COREMEDIA CONTENT CLOUD

Headless Server Manual

COREMEDIA

List of Figures

List of Tables

List of Examples

1. Preface

This manual describes the concepts and configuration of and development with the *Headless Server*.

- Chapter 2, *Overview* [13] describes the aim, concepts and components of the *Headless Server*.
- Chapter 3, *Configuration and Operation* [15] describes the configuration, deployment and preview integration of the *Headless Server*.
- Chapter 4, *Development* [25] describes how to extend the *Headless Server*.
- Chapter 5, *Rich Text Output* [56] describes how to process Rich Text with the *Headless Server*.
- Chapter 6, *eCommerce Extension* [77] describes how to use eCommerce with the *Headless Server*.
- Chapter 8, *Persisted Queries* [91] describes how to configure and use Persisted Queries with the *Headless Server*.
- Chapter 9, *REST Access to GraphQL* [97] describes how to map HTTP GET requests to persisted GraphQL queries and how to transform the result.
- Chapter 10, *Site Filter* [101] describes the usage of site filters to get only content, belonging to one site.
- Chapter 11, *Metadata Root* [102] describes how you can get metadata for fields for preview driven editing functionality.
- Chapter 12, *Frontend Client Development* [105] describes how you can create a progressive web app for the Headless Server using React and describes some main concepts.
- Chapter 13, *Configuration Property Reference* [121] links to the configuration properties for the Headless Server.

1.1 Audience

This manual is intended for architects and developers who want to work with *CoreMedia Content Cloud* or who want to learn about the concepts of the product. The reader should be familiar with *CoreMedia CMS*, *Spring*, *Maven*, *GraphQL* and, optionally, the commerce system to connect with.

1.2 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 1.2.1, “Registration” [3] 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 1.2.1, “Registration” [3] describes how to register for the usage of the services.
- Section 1.2.2, “CoreMedia Releases” [4] describes where to find the download of the software.
- Section 1.2.3, “Documentation” [4] describes the CoreMedia documentation. This includes an overview of the manuals and the URL where to find the documentation.
- Section 1.2.4, “CoreMedia Training” [8] describes CoreMedia training. This includes the training calendar,the curriculum and certification information.
- Section 1.2.5, “CoreMedia Support” [8] describes the CoreMedia support.

1.2.1 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 1.2.5, “CoreMedia Support” [8]) by email to request further access depending on your customer, partner or freelancer status so that you can use the CoreMedia services.

1.2.2 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-10

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 1.2.1, "Registration" [3] for details about the registration process. If the problems persist, try clearing your browser cache and cookies.

Maven artifacts

CoreMedia provides 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 Blueprint Developer Manual .

License files

You need license files to run the CoreMedia system. Contact the support (see Section 1.2.5, "CoreMedia Support" [8]) to get your licences.

1.2.3 Documentation

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

https://documentation.coremedia.com/cmcc-10

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

Manual	Audience	Content
Deployment Manual	Developers, architects, administrators	This manual describes the concepts and usage of the CoreMedia deployment artifacts. That is the deployment archive and the Docker setup. You will also find an overview of the properties required to configure the deployed system.
Elastic Social Manual	Developers, architects, administrators	This manual describes the concepts and administration of the *Elastic Social* module and how you can integrate it into your websites.
Frontend Developer Manual	Frontend Developers	This manual describes the concepts and usage of the Frontend Workspace. You will learn about the structure of this workspace, the CoreMedia themes and bricks concept, the CoreMedia Freemarker facade API, how to develop your own themes and how to upload your themes to the CoreMedia system.
Headless Server Developer Manual	Frontend Developers, administrators	This manual describes the concepts and usage of the *Headless Server*. You will learn how to deploy the Headless Server and how to use its endpoints for your sites.
Importer Manual	Developers, architects	This manual describes the structure of the internal CoreMedia XML format used for storing data, how you set up an *Importer* application and how you define the transformations that convert your content into CoreMedia 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 or the usage of the watchdog component.
Search Manual	Developers, architects, administrators	This manual describes the configuration and customization of the *CoreMedia Search Engine* and the two feeder applications: the *Content Feeder* and the *CAE Feeder*.
Site Manager Developer Manual	Developers, architects, administrators	This manual describes the configuration and customization of *Site Manager*, the Java based stand-alone application for administrative tasks. You will learn how to configure the *Site Manager* with property files and

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

Table 1.1. CoreMedia manuals

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

Email: documentation@coremedia.com

1.2.4 CoreMedia Training

CoreMedia's training department provides you with the training for your CoreMedia projects either 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

1.2.5 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 1.2.1, “Registration” [3]. The support email address is:

Email: support@coremedia.com

Create a support request

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:

- 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.

Support checklist

To put it in a nutshell, CoreMedia needs:

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

Log 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. The location where component log output is written is specified in its `logback.xml` file.

Which Log File?

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

Where do I Find the Log Files?

By default, log files can be found in the CoreMedia component's installation directory in `/var/logs` or for web applications in the `logs/` directory of the servlet container`.` See Section 4.7, “Logging” in *Operations Basics* for details.

1.3 Typographic Conventions

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

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

Code lines in code examples which continue in the next line	\	`cm systeminfo \`
		`-u user`

Table 1.2. 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.
ϟ	Danger: The violation of these rules causes severe damage.

Table 1.3. Pictographs

1.4 Changelog

The following table lists all changes that have been applied to the manual since its first publication.

Section	Version	Description

Table 1.4. Changes

2. Overview

CoreMedia *Headless Server* is a CoreMedia component which allows access to CoreMedia content as JSON through a GraphQL endpoint.

The generic API allows customers to use CoreMedia CMS for **headless** use cases, for example, delivery of pure content to native mobile applications, smartwatches/wearable Devices, Out-of-Home or In-Store Displays or Internet-of-Things use cases.

CoreMedia *Headless Server* provides an additional way of content delivery:

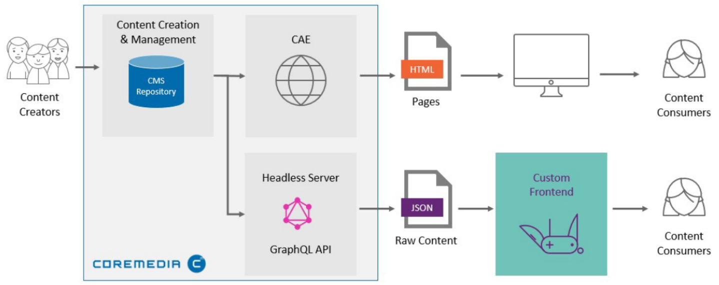


Figure 2.1. Headless Server overview

The *Headless Server* comes with the following feature set:

- Access through a GraphQL endpoint
- GraphQL **schema support** for CoreMedia content types with type inheritance (see Chapter 4, *Developing a Content Type Model* in *Content Server Manual* for details of CoreMedia content types).
- Support for **Spring EL** in GraphQL schemas
- Access to **CoreMedia businesslogic**
- Multi-Site/Language delivery
- Validity/Visibility of Content
- Navigation and Page Grid support
- Responsive Images
- Rich Text Transformation

- Image Maps, Shoppable Videos, Teaser with multiple targets, Videos in Banners
- Full Text Search
- Dynamic Query Lists
- eCommerce integration via *CoreMedia Commerce Hub*
- Studio JSON Preview Client which integrates in CoreMedia Studio
- Deployment as a Spring Boot application with Docker or Chef

3. Configuration and Operation

This chapter describes the configuration and operation of the *Headless Server*.

Deployment

The *Headless Server* is a Spring Boot application that can be deployed with Chef or Docker. See the Deployment Manual for details.

3.1 Configuration of the *Headless Server*

The *Headless Server* can be deployed in preview and live mode.

Together with the *Headless Server*, several tools can be deployed:

GraphiQL	An interactive tool to issue GraphQL queries and browse the GraphQL schema.
Swagger	A tool to query the REST API of the Media Controller.
JSON Preview Client	A Preview Client presenting GraphQL content query results in the Studio preview pane in the form of raw JSON data trees.

The configuration options of the *Headless Server* are listed in the Section 4.3, “Headless Server Properties” in *Deployment Manual*.

3.2 Endpoints of the *Headless Server*

For the *Headless Server* several endpoints are available.

GraphQL

GraphQL is the standard endpoint of the *Headless Server* and available at `/graphql`.

It serves GraphQL requests as specified on graphql.org.

