changes in the component specific subsections. However, the tables in [Section 6.2.1, “Overview of Changed API Elements” \[39\]](#) contain all API

6.2.1 Overview of Changed API Elements

This section lists all API changes ordered by package name.

com.coremedia.blueprint.base.links.impl

Name	Type	Change
Removal		
PropertiesRuleProvider	Class	AbsoluteUrlPrefixRuleProvider
SitesRuleProvider	Class	AbsoluteUrlPrefixRuleProvider

Table 6.1. Changed elements in com.coremedia.blueprint.base.links.impl package

com.coremedia.blueprint.cae

Name	Type	Change
Removal		
richtext.filter.PDivUntanglingFilter	Class	EmbeddingFilter
web.i18n.Setting sPageResourceBundleFactory	Class	If you still like it, keep it in your project code. As an alternative, it is recommended moving resource bundles to resourceBundles, and using LinklistPageResourceBundleFactory instead.
sitemap.SitemapHandler #URI_PATTERN	Field	Replaced with UR_PATTERN_SITEMAP

Name	Type	Change
sitemap.SitemapHandler #ACTION_NAME	Field	No replacement
sitemap.SitemapHandler #SITEMAP_PATH	Field	No replacement

Table 6.2. Changed elements in com.coremedia.blueprint.cae package

com.coremedia.blueprint.themeimporter

Name	Type	Change
Removal		
ThemeImporterImpl #legacyExternalLinkToDocumentName	Method	If you still like it, keep it in your project code. Otherwise, it is recommended using <code>stringToDocumentName</code> instead.

Table 6.3. Changed elements in com.coremedia.blueprint.themeimporter package

com.coremedia.cap.common.infos

Name	Type	Change
Removal		
CapSystemInfo#getInstallationPath	Method	No replacement

Table 6.4. Changed elements in com.coremedia.cap.common.infos package

com.coremedia.cap.multisite

Name	Type	Change
Removal		

Name	Type	Change
Site #getManagerGroupName	Method	getManagerGroupNames().stream().findFirst().orElse(""). However, mind that it is better respecting all group names returned by getManagerGroupNames().
ContentObjectSiteAspect #getIgnoreUpdates()	Method	isIgnoreUpdates()

Table 6.5. Changed elements in com.coremedia.cap.multisite package

com.coremedia.cap.transform

Name	Type	Change
Removal		
undoc.BlobHelper	Class	No replacement
TransformImageService #transformWithDimensions(Content content, Blob data, Transformed Blob operationsForTransformation, String variantName, String extension, Integer width, Integer height)	Method	#transformWithDimensions(Content, String, String, Integer, Integer)
TransformImageService #getTransformationOperations(Content, String, Map<String, String>)	Method	#getTransformationOperations(Content, String) which also resolves the given transformation with the help of ContentOperationsResolver.

Table 6.6. Changed elements in com.coremedia.cap.transform package

com.coremedia.cmdline

Name	Type	Change
Removal		
AbstractUAPIClient #enableOutVerbose	Method	BaseCommandLineClient#enableVerboseLogging
CommandLineParameters #addOptionValidated	Method	Use options.addOption(option) instead, where "options" and "option" refer to the first two arguments of addOptionValidated.
BaseCommandLineClient #addOptionValidated	Method	Use options.addOption(option) instead, where "options" and "option" refer to the first two arguments of addOptionValidated.
management.AbstractVersionSelector #addOption	Method	If you still use it in your project code, switch to options.addOption(option), where "options" and "option" refer to the arguments of addOption().

Table 6.7. Changed elements in com.coremedia.cmdline package

com.coremedia.common.util

Name	Type	Change
Removal		
Consumer	Class	java.util.function.Consumer
Function	Class	java.util.function.Function
Predicate	Class	java.util.function.Predicate
Consumers	Class	No replacement
Functions	Class	No replacement
Deprecation		

Name	Type	Change
Predicates	Class	

Table 6.8. Changed elements in com.coremedia.common.util package

com.coremedia.ldap

Name	Type	Change
Removal		
UserProvider2	Class	UserProvider
LdapMember #getOrganizationalUnit	Class	No replacement
LdapUserProviderConfigura tionProperties #SupportMemberOus	Class	UserProvider
Deprecation		
OrganizationalUnit	Inter- face	

Table 6.9. Changed elements in com.coremedia.ldap package

com.coremedia.objectserver.web

Name	Type	Change
Removal		
links.UriComponentsHelper #addQueryParameters	Meth- od	UriComponentsBuilder#queryP arams (MultiValueMap)
ContentBlobHandlerBase #URIVARIABLE_VERSION	Field	The result map of Content BlobHand lerBase#buildBlobLink contains no entry

Name	Type	Change
		for this key any longer. If you still use it in your project, migrate to <code>ContentBlobHandlerBase#URIVARIABLE_ETAG</code> .
Deprecation		
<code>ContentBlobHandlerBase</code>	Class	No handlers use it any longer. If you still need it in your project code, please contact the CoreMedia support.

Table 6.10. Changed elements in `com.coremedia.objectserver.web` package

com.coremedia.translate

Name	Type	Change
Removal		
<code>item.DefaultContentToTranslateItemTransformer</code>	Constructor	Use the three-argument constructor instead: <code>DefaultContentToTranslateItemTransformer(Function, Collection, boolean)</code> . For the same behavior as before, set the last parameter to <code>true</code> .
<code>xliff.exporter.XliffExporter</code>	Field	See Section 6.2.7, “Removal of Deprecated Version of XliffExporter” [48].
<code>xliff.exporter.XliffExportElement</code>	Field	See Section 6.2.7, “Removal of Deprecated Version of XliffExporter” [48].
<code>xliff.exporter.XliffExportElementBuilder</code>	Field	See Section 6.2.7, “Removal of Deprecated Version of XliffExporter” [48].
<code>xliff.exporter.XliffExportElementCollectionBuilder</code>	Field	See Section 6.2.7, “Removal of Deprecated Version of XliffExporter” [48].
<code>xliff.exporter.CapXliffExportException</code>	Field	See Section 6.2.7, “Removal of Deprecated Version of XliffExporter” [48].

Name	Type	Change
<code>xliff.CapXliffException</code>	Field	See Section 6.2.7, “Removal of Deprecated Version of XliffExporter” [48] .

Table 6.11. Changed elements in `com.coremedia.translate` package

com.coremedia.workflow.plugin

Name	Type	Change
Addition		
<code>FinalAction</code>	Class	See Section 6.7.2, “Final Workflow Actions” [68] for more details.

Table 6.12. Changed elements in `com.coremedia.cap.transform` package

com.coremedia.xml

Name	Type	Change
Removal		
<code>MarkupUtil</code> <code>#isEmptyMarkup</code>	Method	Use <code>EmptyRichtext</code> (for <code>CoreMediaRichtext</code> only) or <code>hasText</code> (for markup of arbitrary grammars) instead.

Table 6.13. Changed elements in `com.coremedia.xml` package

hox.corem.server.media

Name	Type	Change
Addition		

Name	Type	Change
MediaStore #upload	Meth- od	See Section 6.6.2, “MediaStore API” [66] for details.

Table 6.14. Changed elements in `hox.corem.server.media` package

6.2.2 Removal of Deprecated Legacy Rule Providers

The following legacy Rule Providers have been removed from the Java API and the `bp-base-links-legacy.xml` file:

- `com.coremedia.blueprint.base.links.impl.PropertiesRuleProvider`
- `com.coremedia.blueprint.base.links.impl.SitesRuleProvider`

You probably do not use the classes directly in your project code. Instead, check if you still import the legacy configuration file in the `blueprint-links.xml` file in the `cae-base-lib` module, and eventually migrate to the `AbsoluteUrlPre fixRuleProvider`. In case of questions, please contact the CoreMedia support.

6.2.3 Removal of Deprecated LdapMember#getOrganizationalUnit

CoreMedia has finally deleted `LdapMember#getOrganizationalUnit` that has never been used.

If you have implemented the `LdapMember` interface, you can delete the method. You must at least delete the `@Override` annotation.

CoreMedia also deleted `LdapUserProviderConfigurationProperties#supportMemberOus`, which was used to activate `LdapMember#getOrganizationalUnit` (and which was false by default).

With the deletion of `LdapMember#getOrganizationalUnit`, the interface `OrganizationalUnit` became obsolete and has therefore been deprecated.

6.2.4 Implementations of Comparable: Added Type Arguments

For several parts of the public API, the implementation of the `Comparable` interface now uses correct generic type arguments. As part of this change, following interfaces/classes have been updated by adding type parameter.

- `com.coremedia.cap.user.Group`
- `com.coremedia.cap.user.User`
- `com.coremedia.cap.workflow.Process`
- `com.coremedia.cap.workflow.ProcessDefinition`
- `com.coremedia.cap.workflow.Task`
- `com.coremedia.cap.workflow.TaskDefinition`
- `com.coremedia.workflow.WfValue`
- `com.coremedia.workflow.common.expressions.Aggregate`

6.2.5 Removal of Deprecated SitemapHandler#URI_PATTERN

The deprecated Sitemap URI Pattern `/service/sitemap/{sitemapPath}` and the related handler have been removed from `com.coremedia.blueprint.cae.sitemap.SitemapHandler` (module `cae-base-lib`). The same applies to private members `ACTION_NAME` and `SITEMAP_PATH` which are now obsolete.

Use `/service-sitemap-{siteId}-{sitemapFile}` instead. It is defined by `SitemapHandler.URI_PATTERN_SITEMAP`.

If you still require the old mapping or members, ensure to keep your existing `SitemapHandler` in *CoreMedia Blueprint*.

6.2.6 Removal of Consumer, Function, Predicate and Related Classes

The following deprecated classes have been deleted from the Java API:

- `com.coremedia.common.util.Consumer`, `com.coremedia.common.util.Function`
- `com.coremedia.common.util.Predicate`

All usages have been replaced with the according JDK classes of package `java.util.function`. The APIs of the consumer and the function classes are identical.

If you have implemented own consumers or functions, you only have to change the `implements` clauses from `com.coremedia.common.util.[Consumer|Function]` to `java.util.function.[Consumer|Function]`.

If you have own predicates, you must change the `implements` clause and rename the `include` method to `test`.

The deprecated `Functions` and `Consumers` utility classes from the same package have also been deleted, since they make no sense without the actual classes, and were trivial anyway. The `Predicates` class still exists, but has been deprecated and usages can be replaced as described in its API documentation.

6.2.7 Removal of Deprecated Version of XliffExporter

The deprecated interface `com.coremedia.translate.xliff.exporter.XliffExporter` [artifact: `cap-xliff`] and its related interfaces and classes have been removed.

Use the refactored, more flexible `com.coremedia.cap.translate.xliff.XliffExporter` instead.

Changes in Detail

The following classes/interfaces have been removed:

- `com.coremedia.translate.xliff.exporter.XliffExporter`
Use `com.coremedia.cap.translate.xliff.XliffExporter` instead.
- Replaced by `com.coremedia.translate.item.TranslateItem`:
 - `com.coremedia.translate.xliff.exporter.XliffExportElement`
 - `com.coremedia.translate.xliff.exporter.XliffExportElementBuilder`

- `com.coremedia.translate.xliff.exporter.XliffExportElementCollectionBuilder`
- `com.coremedia.translate.xliff.exporter.CapXliffExportException`
- `com.coremedia.translate.xliff.CapXliffException`

Extensions to Ease Upgrade

Some classes and interfaces have been extended, to provide the same behavior as the now removed, previously deprecated `XliffExporter`.

The interface `ContentToTranslateItemTransformer`, which is required as an intermediate step to transform contents (master and derived) to translate-items accepted by the current `XliffExporter`, has been extended by using a default locale mapper, by introducing the following default methods:

- `getDefaultLocaleMapper()`, and
- `transform(masterSources, derivedTargets, strategy)`

This default is the same as it has been used by the removed `XliffExporter`.

The `XliffExportOptions` class has been extended to provide a way to specify a default comment to apply to all translate items, which don't have a comment yet (added to XLIFF as `<note>`). See `XliffExportOptions.DefaultCommentOption` and its usages for details. This implements the same behavior as the removed `XliffExporter`, where you could hand over a comment on XLIFF export – and which also got applied to all translate-items, and thus to all XLIFF `<file>` entries.

And to ease, having a base set of options to modify for example by a new default comment, `XliffExportOptions` now provides a method `builder()` to create a derived set of options from the original one.

Typical Replacement

A typical replacement for the old `XliffExporter` is similar to [Example 6.1, "Typical Modern XliffExporter Usage" \[49\]](#).

```
XliffExportOptions xliffExportOptions = XliffExportOptions.xliffExportOptions()
    .option(
        XliffExportOptions.TargetOption.TARGET_EMPTY
    )
    .option(
        XliffExportOptions.DefaultCommentOption.of("default comment")
    )
    .build();
List<TranslateItem> items = contentToTranslateItemTransformer
    .transform(
        masterContentObjects,
        derivedContents,
```

```
        TransformStrategy.ITEM_PER_TARGET
    )
    .collect(toList());

Xliff xliiff = xliiffExporter.exportXliff(
    items,
    xliiffExportOptions
);
```

Example 6.1. Typical Modern XliffExporter Usage

For details, see section [Section “XLIFF Integration”](#) in *Blueprint Developer Manual* and corresponding JavaDoc.

Replacing setXliffExportElementFilter

The former `XliffExporter` provided a method for filtering contents, which should not be added to the XLIFF file: `setXliffExportElementFilter`.

In general, such filtering should happen when creating the `TranslateItem` instances (this is a so-called pre-processing stage). For details, see [Section “XLIFF Export”](#) in *Blueprint Developer Manual*, especially [Section “XLIFF Integration”](#) in *Blueprint Developer Manual*.

Nevertheless, in contexts, where you cannot directly influence the creation of `translate-items`, a fallback exists for the new `XliffExporter`: `setTranslateItemFilter`. This applies for example to the default XLIFF download as provided in *CoreMedia Studio*. As `translate-items` only provide the information relevant for translating a given content, you may miss some details which were available in the previous pre-processing stage via `XliffExportElementFilter`. If you require more details, you may want to replace the `ContentToTranslateItemTransformer` instead, which defaults to `DefaultContentToTranslateItemTransformer`.

Generate XLIFF in Workflow Instead

The integrated XLIFF download in *CoreMedia Studio* provides only very limited control on XLIFF creation. In general, it is recommended disabling the download button for the given workflow instead and serve the XLIFF via a BLOB property of the workflow. The BLOB property is then set from a custom workflow action, which again can use the rich API as provided by the modern `XliffExporter`.



If it is about not repeating translation for unchanged properties, you may want to set `translate.item.include-unchanged-properties` to `false`, which will be taken into account when generating the `translate-items`. See [section “XLIFF Configuration Properties”](#) in *Blueprint Developer Manual*.

6.3 Studio Client Changes

This section describes the changes in the user interface of CoreMedia Studio.

6.3.1 Modern Studio Layout Always Activated

In *CoreMedia Content Cloud* v10, it was possible to optionally activate the modern *CoreMedia Studio* layout. In v11, *CoreMedia Studio* always uses the modern layout while the classic layout has been removed. In addition, the modern layout received a design update to improve your user experience.

6.3.2 Workflow App

In previous releases workflow management was handled inside Studio in the Control Room. Now, CoreMedia introduces the Workflow App, which opens fullscreen in a new browser tab, and in the new Studio design. With the Workflow App, tasks such as localization and publication have both more room and a cleaner work area, and this translates into greatly improved UX and productivity.

Accompanying the new Workflow App is a new API to add custom publication and translation workflows. The former API was targeted specifically at the Control Room in main Studio. The new API can be used to customize both, main Studio and the Workflow App.

Note that the old API is still in place. As workflows continue to be started in the Control Room, the old API can still be used to customize the start-workflow windows. However, to customize the Workflow App, one must use the new API and we recommend using it for both, main Studio and the Workflow App.

The new API is designed to be more declarative and also independent from Ext JS as the underlying UI framework. For details, please consult [Section 9.26, "Custom Workflows"](#) in *Studio Developer Manual*.

6.3.3 New Content Type Localization API

With *CoreMedia Content Cloud* v11, a new API for localizing content types has been introduced. The background for this change is that content type localizations need to work

in both the Main Studio App and the new Workflow App. The previous way of localizing content types (solely via resource bundle keys) has some drawbacks, so this was the opportunity to rework it. The main drawbacks are:

- The localization resource bundle keys were rarely directly used in code. Instead, usages resulted from string concatenation. This untyped access of bundle keys prevents IDE support and makes assessment of refactorings hard.
- Customizing icons was only supported in the form of style classes. No direct convenient usage of SVG icons was possible.

The new API addresses both issues but at the cost of one extra bit of configuration. The new API is described in-depth in [Section 9.5, "Document Type Model"](#) in *Studio Developer Manual*.

This upgrade section shows how a typical use case can be migrated. In addition, you will learn the usage of a converter tool that migrates existing content type localizations.

Note that the previous API for localizing content types is still in place as a fallback, but it only works for the Main Studio App. Localizing content types for the Workflow App requires using the new API, so it is recommended to use the new API for both apps to take advantage of shared code (see below).

Example Migration

With the previous API, localizing a custom content type was accomplished by overriding the resource bundle `ContentTypes_properties` with a custom resource bundle:

```
new CopyResourceBundleProperties({  
  destination:  
    resourceManager.getResourceBundle(null, ContentTypes_properties),  
  source:  
    resourceManager.getResourceBundle(null, BlueprintDocumentTypes_properties),  
})
```

The custom resource bundle typically included properties like (for a content type named "Media"):

```
Media_text: "Media Object",  
Media_toolTip: "Multimedia object",  
Media_icon: "custom-icons custom-icons--type-media",  
Media_alt_text: "Alternative Text",  
Media_alt_toolTip: "Alternative text shown in case of render failures",  
Media_alt_emptyText: "Enter an alternative text here.",  
Media_caption_text: "Caption",  
Media_caption_toolTip: "Caption of the media object",  
Media_settings.copyright_text: "Copyright",
```

```
Media_settings.copyright_toolTip: "Copyright",
Media_settings.copyright_emptyText: "Enter copyright information here.",
```

The new API still relies on resource bundles to actually localize texts. But it utilizes a new registry API to explicitly register and query content type localizations. The example from above is transferred into:

```
import typeMedia from "../icons/type-media.svg";

contentTypeLocalizationRegistry.addLocalization("Media", {
  displayName: BlueprintDocTypes_properties.Media_displayName,
  description: BlueprintDocTypes_properties.Media_description,
  svgIcon: typeMedia,
  properties: {
    alt: {
      displayName: BlueprintDocTypes_properties.Media_alt_displayName,
      description: BlueprintDocTypes_properties.Media_alt_description,
      emptyText: BlueprintDocTypes_properties.Media_alt_emptyText,
    },
    caption: {
      displayName: BlueprintDocTypes_properties.Media_caption_displayName,
      description: BlueprintDocTypes_properties.Media_caption_description,
    },
    settings: {
      properties: {
        copyright: {
          displayName:
            BlueprintDocTypes_properties.Media_copyright_displayName,
          description:
            BlueprintDocTypes_properties.Media_copyright_description,
          emptyText:
            BlueprintDocTypes_properties.Media_copyright_emptyText,
        },
      },
    },
  },
});
```

The content type icon is directly imported from an SVG file. `contentTypeLocalizationRegistry` is a global constant of type `ContentTypeLocalizationRegistry` which also allows to query for existing localizations. In addition, existing localizations can be updated with the same method `addLocalization` from above.

The Blueprint workspace already contains numerous modules using the new API under `apps/studio-client/shared/js`. It is important to note that these shared modules need to be included in both the Main Studio App as well as the Workflow App. As usual, they can be included as a direct dependency (like `@coremedia-blueprint/studio-client.blueprint-doctypes`) or as an extension dependency (like `@coremedia-blueprint/studio-client.lc`).

Tool Support

Of course it is possible to migrate content type localizations manually to the new API. But especially for larger parts it is more convenient to use a converter tool which takes care of everything, except the icons. SVG icons need to be provided manually and added to content type localization registrations where needed.

Detailed Overview of Changes | New Content Type Localization API

The primary parameter for the tool is the default-locale file of a content type localization resource bundle. The conversion result depends on whether this file belongs to an extension or not.

If the file belongs to an extension, the tool takes care of:

- Creating a new shared localization module under `apps/studio-client/shared/js/extensions`.
- Generating the new localization resource bundle.
- Generating the registration code.
- Adding the new shared module as a dependency to the [extension] module where the input file is located.
- Creating an extension module for the other apps (currently just one, that is the Workflow App).

NOTE

Afterwards, the extension tool needs to be used to (re-enable) the extension.



For example, to run the converter tool for a resource bundle `LivecontextDocTypes_properties`, you switch to the directory `apps/studio-client/tools/localization-converter` and make the following pnpm call:

```
pnpm run convert  
$WORKSPACE_ROOT/apps/studio-client/apps/main/extensions/lc/src/LivecontextDocTypes_properties.ts
```

If the file does not belong to an extension, you need to provide a name for the new shared localization module in addition to the input file. The tool then takes care of:

- Creating a new shared localization module with the given name under `apps/studio-client/shared/js`.
- Generating the new localization resource bundle.
- Generating the registration code.
- Adding the new shared module as a dependency to the module where the input file is located.

Afterwards, you need to add the new shared module as a dependency to the other apps manually. For example, to run the converter tool for a resource bundle `Blueprint DocumentTypes_properties`, switch to the directory `apps/studio-client/tools/localization-converter` and run the following code:

```
pnpm run convert
$WORKSPACE_ROOT/apps/studio-client/apps/main/blueprint/blueprintforms/src/BlueprintDocumentTypes_properties.ts
-- --sharedPackageBaseName=blueprint-doctypes
```

There are further advanced options for the tool which might be needed in some special cases.

- `additionalLocales` (defaults to `["de", "ja"]`)

Which locales, besides the default locale, the generated resource bundle should include.

- `coreVersion`

Use this option to explicitly define the version of the used CoreMedia studio-client core. If none is provided the version is derived from the dependencies of the packages containing the given properties files.

- `jangarooNpmVersion`

Use this option to explicitly define the version of the used Jangaroo npm packages. If none is provided the version is derived from the dependencies of the packages containing the given properties files.

6.3.4 New Toast API

With *CoreMedia Content Cloud* v11, a new toast API has been introduced. Former toasts have been shown directly on buttons. For example, when a search folder was created or content was published, a green bubble was shown next to the button that triggered the actions.

These bubbles have been removed from the buttons and are shown now in the lower left corner of *CoreMedia Studio*. Depending on the validation state of the toasts, they are shown in different colors.

New Toast can be shown via the `toastService` instance:

```
toastService._.showToast(title:string,  
message:string, severity: ValidationState):Toast
```

Example 6.2. Using the ToastService

Additionally, users can now disable these toasts altogether in the user preferences.

6.3.5 View Aborted Workflows in Studio

Workflow definitions for publication, translation, and synchronization workflows have been extended with a new final action `ArchiveProcessFinalAction` which archives aborted workflow processes in the same way as completed processes.

Users can now see their archived processes as finished workflows in Studio.

6.3.6 Consolidated Notifications and Running Jobs

Notification toasts, the notifications menu and the jobs window have been redesigned.

Users can now mark notifications again as “unread”. The new design did not affect the Notifications API but may result in some layout changes for custom notifications.

6.4 Richtext Editor Changes

During the lifetime of *CoreMedia Content Cloud* v11 the underlying framework for richtext editing will change from CKEditor 4 to CKEditor 5. The preparations for the new richtext editor also require some changes to the current editor.

6.4.1 The Future of Richtext Editing

CKEditor 4 has been the solid foundation for richtext editing in CoreMedia Studio for a long time. The latest version CKEditor 5 is a complete rewrite and CKEditor 4 will not receive new features anymore. It will still be available in CMCC v11, but it will only receive maintenance updates and it will be removed in the major release that follows CMCC v11.

As plugins need to be rewritten from scratch, which applies to CoreMedia CKEditor 4 plugins like for data-processing, too, CoreMedia decided to take the chance to revise the complete architecture for CoreMedia Studio integration.

Most important: The toolbar for richtext editing actions will now be a native CKEditor 5 toolbar rather than redirection via an Ext JS toolbar. You will benefit from this, as it is now straightforward adding CKEditor plugins as provided by CKEditor marketplace.

The adaptations for running CKEditor 5 in the CoreMedia CMS eco-system are provided as loosely coupled plugins. These, if possible, provide extension points (mostly in configuration layer) for several use-cases known from customer projects. For example, data-processing, responsible for mapping CoreMedia Richtext 1.0 to HTML and vice versa can now easily be extended as part of the configuration of CKEditor's ClassicEditor instance. Thus, integrating plugins like CKEditor's Highlight Plugin just requires one additional step in CoreMedia CMS eco-system: Provide a mapping from <mark> element to some representation in CoreMedia richtext and vice versa.

In one of the early AEP releases of CMCC v11 the default will change from CKEditor 4 to CKEditor 5. You can then keep the existing version. However, new features will only be implemented for the new rich text editor. As such, we discourage customizing the CKEditor 4 based rich text editor – and if required, ensure to keep a specification as it is likely, that you will have to rewrite the customizations for CKEditor 5. More information is available at <https://support.coremedia.com/hc/en-us/articles/360020791099-Roadmap-to-CKEditor-5-in-CMCC-v11>.

In the early releases of CMCC v11 you can already enable a preview of the current state of the richtext editor based on CKEditor 5 for development purposes. To simplify the switch from version 4 to 5 some changes to configuration of the richtext editor had to be implemented. Please have a look at the breaking changes, described in [Section 6.4.3, "Adapting CKEditor 4 Extensions" \[58\]](#)

6.4.2 Deprecation of CKEditor 4 Extensions

The current version of the richtext editor based on CKEditor 4 in *CoreMedia Studio* will be available during the entire lifetime of *CoreMedia Content Cloud* v11. However, new customizations should be based on CKEditor 5 because the previous editor won't be available anymore in the following major release.

6.4.3 Adapting CKEditor 4 Extensions

Till the production-ready release of the new richtext editor in *CoreMedia Studio* you must still use CKEditor 4 in production. Some classes have been moved to a new package, so you have to adapt your extensions, when you have used these classes.

In *CoreMedia Content Cloud* v11, there are only a few minor breaking changes related to the richtext editor code. The following classes have been moved to the new package `@coremedia/studio-client.main.ckeditor4-components`:

- `AddCKEditorPluginsPlugin.ts`
- `AddCKEditorPluginsPluginBase.ts`
- `CKEditor_properties.ts`
- `CoreMediaRichTextArea.ts`
- `CoreMediaRichTextAreaBase.ts`
- `CustomizeCKEditorPlugin.ts`
- `LinkAction.ts`
- `LinkActionBase.ts`
- `PasteContentAction.ts`
- `PasteContentActionBase.ts`
- `PropertiesAction.ts`
- `PropertiesActionBase.ts`
- `RemoveCKEditorPluginsPlugin.ts`
- `RemoveCKEditorPluginsPluginBase.ts`
- `RichTextAction.ts`
- `RichTextActionBase.ts`
- `RichTextActionToggleButton.ts`
- `RichTextArea.ts`

- `RichTextMenuCheckItem.ts`
- `RichTextMenuCheckItemBase.ts`
- `RichTextMenuItem.ts`
- `RichTextPropertyField.ts`
- `RichTextPropertyFieldBase.ts`
- `TableAction.ts`
- `TableActionBase.ts`
- `TeaserOverlayPropertyField.ts`
- `TeaserOverlayPropertyFieldBase.ts`
- `TeaserOverlayStyleSelector.ts`
- `TeaserOverlayStyleSelectorBase.ts`

Adjust your dependencies and imports accordingly if you are using some of the plugins, actions or components in your Blueprint code.

The deprecated abstract function `registerRichTextSymbolMapping` in `EditorContext` has been removed. Please set `symbolFontMapping` in `CKEditor-Config` via `customizeCKEditorPlugin` instead.

6.4.4 Enable Developer Preview [Optional]

Do not use CKEditor 5 based richtext editor in a production environment

Using CKEditor 5 in this early stage may or will modify your richtext data when loaded into CoreMedia Studio. This is because some valid CoreMedia Richtext 1.0 elements and attributes are not supported yet. When loaded from server, such yet unknown elements and attributes will be removed from XML.



To enable the preview of CKEditor 5 in *CoreMedia Content Cloud* v11, simply add a dependency to the `@coremedia-blueprint/studio-client.main.ckeditor5-plugin` in your Blueprint. Execute `$ pnpm add --filter "@coremedia-blueprint/studio-client.main.base-app" @coremedia-blueprint/studio-client.main.ckeditor5-plugin@1.0.0-SNAPSHOT` from the studio-client root on the command line to enable the package in the Blueprint. If you changed the project version of the Blueprint from "1.0.0-SNAPSHOT" to a different value, please adjust the command accordingly.

Mandatory code adjustments when upgrading to CKEditor 5

NOTE

If you have no custom richtext implementations or CKEditor customizations, you don't need to follow these steps at all. All the core Studio components will use the new CKEditor out-of-the-box.



If you are using richtext in your custom Studio code, you probably depend on one of these Ext JS components:

- `RichTextArea.ts`
- `CoreMediaRichTextArea.ts`
- `RichTextPropertyField`

These components still exist and can be used by changing their imports to

- `@coremedia/studio-client.main.ckeditor4-components/RichTextArea`
- `@coremedia/studio-client.main.ckeditor4-components/CoreMediaRichTextArea`
- `@coremedia/studio-client.main.ckeditor4-components/RichTextPropertyField`

Please notice, changing the imports might not suffice if you have customized the editor by using plugins:

- You will need to convert your CKEditor plugins, actions and toolbar items to CKEditor 5. To quote the [Migration from CKEditor 4 - CKEditor 5](#) documentation:

"The trickiest migration challenge to be faced may be related to custom plugins you have developed for CKEditor 4. Although their concept may stay the same, their implementation will certainly be different and will require rewriting them from scratch."

- The configuration for the default CKEditor instance in CoreMedia Studio is located in `@coremedia-blueprint/studio-client.main.ckeditor5`. You can customize this configuration to fit your needs.

- You will need to add build configurations if you need multiple CKEditor instances with different configurations.

These tasks will be explained in the following sections.

CKEditor Plugins

WARNING

Please note: The way to register different types of editors as described in this section is highly experimental and will probably change in future releases. Customers should not rely on any CKEditor Studio API for now. (Such as `ckeditorFactory` or `CKEditor5Wrapper`)



The CKEditor ecosystem has changed a lot between CKEditor 4 and CKEditor 5. CoreMedia adjusted their implementation as well and that is why customizations and plugins are now handled a little differently as well.

CKEditor plugins are now imported directly to the build configuration as npm packages. CoreMedia provides all its plugins as npm packages, and you will also have to convert your custom plugins to packages.

These packages can then be installed and imported in the `@coremedia-blueprint/studio-client.main.ckeditor5` package in the Blueprint. See the [CKEditor 5 Quickstart Guide](#) and [Section 9.6.6, "Activating and Customizing CKEditor 5 Preview"](#) in *Studio Developer Manual* to learn more about how to create your own CKEditor 5 configuration.