GraphiQL

GraphiQL is a graphical interactive in-browser GraphQL IDE. See the GraphiQL GitHub repository for details.

The GraphiQL endpoint is, by default, enabled for the *Headless Server* in preview mode and available at `/graphiql`.

Swagger UI

Swagger UI is a tool to visualize and interact with REST resources. More information can be found at https://swagger.io/tools/swagger-ui/.

For the *Headless Server*, media objects are delivered via REST and can be inspected with Swagger UI.

Swagger UI is only available, if configured. Default, it is enabled for the *Headless Server* in preview mode and available at `/swagger-ui.html`.

JSON Preview and Preview URL Service

The JSON Preview and corresponding Preview URL Service are only available in *Headless Server* preview mode and provide a preview integration into CoreMedia *Studio*. See Section 3.3, “Preview” [19] for details.

Endpoints for JSON Preview and Preview URL Service are `/preview` and `/previewurl`.

REST

Persisted queries (see Chapter 8, *Persisted Queries* [91]) may be accessed by simple HTTP GET requests. As the persisted queries are customizable and freely definable by name, the endpoints are exposed dynamically relatively to the endpoint `/caas/v1/`. See Chapter 9, *REST Access to GraphQL* [97] for details.

All endpoints to persisted queries are documented automatically within the Swagger UI.

Site Filter

A site filter restricts the access of GraphQL queries to content objects of one site only. See Chapter 10, *Site Filter* [101] for details.

3.3 Preview

Data delivered by CoreMedia *Headless Server* can be previewed in CoreMedia *Studio* by integrating a corresponding preview client.

A basic preview client that renders *Headless Server* data as a raw JSON data tree is available in this workspace.

To display multiple Previews in Studio, the Multiple Previews Feature needs to be configured.

How to enable the multiple previews feature is described in Section 7.28, “Multiple Previews Configuration” in *Studio Developer Manual*.

3.3.1 JSON Preview Client

The JSON Preview Client is available together with a Preview URL Service in the module json-preview-client.

Deployment

The JSON preview client is deployed together with the *Headless Server*. The *Headless Server* has a dependency to `json-preview-client`. It is activated with property `caas.preview=true` which is set for headless-server-preview.

To remove the JSON Preview Client, either remove the dependency to `json-preview-client` or adapt `JsonPreviewClientConfig.java.`

JSON Preview Client Configuration

A JSON Preview is available for all content types that have a preview configured. To support specific documenttype properties, corresponding queries can be added in `content.graphql`.

As the JSON Preview Client is deployed together with the headless-server-preview, the following configuration needs to be applied to headless-server-preview:

- Endpoint of the Headless Server for the JSON Preview Client:

```
previewclient.caasserver-endpoint=http://[hostname]:41180/graphql
```

3.3.2 Custom Preview Client

For a custom preview client, a corresponding preview URL service needs to be set up. It should respond to preview URL requests for a given content ID with a URL where to fetch the actual preview HTML from the custom preview client.

The preview client needs to include `coremedia.preview.js` to enable communication with Studio.

To enable Preview Driven Editing (PDE), preview metadata tags need to be set (data-cm-metadata).

3.4 Security

The *Headless Server* is usually exposed to public access. Therefore the *Headless Server* needs an efficient protection.

GraphQL offers a self-descriptive approach to deliver data to client applications. This makes it easy to any client to use and visualize this data in any way, without the need to have an exact knowledge about the underlying data model, thus reducing the need for support. On the other hand, clients may request as much data as they wish, creating potentially high load on the server.

Protecting the *Headless Server* can be realized by two general approaches:

- Externally, before the *Headless Server* is actually invoked, using hardware (load balancers, firewalls) or a web server (gateway) in front of the *Headless Server*.
- On the application layer of the *Headless Server* by means of configuration.

The external approach is usually very efficient. You may enforce certain access restrictions by employing some kind of authorization and/or authentication or define IP access restrictions. However, this approach implies, that the clients are in some kind 'known' by the server. If you want to allow to access data by any client, this approach is hard to enforce.

Whenever it is not possible or not desirable to restrict access to known clients, you might use the application layer approach.

The *Headless Server* offers these options to employ security measures:

- Whitelisting of persisted GraphQL queries described in Section 3.4.1, “Whitelisting of GraphQL Queries” [22].
- Limiting the size of a search result described in Section 3.4.2, “Limiting the Size of a Search Result” [22].
- Limiting the depth of a GraphQL query described in Section 3.4.3, “Limiting the Depth of a GraphQL Query” [22].
- Limiting the complexity of a GraphQL query described in Section 3.4.4, “Limiting the Complexity of a GraphQL Query” [23].
- Enforce an execution timeout for GraphQL queries described in Section 3.4.5, “Enforcing an Execution Timeout for GraphQL Queries” [23].

All the above measures may be used to protect the server from expensive queries or malicious attacks.

NOTE

In order to provide a certain amount of protection by default, the size of a search result is limited to 200 hits and the maximum query depth is set to 30. Especially on protected preview servers, these limits are possibly not desirable, while developing or testing GraphQL queries and therefore should be reconfigured to suite your needs.

3.4.1 Whitelisting of GraphQL Queries

Query whitelisting means that only the persisted queries that reside on the server are allowed to execute. All other GraphQL queries are denied.

The whitelisting may be enabled by setting the configuration property `caas.persisted-queries.whitelist` to `true`. See Section 8.2, "Query Whitelisting" [95] for more details.

3.4.2 Limiting the Size of a Search Result

Allowing unlimited result sizes on search queries is probably the easiest way to produce high load on the server. Therefore, limiting the size of a search result to a maximum value is almost imperative. Whenever the requested limit exceeds the maximum allowed limit, the requested limit is overwritten by the maximum value before the search query is invoked.

The maximum search result limit is enabled by setting the configuration property `caas.graphql.max-search-limit` to a value greater than `0`. The default maximum search limit is 200.

When not requesting an explicit limit within a query, the default limit is 10. A configured maximum search limit which is smaller than the default limit, overwrites the default limit.

3.4.3 Limiting the Depth of a GraphQL Query

Any opening curly bracket in a GraphQL query marks the start of a new nesting level of the query. The depth of a query is then simply the deepest nested level. By limiting the depth of a query to a certain value, the size of the data is limited correspondingly. Furthermore, indefinite querying of circularly linked content is prevented. As the depth is calculated before actually invoking the query, the counter measure is quite efficient.

The depth limit is enabled by setting the configuration property `caas.graphql.max-query-depth` to a value greater than `0`. The default depth limit is 30.

3.4.4 Limiting the Complexity of a GraphQL Query

The higher the complexity of a query is, the higher is the resulting potential load on the server. The complexity of a query may be limited by a `MaxQueryComplexityInstrumentation` which is provided by the `graphql-java` framework. By default, the complexity of a query is calculated by summing up the number of requested fields and nested levels. A more sophisticated complexity calculator may be added to the Spring configuration by implementing the `FieldComplexityCalculator` interface from `graphql-java`. Like the query depth, the complexity of a query is calculated before actually invoking the query.

The complexity limit can be enabled by setting the configuration property `caas.graphql.max-query-complexity` to a value greater than `0`. The default is `0` which means that this check is disabled.

3.4.5 Enforcing an Execution Timeout for GraphQL Queries

As a last resort, it is possible to enforce a maximum time to process a GraphQL query. Whenever that time is exceeded, a timeout kicks in, aborting the query execution. As at this point of time the query was already invoked, this type of counter measure should be considered as a last resort. If the server is under such a high load, instead of enforcing an execution timeout, please consider counter measures outside of the *Headless Server*, as mentioned above. Besides, if there is no malicious attack, the server resources like the number of processors, or RAM size may be sized too small. In such cases, raising limited resources or deploying another instance of the *Headless Server* may be fitting solutions.

The timeout is implemented by the `ExecutionTimeoutInstrumentation` provided by CoreMedia and bundled with the *Headless Server*. It can be enabled by setting the configuration property `caas.graphql.max-query-execution-time` to a value greater than `0`. The default value is `0` which means that no timeout is checked.

The timeout is set in milliseconds. A reasonable value may be `2000` or `3000` (that is, 2 or 3 seconds). Also keep in mind, that the first invocation of a query on a new instance of the *Headless Server* may take much longer than the follow-up queries due to caching effects.

4. Development

This chapter shows how to use the *Headless Server* for your frontend applications.