These configurations must be registered as a certain editor type in one of the Studio plugins:

```
init(): void {
  ckEditorFactory.registerConstructor("default", (editorType)=>{
    return new CKEditor5Wrapper((domElement:Element) =>
      initEditor(domElement));
  }, 5);
}
```

Example 6.3. Create editor

Now you can use this editor type in `RichTextPropertyFields` or `RichTextAreas`:

```
Config(RichTextPropertyField, {
  itemId: "myRichTextPropertyField",
```

```
    propertyName: ConfigUtils.asString(PROPERTY_NAME),  
    editorType: "default"  
  }},
```

Example 6.4. Use CKEditor in field

Note that you could omit the `editorType` here because it defaults to "default" anyway. Have a look at [Section 9.6.6, "Activating and Customizing CKEditor 5 Preview"](#) in *Studio Developer Manual* to learn more about registration of editor types.

Have a look at the following links to learn more about how to create your own plugins for CKEditor 5:

- [Installing plugins - CKEditor 5 Documentation](#)
- [Creating a simple plugin - CKEditor 5 Documentation](#)
- [CoreMedia/ckeditor-plugins](#)

Studio Plugins

CAUTION

The CKEditor Studio plugins for editor customizations will not work anymore. With CKEditor5 CoreMedia provides a whole new way of customizing the editor by letting you create your own build and register it in Studio.



Within the `@coremedia-blueprint/studio-client.main.ckeditor5` package, you can either change the editor's configuration, toolbar items or add own or remove unwanted plugins. Please visit CKEditor 5 documentation to learn more about how to configure CKEditor builds or write own plugins.

CAUTION

You may not remove CoreMedia specific plugins (such as the `CoreMediaStudioEssentials` plugin) to keep the editor working correctly in Studio.



The following table shows the replacements for the CKEditor Studio plugins. Please change the configuration in the Blueprint module directly if you have changed the default behavior of your editor before:

Plugin in v10	Upgrade Path
AddCKEditorPluginsPlugin	You are now able to add plugins directly to the CKEditor in the <code>@coremedia-blueprint/studio-client.main.ckeditor5</code> package.
RemoveCKEditorPluginsPlugin	See AddCKEditorPluginsPlugin. You can also remove unwanted plugins directly inside the <code>@coremedia-blueprint/studio-client.main.ckeditor5</code> package.
CustomizeCKEditorPlugin	Just like adding and removing plugins, you can also change the configuration of the CKEditor in the <code>@coremedia-blueprint/studio-client.main.ckeditor5</code> package.

Table 6.15. Replacements for Studio plugins

Multiple Configurations in Different Components

You may also want to use whole different configurations of the CKEditor in different components. This could previously be achieved by using the above plugins. If you need a different configuration for a custom component, you can also add another build configuration for a custom editor type to the `@coremedia-blueprint/studio-client.main.ckeditor5` package and export a function that exports your editor configurations.

See [Section 9.6.6, “Activating and Customizing CKEditor 5 Preview”](#) in *Studio Developer Manual* to learn more about how to create different CKEditor builds to your workspace.

6.5 Studio Server Changes

This section describes changes applied to the studio server.

6.5.1 Consolidating Group IDs and Versions

The groupId of the content-hub core artifacts has been changed from `com.coremedia.content-hub` to `com.coremedia.cms` in order to adjust to naming conventions.

6.5.2 Editorial Comments Cache Properties Removed

The configuration properties `editorial.comments.data.cache.*` have been removed.

Old Property	New Property
<code>editorial.comments.data.cache.comments-cache-size</code>	<code>cache.capacities.com.coremedia.editorial.comments.data.editorial.comments</code>
<code>editorial.comments.data.cache.qualified-cache-size</code>	<code>cache.capacities.com.coremedia.editorial.comments.data.editorial.comments.qualified</code>
<code>editorial.comments.data.cache.threads-cache-size</code>	<code>cache.capacities.com.coremedia.editorial.comments.data.editorial.comments.commentThreads</code>

Table 6.16. Changed properties for editorial.comments

6.5.3 Validators by Configuration

You can declare validators not only as Spring beans, but alternatively also by JSON configuration files now. This is supported for most core validator classes, except of those which make only sense as singletons and which are provided by default, like the `AvailableLocalesValidator`. Most Blueprint validator classes can easily be enabled for JSON configuration in the project, if needed.

See [Section “Declaration of Validators”](#) in *Studio Developer Manual* for details.

Unfortunately, the option of JSON configuration required some API changes. The validator classes have been widely stripped off their Spring features, especially `@Autowired` and `InitializingBean`. Not expecting validators to be Spring beans, but just simple POJOs, makes it much easier to instantiate them from JSON configuration files. And the Spring features that were actually used could easily be replaced by other means, like non-null constructor arguments in favor of `@Autowired`, setters and `afterPropertiesSet` checks.

The abstract base classes `PropertyValidatorBase` and `ContentTypeValidatorBase` have been replaced by the Spring-free `AbstractPropertyValidator` and `AbstractContentTypeValidator`, respectively. This affects the constructor signatures and setters of their subclasses, since mandatory configuration is required as constructor arguments now, in favor of setters. The changes are most obvious in the `ValidatorsConfiguration` class in the Blueprint.

If you want to use JSON configuration also for custom validators, the validator classes should refrain from using Spring features like `@Autowired` or `InitializingBean`. Otherwise, you must implement more complex validator factories.

6.6 Content Server Changes

This section describes changes applied to the different content servers.

6.6.1 Database Schema

The database table, `BlobData` has been enhanced with a `uuid` column. The UUID of a blob is optionally assigned and consumed internally by the contentserver.

CAUTION

It is not possible to start a Content Server of a previous release on the database after the new Content Server has adapted the database tables. You should always make a backup of the Content Server database before upgrading.



6.6.2 MediaStore API

Depending on the particular backend, a `MediaStore` implementation is possibly not able to upload blobs without a content type. Therefore, `CoreMedia` introduced a new variant of the interface method `MediaStore#upload` with an additional content type argument and deprecated the old variant. Moreover, the new method returns an `UploadResult`, which consists of the size of the blob and an optional UUID. It is a default method which delegates to the old one, so you do not need to change your `MediaStore` implementations immediately.

There is also a new variant of the `MediaStore#download` method, which receives the UUID that has been assigned to the blob by the upload method. It is a backward compatible default method.

6.6.3 Content Server Does Not Use `corem.home` Anymore

The `corem.home` directory and system property are no longer used by the Content Server.

For the Content Server, a new Spring property `cap.server.base-dir` has been introduced to control the base directory for other configuration properties when using relative paths. By default, the property is set to the value of the `user.dir` system property. For the Blueprint Docker images this is the `/coremedia` directory.

The following properties have been adapted and support relative file paths (some defaults have changed):

Property	Value Type	Default
<code>cap.server.license</code>	File/URL	license.zip
<code>cap.server.document-types</code>	File/URL patterns	classpath*:framework/doctypes/*.xml
<code>cap.server.login.authentication</code>	File/URL	classpath:coremedia-jaas.conf resp. classpath:blueprint-jaas.conf
<code>cap.server.login.bouncers</code>	File	Empty
<code>cap.server.encrypt-passwords-key-file</code>	File	etc/keys/DATABASE_NAME.DATABASE_USER.rijndael
<code>replicator.tmp-dir</code>	Directory	var/tmp

Table 6.17. Changed properties

6.6.4 Secure Blob Access

Blob access by clients is now protected against URL guessing. If you want to use components from earlier CMCC releases with new Content Servers, you must set the application property `cap.server.blob-md5-permission-check` to "false" for the new Content Servers.

6.7 Workflow Server Changes

This section describes changes applied to the Workflow Server and the deployed workflows.

6.7.1 Upgrading Workflow Definitions

CoreMedia added support for final actions to workflow definitions as described in [Section 6.7.2, “Final Workflow Actions” \[68\]](#). Predefined translation, publication, and synchronization workflows were adapted to use such final actions, so that *Studio* is now able to show aborted workflows in the UI, see [Section 6.3.5, “View Aborted Workflows in Studio” \[56\]](#) for details.

You must re-upload changed workflow definitions using the *cm upload* tool to make use of this new functionality. See [Section 3.5.4, “Upload”](#) in *Workflow Manual* for a description of the *cm upload* tool.

The standard workflows `studio-simple-publication.xml`, `studio-two-step-publication.xml`, and `/com/coremedia/translate/workflow/synchronization.xml` can be uploaded by passing their name to the `-n` option of the *cm upload* tool.

The workflow definition `translation.xml` from *Blueprint* directory `apps/workflow-server/modules/cmd-tools/wfs-tools-application/src/main/app/properties/corem/workflows` can be uploaded by passing the file name to the `-f` option of the *cm upload* tool.

The updated workflow definitions use the new `ArchiveProcessFinalAction` instead of the now deprecated `ArchiveProcess` action. It is recommended to replace usages of the deprecated `ArchiveProcess` action class in custom workflow definitions in the same way as in the predefined workflow definitions.

6.7.2 Final Workflow Actions

Workflows can now declare custom final actions, which are executed after a workflow process has completed successfully or was aborted. To this end, a workflow definition can contain `<FinalAction>` elements with custom classes that implement the interface `com.coremedia.cap.workflow.plugin.FinalAction`.

For more details, have a look at the [Section 6.10.5, “Final Actions”](#) in *Unified API Developer Manual*.

6.7.3 Workflow Server Does not Use `corem.home` Anymore

The `corem.home` directory and system property are no longer used by the Workflow Server.

The command-line-tools still use `corem.home`, and as the Docker image for the Workflow Server also contains the workflow converter tool, it still sets `corem.home` but it is not used in the server itself.

The Workflow Server has only one new property which has not been configurable before, that allows a relative filepath, which is now relative to the value of the `user.dir` system property:

- `workflow.working-dir` (directory, default: `var/tmp`)

The Workflow Server does not use `cap.client.server.ior.url` anymore. You must use `workflow.ior-url` instead.

The deprecated method `CapSystemInfo#getInstallationPath` has been removed. The following modules that were used to create a `corem-home` directory structure for the servers have been removed:

- `content-server-config`
- `content-server-blueprint-config`
- `workflow-server-config`
- `workflowserver-blueprint-config`

6.8 Content Application Engine Changes

This section describes changes applied to the Content Application Engine (CAE).

6.8.1 Moved Internal Controller to Management Port

In *CoreMedia Content Cloud* v10 the CAE exposed several endpoints below `/blueprint/servlet/internal` which had to be handled separately when setting up URL rewrite rules so that they cannot be accessed from the public internet.

In CM11 they are now configured to be included in the Spring Boot management context (together with the actuators) and are available on a different port (by default in blueprint deployment port 8081 on the CAE Docker Container and mapped to 40981 for preview and 42181 for live CAE).

Affected endpoints are concerned with URL generation [`/blueprint/servlet/internal/service/url`], sitemap generation [`/blueprint/servlet/internal/<siteid>/sitemap-org`] and test URL generation [`/blueprint/servlet/internal/<siteid>/testurls`]. If you use them in your project, please make sure to access them from your internal network using the new port and a path without the `/blueprint/servlet` prefix. For example: `http://<hostname>:42181/internal/service/url` for accessing the URL generation on the Live CAE.

6.9 Feeder and Search Engine Changes

This chapter contains detailed information about necessary upgrade steps for search components.

6.9.1 Configuration Changes and Reindexing

The Solr index schema for Content Feeder and CAE Feeder, and the configuration of the Content Feeder was changed to enable the search filter for validation issues by default, and to also support Solr nested documents in the CAE Feeder. See also the updated section [Section 4.1.3, "Content Issues"](#) in *Search Manual*.

Solr Configuration Changes

The Solr index fields `_root_` and `_nest_path_` were added in the index schema for both Content Feeder and CAE Feeder indexes in `content/conf/schema.xml` and `cae/conf/schema.xml`. These files are located in the Blueprint below `apps/solr/modules/search/solr-config/src/main/app/configsets`.

The fields were optional since version 2104 but not defined by default. Adding them requires recreation of indexes from scratch. If you create new indexes with this version, or if you have already created new indexes with previous versions and these fields defined, then you should add/keep the fields in the schema. For the Content Feeder index, these fields are required if indexing of validation issues is enabled. It's important that you don't change the definition of these fields, if you keep an existing index.

Several deprecated configuration properties have been removed for the Content Feeder, the CAE Feeder, and for applications that connect to Solr for searching.

If you still use any of these properties, you have to replace them with the property that is listed right next to it in the following list (sorted alphabetically):

- `feeder.backgroundFeed.delay` → `feeder.content.background-feed-delay`
- `feeder.executorQueueCapacity` → `feeder.core.executor-queue-capacity`
- `feeder.executorRetryDelay` → `feeder.core.executor-retry-delay`

- `feeder.indexDeleted` → `feeder.content.index-deleted`
- `feeder.indexGroups` → `feeder.content.index-groups`
- `feeder.indexNameInTextBody` → `feeder.content.index-name-in-textbody`
- `feeder.indexReferrers` → `feeder.content.index-referrers`
- `feeder.management.password` → `feeder.content.management.password`
- `feeder.management.user` → `feeder.content.management.user`
- `feeder.maxBatchBytes` → `feeder.batch.max-bytes`
- `feeder.maxBatchByteSize` → `feeder.batch.max-bytes`
- `feeder.maxBatchSize` → `feeder.batch.max-size`
- `feeder.maxOpenBatches` → `feeder.batch.max-open`
- `feeder.maxProcessedBatches` → `feeder.batch.max-processed`
- `feeder.partialUpdate.aspects` → `feeder.content.partial-update-aspects`
- `feeder.retryConnectToIndexDelay.seconds` → `feeder.content.retry-connect-to-index-delay`
- `feeder.retrySendIdleDelay` → `feeder.batch.retry-send-idle-delay` (see note below)
- `feeder.retrySendMaxDelay` → `feeder.batch.retry-send-max-delay` (see note below)
- `feeder.sendIdleDelay` → `feeder.batch.send-idle-delay` (see note below)
- `feeder.sendMaxDelay` → `feeder.batch.send-max-delay` (see note below)
- `feeder.tika.timeout.milliseconds` → `feeder.tika.timeout`
- `feeder.tika.warn.milliseconds` → `feeder.tika.warn-time-threshold`
- `feeder.updateGroups.background.delay` → `feeder.content.background-feed-delay`
- `feeder.updateGroups.immediately` → `feeder.content.update-groups-immediately`
- `solr.collection.cae` → `solr.cae.collection`
- `solr.collection.content` → `solr.content.collection`
- `solr.configSet` → `solr.cae.config-set` (for the CAE Feeder), `solr.content.config-set` (for the Content Feeder)

- `solr.partialUpdates` → `feeder.solr.partial-updates.enabled`
- `solr.partialUpdatesSkipIndexCheck` → `feeder.solr.partial-updates.skip-index-check`

Note, that the old properties `feeder.sendIdleDelay`, `feeder.sendMaxDelay`, `feeder.retrySendIdleDelay`, and `feeder.retrySendMaxDelay` took a value in seconds for the `_Content Feeder_`, but a value in milliseconds for the `_CAE Feeder_`. The new properties `feeder.batch.send-idle-delay`, `feeder.batch.send-max-delay`, `feeder.batch.retry-send-idle-delay`, and `feeder.batch.retry-send-max-delay` take milliseconds for both Content Feeder and CAE Feeder (if the value does not specify a different Spring Boot duration unit like 's').

Solr Terminology Changes

Apache Solr has replaced the names of their replication setup from Master/Slave to Leader/Follower. These names have now been changed in Solr configuration files `solrconfig.xml` and CoreMedia documentation. For details of the change in Solr, see [Solr Upgrade Notes](#).

The following system properties used to configure Solr replication have been renamed. If you are using the old names in your project, you must update them accordingly:

- `solr.master.url` → `solr.leader.url`
- `solr.master` → `solr.leader`
- `solr.slave` → `solr.follower`

Environment variables for the Docker image have also changed. The old environment variables still work, but are deprecated:

- `SOLR_MASTER` → `SOLR_LEADER`
- `SOLR_MASTER_URL` → `SOLR_LEADER_URL`
- `SOLR_SLAVE` → `SOLR_FOLLOWER`
- `SOLR_SLAVE_AUTOCREATE_CORES` → `SOLR_FOLLOWER_AUTOCREATE_CORES`
- `SOLR_SLAVE_AUTOCREATE_CORES_LIST` → `SOLR_FOLLOWER_AUTOCREATE_CORES_LIST`
- `SOLR_SLAVE_AUTOCREATE_THRESHOLD` → `SOLR_FOLLOWER_AUTOCREATE_THRESHOLD`

The Maven commands to start and stop a Solr follower node for local development in `apps/solr/modules/search/solr-config/pom.xml` have also changed

Detailed Overview of Changes | Configuration Changes and Reindexing

to use "follower" instead of "slave". For details, see the comments in that `pom.xml` file.

In addition to the system properties and environment variables, the following changes have been applied:

- The default log-location for Solr followers changed: `slave-logs/` → `follower-logs/`
- To start a Solr follower locally, you now need to call `mvn exec:exec@start-solr-follower` (instead of `mvn exec:exec@start-solr-slave`).

Content Feeder Configuration for Issue Feeding

The configuration of the Content Feeder was changed to index issues, by default. You can disable issue indexing with configuration property `feeder.content.issues.index=false`, for example if you have decided to not add the fields to the Solr schema to avoid reindexing, or if you are not using the default Solr integration but a custom Indexer that does not support indexing issues as nested documents.

Recreate Index from Scratch for Improved CAE Feeder Database Usage

The CAE Feeder was improved to store data more efficiently in the database, which reduces database disk space requirements and improves throughput. To this end, the data type of some database columns was changed from a string type to a binary type. This change also applies to custom applications based on the Proactive Engine.

When the CAE Feeder is started with an empty database to feed everything from scratch, it will automatically create tables with improved definitions. If started with existing database tables from a previous release, then the CAE Feeder will work as before without improved database usage. Existing database tables are not upgraded. For best performance, it is recommended to start feeding with an empty database.

6.10 Personalization Changes

This section describes changes applied to the Personalization integration.

6.10.1 Client-Side Personalization API

A new extension for integrating third party web personalization, optimization and testing tools has been released. It allows for the following use cases to be added to a site (depending on the capabilities of the integrated tool]):

- A/B/n testing and targeting of content fragments
- Personalization of content fragments with machine learning
- Delivering content fragments based on user segments

The integration follows the 'client-side' approach by adding vendor specific JavaScript code to the generated web pages and 'linking' it to an extension brick.

In CoreMedia Studio content types are added to allow for the simple creation and management of new testing and segmentation experiences.

Out of the box adapters for connecting Monetate, Dynamic Yield and Evergage are available. Usage of this extension requires the customer to have an active Adaptive Personalization license.

The extension's source code and documentation can be accessed on GitHub: <https://github.com/CoreMedia/p13n-core>

6.11 Commerce Integration Changes

This section describes changes applied to the different commerce integration components.

6.11.1 Commerce Hub API v2

In CoreMedia v11 only Commerce Adapters with version 2.x.x are supported and these adapters are not backward compatible to v10. There are several incompatible changes as described in this chapter and in "Various Commerce Hub API removals of deprecated code", "Salesforce Commerce Cloud OCAPI Changes" and "Removal of Legacy Commerce Search Facet API and Demo Content". If you use the default Commerce Adapters out of the box, you still need to consider some configuration steps mentioned in the denoted chapters. If you have customized a Commerce Adapter in version 10, you need to adapt to the new API in your customization code.

6.11.2 Commerce Adapter Configuration Properties

The out-of-the box commerce adapter configuration has been improved and several mandatory properties have been marked with `@NotBlank` and thus must be provided at startup. When upgrading to the new adapter version, it is most likely that you have already set these properties in your old v10 setup or by default using the Blueprint Docker compose setup. The validation should help you to find missing configuration faster.

SAP Commerce Cloud / Hybris

The following configuration properties have been marked as `@NotBlank` and must be provided:

- `hybris.host`
- `hybris.user`
- `hybris.password`
- `hybris.link.storefront-url`
- `hybris.link.asset-url`

- `hybris.oauth.client-id`
- `hybris.oauth.client-secret`

These Properties have been removed without replacement:

- `hybris.defaultCatalogVersionLive`
- `hybris.productSearchMaxResults`
- `hybris.link.storefrontHost`

If you use the default SAP Hybris commerce setup you may configure the old default values as follows:

- `hybris.user=admin`
- `hybris.password=nimda`
- `hybris.oauth.client-id=coremedia_preview`
- `hybris.oauth.client-secret=secret`

HCL Commerce / WCS

The following configuration properties have been marked as @NotBlank and must be provided:

- `wcs.url`
- `wcs.secure-url`
- `wcs.secure-search-url`
- `wcs.auth-header-name`
- `wcs.username`
- `wcs.password`

The following properties have been removed:

- `livecontext.ibm.wcs.url`
- `livecontext.ibm.wcs.rest.secureUrl`
- `livecontext.ibm.wcs.rest.search.url`
- `livecontext.ibm.wcs.rest.search.secureUrl`
- `livecontext.rest.connector.authHeaderName`
- `livecontext.rest.connector.authHeaderValue`
- `livecontext.service.credentials.username`
- `livecontext.service.credentials.password`

- `wcs.host`

Salesforce Commerce Cloud / SFCC

The following configuration properties have been marked as @NotBlank and must be provided:

- `sfcc.link.storefront-url`
- `sfcc.ocapi.host`
- `sfcc.oauth.client-id`
- `sfcc.oauth.client-password`

The following configuration properties have been marked as @NotBlank but have a default value:

- `sfcc.vendor-version`
- `sfcc.oauth.protocol`
- `sfcc.ocapi.protocol`
- `sfcc.ocapi.version`

These Properties have been removed without replacement:

- `sfcc.host`

has been removed due to no usages.

Base Adapter / Mock

The base adapter configuration property `metadata.vendor` has been marked as @NotBlank and must be provided. The mock adapter configuration property `mock.link.storefrontUrlPreview` has been marked as @NotBlank and must be provided. The base adapter configuration property `metadata.supportsMultiCatalog` has been removed without replacement. The following mock adapter configuration properties have been removed without replacement:

- `cookie.user.pattern`
- `mock.link.storefrontUrl`

6.11.3 Commerce Adapter Cache Configuration changes

The commerce cache configuration keys for all commerce adapters have been harmonized and now all start with "cache." - without a prefix for a concrete vendor (for example "sfcc.cache.").

This change was already announced, and old configuration properties were deprecated with previous adapter versions, but the fallback was kept until now. Please make sure that your custom cache configuration of your commerce adapter only uses the new properties. Otherwise, the default values will apply and your previous custom settings are ignored.

6.11.4 Salesforce Commerce Cloud OCAPI Changes

The Salesforce Commerce Cloud Adapter v2 is now using Shop API OCAPI calls wherever possible, especially for requesting products and categories. The Shop API is much more reliable and performant. Please note, since it is meant to be used to render storefronts, only online items are visible. There are a few parts that are not covered by the Shop API. The Data API is still used to read catalogs and associations between category and products (products as direct children of categories).

This change reflects Salesforce's recommendation to use Shop API for all end user related activity as it scales better and is more reliable.

Due to this change, the Shop-API OCAPI read permission for `/categories/*` need to be added to your Salesforce Commerce Cloud instance.

You will find the current required OCAPI permissions in the cartridge workspace documentation. There are also minor cartridge changes beside the OCAPI permissions but they are not mandatory to integrate your SFCC installation with *CoreMedia Content Cloud* v11 and the new v2 adapter.

On the SFCC adapter configuration the version to be used to access OCAPI can be configured by using the `sfcc.ocapi.version` configuration property. The default is changed to "v21_9".

6.11.5 LiveContext HCL and B2B Removal

The LiveContext HCL/IBM Blueprint extension was deprecated in *CoreMedia Content Cloud* v10 and has now been removed. It is replaced by the Commerce Hub Adapter for HCL Commerce.

Connections to an eCommerce system are no longer established directly through implementation of the eCommerce API in the Blueprint, but only via Commerce Hub/Adapters. With this removal, also the HCL/IBM B2B integration is removed. Thus, the example Aurora B2B sites have been removed from the demo content. Additionally, features that were part of the HCL/IBM B2B support have also been omitted. Note, there is still no replacement in the Commerce Hub for some of them. In detail, these are the handling of HCL/IBM workspaces and contracts.

If you have customizations based on the LiveContext Blueprint extension for HCL/IBM, you should consider migrating them to a customized Commerce Hub Adapter. Please refer to the [Connector for HCL Commerce Manual](#). Alternatively, you can turn the v10 LiveContext Blueprint extension into custom code, migrate it to v11 yourself and maintain it as a project solution.

6.11.6 Various Commerce Hub API Removals of Deprecated Code

In *CoreMedia Content Cloud* v11, only Commerce Adapters with version 2.x.x are supported. Beside the Search API changes described in [Section 6.11.7, "Removal of Legacy Commerce Search Facet API and Demo Content" \[81\]](#) these additional API changes in Commerce Hub were also made to resolve deprecations.

The following changes are only relevant if you have either customized or implemented a project-specific commerce adapter.

- The `catalogId` was removed from `Invalidation` because invalidations do not depend on the catalog.
- The unused fields `ContractIds` and `SegmentIds` were removed from `EntityTypeParam`.
- The `Price` constructor was removed. Use the builder instead.
- Information about service features can be gathered via gRPC server reflection. Its `Feature` implementation has been removed.
- The deprecated `ConnectorApplication` was removed.

- `com.coremedia.commerce.adapter.base.api.Messages` was moved to `com.coremedia.commerce.adapter.base.entities.Messages`.
- `com.coremedia.commerce.adapter.base.v2.Messages` was removed. The utility methods of this class were moved to `com.coremedia.commerce.adapter.base.entities.Entities`.
- The Commerce Hub API for preview URLs has been removed from `com.coremedia.commerce.adapter.base.repositories.LinkRepository`. Now preview URLs are generated the same way as all other commerce related links via link templates and dynamic replacement values. If you run your own commerce adapter implementation or if you do have customizations regarding link generation for Studio preview, you may have to update your `LinkRepository` implementation.

`com.coremedia.commerce.adapter.base.repositories.LinkRepository` does not provide dedicated methods for preview URLs anymore. Instead all links are handled via `com.coremedia.commerce.adapter.base.repositories.LinkRepository#getLinkTemplates` from now on. A set of preview specific `StorefrontRefKeys` has been added (see `com.coremedia.commerce.adapter.base.entities.links.StorefrontRefKeysPreview`). The `LinkRepository` needs to provide these link templates in order to support the preview feature in *CoreMedia Studio*. Highly dynamic preview specific url tokens are handled via `com.coremedia.commerce.adapter.base.repositories.LinkRepository#getLinkVariablesForRequest` from now on.

On the CMS side the `com.coremedia.livecontext.ecommerce.link.PreviewUrlService` has been deleted and there are changes in the `com.coremedia.livecontext.fragment.links.CommerceStudioLinks`.

If you didn't do any customization regarding URLs for Studio preview, nothing needs to be changed on your side.

6.11.7 Removal of Legacy Commerce Search Facet API and Demo Content

In *CoreMedia Content Cloud* v11 only commerce adapters with version 2.x.x are supported. The deprecated legacy search API was removed in favor of the new v2 search API. Legacy searches are not supported anymore. That especially means, that only the new multi-facet-capable search v2 API is used that was introduced in 2104.1.

If you have not customized the commerce adapter in your project you only have to take care of existing Product Lists in your content repository. If you have created them prior to 2104.1 and have not yet converted them in v10 Studio using the **[Enable multiple selection]** button, they might be broken in v11. The reason is that besides enabling multiple selection of facet filters, also the storage format in the Struct settings has changed. Validators will help you to identify these broken Product Lists in Studio. You can for example use the library search with issue filtering by type "Error" and selected content type "Product List". CoreMedia recommends converting them before updating to v11 in v10 Studio as there is a best-effort guessing to find the corresponding new facet value in the new multi-facet API using the **[Enable multiple selection]** button. In v11 Studio you can only set the facet values completely new, without conversion support. The validator message will tell you the stored value that is now invalid, but you need to select it newly with the v11 form.

If you have customized either the commerce adapter or make use of the commerce adapter Search API in your Blueprint code, you should be aware of these changes and removals:

- **SearchQuery**
 - The deprecated `getFacet` method was removed. A list of facets can be retrieved via `getFilterFacets`.
 - The flag `isFacetSupport` was removed. Use `isIncludeFacets` instead.
 - The `SearchQuery` no longer holds information about the `pageNumber`. To gather this information the limit and offset can be used.
- **SearchResult**
 - The `SearchResult` doesn't hold information on the `pageNumber` and `pageSize` anymore. This was redundant information, which can be gathered from the `SearchQuery`.
 - The constructors were removed. Use the builder instead.
- The deprecated `ProductRepository#getFacetsForProductSearch` was removed. Instead you can use `SearchQuery#search` with limit 0 and `includeFacets` true.

6.11.8 Updated Multi Catalog Configuration

The catalog alias configuration for the multi catalog feature has been unified with the catalog configuration for a single catalog configuration. In the former Studio content configuration setting both the `catalogConfig` struct and the `livecontent.catalogAliases` list could define catalog aliases. This was very misleading and has been unified now.

Detailed Overview of Changes | Updated Multi Catalog Configuration

If you do not use the multi catalog feature there is no action required.

If you use multiple catalogs in a site, the catalog aliases must be configured via the `additionalCatalogConfigs` struct list below the `commerce` struct from now on. The old configuration via `livecontext.catalogAliases` needs to be transferred to the new struct list `additionalCatalogConfigs`. Each catalog configuration consist of alias, name and id. At least the alias and one of name or id must be configured.

Old format:

```
<StructProperty Name="commerce">
  <StructProperty Name="catalogConfig">
    <StringProperty Name="alias">catalog</StringProperty>
    <StringProperty Name="name">Default Catalog</StringProperty>
    <StringProperty Name="id">12345</StringProperty>
  </StructProperty>
  ...
</StructProperty>
<StructProperty Name="livecontext.catalogAliases">
  <Struct>
    <StringProperty Name="catalog">Default Catalog</StringProperty>
    <StringProperty Name="asset">Asset Catalog</StringProperty>
  </Struct>
</StructProperty>
```

New format:

```
<StructProperty Name="commerce">
  <StructProperty Name="catalogConfig">
    <StringProperty Name="alias">catalog</StringProperty>
    <StringProperty Name="name">Default Catalog</StringProperty>
    <StringProperty Name="id">12345</StringProperty>
  </StructProperty>
  <StructListProperty Name="additionalCatalogConfigs">
    <Struct>
      <StringProperty Name="alias">asset</StringProperty>
      <StringProperty Name="name">Asset Catalog</StringProperty>
      <StringProperty Name="id">6789</StringProperty>
    </Struct>
  </StructListProperty>
</StructProperty>
```

Consult the commerce adapter documentation for more details.

NOTE

The default catalog is configured via `commerce#catalogConfig`, whereas all additional catalogs are configured via `commerce#additionalCatalogConfigs`.



6.12 Blueprint Changes

This section describes changes in the CoreMedia Blueprint workspace.

6.12.1 Content Issue Search Enabled by Default

The content issue search in *CoreMedia Studio* is available as an opt-in feature since *CoreMedia Content Cloud* v10.2104.1. It is now enabled by default, and you will have to follow the changes described in [Section 6.9.1, "Configuration Changes and Reindexing" \[71\]](#) to see it in action.