The CoreMedia Headless Server is a GraphQL service implementation written in Java. It leverages the GraphQL Java Open Source framework and the accompanying Spring integration.

4.1 Defining the GraphQL Schema

GraphQL features a type system (see https://graphql.org/learn/schema/) which is independent of a particular implementation language. Types are written in a special formal language, the *Schema Definition Language (SDL)*. The set of types defined with this SDL is then collectively called the *GraphQL schema*.

Define the data to be queried

The schema essentially defines what data can be queried from the GraphQL server. Therefore, the GraphQL schema is one way to restrict the information a possible client can retrieve from the *Headless Server*. Another way would be to use a different CoreMedia CMS user with a different set of rights.

Each query is validated against the schema before query execution, any query that fails this validation is rejected by the *Headless Server*.

A GraphQL schema for a subset of the CoreMedia Blueprint document model is defined in the file `content-schema.graphql`. In this schema file, the GraphQL query root type `Query` is defined and contains a field `content` of type `ContentRoot`. This root object supports CoreMedia CMS content repository access. It is implemented by the Spring bean `content` of class `ContentRoot`. Further content fields can be added to the GraphQL. These must then be implemented either by a `@fetch` directive (see section Section 4.2, "The @fetch Directive" [28]), or by subclassing the `ContentRoot` class. In the latter case, an instance of the new subclass must replace the `ContentRoot` instance in the Spring configuration.

The query root type is extensible by adding Spring beans with the qualifier `QueryRoot`. This is used in the eCommerce integration module to add a new query root field `commerce`, as described in Chapter 6, *eCommerce Extension* [77].

Extending the schema, additional files

A GraphQL schema for the *Headless Server* may be split into several files. So, additional GraphQL types and interfaces can either be added to the schema by extending the file `content-schema.graphql`, or by adding more GraphQL schema resource files with the name pattern `*-schema.graphql` to the Java resources folder. During startup, the *Headless Server* looks for files wich follow the location pattern in the classpath. These are then merged together, yielding the complete schema.

Further `adapters` (Section 4.6, "Adapter" [33]), `model mappers` (Section 4.4, "Model Mapper" [31]), filter predicates (Section 4.5, "Filter Predicates" [32]), or GraphQL scalar types can be defined as Spring beans in `CaasConfig.java`, or by adding a new Spring configuration class.

All these extensions can be made within the `headless-server-base` module within the blueprint workspace. However, a better practice is to add a new Maven module with its own Spring configuration and schema resource file. This separates your extensions from future changes within the `headless-server-base` module. Again, the eCommerce integration module described in Chapter 6, *eCommerce Extension* [77] may serve as an example.

4.2 The @fetch Directive

The standard GraphQL Java `@fetch` directive has been extended by CoreMedia to support the Spring Expression Language. This gains a lot of flexibility for implementing data fetching logic, often avoiding the need to extend a Java class with corresponding properties.

As a simple example, assume you want to make the `name` field of some object to be available as is and, additionally, with all characters converted to upper case:

```
type SomeObjectType {
      name
      uppercaseName: @fetch(from: "name.toUpperCase()")
}
```

The special SpringEL variables `#this` and `#root` are initially bound to the target object of the field. Note that, according to SpringEL semantics, the `#root` variable remains to be bound to this object during expression evaluation, while the `#this` variable may change, depending on expression context.

The following fields all fetch the same value:

```
      name
      name2: @fetch(from: "name")
      name3: @fetch(from: "#root.name")
      name4: @fetch(from: "#this.name")
```

With the original `@fetch` directive from GraphQL Java, only the first, simple form is allowed, the "expression language" is restricted to simple identifiers. The CoreMedia *Headless Server* `@fetch` directive implements a strict superset.

In GraphQL, fields may take arguments. Inside the fetch expression, these are available as SpringEL variables of the same name:

```
type Query {
  add(x: Int!, y: Int): Int! @fetch(from: "#x + #y")
}
```

Other SpringEL variables may be defined by adding Spring beans with the qualifier `globalSpelVariables`. Moreover, SpringEL variables may also be bound to functions. Such functions might help to keep the SpringEL expressions short and concise. For example, in `CaasConfig.java`, a SpringEL function `#first` is defined with a static method from class `SpelFunctions`. It retrieves the first element of a list, or null if the list is itself null or empty:

```
  @Bean
  @Qualifier("globalSpelVariables")
  public Method first() throws NoSuchMethodException {
    return SpelFunctions.class.getDeclaredMethod("first", List.class);
  }
```

A `@fetch` directive utilizing this function may look like this:

```
      authors: [CMTeasable]
      author: CMTeasable @fetch(from: "#first(authors)")
```

The same functionality might be expressed with a rather lengthy expression using the ternary (conditional) operator:

```
      authors: [CMTeasable]
      author: CMTeasable
        @fetch(from: "authors?authors.length>0?authors[0]:null:null")
```

Note that SpringEL variables all share the same name space, so be aware of possible name clashes.

The GraphQL schema `content-schema.graphql` contains many more examples for Spring EL expressions.

When accessing settings or nested properties there are two ways to do so. Firstly, it is possible to access the properties via the spring expressions:

```
      @fetch(from: "structName?.pathSegmentA?.pathSegmentB?.propertyName")
```

Using this will however result in an error if one of the path segments or the property itself does not exist on the object. A more reliable way of accessing settings and properties would be to use the SettingsAdapter and the StructAdapter (see Section 4.6, "Adapter" [33]) for access to these kinds of properties. They take care of existing properties, it is possible to query multiple properties at once and to pass them default values. Additionally, they provide the option to wrap a value in its path, which means that the adapter does not return the value directly, but instead wrapped in a hierarchical structure, representing the path.

```
     @fetch(from: "@structAdapter.to(#root).getWrappedInStruct('structName',
 {'pathSegmentA','pathSegmentB','propertyName'}, 'defaultValue')"
```

The result would be something like: `{structName:{pathSegmentA:{pathSegmentB:{propertyName:propertyValue}}}}`

4.3 The @inherit Directive

Inheritance relationships between interfaces or object types may be expressed with the `@inherit` directive. This obviates the need to repeat fields of supertypes or interfaces in subtypes or subinterfaces, respectively.

As an example, define an interface `Shape` with a field `area`, and a subinterface `Circle` which inherits the `area` field and adds another field `radius` to the interface type:

```
interface Shape  {
  area: Float!
}

interface Circle @inherit(from: "Shape") {
  radius: Float!
}
```

In effect, the `Circle` interface includes both fields. The `@inherit` directive works similarly for object types.

You might be surprised that the GraphQL SDL language itself does not support field inheritance in some way. So far, the GraphQL language designers rejected the introduction of such a language feature. They argue that this would violate a fundamental design goal of GraphQL, namely to favor readability over writability.

This is debatable, as with the absence of field inheritance you have to repeat each field of all supertypes in each subtype, and the fact that the same field occurs in multiple types in exactly the same way has to be inferred by the reader. The content schema makes heavy use of inheritance in order to mirror the inheritance relationships within the document type model. CoreMedia found that this improves the readability of the schema and is less error prone when modifying the schema.

However, if you do not like the `@inherit` directive, don't use it. You can achieve exactly the same effect by copying field definitions to each related type. This is semantically equivalent to what the implementation of the `@inherit` directive does when the schema definition file is parsed: it adds all fields of supertypes or superinterfaces to the subtype or subinterface, respectively, to the internal representation of the schema. When this schema is then queried by a client like GraphiQL (by an introspective query), this expansion has already taken place, and there are no more `@inherit` directives in the schema visible to clients.

In the *Headless Server*, a GraphQL schema file is parsed by an extended `graphql.schema.idl.SchemaParser` that adds support for this `@inherit` directive.

4.4 Model Mapper

A model mapper can be used to convert domain model objects to a more suitable representation.

For example, a `Calendar` object can be converted to a `ZonedDateTime` Object.

```
@Bean
public ModelMapper<GregorianCalendar, ZonedDateTime> dateModelMapper() {
  return gregorianCalendar -> Optional.of(gregorianCalendar.toZonedDateTime());
}
```

Example 4.1. Creating a ModelMapper for Calendar objects

Beans of type `ModelMapper` are picked up automatically and configured in the GraphQL wiring Factory.

4.5 Filter Predicates

A filter predicate can be used to filter beans by predicate.

For example, a validity date filter predicate can be defined to filter content items by their validity date.

```
@Bean
@Qualifier("filterPredicate")
public Predicate<Object> validityDateFilterPredicate() {
  return new ValidityDateFilterPredicate();
}
```

Example 4.2. Creating a filter predicate

Beans with Qualifier `filterPredicate` are picked up automatically and applied to `ModelMappers`.

4.6 Adapter

An Adapter can be used to enhance domain model objects with

- Business logic from blueprint-base
- Aggregation, Recomposition
- Fallbacks

An Adapter is defined as Spring Bean and can be accessed from the GraphQL schema.

There are several predefined Adapters in the *Headless Server*, that can be accessed in the GraphQL schema.

For example, to access the settings of a content object, the `SettingsAdapter` can be used.

The `StructAdapter` provides access to values in structs. The Adapter expects a name of the struct which is to be accessed and a list specifying the path including the property to be found. This list shouldn't contain the name of the struct. Additionally it is possible to provide a default value which is used in case the struct value wasn't found.

```
      type CMSettingsImpl implements CMSettings ... {
  settings(paths: [[String]]): JSON @fetch(from: "#paths == null ?
#this.settings : @structAdapter.to(#root).getWrappedInStruct('settings',#paths,
 null)")
}
```

Example 4.3. Retrieve a value from a struct with the StructAdapter

The `SettingsAdapter` provides functionality to retrieve settings via the `SettingsService` from blueprint-base packages. By doing so, it can find local and linked settings on content objects. The SettingsAdapter covers a specific case of values in a Struct. Inheritance of settings is not supported at the moment.

```
      settings(paths: "commerce")

settings(paths: ["commerce"])

settings(paths: [["commerce"]])

settings(paths: ["commerce","endpoint"])

settings(paths: [["commerce","endpoint"],["commerce","locale"]])
```

Example 4.4. Different ways to pass the paths parameter to the settings field from the GraphQL perspective

```
@Bean
public SettingsAdapterFactory settingsAdapter(@Qualifier("settingsService")
 SettingsService settingsService) {
  return new SettingsAdapterFactory(settingsService);
}
```

Example 4.5. Define SettingsAdapter as bean

```
type CMTeasableImpl implements CMTeasable ... {
  customSetting: String @fetch(from:
    "{!@settingsAdapter.to(#root).get({'customSetting'},'')}")
}
```

Example 4.6. Retrieve settings with the SettingsAdapter

There are several Adapters available, for example:

structAdapter	Retrieve values from a Struct at a content object.
responsiveMediaAdapter	Retrieve the crops for a Picture.
mediaLinkListAdapter	Retrieve the media for a content object, for example, picture(s), video(s).
pageGridAdapter	Retrieve the pagegrid.
imageMapAdapter	Retrieve image maps.
navigationAdapter	Retrieve the navigation context.

4.7 Building Links

In GraphQL, Objects may contain cross references (relations, "links") to other objects:

- Typically, a special field holds some kind of identifier or ID of this object, and other objects refer to it with the value of this ID.
- Another kind of reference is a hyperlink, for example the URL of some binary resource.

Link Composer

The CoreMedia *Headless Server* supports both types of references by a unified *`Link Composing API`*. This API is a generalization of the CAE link schemes and post processors (see Section “Writing Link Schemes” in *Content Application Developer Manual* and Section “Post Processing Links” in *Content Application Developer Manual*).

A `LinkComposer` is a `PartialFunction` from some domain object type to a resulting link type.

All link composers are partial functions: if they are not able to map an object to a proper link of the given target type, they return an empty `Optional`. If no configured link composer returns a non-empty `Optional`, the GraphQL query response will contain a `null` value for the link.

4.7.1 Link Composer for ID links

Link composers for ID links are mapping arbitrary Java objects to a `GraphQLLink` object, which is an (extensible) record of a type-specific, opaque ID and a type name.

The `ContentLinkComposer` class implements these for Unified API Content objects.

4.7.2 Link Composer for hyperlinks

Link composers for hyperlinks are mapping arbitrary Java objects to Uniform Resource Identifiers (URIs).

The `ContentBlobLinkComposer` class implements these for blob properties of content objects. The resulting URIs point to the appropriate controller inside the *Headless Server*. This controller serves blob data as-is, or picture data transformed with the *CoreMedia Image Transformation Framework*.

4.7.3 Implementing Custom Link Composer

Custom link composers can be added by implementing the corresponding interface in a Spring bean: `LinkComposer<?, ? extends GraphQLLink>` for GraphQL links, and `LinkComposer<?, ? extends UriLinkBuilder>` for hyperlinks.

The latter kind of link composers need to be added for `Content` objects if you want to see hyperlinks within internal links in CoreMedia Rich Text markup which are described in Section 5.9, “Internal Links” [74]. A sample LinkComposer for Content objects might look like this:

```
@Bean
public LinkComposer<Content, UriLinkBuilder> contentUriLinkComposer() {
  return content -> {
    String contentType = content.getType().getName();
    int numericContentId = IdHelper.parseContentId(content.getId());
    return Optional.of(new UriLinkBuilderImpl(
            UriComponentsBuilder.newInstance()
                    .scheme("coremedia")
                    .pathSegment(contentType, ""+numericContentId)
                    .build()));
  };
}
```

Such a link composer will then generate URIs of the form `coremedia:/`*`content type`*`/`*`content id`*, for example, `coremedia:/CMPicture/1726`. Converting and rendering this URI as a clickable hyperlink (URL) is then the duty of the client. For example, in a React client using React Router, the URI may map to a corresponding route.

Link `PostProcessors` are not currently configured in the *Headless Server*. If required, post processors can be added to the configuration of the `uriLinkComposer` and/or `graphQlLinkComposer` beans.

4.8 Content Schema

The types and interfaces in the schema file `content-schema.graphql` define a subset of the CoreMedia Blueprint document model in GraphQL SDL terms. The Blueprint document types are mapped to GraphQL interfaces of the same name, while an object type with the suffix `Impl` serves as the implementation of these interfaces. From the GraphQL field `content` of query root type, data of CoreMedia CMS documents is reachable via GraphQL queries (some fields omitted for brevity):

```
type Query {
  content: ContentRoot
}

type ContentRoot {
  content(id: String!, type: String): Content_
    @fetch(from: "getContent(#id,#type)")
  article(id: String!): CMArticle
    @fetch(from: "getContent(#id, 'CMArticle')")
  picture(id: String!): CMPicture
    @fetch(from: "getContent(#id, 'CMPicture')")
  page(id: String!): CMChannel
  pageByPath(path: String!): CMChannel
    @fetch(from: "@pageByPathAdapter.to().getPageByPath(#path)")
  site(siteId: String, id: String  @deprecated(reason: "Arg 'id' is deprecated.
 Use 'siteId' instead.")): Site
  sites: [Site]!
}
```

The following sections will discuss some example queries using these content root fields.

4.8.1 Simple Article Query

The following GraphQL query is a simple example for fetching data from a CMArticle document. It is based on the GraphQL schema defined in the file `schema.graphql`:

```
query ArticleQuery {
  content {
    article(id: "2910") {
      title
      teaserTitle
      teaserText
      picture {
        name
        creationDate
        alt
        uriTemplate
      }
    }
  }
}
```

Of course, you will have to change the article id parameter to a value which is valid in your content server.

Note that the query includes a field called uriTemplate that can be used by a client to construct a URL to the cropped image data by substituting the cropName and width parameters.

4.8.2 Article Query with Fragments and Parameters

The following example is a more complex article query. It uses GraphQL query fragments to factor out repeating parts, and a query parameter $id which can easily be passed in the variables field of the HTTP request:

```
query ArticleQuery($id: String!) {
  content {
    article(id: $id) {
      ...Reference
      teaserTitle
      teaserText
      pictures {
        ...ContentInfo
        alt
        uriTemplate
      }
      navigationPath {
        ...Reference
      }
    }
  }
}

fragment ContentInfo on Content_ {
  name
  creationDate
}

fragment Reference on CMLinkable {
  ...ContentInfo
  title
  segment
  link {
    id
    type
  }
}
```

Query parameters are called variables in Graphql. In GraphiQL, you pass query variables in the QUERY VARIABLES input field below the query input field, as a JSON object, for example,

```
{
  "id": "6494"
}
```

Note that, in this query, the picture fields has been replaced with the pictures field. This way, the result will hold a list of all (valid) CMPicture links within the pictures property instead of just the first one. Moreover, the response contains the navigation path of the article up to the root.