6.12.2 Guava Optional, Function, Predicate, Supplier Have been Replaced With Their JDK Counterparts

Guava `Optional`, `Function`, `Predicate`, `Supplier` and most Guava utilities from `com.google.common.collect` have been replaced with their JDK counterparts. Most of these Guava features are available in the JDK by now, so it is unnecessary to bother with a third-party library.

Although it is not required, CoreMedia recommends that you also refactor these Guava usages in your code for consistency.

6.12.3 New Blueprint Parent

A new POM parent `cms-blueprint-parent` has been introduced and is used as (transitive) parent for every Blueprint module. It contains basic plugin management, especially for the install plugin and deploy plugin for which it is important that all modules use the same versions. If you need a custom parent in your Blueprint, you can simply search and replace our parent element.

You could use the `cms-blueprint-parent` as the parent of your custom parent or integrate the plugin management as necessary.

6.12.4 Consolidating Blueprint Group IDs and Versions

The Blueprint now uses the group ID `com.coremedia.blueprint` for every module and Blueprint modules are now always referenced by using `${project.version}`.

The following group IDs are no longer used:

- `com.coremedia.blueprint.boot`
- `com.coremedia.blueprint.docker`
- `com.coremedia.blueprint.content-hub`

The following properties have been removed:

- `blueprint.groupId`
- `blueprint.version`
- `blueprint.boot.groupId`
- `Blueprint.boot.version`

This makes it easier to replace the CoreMedia group ID with a custom one and allows for setting the version via *maven-release-plugin* or *versions-maven-plugin* which should also make it easier to create releases of your Blueprint.

6.12.5 Replacing dockerfile-maven-plugin with google-jib-plugin

The *dockerfile-maven-plugin* has been replaced with the *google-jib-plugin* to build OCI conform container images, because the *dockerfile-maven-plugin* is no longer maintained.

The *google-jib-plugin* plugin integrates well with Spring Boot applications and allows to build images securely without Docker being installed. This allows for a rootless, daemonless build but restricts the image building process to use only adding image layers by adding resources. There is no possibility to execute any RUN steps during the build process. Building the images without Docker now requires a configured container registry although it is still possible to build images in the local Docker daemon, without the need of a registry.

The switch to the new image build process is tightly coupled to the recent change to build all artifacts in a reproducible manner, resulting in identical image digests when

the source has not changed. With Spring Boot layered applications, this results in a faster build and smaller upload size to your container registry. Because file and image creation date affect the resulting image digest, jib sets the date to Unix epoch (00:00:00, January 1st, 1970 in UTC). If you do not like this, you can enable the current timestamp in the *jib-maven-plugin*, but this sacrifices reproducibility since the timestamp will change with every build.

```
<container>
  <useCurrentTimestamp>true</useCurrentTimestamp>
</container>
```

As a result of this change, the image build process moved to the application modules and the Docker directory hierarchy has been removed.

All Spring Boot application images are still based on the `coremedia/java-application-base` images will run with only some small breaking changes to the previous images. The breaking changes include:

- The application root directory `/coremedia` is now write protected as it is intended by the *google-jib-plugin* plugin.
- The log files are now written to `/coremedia/log/application.log` if the dev profile is activated.

To build the images, the properties and Maven profiles have changed:

- The Maven profile to build the image is now named `default-image` with the property `jib.goal` to be either set to `build` or `dockerBuild`, depending on whether you want to use the Docker daemon or the registry as a build target.
- The property to define the registry has been renamed from `docker.repository.prefix` to `application.image-prefix`.
- The property to define the image tag has been renamed from `docker.image.tag` to `application.image-tag`.
- The property to define the base image has been renamed from `docker.java-application-base-image.repo` to `application.image-base`.
- The property to define the image name has been renamed from `docker.repository.suffix` to `application.image-suffix`.

The new images do not contain any health checks directive any more. If you rely on the Docker health checks, please define them either on the command line or in your `docker-compose` file.

```
services:
  foo:
    test: [ "CMD", "curl", "-Lf", "http://localhost:8081/actuator/health" ]
    interval: 30s
    timeout: 10s
```

```
retries: 3
start_period: 40s
docker run --health-cmd='curl -Lf http://localhost:8081/actuator/health ||
exit 1' \
--health-interval=30s \
--health-timeout=10s \
--health-start-period=40s \
--health-retries=3 <image>
```

Example 6.5. Healthcheck

For the Commerce Adapter the health check looks like this:

```
services:
commerce-adapter-mock:
test: [ "CMD", "curl", "-Lf", "http://localhost:8081/actuator/health" ]
interval: 30s
timeout: 10s
retries: 3
start_period: 40s

docker run --health-cmd='curl -Lf http://localhost:8081/actuator/health ||
exit 1' \
--health-interval=30s \
--health-timeout=10s \
--health-start-period=40s \
--health-retries=3 <image>
```

Example 6.6. Commerce Adapter health check

6.12.6 Content Hub and Feedback Hub Adapters as Plugins

The following extensions for the Content Hub Adapters have been removed from the Blueprint.

- content-hub-adapter-rss
- content-hub-adapter-youtube

The following extension for the Feedback Hub Adapter has been removed from the Blueprint:

- feedback-hub-adapter-imagga

These adapters are now released as separate plugins which are available on GitHub.

- <https://github.com/CoreMedia/content-hub-adapter-rss>
- <https://github.com/CoreMedia/content-hub-adapter-youtube>

- <https://github.com/CoreMedia/feedback-hub-adapter-imagga>

By default, the plugins are bundled with the Blueprint.

For details on bundled plugins see `workspace-configuration/plugins/README.md` in the Blueprint workspace and section "Plugin Descriptors and Bundled Plugins" in the Blueprint Developer Manual.

To remove these bundled plugins, go to `workspace-configuration/plugins/` remove the entries from the file `plugin-descriptors.json` and run `mvn generate-resources`.

If you have customized the adapters, you can keep your customized extensions, but must remove the bundled plugins as described above. Nevertheless, in the long run, you should consider migrating your extensions to plugins as well.

6.12.7 Commerce "Candy Shop" Developer Setup Removed

The developer feature "Candy Shop URLs" was already deprecated in *CoreMedia Content Cloud* v10 and has now been removed from the Blueprint.

This feature used to provide automatic retrieval of commerce fragments from a local CAE by using a dedicated hostname and Apache proxy. There is no replacement planned, developers will need to configure their local machine in the shop configuration to retrieve fragments from their localhost. Frontend development is easily possible through the Frontend Developer workflow and uploading of local Themes to the CMS.

6.12.8 Using Standard Spring Boot Logging Configuration

The logging setup has changed to implement a more standardized setup. This includes:

- Disabling the file appender by default.
- Removed any third-party logback appenders to push logs events directly to a log aggregation like elasticsearch.
- Removed the Spring-Boot elk profile to activate elastic log aggregation.
- Removed the `com.coremedia.blueprint:logging-config` Maven module.
- Removed any `logback-spring.xml` file in favor of Spring Boot standard setup.

- Removed the Studio Console logfile endpoint.
- The logging pattern has been changed to integrate better with container based deployments and cloud native logging solutions like the Elastic stack and the Grafana Loki Stack.

The new logging pattern for the console appender no longer contains a timestamp. In the recommended setup, the log events are digested from the container runtime and then shipped to the log aggregation service. Since all container runtimes add a timestamp by default, adding a timestamp with logback is redundant. The logging pattern also does no longer contain coloring, since that unnecessarily complicates the processing of the log events.

Removal of Logback appenders

Instead of using this approach please use one of the many log shippers available to send logfiles to an elastic stack.

Removal of logback-spring.xml files

If you need to customize your logging you can add `logback-spring.xml` files again to the `*-app/src/main/resources` folder and follow the standard customization steps described in the Spring Boot documentation.

Removal of the Studio console endpoint

The endpoint was intended to be used similar to the logfile endpoint of the standard Spring Boot actuators. To view the studio-console logger, please use standard log queries in your log aggregation stack of your choice to separate the studio-console logger from the studio-server logs.

Standard console logging pattern

For an easier integration with log shippers, the logging pattern has been changed:

- Any coloring has been removed for console and file logging except for local profiles, when the application is started using the IDE or the *spring-boot-maven-plugin* plugin.
- The timestamp has been removed for standard console logging. The container logging always contains the timestamp and the shippers parse this timestamp anyway.
- The pattern has been reordered so that the message is the last entry. This simplifies the handling of multiline strings.

The new logging pattern for the file logger will be:

```
%d{yyyy-MM-dd HH:mm:ss} %-7([%level]) \\(%thread\\) %logger [%X{tenant}] - %message%n
```

The new logging pattern for the console logger will be:

```
%-7([%level]) \\(%thread\\) %logger [%X{tenant}] - %message%n
```

If you want to keep the old pattern, set the following property respectively environment variable `logging.pattern.file`, `LOGGING_PATTERN_FILE` to the following:

```
%d{yyyy-MM-dd HH:mm:ss} %-7([%level]) %logger [%X{tenant}] - %message \\(%thread\\) %n
```

For the console logger set `logging.pattern.console` or `LOGGING_PATTERN_CONSOLE` to the following:

```
%clr(%d{yyyy-MM-dd HH:mm:ss} -){faint} %clr(%7([%level])) %clr(%logger){cyan} [%X{tenant}] %clr(-){faint} %message \\(%thread\\) %n
```

6.12.9 Removed Optimizely Integration

The extension Optimizely has been removed from the applications CAE and Studio Client. The settings starting with "optimizely." have no effect anymore and can be removed from custom content. Additionally, the helper class `ExtensionAspectUtil.java` has been removed.

6.12.10 Removed Chef Deployment

The deployment based on Chef has been removed from *CoreMedia Content Cloud*. Use a container based deployment scenario.

6.12.11 Removed Watchdog/Probedog

The Watchdog and Probedog components have been removed. To implement the same functionality, you can use Spring Boot actuators together with an orchestration solution. In addition to the standard actuators of Spring Boot, CoreMedia adds a set of custom

endpoints to represent the health state of the applications. You find the documentation in the [Operations Basics](#).

6.12.12 Removed Dynamic Packages Proxy App

The Studio Packages Proxy App has been removed. The new plugin mechanism can be used instead. See the [Blueprint Developer Manual](#) for details.

6.12.13 blueprint-doctypes-xmlrepo.xml moved

The schema definition for tests using the XML Repository has been moved from Blueprint Base module `com.coremedia.blueprint.base:bpbase-test-util` to Blueprint module `com.coremedia.blueprint:test-util`.

All corresponding tests in Blueprint got their references adapted from the old artifact to the new artifact:

- **was:** `classpath:com/coremedia/blueprint/base/testing/blueprint-doctypes-xmlrepo.xml`
- **is:** `classpath:com/coremedia/testing/blueprint-doctypes-xmlrepo.xml`

The new artifact contains the very same definitions as the original XML-Repository schema.

The old artifact is still available, but it is deprecated and will be removed in upcoming versions.

Upgrade: Changes will be automatically applied by Git update to existing tests. For custom tests, it is strongly recommended updating the reference to the new content-type definitions.

General Note: The motivation for moving the definition is, that if you customized your content-type definitions for your project, you may have experienced especially CAE ContentBean tests to struggle with insufficient definitions. Thus, by nature of the existing tests you may have been forced providing a custom schema. With this change, this will be feasible without the need to extract the definitions from JAR files.

In general, we recommend providing a custom schema for each test with the specific content-types and properties required by that test. This increases maintainability. In

addition to that, you may want to use standard content-type definitions in the same format supported by the *Content Server*. How to do that, and for more information read the corresponding JavaDoc:

- `XmlRepoConfiguration`
- `XmlUapiConfig`
- `XmlUapiConfig.Builder`

`XmlUapiConfig.Builder.withContentTypes(...)` is the method to use to refer to content-type definitions as supported by the *Content Server*.

6.13 Frontend Workspace Changes

This section describes changes in the CoreMedia Frontend workspace.

6.13.1 Using pnpm Instead of Yarn

The package manager of the frontend workspace has been changed from *Yarn* to *pnpm*. *Yarn* in version 1 is not maintained anymore and *pnpm* is fast, efficient, strict and supports monorepos without additional libraries like *lerna*. Changes to the tooling should not affect any bricks or themes in your project.

You need to install **pnpm** in the latest 6.x version and use the corresponding pnpm commands.

Yarn (old)	pnpm (new)
yarn install	pnpm install
yarn build	pnpm build
yarn test	pnpm test
yarn create-theme	pnpm create-theme
yarn create-brick	pnpm create-brick

Table 6.18. *pnpm commands comparison*

Changes

- Support for building the frontend workspace with maven or other package manager has been removed.
- The lock file for pinned versions changed from `yarn.lock` to `pnpm-lock.yaml`.
- A new file for the workspace structure has been introduced: `pnpm-workspace.yaml`.

Since *pnpm* uses a non-flat `node_modules` structure, it is important to check for missing dependencies in custom themes and bricks. Please remove all `node_mod`

ules in your project workspace before running `pnpm install`. If a build fails, add the missing dependency to your theme or brick according to the error message.

6.13.2 Using NodeJS 16 [LTS]

You need to install *NodeJS* in latest version 16. *Node-Sass* has been updated accordingly to work with *NodeJS* 16.x.

6.13.3 Removal of Deprecated FreemarkerFacade Methods

Both deprecated methods `createBeansFor()` and `createBeanFor()` have been moved from `BlueprintFreemarkerFacade` to `LivecontextFreeMarkerFacade`, because of the remaining usages in `LiveContext` templates. We recommend to not use the functions in projects.

Additionally, the deprecated methods `BlueprintFreemarkerFacade.getContainerMetadata` and `BlueprintFreemarkerFacade.getDisplaySize` were removed. Please use `Container#getContainerMetadata()` and `BlueprintFreemarkerFacade.getDisplayFileSize` instead.

6.13.4 Optimized Rendering of Embedded Content in Richtext

When embedding content (especially images) in richtext, `asRichtextEmbed` templates are likely to break HTML validity by rendering `<div>` blocks inside the richtext's `<p>` context. So far, the Blueprint's `PDivUntanglingFilter` tried to be smart and adjust the HTML accordingly by closing and reopening the `<p>` context around an embedded object. However, this magic never really matched projects' expectations about the result, because it sometimes led to unwanted additional paragraphs or empty links.

Therefore, `CoreMedia` removed the `p/div` adjusting features from the `PDivUntanglingFilter` and leaves it to the `asRichtextEmbed` templates to provide HTML fragments that are suitable to be used within a `<p>` context. Consequently, the `PDivUntanglingFilter` has been renamed to `EmbeddingFilter`.

If the old `PDivUntanglingFilter` is useful for you, you can keep it in your project code, and also preserve the `LinkEmbedFilter` and `ImageFilter` classes in their old versions.

The following templates in the example bricks and themes are changed:

- `CMTeasable.asRichtextEmbed.ftl` closes and opens the `<p>` elements before and after the include manually.
- `CMPictureasRichtextEmbed.ftl` changed from `<div>` to ``.
- `CMPicture.media.ftl` uses the `<picture>` element for the responsive container instead of a `<div>`.

6.14 Command Line Tool Changes

This section describes changes applied to the command line tools.

6.14.1 Deleted Deprecated Options of some Commandline Tools

The command line tools *publish*, *destroy* and *query* do not support the option `paths` any longer. Use `path` instead (also CSV capable, despite the singular name). `paths` has already been undocumented for a while, so you might not even be aware that it existed.

7. Studio Client to TypeScript Migration

Studio Client prior to *CoreMedia Content Cloud* v11 was programmed with ActionScript and MXML. However, ActionScript is based on ECMAScript 3 and has not made much progress since 2011, when it was donated to Apache. FlashPlayer is end-of-life, so most browsers no longer support it, and Apache's efforts to create their own compiler to JavaScript (Apache Royale) has not been overly successful. Therefore, Studio Client has been migrated to TypeScript.