4.8.3 Querying all available Sites

To query all available sites, issue a query to the sites field of the content root:

```
{
  content {
    sites {
      id
      name
      locale
      root {
        ...Reference
      }
      crops {
        name
        aspectRatio {
          width
          height
        }
        sizes {
          width
          height
        }
      }
    }
  }
}

fragment ContentInfo on Content_ {
  name
  creationDate
}

fragment Reference on CMLinkable {
  ...ContentInfo
  title
  segment
  link {
    id
    type
  }
}
```

4.8.4 Site Query

To query a specific site, issue a query containing the content/site field, with the site ID as a parameter (for example, ID of the Corporate home page, "abffe57734feeee") You will find the site ID in the Site Indicator content of the site):

NOTE

The former argument 'id' is deprecated as of version 2004 in favor of the more specific argument name 'siteId'. The argument 'id' may still be used, but will be removed in future versions!

```
query SiteQuery($id: String!) {
  content {
    site(siteId: $id) {
      name
      id
      root {
        ...Reference
        ...Navigation
      }
      crops {
        name
        aspectRatio {
          width
          height
        }
        sizes {
          width
          height
        }
      }
    }
  }
}

fragment ContentInfo on Content_ {
  name
  creationDate
}

fragment Reference on CMLinkable {
  ...ContentInfo
  title
  segment
  link {
    id
    type
  }
}

fragment NavigationEntry on CMLinkable {
  ...Reference
  title
}

fragment Navigation on CMNavigation {
  ...NavigationEntry
  ... on CMNavigation {
    children {
      ...NavigationEntry
      ... on CMNavigation {
        children {
          ...NavigationEntry
          ... on CMNavigation {
            children {
              ...NavigationEntry
              ... on CMNavigation {
                children {
                  ...NavigationEntry
                }
              }
            }
          }
        }
      }
    }
  }
}
}
```

4.8.5 Querying derived Sites

Derived sites are part of any Site object of the content schema by means of the field derivedSites (see Section 5.5, “Localized Content Management” in *Blueprint Developer Manual* for details).

```
query DerivedSitesQuery($id: String!) {
  content {
    site(siteId: $id) {
      name
      id
      locale
      derivedSites {
        name
        id
        locale
      }
    }
  }
}
```

4.8.6 Page Query

As a more complex example, the following query returns a complete page (CMChannel), including data for all page grid placements, with image and video links (if present). Also included in the response: image map data.

```
query PageQuery($id: String!) {
  content {
    page(id: $id) {
      __typename
      ...Reference
      title
      teaserTitle
      teaserText
      creationDate
      grid {
        cssClassName
        rows {
          placements {
            name
            viewtype
            items {
              ...Teasable
              ...ImageMap
              ... on CMCollection {
                viewtype
                items {
                  ...Teasable
                }
              }
            }
          }
        }
      }
    }
  }
}

fragment ContentInfo on Content_ {
  name
  creationDate
}

fragment Reference on CMLinkable {
  ...ContentInfo
  title
  segment
  link {
    id
    type
  }
}

fragment ImageMap on CMImageMap {
  displayTitle
  displayShortText
  displayPicture
  transformedHotZones {
    crops {
      name
      coords {
        x
        y
      }
    }
    points {
      x
      y
    }
    alt
    shape
    target
    displayAsInlineOverlay
    inlineOverlayTheme
    linkedContent {
      ...Reference
      ...QuickInfo
    }
  }
}

fragment Teasable on CMTeasable {
  ...Reference
  teaserTitle
  teaserText
  teaserTarget {
    ...Reference
  }
  teaserTargets {
    target {
      ...Reference
    }
    callToActionEnabled
    callToActionText
  }
  teaserOverlaySettings {
    style
    enabled
    positionX
    positionY
    width
  }
  picture {
    ...Picture
  }
  video {
    ...Video
  }
}

fragment QuickInfo on CMTeasable {
  ...Reference
  teaserTitle
  teaserText
  picture {
    ...Picture
  }
}

fragment Picture on CMPicture {
  ...ContentInfo
  title
  alt
  link {
    id
    type
  }
  uriTemplate
  base64Images {
    cropName
    base64
  }
}

fragment Video on CMVideo {
  ...ContentInfo
  title
  alt
  link {
    id
    type
  }
  data {
    uri
  }
  dataUrl
}
```

Page queries accept the numeric content ID of a CMChannel document as well as a site ID. In the latter case, the home page of the site will be returned, for example, for the Calista demo site (query variables: { "id": "ced8921aa7b7f9b736b90e19afc2dd2a"}).

Alternatively, a page may be queried by its navigation path, using the 'pageByPath' query.

```
{
  content {
    pageByPath(path: "corporate/for-professionals") {
      id
      title
    }
  }
}
```

The path argument in the (abbreviated) example above consists of the segment path starting with the homepage segment 'corporate', the path separator '/' and the subpage segment 'for-professionals'. If the query is invoked using a site filter endpoint, like '/corporate/graphql', the homepage-segment of the path may be omitted, e.g. simply 'for-professionals'.

4.8.7 Download Query

For a CMDownload, the corresponding blob data (URI, contentType, size and eTag) can be queried as follows:

```
{
  content {
    content(id: "6600", type: "CMDownload") {
      ... on CMDownload {
        data {
          uri
          contentType
          size
          eTag
        }
      }
    }
  }
}
```

4.8.8 Querying localized variants

Localized variants of any content object can be obtained using either the field localized-Variants or localizedVariant. The first will return all existing variants of a content object while the latter requires specific locale parameters in order to retrieve the variant of a specific variant.

```
query LocalizedVariants($path: String!) {
content {
  pageByPath(path: $path) {
    title
    type
    localizedVariants {
      ... on CMChannel {
        repositoryPath
        locale
      }
    }
    localizedVariant(language: "en", country: "us", variant: "") {
      ... on CMChannel {
        repositoryPath
        locale
      }
    }
    }
}
    }
```

With the locale specific approach, the parameter language is mandatory, while country and variant are optional. Please note, that if a given combination of locale parameters does not exist you may get an empty object. When skipping the parameter country however, the first variant matching the language will be returned.

4.9 Search

To use Headless Server search, an existing Solr with an index created by a CAE Feeder needs to be provided.

Search is handled with the `searchServiceAdapter` adapter. The following functionality is supported:

- Full text search
- Paging
- Limit
- Filter by content type, optionally including their sub types
- Predefined sort fields with order
- Limitation to a site
- Valid from and valid to conditions are applied to search filters automatically

The following GraphQL query is a simple example for fetching a search result.

```
{
  content {
    search(query:"Perfect") {
      numFound
      result {
        name
      }
    }
  }
}
```

Several parameters can be passed to the `SearchServiceAdapter` to customize the search:

- query: The search query.
- offset: The offset.
- limit: The limit of search result.
- docTypes: Content types to restrict the search result.

  Misspelled content types are ignored. When passing an abstract content type, the subtypes are retrieved, if the parameter includeSubTypes = true. The search result does not contain abstract document types.
- sortFields: List of sort field with order, separated by '_', in upper case, e.g. ID_ASC.

  The set of available sort fields is limited to the enum `SortFieldWithOrder` defined in the content schema: ID, DOCUMENTTYPE, TITLE, TEASER_TITLE, MODIFICATION_DATE, CREATION_DATE, EXTERNALLY_DISPLAYED_DATE

  The fields of the enum above however can be extended by more fields. Currently the `SearchServiceAdapter` supports these sort fields: ID, DOCUMENTTYPE, NAVIGATION_PATHS, NOT_SEARCHABLE, SUBJECT_TAXONOMY, LOCATION_TAXONOMY, TITLE, TEASER_TITLE, TEASER_TEXT, KEYWORDS, MODIFICATION_DATE, CREATION_DATE, TEXTBODY, SEGMENT, COMMERCE_ITEMS, CONTEXTS, AUTHORS, HTML_DESCRIPTION, VALID_FROM, VALID_TO, EXTERNALLY_DISPLAYED_DATE

Possible order field values: ASC, DESC

- siteId: The siteId can be passed as parameter to restrict search per site.
- includeSubTypes: A boolean flag, indicating to include the sub types of the given doc types in the search. Defaults to 'false'.

The query parameter supports the following syntax:

- The + and - characters are treated as "mandatory" and "prohibited" modifiers for terms.
- Quoted expressions, like "Foo Bar" are treated as a phrase
- An odd number of quote characters is evaluated as if there were no quote characters at all.
- The wildcard character '*' supports the search for partial terms like 'frag*', which would find e.g. the terms 'fragment' and 'fragile' as well. When used exclusively as a search query, the search is executed with all other search parameters but without an explicit search expression.

By default, the `SearchServiceAdapter` uses the caeSolrQueryBuilder spring bean, which in turn invokes searches on solr on the `cmdismax` endpoint. For details please see the Section 3.8.1, "Details of Language Processing Steps" in *Search Manual*.

The used endpoint can be customized in `CaasConfig`. See section customization at the end of this chapter for details.

The following GraphQL query is a more complex example for fetching a search result.

```
{
  content {
    search(query: "Perfect", offset: 3, limit: 5, docTypes: ["CMArticle",
"CMPicture"], sortFields: [CREATION_DATE_ASC, MODIFICATION_DATE_ASC], siteId:
 "abffe57734feeee", includeSubTypes: true) {
      numFound
      result {
        name
        type
      }
    }
  }
}
```

If docTypes or limit is not passed as parameter, the following search configuration is taken into account, which is read from CMS content using settings. See general search configuration for details in Blueprint Developer Manual .

- searchDoctypeSelect, search.doctypeselect: document types to restrict the search result

- searchResultHitsPerPage, search.result.hitsPerPage: limit of the search result

Valid from and valid to conditions are applied to search filters automatically.

Configuring Search

The connection to Solr is defined with `solr.url`

The search index is specified with property `caas.solr.collection`

Caching is only performed in live mode and can be configured with `caas.search.cache.seconds`

Customization

If search should be performed with a different index, you have to implement `SolrQueryBuilder` in your own class. The searchHandler can be passed as a constructor argument.

4.10 Dynamic Query Lists

To use Dynamic Query Lists with *Headless Server*, *Headless Server* Search needs to be set up (see Section 6.1.1, "Content Query Form" in *Blueprint Developer Manual* for details about Dynamic Query List content).

Dynamic Query Lists are handled with the `queryListAdapter`. The following functionality is supported:

- Paging
- Limiting the result size
- Filter by predefined fields
- Sort by predefined fields

The following GraphQL query is a simple example for fetching data from a CMQueryList document.

```
{
  content {
    queryList(id: "7692") {
      title
      items {
        title
      }
    }
  }
}
```

The following parameter can be passed to the `QueryListAdapter` to customize the Dynamic Query List result:

- offset: The offset for paging. Available as `pagedItems` in graphql schema.

The following GraphQL query is a simple example for fetching paged data from a CM-QueryList document.

```
{
  content {
    queryList(id: "7692") {
      title
      pagedItems(offset: 3) {
        title
      }
    }
  }
}
```

Dynamic Query List configuration is read from the content using configuration that can be applied in Studio.

General configuration:

Content Types	A selection of content types.
Limit	Limit of the Dynamic Query List items.
Sort Field	The field to sort on.
Order	The sort order

Search filter configuration:

Authors	The authors of the document.
Context Documents	The context of the document.
Modification Date	The modification date defines as interval.
Location Tag	The content is tagged with the given location tag.
Subject Tag	The content is tagged with the given subject tag.
Tag Context	The content is tagged with one of the tags of the query list's context.

Valid from and valid to conditions are applied to search filters automatically.

Dynamic Query List Configuration

Caching for dynamic query lists is only performed in live mode and can be configured with `caas.querylist.search.cache.seconds`

4.11 Product Lists

To use Product Lists with *Headless Server*, *Headless Server* Search needs to be set up.

Product Lists are handled with the `productListAdapter`. The following functionality is supported:

- Paging
- Limiting the result size
- Filter by Subcategories with a specific value

The following GraphQL query is a simple example for fetching data from a CMQueryList document.

```
{
  content {
    productList(id: "7692") {
      items {
        ... on CMTeasable {
          teaserTitle
          teaserText
        }
        ... on Product {
          name
          shortDescription
        }
      }
    }
  }
}
```

Product List configuration is done in CoreMedia Studio, such as:

- First Displayed Position: The position of the first item to be displayed (for paging)
- Limit: Limit of the products in the Product List
- Order: The sort order

Please consult the Studio User Manual for details.

Valid from and valid to conditions are applied to search filters automatically.

Product List Configuration

Caching for Product lists is only performed in live mode and the caching time can be configured with `caas.querylist.search.cache.seconds`

4.12 Using Time Dependent Visibility

The time at which a published content should be visible to the customer can be controlled by validity or visibility. For more information see Section 4.6.14, “Time Dependent Visibility” in *Studio User Manual*.

To enable time dependent visibility, you have to pass a request header with the **view date** to the *Headless Server*. Note that this is only possible in preview mode.

The **view date** request header is passed as:

- Header Name: `X-Preview-Date`
- Value: Date object in HTTP Date Header standard format, see RFC 7231 for specification

The `RequestDateInitializer` evaluates the **view date** header and either sets the passed date, or, if not available, the current date as request attribute. The view date is then set in the GraphQL context and can be retrieved via `ContextHolder#getViewDate()`.

The validity check of content items is performed in the `ValidityDateFilterPredicate` which is configured in `CaasConfig`.

The visibility of content items is checked in the `PageGridAdapter`.

4.13 Remote Links

Overview

The *Headless Server* is able to retrieve links for content objects like pages, articles or pictures, that are generated by a remote system, like a CAE. These links can be used by a client to link to that remote content.

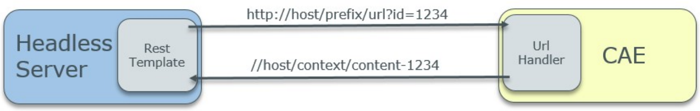


Figure 4.1. Remote Links

A request is executed from the *Headless Server* to a configured remote handler in a CAE with a list of content IDs together with optional properties (site, context). The handler generates corresponding links and returns them to the *Headless Server*.

GraphQL Schema

In the GraphQL schema, the `remoteLink` property is defined on type CollectionItem:

```
interface CollectionItem {
  remoteLink(siteId:String, context:String): String
}
```

All types that inherit from CMLinkable or implement CollectionItem can access this field. For example, the following query retrieves a remote link for an article:

```
{
  content {
    article(id: "7456") {
      remoteLink
    }
  }
}
```

The query fragment in the GraphQL schema to retrieve a remoteLink contains the **contentId** implicitly via the `RemoteLinkDataFetcher`. Additionally, the following parameters can be set:

siteId	(optional): Defines, for which site the link is generated, as a content can be located in multiple sites.

context	(optional): Defines, for which context the link is generated, as a content can be located in different contexts within a site.

The following example retrieves an article within a site and a specific context:

```
{
  content {
    article(id: "7456") {
      remoteLink(siteId:"abffe57734feeee", context: "7950")
    }
  }
}
```

A typical use case is the retrieval of a page content object:

```
query getPageById($pageId: String!, $siteId: String) {
  content {
    page(id: $pageId) {
      title
      remoteLink(siteId: $siteId)
      pictures {
        title
        remoteLink(siteId: $siteId)
      }
    }
  }
}
```

CAUTION

Please note, that links for commerce objects currently cannot be resolved. For technical reasons it is nonetheless possible to use the remoteLink fragment for commerce object already. A query for remote links for commerce object will always resolve to a null object!

Batch loading mechanism and caching

In order to achieve a reasonable performance when resolving remote links, the *Headless Server* uses a so called batch loader, which is able to resolve all remote links with only one remote request to the CAE per query level and caches the results (time based eviction).

Configuration

The following configuration options are available, see Section 4.3.3, "Remote Service Adapter Properties" in *Deployment Manual* for details:

caas.remote.baseurl	Base URL of the remote handler.
caas.remote.httpClientConfig.*	Configuration options of the HttpClient used by the RestTemplate.

caas.cache-specs[remote-links]	[Caffeine Cache] configuration for the remote link cache.

CAE Handler

The CAE `UrlHandler` handles requests to `/internal/service/url` and generates links using the CAE link building mechanisms.

As the remote handler for link building is configurable, a custom service can be set up, that handles requests with the given parameters and returns URLs in json format with entities of type `UrlServiceResponse`.

Deployment

It is assumed that the remote system, that is the CAE, is located in the same trusted network as the *Headless Server* and so the systems communicate via HTTP. If communication should be established via HTTPS, security configuration needs to be applied to the servers accordingly.