TypeScript is a very popular programming language with an active community. There are many experienced developers that will be able to support you with developing your Studio extensions in the future.

This chapter describes the changes in the workspace and in programming. It also explains the necessary upgrade steps for your own Studio Client customizations.

- [Section 7.1, “Studio Client in TypeScript/npm” \[98\]](#) describes why CoreMedia has switched to TypeScript for Studio client development and what has changed.
- [Section 7.2, “Upgrading Studio Client to CoreMedia Content Cloud v11” \[100\]](#) describes how you migrate your own Studio Client customizations to TypeScript.
- [Section 7.3, “Changes Between ActionScript and TypeScript for Developers” \[111\]](#) describes how development with TypeScript differs from development in ActionScript and MXML.

7.1 Studio Client in TypeScript/npm

This section gives an overview of the new *Studio Client* workspace, programming language and tools. Since this is one of the major changes in *CoreMedia Content Cloud* v11, here is a concise summary of what's new:

- *Studio Client* is now implemented in TypeScript instead of ActionScript/MXML
- *Studio Client* is now built with [p]npm instead of Maven
- Resulting compiled JavaScript and Docker image are very similar to the former artifacts
- CoreMedia provides a tool to automate conversion of your customized Blueprint workspace to TypeScript/npm
- CoreMedia provides npm-based tooling to build the new workspace and start *Studio Client*
- Using TypeScript leads to extended IDE support: JetBrains IntelliJ IDEA Ultimate, WebStorm or Microsoft Visual Studio Code

Please read on if you are interested in more details.

Up to *CoreMedia Content Cloud* v10, *Studio Client* is implemented in ActionScript and MXML, the programming languages of *Apache Flex*. CoreMedia's Open Source tool *Jangaroo* is used to compile ActionScript/MXML to JavaScript that runs in the browser directly, unlike original *Flex*, which uses *FlashPlayer*. To invoke the *Jangaroo* compiler and package resources, *Maven* is used as the underlying build tool.

Why Studio Client has been Migrated to TypeScript

ActionScript and MXML offer considerable advantages over pure JavaScript, mainly static typing, which leads to superior IDE support, especially when developing using comprehensive APIs like those of *CoreMedia Content Cloud* and *Sencha Ext JS*. However, ActionScript is based on ECMAScript 3 and has not made much progress since 2011, when it was donated to Apache. FlashPlayer is end-of-life, so almost all browsers no longer support it, and Apache's efforts to create their own compiler to JavaScript (*Apache Royale*) has not been overly successful. In contrast, in the same period, ECMAScript/JavaScript, after years of stagnation, flourished with innovation, leading to many modern language features being introduced in frequent major ECMAScript updates. Also, TypeScript established itself as the de-facto standard to add static typing to the ECMAScript language.

Since its introduction in 2012, TypeScript took years to become on par with ActionScript, but in 2019, it even turned out to have overtaken it, so it was the right time to start finding a migration path for the *Studio Client* code from ActionScript to TypeScript. Although ActionScript and TypeScript are quite similar, this proved to be quite a challenge, but CoreMedia succeeded in building a tool that converts ActionScript, MXML and properties source code to TypeScript. This tool is actually the new version 4.1 of the *Jangaroo* compiler, until then only used to compile to JavaScript, now supporting TypeScript as a new output format.

Automating TypeScript Conversion

Using TypeScript, building *Studio Client* with Maven no longer seemed appropriate, either, so now, the standard packaging format and build tool for JavaScript, *npm*, to be more precise, *pnpm*, is used. Converting *Maven pom.xml* to *npm package.json* has been included in the conversion tool as well.

To summarize, CoreMedia provides a tool to convert your custom Blueprint workspace from ActionScript/MXML/properties/Maven to TypeScript/pnpm completely automatically, so that it can be successfully built and run right after conversion. You essentially convert your *CoreMedia Content Cloud* v10.2107 *Studio Client* to TypeScript and, as a check-point, run it with your existing 2107 system. You then move on to v11 as usual, by merging-in CoreMedia's v11 Blueprint workspace. This process is described in more detail in [Section 7.2, "Upgrading Studio Client to CoreMedia Content Cloud v11" \[100\]](#).

Once converted to TypeScript/npm, your *Studio Client* workspace comes with a modern, superior developer experience. TypeScript is in the 2021 top ten of the worldwide [PYPL Popularity of Programming Language Index](#). Several IDEs (JetBrains IDEA and WebStorm, Microsoft Visual Studio Code) support TypeScript out of the box. *npm* is JavaScript's de-facto standard for package and dependency management. *pnpm* is a new, popular, very efficient and convenient tool to build a *npm* workspace.

*The New Workspace:
TypeScript and pnpm*

This combination of modern programming language and tool has been complemented by a minimal set of tools provided by CoreMedia to support the integration with *Sencha Ext JS* and add even more developer convenience for building, testing, starting and publishing *npm* packages with TypeScript code. These tools are themselves published as *npm* packages, containing scripts that are invoked via *pnpm*, taking advantage of established standards and the power and flexibility of *pnpm*.

7.2 Upgrading Studio Client to *CoreMedia Content Cloud* v11

This chapter describes how you convert your Studio Client customizations to TypeScript using CoreMedia's workspace conversion tool, as a prerequisite for upgrading to *CoreMedia Content Cloud* v11. After the conversion and upgrade to v11, there are certain clean up and error fixing tasks that might be necessary.

7.2.1 Converting Studio Client to TypeScript

1. Add extNamespace Property

Iterate over all your custom *Studio Client* module POMs: Add a property `extNamespace` to the configuration of `jangaroo-maven-plugin`. The `extNamespace` should specify the longest common ActionScript package name prefix of all classes contained in that package. This is important to keep TypeScript import identifiers concise. Since these already use the npm package name, the common ActionScript package prefix would be redundant.

For example, if your module contains classes with ActionScript fully-qualified names `com.acme.studio.foo.Foo`, `com.acme.studio.foo.bar.FooBar` and `com.acme.studio.bar.Bar`, and the npm package name of the module will be `"@acme/studio"`, setting the common prefix to `"com.acme.studio"` would result in the import identifiers for the three classes being `"@acme/studio/foo/Foo"`, `"@acme/studio/foo/bar/FooBar"` and `"@acme/studio/bar/Bar"`, respectively.

Not setting an `extNamespace` at all would result in long, redundant import identifiers like `"@acme/studio/com/acme/studio/foo/bar/FooBar"`. This is why we require an `extNamespace` to be set. If you are determined to not set an `extNamespace`, you can switch off the check in `blueprint-parent/pom.xml` via the property `extNamespaceRequired`. Configuring an `'extNamespace'` in a module where there are some ActionScript or MXML files whose fully-qualified name does not start with that prefix will result in a build error. You must choose another `'extNamespace'` or change the fully-qualified names (rename refactoring) so that they start with the prefix.

Finish this step by committing all added `extNamespace` properties, so that you can start the next step with a clean state:

```
$ git add -A
$ git commit -m "Complement extNamespace values"
```

2. Enable All Extensions

Enable all extensions, so that disabled extensions with *Studio Client* modules are converted, too.

```
$ cd workspace-configuration/extensions
```

Save current extension enabling state to restore it after conversion:

```
$ mvn extensions:list -DextensionsFile=extensions.txt
```

Enable all extensions:

```
$ mvn extensions:sync "-Denable=*"
```

Return to the top workspace directory:

```
$ cd ../../
```

3. Patch Studio Client blueprint-parent POM [2107.3 only]

For the Pre-Release, a minor adjustment is necessary to reference the matching CoreMedia core artifacts. In file `apps/studio-client/blueprint-parent/pom.xml`, complement the XML element `<npmDependencyOverrides>` [line 523] by the following entry:

```
<npmDependencyOverride>
  <search>^(@coremedia/studio-client[.].*).*$</search>
<replace>$1:^[cm.studio-client.core.version]-${jangaroo.version}</replace>
</npmDependencyOverride>
```

Alternatively, to automate this, you can use the following `sed` command:

```
$ sed -i
's#^\(\\s*\)</npmDependencyOverride>#\\1</npmDependencyOverride>\\n\\1<npmDependencyOverride>\\n\\1
<search>^(@coremedia/studio-client[.].*).*$</search>\\n\\1
<replace>$1:^[cm.studio-client.core.version]-${jangaroo.version}</replace>\\n\\1</npmDependencyOverride>#'
apps/studio-client/blueprint-parent/pom.xml
```

4. Invoke Workspace Conversion Tool

Next, the workspace conversion tool is invoked. It is integrated into the 4.1 version of Jangaroo, and as such invoked through Maven.

Change into the *Studio Client* workspace:

```
$ cd apps/studio-client
```

Invoke the conversion tool:

```
$ mvn clean install
net.jangaroo:jangaroo-maven-plugin:4.1.17:convert-workspace -DskipTests
-Dmaven.compiler.migrateToTypeScript
-Dmaven.compiler.suppressCommentedActionScriptCode
-DprojectExtensionWorkspacePath=$PWD
-DconvertedWorkspaceTarget=../studio-client-ts -Djangaroo.version=4.1.17
-Djangaroo.libs.version=4.1.8
```

CAUTION

Under Windows, `$PWD` must be replaced by `%CD%`.



This runs for a few minutes, generating an npm package under `'../studio-client-ts'` for each Maven module of your workspace, as well as some global files directly in `'../studio-client-ts'`.

If the conversion finishes without error, copy the generated workspace over the former *Studio Client* workspace.

Return to the top workspace directory:

```
$ cd ../../
```

Remove module reference to *Studio Client* Maven project in root POM:

```
$ sed -i '/<module>apps\/studio-client<\/module>/d' pom.xml
```

Remove old *Studio Client* workspace:

```
$ git restore apps/studio-client
$ git rm -r apps/studio-client/
$ rm -rf apps/studio-client/
```

Move new *Studio Client* workspace to its final destination:

```
$ mv apps/studio-client-ts apps/studio-client
```

5. Reformat TypeScript Code

The generated TypeScript code keeps some formatting of the original ActionScript/MXML code. To end up with cleanly formatted code, you should run a "linter" tool. Since this is invoked through *npnm*, you must first initialize the workspace, only then you can run the linter:

```
$ cd apps/studio-client
$ pnpm install
$ pnpm -r run lint --no-bail
```

The preconfigured linter rules (`.eslintrc.js`) take care of standardized indentation, spacing before and after certain keywords or tokens, replacing `var` by `let` or `const` as appropriate and consolidating and sorting imports, removing unused ones.

Return to the top workspace directory:

```
$ cd ../../
```

6. Disable Previously Disabled Extensions

After conversion, the enabled/disabled state of your workspace extensions should be restored.

Change into the extensions directory:

```
$ cd workspace-configuration/extensions
```

This change applies to 2107.3 only, in newer AMPs, its has already been applied. Extend extension tool configuration in its `pom.xml` by the `apps/studio-client` `npm` workspace root:

```
$ sed -i -E 's#(\s*)<projectRoot>../../</projectRoot>#\1<projectRoot>\n\1<projectRoot>../../</projectRoot>\n\1<projectRoot>../../apps/studio-client</projectRoot>\n\1</projectRoot>#' pom.xml
```

Alternatively, if you want to change this manually, in the `pom.xml`, replace the line

```
<projectRoot>../../</projectRoot>
```

by the lines

```
<projectRoot>
<projectRoot>../../</projectRoot>
<projectRoot>../../apps/studio-client</projectRoot>
</projectRoot>
```

Invoke the extensions tool to restore the former state:

```
$ mvn extensions:sync -Dextensions-maven-plugin.version=4.5.1
-DextensionsFile=extensions.txt
```

Remove the extensions file, as it is no longer needed:

```
$ rm extensions.txt
```

Again, return to the top workspace directory:

```
$ cd ../../..
```

7. Save Workspace State

Now would be a good point to save your work by committing it on your v11 upgrade branch.

Add all changed and new files and commit everything:

```
$ git add -A
$ git commit -m "Conversion of Studio Client to TypeScript"
```

8. Build the Converted Workspace

The generated npm workspace should now work and can be built the way a CoreMedia v11 *Studio Client* is built from now on. Follow the prerequisites given in [Section 3.1, "Prerequisites for the Upgrade"](#) [17]. Optionally consult the updated [Blueprint Developer Manual](#) and [Studio Developer Manual](#) for more information.

Change into the *Studio Client* workspace:

```
$ cd apps/studio-client
```

To build the workspace, run

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

Both commands output several messages, should not report (fatal) errors and end with a success message. In case of errors, please double-check you followed the instruction up to this point, check the CoreMedia help center for updates or contact CoreMedia support.

NOTE

During the build some files might get changed or created, for example, `tsconfig.json` files. This is normal and expected. Commit these changes into your workspace.



9. Start Converted Studio Client

Next, try to start your *Studio Client* from npm for the first time. You need a running 2107 *Studio Server* to do so, either on your local machine or on any server reachable from your machine.

If Studio Server runs locally, the command to start the local *Studio Client* server is


```
$ pnpm -r run start --filter ./apps/main/app
```

If Studio (Server) runs under some URL `https://studio.acme.com/`, the command must be complemented like this:

```
$ pnpm -r run start --filter ./apps/main/app --  
--proxyTargetUri=https://studio.acme.com/
```

In any case, you can now open the given *Studio Client* URL <http://localhost:3000> in a supported browser (Firefox, Microsoft Edge, any Chromium-based browser for development) and you should see a 2107 Studio that looks exactly as before.

7.2.2 Completing Studio Client v11 Upgrade

After merging your converted custom Blueprint workspace with CoreMedia v11 as described in [Merge with v11.2110.1 Workspace \[26\]](#) is finished successfully, there are some mandatory and some optional steps to take as follows.

Before you start, make sure your *Studio Client* workspace root is the current directory:

```
$ cd apps/studio-client
```

1. Update Core Dependency Versions

Even after merging with a v11 Blueprint workspace, dependencies of your custom packages [former custom Maven modules] still reference the 2107 CoreMedia core *npm* artifacts [for example 2107.6.0-4.1.15] in their `package.json`. Such dependencies must be updated to the corresponding v11 artifacts, which is easily achieved using the new *pnpm* tooling:

```
$ pnpm -r update "@coremedia/studio-client.*@2110.1.0"
```

2. Build the Workspace

Continue by building the *Studio Client* *pnpm* workspace:

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

You may encounter further errors resulting from incompatibilities between your customizations and v11 [breaking] changes or as a result of type errors caused by your changes in existing Blueprint packages. See other sections of this Upgrade Guide, release notes for component "Studio Client" and [4 \[106\]](#) for guidance on how to fix such errors.

3. Start Studio Client from Custom v11 Workspace

Now that the v11-upgraded workspace has been built, again try to start *Studio*. You now of course need a running v11 system, either started locally or somewhere in your CI. Use the same command as above to start *Studio Client*:

```
$ pnpm -r run start --filter ./apps/main/app --  
--proxyTargetUri=https://studio.acme.com/
```

and open <http://localhost:3000> in the browser. Your customized v11 Studio should load.

Although TypeScript can find more errors at build-time, many of these are still ignored (see [Fixing TypeScript Type Errors \[106\]](#)), and of course, some bugs only occur at run-time. Thus, you may encounter JavaScript errors in the browser, resulting from incompatibilities between your customizations and v11 [breaking] changes. Try to narrow down which component is causing the error by analyzing error stack traces, debugging and adding console output. Again, consult all sections of this Upgrade Guide as well as all relevant release notes to resolve these issues. Contact CoreMedia support to help you with intractable errors.