The handler path of the UrlHandler `/internal/service/url` needs to be configured if required for preview and live environments (for example, traefik, rewrite rules).

Development

For debugging SSL connections, the option `caas.remote.httpClientConfig.trustAllSslCertificates` can be set to true. This should only be done in a development environment.

5. Rich Text Output

Delivering CoreMedia Rich Text properties requires a transformation of the internally stored markup format into a format that can be serialized to JSON output and that matches the requirements of the client. This process is handled by a configurable set of *Rich Text Transformers* per RichTextTransformerRegistry. Each transformer handles a specific transformation aspect required by the client, for example:

- Generate a text only teaser from the first paragraph of a richtext property.
- Generate a full HTML representation of a detail text including embedded images and internal links.

Transformers are applied to the raw content of a GraphQL field on either of these types:

- `String`: A string representation of the complete Markup.
- `RichTextTree`: A custom scalar GraphQLType that defines a tree based representation of the Markup.

The output format may be specified by the transformation name in a GraphQL query with a view clause, where the name of the view is equivalent to the transformation name.

Please note, that the term 'view' is not connected in any way to the views of the CAE used for rendering the same content for different display purposes!

```
Syntax:
graphQL-field-name(view: "transformation-name")

Example:
detailText(view: "plainFirstParagraph")
```

Names of the currently predefined views are:

default	Delivers the complete content of the requested field, consisting of all embedded markup, links and images, for instance. This view is the default, if no view is specified.
simplified	Delivers the complete content of the requested field, where special embedded markup like links and images is replaced by a plain version.
plainFirstParagraph	Delivers the first paragraph of the requested field without any embedded markup.

Please also note that, for technical reasons, the delivered content in all views is always nested in a <div> tag

Rich text transformers are fully configurable via YAML configuration files. Each configuration defines the following elements:

name	The transformer's view name.
elements	List of rich text elements. Is included at the start of the YAML definition. Individual elements are accessed by reference from following handlers.
classes	List of known rich text CSS class names. Is included at the start of the YAML definition. Individual names are accessed by reference from following handlers.
contexts	List of processing contexts. Each context defines a list of handlers, which are responsible for: • Processing opening and closing elements. • Processing text nodes. • Transforming elements and attributes.
initialContext	Defines the root context.
handlerSets	An optional mapping of named handler lists. Allows grouping and reusing handlers in different contexts.

Writing a new transformer is easily accomplished. First, create a YAML text file and place it in the Blueprint in the folder `resources/richtext`. The name of the file should match the name of the view used later in your GraphQL queries, for example a transformer named 'myView':

```
resources/richtext/myView.yml
```

As a starting point, add this basic content to your transformer file:

```
#!import file=includes/elements.yml
#!import file=includes/classes.yml
#!import file=includes/attributes.yml

name: myView
contexts:
  - &root !RootContext
    name: root
    handlers:
      - - !Handler
          eventMatcher:   !Matcher {qname: }
          outputHandler:  !ElementWriter {writeCharacters: true}
initialContext: *root
```

Note that the file name (without the suffix) matches the 'name' property. As mentioned above, any transformer consists of the top level YAML properties 'name', 'elements', 'classes', 'contexts', 'handlerSets' and 'initalContext', which are all included in this basic example file.

When writing a configuration in YAML style, indentions are most important. For a reference about YAML you may refer to https://yaml.org/.

5.1 The Include Directive

A directive to include the contents of a supporting YAML file. Used to provide reusable definitions in a separate file. CoreMedia provides a set of include files reflecting the CoreMedia Rich Text Markup. They contain the used tags and CSS classes.

```
Syntax:
#!import file=<relative-path-to-include-file>/<name-of-include-file>

Example:
#!import file=includes/elements.yml
```

As best practice, always include these standard includes! Note, that all following example code snippets do rely on these includes!

```
#!import file=includes/elements.yml
#!import file=includes/classes.yml
#!import file=includes/attributes.yml
```

5.2 YAML Anchors and Aliases

When using includes, using YAML anchors and aliases is imperative. The contents of the includes should make use of anchors in order to reference the anchored definitions by an alias.

```
# Example: anchor a scalar value
anyProperty: &nameAnchor anchoredContent

# reuse it by alias:
anyOtherPropery: *nameAnchor

# which is equivalent to:
anyOtherPropery: anchoredContent
```

Example: anchor a code snippet

```
# define a code snippet anchor
anyProperty: &codeSnippetName
  - a
  - b
  - c

# reuse the snippet
myProperty: *codeSnippetName

# which is equivalent to
myProperty:
  - a
  - b
  - c
```

5.3 Code Comments

YAML comments are introduced by the '#' character at any column in a row.

```
# this is a comment, not to be confused with the include directive!
```

5.4 Name Property

A top level YAML property, defining the name of a transformer.

```
name: myTransformerName
```

5.5 Elements Property

Defines a list of rich text elements (tags) to be considered when parsing the raw markup. Usually included by the include directive (see https://yaml.org/) but not necessarily. As best practice, all listed elements should be anchored.

Only elements listed and anchored here can be used for the transformation contexts and handlers.

Example:

```
elements:
  - &div  !QName [ "http://www.coremedia.com/2003/richtext-1.0" , "div" ]
  - &p    !QName [ "http://www.coremedia.com/2003/richtext-1.0" , "p" ]
   ...
  - &sup  !QName [ "http://www.coremedia.com/2003/richtext-1.0" , "sup" ]
```

5.6 Classes Property

A list of CSS classes to be considered when parsing the raw markup. Usually included by the include directive (see Section 5.1, "The Include Directive" [59]) but not necessarily. As best practice, all listed classes should be anchored.

Only classes listed and anchored here can be used for the transformation contexts and handlers.

Example:

```
classes:
  - &headline_styles !!java.util.ArrayList
  - &headline_1_style p--heading-1
  - &headline_2_style p--heading-2
  - &headline_3_style p--heading-3
  - &headline_4_style p--heading-4
  - &headline_5_style p--heading-5
  - &headline_6_style p--heading-6
...
```

5.7 Contexts and InitialContext Property

A context defines how to transform a specific element node of a rich text document. For this task it has a number of registered event handlers, which apply to its subnodes.

Rich text processing always starts with a Root Context, where the root tag of the markup is processed. Contexts are stacked, that is when encountering the start of a paragraph, a new context for handling the elements within that paragraph is pushed on top of current context and removed when the paragraph ends.

Defining one or more contexts is achieved with the contexts property, followed by a YAML list of context definitions.

Syntactically, a context definition consists of a context type, a name and various handlers.

Syntax:

```
contexts:
  - !context-type
    name: context-name
    defaultHandler:
      !Handler
      eventMatcher:   ...
      contextHandler: ...
      outputHandler:  ...
    handlers:
    - list of additional handlers
  ...
initialContext:
  - !context-type ...
```

Example: Define three named contexts and reference context 'root' as initial context.

```
contexts:
  - !Context
    name: headline
    defaultHandler:
      !Handler
      outputHandler: !ElementWriter {writeCharacters: true}
    handlers:
      - *text_handlers

  - !Context
    name: paragraph
    defaultHandler:
      !Handler
      outputHandler: !ElementWriter {writeCharacters: true}
    handlers:
      - *text_handlers
      - *inline_handlers

  - &root !RootContext
```

```
        name: root
        handlers:
          - *headline_handlers
          - *block_handlers
          - *blockquote_handlers

initialContext: *root
```

5.7.1 Context Types

Context Type	Description
!RootContext	Context type used for initial contexts only!
!Context	Context type for all other parsing events.

Table 5.1. Available context types for the contexts section.

Both context types share the same properties:

Property	Description
name	The context's name.
handlers	A list of handlers.
defaultHandler	An optional default event handler which is executed if none of the other handlers applies.

Table 5.2. Available properties for !Context and !RootContext.

5.7.2 Handlers

A handler is always introduced by this start element:

```
!Handler
```

Handlers consist of up to three properties:

- An event matcher

- A context handler
- An output handler

Event Matcher

An event handler applies to a specific start element event within the XML event stream (except for default handlers).

```
        !Matcher
```

Property	Description
qname	The qualified name of the start element event.
classes	Optional style classes. Matches if the event's attribute class contains any of the styles.

Table 5.3. Available properties for !Matcher.