4. Fixing TypeScript Type Errors

The converter cannot always generate "green" TypeScript code from ActionScript code, for several reasons:

- TypeScript supports much more precise types through generics, overloading, function signatures etc. Using a more precise type may identify existing type errors or implicit assertions that lead to type errors.
- TypeScript's DOM API, using these features, provides stronger typing, reporting type errors that went unnoticed using jangaroo-browser's ActionScript DOM API.
- Like for ActionScript, CoreMedia generated a TypeScript API from the official *Sencha Ext JS* documentation. Since the Ext JS ActionScript API is named "Ext AS", the new API is consequently called "Ext TS". Again, errors may surface because TypeScript can express (JavaScript) types more precisely. For example if a parameter is documented be a `string or a number`, the only possible ActionScript type is "any" `[*]`, while TypeScript allows to describe this using `string | number`. Such differences between Ext AS and Ext TS may lead to type errors in your code. Also, there may still be bugs and inconsistencies in the new Ext TS API. Please report such issues to CoreMedia support if you think you encountered one.
- While in ActionScript, an Ext JS Config object type was inaccurately represented by the class itself, the two are now clearly separated. The converter cannot always determine whether the ActionScript type 'Foo' is supposed to refer to the class 'Foo' or its Config type. In ambiguous cases, the type remains the class type, causing type errors if actually the Config type was meant.

It is not mandatory to fix all type errors to upgrade your *Studio Client* customizations to v11, but it is highly recommended for mid- to long-term maintenance of your code. Each npm package contains a file `jangaroo.config.js`, which again contains

a setting `ignoreTypeErrors`. This setting is true after initial conversion, so that you can build and run Studio in spite of type errors. However, if your customizations in an existing Blueprint *npm* package, which in v11 does not set `ignoreTypeErrors`, cause type errors, you must either re-introduce the property or fix these type errors.

When fixing type errors, start from a "bottom" package in the workspace dependency hierarchy, i.e. a package with no dependencies to other packages in your local *npm* workspace. Build, fix type errors, until no more remain, then set `ignoreTypeErrors` to false [or remove the entry altogether] in the corresponding `jangaroo.config.js` to save this state. Continue working up your way in the dependency hierarchy, until eventually, the whole *Studio Client* workspace builds without type errors.

Before starting to fix type errors, you should make yourself familiar with the new TypeScript syntax and specific utilities CoreMedia offers to integrate with Ext JS. [Section 7.3, "Changes Between ActionScript and TypeScript for Developers" \[111\]](#) contains a comprehensive description of how Studio ActionScript, MXML and properties files are (automatically) converted to TypeScript. This should help you as an experienced *Studio Client* developer to transfer your knowledge to the new syntax.

7.2.3 Type Error Fixing Hints

This section presents different types of (type) errors that may occur during and after conversion to TypeScript and gives guidance on how to fix them. The cases have been collected from our own experience with converted *CoreMedia Studio* code.

Property does not exist/Type incompatible: Maybe Config type was meant

In many cases where the TypeScript compiler complains about an undefined property or that the actual type is not compatible to the required type, the reason is that the instance type was used as the ActionScript surrogate of the 'Config' type.

In ActionScript/Jangaroo 4, an Ext component's Config type and its actual (instance) type is, while inaccurate, represented by the same type/class (for details, see [Section "Using the Ext Config System" \[120\]](#)). For Ext in TypeScript, the distinction has been restored. However, the Jangaroo ActionScript-to-TypeScript converter cannot always determine whether the ActionScript type is supposed to refer to the class or its Config type. When in doubt, it chooses the class. If this choice was wrong, the resulting JavaScript code still works, but you end up with type errors like the ones described above.

One case is, that the class type that is supposed to be a Config type is handed in to a function that requires a Config type, like a component constructor. The other case is,

that the variable or expression of that type is used to access `Config` properties that do not exist on the class, or only as private members.

The typical fix is to change type `Foo` to `Config<Foo>` in such cases. Maybe you must add an `import` directive for `Config` (from `"@jangaroo/runtime/Config"`), if it is not already there. The `Config` type is computed based on the class by the utility type `Config`. When using a class as well as its `Config` type, this saves you from having to use two separate imports. For details, see [Section "Using the Ext Config System" \[120\]](#).

Callback function parameter is incompatible

Many methods, most often for event listener registration, take a callback function as a parameter. In `ActionScript`, such parameters have the general type `Function`, and it is impossible to express the function signature. In `TypeScript`, however, the function signature can be expressed, and for example the `Ext JS` documentation actually documents most callback function signatures, so they have been included in `Ext TS`.

Having callback signature leads to improved IDE assistance and also reveals (subtle) type errors in `ActionScript` code after conversion to `TypeScript`. In most cases, the signature match failure is quite trivial and easy to fix. Read the `TypeScript` error message carefully, and it tells you which parameter must have a different type or is altogether obsolete.

In rare cases, callback type errors may also be the result of an error in `Ext TS`, which again usually results from an API documentation error in `Ext JS`. If you think you found such an error, please report it to `CoreMedia Support`. One shortcoming we are aware of is, that `Ext` event listeners receive the "options" object (`eOpts`) that was used when attaching the listener as an additional last argument. `Ext TS` callback signatures currently lack this parameter, because it is rarely used and only complicates declaring event signatures. In most cases, using the options in the event callback is not necessary. The event callback function is typically a method of the class that attaches it as event listener, so a private class field can be used to store additional information the listener needs.

Some Jangaroo utilities are no longer supported in TypeScript

Jangaroo 4's runtime library contains some remains of Jangaroo 2, which were not properly deprecated, but in the context of *CoreMedia Studio*, there was no real need to use them. If, however, you still used certain parts of this API, you will end up with compiler errors already when trying to convert to `TypeScript`. These errors are, in contrast to `TypeScript` type errors, "fatal" in the sense that they cannot be ignored. This section lists removed Jangaroo API and explains how to replace certain usages.

jojo.JavaScriptObject

Inheriting from this class used to signal Jangaroo 2 that instances of your class are considered primitive JSON objects. Jangaroo 2 supported limited ActionScript language features for such subclasses. Jangaroo 4 only kept the class for (partial) compatibility, but did not implement these semantics. Thus, in Jangaroo 4, classes extending `JavaScriptObject` were no different than classes extending nothing, `Object` or `Ext.Base`.

The solution is to just remove the `extends` clause from your class, or, to keep it even more compatible in some edge cases, use `Ext.Base` as the superclass. The only feature of `JavaScriptObject` your class might use is its convenient constructor, which copies all enumerable properties from the parameter object to `this`. This can be implemented in TypeScript simply by

```
for (let key in config) this[key] = config[key];
```

or, if you know the `config` value does not use the prototype chain, even simpler by

```
Object.assign(this, config);
```

jojo.DynamicClassLoader

If at all, this Jangaroo 2 utility class is used to dynamically load classes (because there are no compile-time usages that trigger their loading). With Jangaroo 4, all class loading and script loading was delegated to `Ext`, but the API still existed for compatibility. This mechanism is used internally by the *CoreMedia Studio* plugin implementation, but should usually not occur in your plugin code.

If you used it anyway, you probably mimicked the pattern used by Studio core code, namely import several classes using `DynamicClassLoader.INSTANCE.import_ ("...")`; then adding a callback for when these classes have been loaded using `DynamicClassLoader.INSTANCE.complete (callback)`;

This pattern can be replaced by using `Ext`'s API directly:

```
Ext.require ("...");
Ext.onReady (callback);
```

jojo.getOrCreatePackage()

This utility method returns or creates the JavaScript object representing an ActionScript package at runtime. In Jangaroo 4, an ActionScript package is implemented as an `Ext` "namespace". In clean code, you should never need to access an ActionScript package at runtime, but it sometimes was used as a trick to initialize "global" (package-scope) variables by privileged code and let it appear read-only (`const`) for all other code.

If you used `getOrCreatePackage ()` for this trick: It no longer works in TypeScript. To compensate for that, there is a new utility function `lazyConst`, which is used like so:

```
import { lazyConst } from "@jangaroo/runtime";
import MyServiceInterface from "...";
import MyServiceImpl from "...";

const myService: { readonly _: MyServiceInterface } = lazyConst(() => new
MyServiceImpl());
export default myService;
```

For all other usages of `getOrCreatePackage(...)`, try replacing it by the corresponding Ext API, namely `Ext.namespace(...)` [or its alias `Ext.ns(...)`].

7.3 Changes Between ActionScript and TypeScript for Developers

This section is intended for developers, who already have experience with Studio Client programming in ActionScript/MXML. You will learn the differences to the new TypeScript approach.

7.3.1 ActionScript → TypeScript

Basics

- `public` is the default in TypeScript, so all `public` modifiers are gone
- Class members do *not* use `function`, `var` or `const` keywords in TypeScript. Methods use parenthesis, `const` becomes `readonly`.
- The constructor is not named like the class, but uses the well-known name `constructor`. No return type is specified.
- This built-in ActionScript classes `String`, `Number`, `Boolean` are usually used in the "unboxed", lower-case variants `string`, `number`, `boolean` in TypeScript.
- `this` must always be used explicitly (like in JavaScript)
- `private` is available in TypeScript, but only affects the compiler. To be private for subclasses and at runtime, there is a new JavaScript syntax: Using a member name that starts with `#` makes it private.
- A `set` accessor may not specify a `void` return value in TypeScript.
- The ActionScript "untyped type" `*` maps to TypeScript's `any`.
- In ActionScript, parameter default values are only applied if the function/method is called without an argument for that parameter. In TypeScript, parameter default values are also applied if the function/method is called with a given argument that is `undefined`. This is a subtle difference and only rarely results in different semantics, but note that the conversion tool does not take care of these rare cases, so they can lead to undesired behavior and must be fixed manually.
- While in ActionScript, typed class fields are implicitly assigned a default value (for example, `0` for an `int`), in TypeScript, all default values must be given explicitly. Otherwise, the field would remain `undefined`.

- ActionScript classes may contain "top-level" code, optionally enclosed in curly braces, which is executed when the class is first loaded / used. In TypeScript, such code is put in a block prefixed by the `static` keyword.

ActionScript

```
public class Foo extends SuperFoo {
    public static const FOO: * = "FOO";
    public var foo: String;
    private var _bar: Number;

    public function Foo(newBar: Number) {
        super();
        _bar = newBar;
    }

    public function get bar(): Number {
        return _bar;
    }

    public function set bar(value: Number): void {
        _bar = value;
    }

    protected function hook(): Boolean {
        return false;
    }

    Registry.register(Foo);
}
```

TypeScript

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

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

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

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

    protected hook(): boolean {
        return false;
    }

    static { Registry.register(Foo); }
}
```

Table 7.1. Basic ActionScript-TypeScript example comparison

Interfaces

While ActionScript and TypeScript have a notion of interfaces and even use the same keyword, the semantics are different.

In ActionScript, an interface must be implemented by a class explicitly. It can be checked at run-time whether an object is an instance of a given interface, using the ActionScript built-in operator `is`. This means that an interface must have some run-time representation.

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. To simulate this, you have to provide custom functions that test a given object (*type assertions*).

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 is compiled to JavaScript using the Ext class system, there already is 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 `mixin`s hierarchy of the object's class.

We use a similar approach in TypeScript. An 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 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 utility function `mixin`, which is usually imported as a *named import*, so you'll see it simply as `mixin(Claazz, Interface1, ..., InterfaceN)` after the class declaration.

Furthermore, in ActionScript interfaces, only methods may be declared. TypeScript allows to declare fields in interfaces, too. Note that a field and an accessor pair is considered two different things in both languages.

The following example illustrates the differences. Assume that accessor pair `foo` was meant as a field (which ActionScript cannot express in an interface), while `bar` was intended to be implemented as an accessor pair. The underscore prefix of `_bar` is left out in TypeScript, as `#` private names do not clash with public names, anyway.

ActionScript

```
interface IFoo {
    function get foo(): String;
    function set foo(value: String): void;

    function get bar(): Number;
    function set bar(value: Number): void;

    function isAFoo(obj: *): Boolean;
}

public class Foo implements IFoo {
    private var _foo: String;
    private var _bar: Number;

    public function get foo(): String {
        return _foo;
    }

    public function set foo(value: String): void {
        _foo = value;
    }
}
```

TypeScript

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

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

    abstract isAFoo(obj: any): boolean;
}

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

    get bar(): number {
```

ActionScript

```
public function get bar(): Number {
    return _bar;
}

public function set bar(value: Number): void {
    _bar = value;
}

public function isAFoo(obj: *): Boolean {
    return obj is IFoo;
}
```

TypeScript

```
return this.#bar;
}

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

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

mixin(Foo, IFoo);
```

Table 7.2. Interfaces ActionScript-TypeScript example comparison

Note that `is` and `mixin` must be imported from Jangaroo Runtime as a named import, so they can be used simply as `is` and `mixin`. See below for details about importing and exporting in TypeScript.

Imports and Exports

In ActionScript, each *compilation unit* (usually a class, but interfaces and global variables, constants or functions are also compilation units) has a *fully qualified name* that is *globally unique*. This name is used to reference other compilation units when *importing* them.

As TypeScript is an extension of ECMAScript, it uses the ECMAScript module system. Since ES5, any source file that contains `imports` and/or `exports` is a *module*. In `import` statements, 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.

NPM packages are the counterparts of Maven modules, and ActionScript packages correspond to TypeScript (ECMAScript) modules, so there is quite some potential for terminology confusion!

So each Jangaroo ActionScript Maven module has been mapped to a Jangaroo NPM package, and each different ActionScript package within such module has been mapped to a relative path to a TypeScript module within that package.

Thus, a main difference between Jangaroo Maven/ActionScript and NPM/TypeScript is that because ActionScript fully-qualified names are globally unique, import references in ActionScript are agnostic of the Maven module (there only must be a Maven dependency), while import references in TypeScript contain the target's NPM package name.

We map Jangaroo Maven module names to NPM packages names through same pattern-matching-magic, considering that NPM package names should use a *scope* [here: `@coremedia` for core and `@coremedia-blueprint` for Blueprint packages], separated by a slash from the actual NPM package name.

To avoid unnecessary deep paths for TypeScript modules in NPM packages, we mapped the ActionScript package names within one Maven module to a shortened relative path in the NPM package. We configured the longest possible common ActionScript package name prefix to be left out for the relative path.

As an example, we use the Maven module `com.coremedia.ui.toolkit:client-core`, which is mapped to `@coremedia/studio-client.ext.client-core`. The Maven module contains ActionScript packages `com.coremedia.ui.data`...and `com.coremedia.ui.util`. We tell the `jangaroo-maven-plugin` that the desired redundant prefix to remove during TypeScript/NPM conversion is `com.coremedia.ui` through the configuration option `extNamespace`:

```
<plugin>
  <groupId>net.jangaroo</groupId>
  <artifactId>jangaroo-maven-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <extNamespace>com.coremedia.ui</extNamespace>
    ...
  </configuration>
</plugin>
```

Example 7.1. Removing redundant prefix configuration

Imports

Taken all together, as an example, the ActionScript interface `com.coremedia.ui.data.RemoteBean` is converted to the TypeScript module reference `@coremedia/studio-client.ext.client-core/data/RemoteBean`.

Here is the ActionScript and TypeScript import syntax in direct comparison:

ActionScript	TypeScript
<pre>// from within the same Maven Module and // ActionScript package: // no import necessary! // from with the same Maven Module, but // ActionScript package com.coremedia.ui.util: import com.coremedia.ui.data.RemoteBean;</pre>	<pre>// from within the same NPM package and source // folder: import RemoteBean from "../RemoteBean"; // from within the same NPM package, but source // folder /util: import RemoteBean from "../data/RemoteBean";</pre>

ActionScript	TypeScript
<pre>// from another Maven module: import com.coremedia.ui.data.RemoteBean;</pre>	<pre>// from another NPM package: import RemoteBean from "@coremedia/studio-client.ext.client-core/data/RemoteBean";</pre>

Table 7.3. Imports ActionScript-TypeScript example comparison

Export

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

While it is possible to combine the declaration and the [default] export of a class, the conversion tool does not do so, because later you'll see cases where that is not possible or more redundant. So for consistency, the conversion tool always ends each source file with the default export:

```
default export Foo;
```

A Complete Example Class

Using all the ingredients of the previous sections, the complete example class could look like so:

```
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 7.2. TypeScript example class

Mixins

Ext JS allows mixins to achieve multiple inheritance between classes. Since neither ActionScript nor TypeScript support mixins out of the box, we had to find some way to represent them in both languages.

Mixins in ActionScript

In ActionScript, a class can only extend one other class, but it can implement multiple interfaces. So for a class to be used as a mixin, we extract an interface from that class with a custom ActionScript annotation `[Mixin("fully-qualified name of mixin implementation class")]`. Any class `MixinClient` that implements such an interface `acme.IMyMixin` annotated with `[Mixin("acme.MyMixin")]` becomes a mixin client class, in other words, `MyMixin` is mixed into `MixinClient`. Because ActionScript tools (asdoc, IDEA) do not know of this magic annotation, `MixinClient` also must implement all `IMyMixin` methods to comply with ActionScript semantics. As we do not *really* want to implement these methods, we just declare them, using the `native` keyword.

```

// ./acme/IMyMixin.as
package acme {

[Mixin("acme.MyMixin")]
public interface IMyMixin {
    [Bindable]
    function get mixinConfig(): String;

    [Bindable]
    function set mixinConfig(value: String): void;

    function doSomething(): Number;
}

```

```

}
}

// ./acme/MyMixin.as
package acme {

public class MyMixin implements IMyMixin {
    private var _mixinConfig: String = "";

    [Bindable]
    public function get mixinConfig(): String {
        return _mixinConfig;
    }

    [Bindable]
    public function set mixinConfig(value: String): void {
        _mixinConfig = value;
    }

    public function doSomething(): Number {
        return _mixinConfig.length;
    }
}

}

// ./MixinClient.as
import acme.IMyMixin;
import ext.Component;

public class MixinClient extends Component implements IMyMixin {
    [Bindable]
    public native function get mixinConfig(): String;

    [Bindable]
    public native function set mixinConfig(value: String): void;

    public native function doSomething(): Number;

    public function MixinClient(config: Object = null) {
        super(config);
        doSomething();
    }
}

```

Example 7.3. Mixin in ActionScript example

Mixins in TypeScript

In TypeScript, we use a quite similar approach like in ActionScript, but fortunately, the syntax is much more elegant.

To understand how mixins work, it helps to know that in TypeScript, a class consists of a its run-time JavaScript *value* and a *type*, which is only relevant for type checking / 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 run-time extend JavaScript class **B**] and the type [TypeScript type **A** will at compile-time be a sub-type of type **B**]. When using a class identifier behind a colon or in the

`implements` clause of a class, only its type aspect is 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. In this case, TypeScript does *not* complain about the class not implementing the additional interface methods. We call such an interface a *companion interface* of the class, as it comes together with the class and adds more declarations (the ones we had to declare as `native` in ActionScript).

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

As in Ext JS, a mixin is a usual 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 run-time. We 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. This is much more elegant in TypeScript than in ActionScript: Instead of re-declaring every single member using the `native` keyword, we just declare a *companion interface* of the mixin client class and let that extend the mixin class interface. We could even leave out the `implements` clause of the mixin client class itself, but to emphasize what's going on (and to help some IDEs that don't really support declaration merging completely), during conversion, we generate both clauses.

All in all, the above example results in the following quite more compact TypeScript code.

```
// ./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 7.4. Mixins in TypeScript example

Using the Ext Config System

A major part of Studio Client ActionScript/MXML code deals with Ext JS components, plugins, actions, and other Ext JS classes that have in common that they use the Ext Config system.

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 Configs in Ext 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 – not really 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.

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 7.5. 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.

"Bindable" Configs in Ext 6

Things became more complicated with the new class and Config system introduced with Ext 4 (CoreMedia upgraded directly to Ext 6, later to 7). Configs now can be declared for an Ext class and then trigger some magic: For every Config property `foo`, Ext generates methods `getFoo()` and `setFoo(value)`. Note that these are *not* accessors, but "normal" methods, as Ext 4 came out when browser support for accessors was not yet mainstream. They never managed to update their Config system to "real" accessors.

To make things "easier" [?] for the developer, things get even more complicated: 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)`

Neglecting this additional "convenience", this 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 when the text is changed afterwards:

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

var label = Ext.create({ xtype: "acme.label", text: "Hi!" });
```

```
label.setText(null);
console.log(button.getText()); // logs the empty string (""), not "null"
```

Example 7.6. Bindable configs

Using the Ext 6 Config System in ActionScript

The goal of using ActionScript for Ext JS was to control the comprehensive framework with *static typing*.

So we really didn't like magically appearing methods, and we also didn't want the developer having to declare five members for one config (the config property itself and its get-, set-, update- and apply-method). Thus, for ActionScript, we "faked" Ext being more modern than it was [and is] and pretended that you could use "real" accessors to access a Config's get- and set-method, and to overwrite these methods.

Declaring Ext Configs in ActionScript

In ActionScript, the above example would look like so:

```
package acme {
public class Label extends Component {
    public static const xtype: String = "acme.label";

    private var _text: String = "";

    public function Label(config: Label = null) {
        super(config);
    }

    [Bindable]
    public function get text(): String {
        return _text;
    }

    [Bindable]
    public function set text(value: String): void {
        text = typeof value === "string" ? value : value ? String(value) : "";
        // if rendered, update my DOM node with 'value'
    }
}
}

// ActionScript code using this component:

var label = new acme.Label(acme.Label({ text: "Hi!"}));
label.text = null; // <= !!! translated to JS: label.setText(null) !!!
console.log(button.text); // <= !!! translated to JS: label.getText() !!!
```

Example 7.7. Declaring Ext configs in ActionScript

The `[Bindable]` annotation is the hint for Jangaroo to convert ActionScript accessors not to JavaScript accessor, but to Ext get/set *methods*. So in ActionScript, the Config

access looks like a property, but really calls the get/set methods. To be more precise, this decision is made at run-time, by rewriting such code to use the utility methods `getBindable(obj, config)` or `setBindable(obj, config, value)`.

To know when to replace what looks like a property access by a "bindable" access, the Jangaroo compiler resolves the declaration of the property and looks for a `[Bindable]` annotation. Unfortunately, in TypeScript, this is no longer possible, so we had to find a way to represent "bindable" Configs in TypeScript which a) looks like meaningful TypeScript code and b) can be translated to the Ext semantics.

Long story short, we found a way to add "real" accessors for all Configs defined by ActionScript code at runtime! So in TypeScript, when we write `label.text`, it actually resolves to an accessor that delegates to the `getText` or `setText` method. We even introduced this change for the "normal" JavaScript output mode in Jangaroo 4.1, and it seems to work fine!

One more complication is that in real-world Studio ActionScript code, we used several same-same-but-different patterns to declare `[Bindable]` Configs. The code above is probably the most clean way to do it, but unfortunately another field `_text` is created in addition to the "internal" Config field `text`, which is less efficient and can even lead to confusion. We tried to capture all patterns we identified and automatically translate them to a single TypeScript pattern for bindable Configs, which is presented in the next section.

As you can see in the constructor of the example class, in ActionScript, we use the same type for the class and for the Config type. This is actually cheating. The class has far more properties (methods!) than the Config type, and in Ext, some class properties even have a different type than their Config property counterpart (for example, `Container#items` is an `Array` in the Config type, but a `MixedCollection` in the class).

The reason we do not declare two separate types (as we did in Jangaroo 2) is that ActionScript only allows one exported declaration per file. So to declare the Config type separately, every class would need two files, and both would still have to be consistent (extend the same superclass, use the same mixins). We did not want developers to deal with that.

Creating Ext Config Objects in ActionScript

Ext often uses the concept of creating Config objects first, and instantiating them later. To be able to do so, a Config object must contain an property that indicates which class to instantiate (later). 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>`, plugins use `plugin.<type>`, GridColumns use `gridcolumn.<type>`. The `type` property used for that purpose before introducing `alias` has been deprecated.
- `xclass` — Introduced last, this is the most straight-forward 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.

That said, in ActionScript, you need not worry about all that. We introduced special semantics to ActionScript *type casts* when using them on object literals. (This is quite some cheating, too.) To create a Config object for a class `MyClass`, instead of calling its constructor, you type-cast an object code into that class:

```
var myClassConfig: MyClass = MyClass({
    id: "4711", // inherited from Component.
    configOption1: "bar", // MyClass Config property
    configOption2: 42 // the other MyClass Config property
});
```

Example 7.8. Type-cast object code into class

This adds the appropriate `xtype`, `alias`, and/or `xclass` attributes.

However, in ActionScript, there is no way to type object literals. They are always of type `Object` and any properties are allowed. That's why you should use the following pattern to populate a Config object in ActionScript:

```
var myClassConfig = MyClass({});
myClassConfig.id = "4711";
myClassConfig.configOption1 = "bar";
myClassConfig.configOption2 = 42;
```

Example 7.9. Populate Config object in ActionScript

While this adds a bit more code, you now have type checks and gain IDE support like completion, documentation lookup and navigation.

If needed, you can then use different ways to instantiate the corresponding class:

```
var myClassInstance: MyClass = new MyClass(myClassConfig);
// OR
var myClassInstance: MyClass = Ext.create(myClassConfig); // careful: no type
check!
```

```
// OR
var myClassInstance: MyClass = MyClass(Ext.create(myClassConfig)); // at
least a run-time type check
```

Example 7.10. Different ways to instantiate class

Using the Ext 6 Config System in TypeScript

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 re-appear on the class itself. Also, we need to distinguish simple Configs and bindable Configs. Last but not least, Config objects usually only contain 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 `?`.

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);
    }
}
```

```

    }
}

```

Example 7.11. Config properties in class

To also *export* the additional interface, the most straight-forward option seemed like using a named export. But this has disadvantages when using both the class and its Config type, because you need two import identifiers, especially 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 named `Config`.

```

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

```

Example 7.12. Declaring a virtual class member

This allows to access the Config type by importing the class and then use the utility type also called `Config` (import from `@jangaroo/runtime/Config`). As this pattern is followed by all classes using the Ext Config System, also the Ext framework components, we can complement the example by extending the superclass Config type:

```

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

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

...

```

Example 7.13. 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 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 be added without a type error:

```

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

...

```

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

Example 7.14. TypeScript detecting type errors for existing properties

Being able to use undeclared properties without warning is not desirable. Fortunately, 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";

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

Example 7.15. Preventing use of untyped properties

We just added the type of `myClassConfig` for clarity, you can 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 that Config object complemented by `xclass` / `alias` / `xtype` properties.

Since TypeScript is more strict when checking the type of function argument 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 “Creating Ext Config Objects in ActionScript” \[123\]](#)]. So we need a counterpart of the special type cast semantics we introduced in ActionScript.

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";

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

```

Example 7.16. 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, it should never be necessary to specify it explicitly, just leave it to the 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.

As TypeScript *can* type object literals, it is no longer recommended to populate a `Config` object property-by-property [see [Section “Creating Ext Config Objects in ActionScript” \[123\]](#)]. “Not recommended” means, this is of course still possible, and still results in strictly typed code. Note that the AS→TS conversion compiler does not (yet?) rewrite such code to use object literals.

In the rare case 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";

    const myClassInstance: MyClass = new MyClass(myClassConfig); // xclass
of Config object is ignored
    // OR
    const myClassInstance: MyClass = Ext.create(MyClass, myClassConfig); //
xclass of Config object is ignored
    // OR
    const myClassInstance: MyClass = Ext.create(myClassConfig); // must repeat
target class, but incompatible class and Config type would be reported
    // OR
    const myClassInstance = Ext.create<MyClass>(myClassConfig)); // must
repeat target class, but incompatible class and Config type would be reported
```

Example 7.17. 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 now 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 7.18. 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(...)`.

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(Object.assign(Config<MyClass>({
        id: "4711",
        configOption1: "bar",
        configOption2: [42, 24]
    })), config));
}

```

Example 7.19. Typical work of constructor done in TypeScript

However, there is a special utility class named `ConfigUtils` (formerly in ActionScript: `Exml`) 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, in your constructor, you should use the following:

```

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

```

Example 7.20. Using ConfigUtils utility class

Any client using such a component can then use

```

import ConfigUtils from "@jangaroo/runtime/ConfigUtils";
...
    Config(MyClass, {
        id: "4711",
        configOption1: "bar",
        ...ConfigUtils.append({
            configOption2: [12]
        })
    })), config));
}

```

Example 7.21. 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 surrounding object code, the spread operator `...` is used.

7.3.2 MXML → TypeScript

MXML is part of Flex and used as a more declarative alternative to specify ActionScript classes. Each MXML source file is internally translated to ActionScript.

This is also true for the conversion to TypeScript: MXML is first internally converted to ActionScript and then to TypeScript. However, there are some specialties that only apply in this mode, described in this chapter.

How MXML Translates to ActionScript

MXML is, like the name suggests, an XML format. This is also the file extension used, so the following example would be located in `acme/MyMxmlClass.xml`. Let's have a look at its basic structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Class documentation, corresponds to /** .. */.
-->
<acme:MyClass xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:exml="http://www.jangaroo.net/exml/0.8"
  xmlns="exml:ext.config"
  xmlns:acme="acme.*"
  configOption1="foo"
  configOption2="{41 + 1}">
  <fx:Script><![CDATA[
    public static const xtype:String = "acme.myMxmlClass";

    private var config: MyMxmlClass;

    public native function MyMxmlClass(config: MyMxmlClass = null);
  ]]></fx:Script>

  <fx:Declarations>
    <!--
      Config option 3 documentation.
    -->
    <fx:String id="configOption3">default value for 3</fx:String>
  </fx:Declarations>
  <!-- complex Config property values are specified as nested elements:
-->
  <acme:items>
    <!-- multiple elements result in an Array: -->
    <exml:Button id="myButton" label="Click me!" />
    <exml:Separator />
  </acme:items>
  <acme:plugins exml:mode="append">
    <acme:MyPlugin .../>
  </acme:plugins>
</acme:MyClass>
```

Example 7.22. Example MXML file

This example defines an ActionScript class `MyMxmlClass` in MXML:

- Several *namespaces* are declared to access ActionScript packages (`"acme.*"`) and so-called *libraries* (URL format). Libraries aggregate several classes from different packages and may define an alias for the class name (to prevent name-clashes).
- `MyMxmlClass` inherits from `MyClass` - this is specified by using `MyClass` as its *root node*.
- Config properties are given as either XML attributes (`configOption1`, `configOption2`) or as XML sub-elements (`<acme:items>`, `<acme:plugins>`).
- *Interpolations* are used to compute property values. They use curly braces `{ ... }` that can contain arbitrary ActionScript expressions.
- The `<fx:Script>` element contains additional class members as ActionScript code. "Real" Flex MXML may not define a custom constructor, but to declare an Ext JS specific constructor signature, receiving the `config` argument, it is possible to declare a *native* constructor. To be able to access this `config` parameter in interpolation expressions, it is re-declared as a "virtual" field. Since you cannot define *static* members in declarative MXML, the script block is also used to set the component's `xtype`.
- Additional class fields can be declared and/or initialized using the `<fx:Declarations>` element. The `id` attribute specifies the name of the field. The initial field value is given like any other MXML element, that is the type or class to instantiate is determined through the element name, and the value or the constructor parameter's Config properties are given as XML attributes or sub-elements. If a field with the given name already exists (either inherited or defined in the `<fx:Script>` block), only this initial value is used and assigned to the existing field in generated constructor code.
- Finally, there is a Jangaroo addition to MXML, the property element attribute `exml:mode`. Together with utility functions in class `net.jangaroo.exml.Exml`, it takes care of the given array value being appended or prepended to the property's current value, rather than replacing it (see [Section "Merging Config Objects" \[129\]](#)).