Example:

```
contexts:
  - !Context
    name: headline
    defaultHandler:
      !Handler
      eventMatcher: !Matcher { qname: *p, classes: *headline_styles }
      ...

# alternative (equivalent) YAML style
contexts:
  - !Context
    name: headline
    defaultHandler:
      !Handler
      eventMatcher:
        !Matcher
        qname: *p
        classes: *headline_styles
        ...
```

Context Handlers

Context Handlers (not to be confused with the context type) define a modification on the context stack, whenever the rule of the corresponding event matcher applies. There are currently two styles of context handlers:

```
!Push
# or
!ReplacePush
```

Property	Description
contextName	The name of the context to install on top of the stack.
replacementName	For !ReplacePush context handler only! The name of the context to replace the current context with.

Table 5.4. Available properties for !Push and !ReplacePush.

Example:

```
contexts:
  - !Context
    name: headline
    defaultHandler:
      !Handler
      eventMatcher: ...
      contextHandler: !Push { writeCharacters: true }
      ...
```

Output Handlers

Output Handlers define the generated output for an element node. Available output handlers are:

```
!ElementWriter
!EmptyElementWriter
!ImgWriter
!LinkWriter
```

ElementWriter

The default output handler for element nodes, introduced by:

```
  !ElementWriter
```

Property	Description
writeElement	Boolean flag indicating if the start and stop element should be written to the output. Defaults to false.
writeCharacters	Boolean flag indicating if the character nodes of an element should be written. Defaults to false
elementTransformer	Optional transformation rules for the element.
attributeTransformers	Optional transformation rules for the element's attributes.

Table 5.5. Available properties for !ElementWriter.

Example:

```
contexts:
  - !Context
    name: headline
    defaultHandler:
      !Handler
      eventMatcher:   ...
      contextHandler: ...
      outputHandler: !ElementWriter { writeCharacters: true }
      ...
```

Empty Element Writer

Output handler for empty elements, for example, br. Does not support any properties.

```
!EmptyElementWriter
```

Example:

```
contexts:
  - !Context
    name: headline
    defaultHandler:
      !Handler
      eventMatcher:   ...
      contextHandler: ...
      outputHandler: !EmptyElementWriter
      ...
```

Image Writer

Output handler that generates embedded image tags. Uses the default link builder.

```
!ImageWriter
```

The output format is fixed to:

```
<img data-src="[LINK-URI]" alt="[IMG-ALT-TEXT]"/>
```

Property	Description
attributeTransformers	Optional transformation rules for the element's attributes.

Table 5.6. Available properties for !ImageWriter.

Example:

```
contexts:
  - !Context
    name: headline
    defaultHandler:
      !Handler
      eventMatcher:   ...
      contextHandler: ...
      outputHandler: !ImageWriter
      ...
```

Link writer

Output handler that generates embedded link tags. Uses the default link builder.

```
!LinkWriter
```

The output format is fixed.

For internal links:

```
<a data-href="[LINK-URI]">...</a>
```

For external links:

```
        <a href="[LINK-URI]">...</a>
```

Property	Description
attributeTransformers	Optional transformation rules for the element’s attributes.

Table 5.7. Available properties for !LinkWriter.

Example:

```
    contexts:
      - !Context
        name: headline
        defaultHandler:
          !Handler
          eventMatcher:   ...
          contextHandler: ...
          outputHandler: !LinkWriter
          ...
```

Defining special transformation rules for output handlers

As mentioned above, the output handlers !ElementWriter, !ImgWriter and !LinkWriter support special additional properties in order to describe the transformation of an element or attribute.

An ElementWriter may define the properties 'elementTransformer' and 'attributeTransformers', whereas ImgWriter and LinkWriter only support the 'attributeTransformers' property.

Element Transformer

An Element Transformer allows to generate an alternate element name based on the element styles. It is used, for example, for generating HTML headlines from the rich text headlines, which are internally stored as paragraphs with custom style classes.

Example: mapping from a style's name to element qualified name.

```
          ...
            elementTransformer:
              !ElementFromClass
              mapping:
                  *headline_1_style: h1
                  *headline_2_style: h2
                  *headline_3_style: h3
                  *headline_4_style: h4
                  *headline_5_style: h5
                  *headline_6_style: h6
```

Attribute Transformers

An Attribute Transformer allows to add/remove/modify attributes of an element node. Currently there is only one transformer for filtering style classes.

Example: filtering / passing only the declared styles to the output.

```
            ...
              attributeTransformers:
                !PassStyles
                styles:
                  *float_styles
```

5.8 HandlerSets Property

Using handler sets allows to group and reuse handlers in different contexts. In order to achieve this goal the YAML way, one or more handlers are grouped into a list of handlers, that is a handler set.

Example:

```
handlerSets:
  text: &text_handlers
    - !Handler
      eventMatcher:  !Matcher {qname: *em}
      outputHandler: !ElementWriter {writeElement: true, writeCharacters:
true }
    - !Handler
      eventMatcher:  !Matcher {qname: *strong}
      outputHandler: !ElementWriter {writeElement: true, writeCharacters:
true }
```

- The subsequent property 'text' is up to the author and may be named accordingly to the YAML rules.
- The list of handlers is anchored to the alias 'text_handlers'.

Example: reuse '*text_handlers' for a named context

```
      contexts:
        - !Context
          name: headline
          defaultHandler:
            !Handler
            outputHandler: !ElementWriter {writeCharacters: true}
          handlers:
            - *text_handlers
      ...
```

5.9 Internal Links

Inside CoreMedia Rich Text markup, links to other content objects may be embedded inside anchor and image elements. These are called *internal links*. Internal links are built by the configured `LinkComposer` for String-valued hyperlinks. Link composers are described in Section 4.7, "Building Links" [35].

For each anchor (`<a>`) element, two attributes are added:

data-href	Contains the generated link.
data-show	Contains the link behaviour.

Possible values for link behaviour as specified in http://www.w3.org/XML/2008/06/xlink.xsd are:

- `new`
- `replace`
- `embed`
- `other`
- `none`

For each image (`<img>`) element, a

- `data-src` attribute is added, with the generated link and a
- `data-uritemplate` attribute with the result of composing a link to a ResponsiveMediaAdapter wrapped around the data blob of the image. It has variables for both the crop name and the desired image width. When expanded with valid values for these variables (as configured in the responsive media settings for the site), this URI template will yield a URL pointing to the MediaController running inside the *Headless Server*. Note that this might be (and usually is) a URL relative to the *Headless Server* endpoint.
- `alt`: The alt property of CMMedia objects (or subtypes).

Here is an excerpt of some article detail text with an internal link to a picture document:

```
<p>ChefSupply RGB LED Strip</p>
<p><img data-src="coremedia:/CMPicture/4790"
        data-uritemplate=
          "/caas/v1/media/4790/data/826be46e8a8896e07646/{cropName}/{width}"

        alt="ChefSupply RGB LED Strip 01"/></p>
```

Note that an example link composer `contentUriLinkComposer` for content objects is configured in `CaasConfig.java` and may need customization as described in Section 4.7, “Building Links” [35]. This example link composer generates links that contains the content id:

```
<a data-href="coremedia:///cap/content/7246" data-show="embed">
```

5.10 External Links

Inside CoreMedia Rich Text markup, external links may be embedded inside anchor elements.

For each anchor (`<a>`) element, the following attributes are added:

href	Contains the the external link.
data-show	Contains the the link behaviour.
data-role	Contains the the target frame identifier, if available.

Possible values for link behaviour as specified in http://www.w3.org/XML/2008/06/xlink.xsd are:

- `new`
- `replace`
- `embed`
- `other`
- `none`

6. eCommerce Extension

All eCommerce functionality of the *Headless Server* is bundled within the Blueprint module `headless-server-ec`. It contains a GraphQL schema extension within the file `commerce-schema.graphql` and Java code and Spring configuration to implement this schema extension.

The GraphQL schema extension is essentially a mapping of the *CoreMedia Commerce API* onto GraphQL terms, with some additions to support augmented products and categories.

The schema extension uses the GraphQL extension mechanism to add a new field `commerce` of type `CommerceRoot` to the query root object. Using this new root field, it is possible to query for catalogs, categories and products, including product variants.

eCommerce catalogs are always associated with a specific site, so you have to specify a site ID in each `CommerceRoot` query.

6.1 Products

One way to query products is to retrieve a list of products as a part of a category or catalog query, or by navigating from a `CMProductTeaser` or `CMExternalProduct` content query to the corresponding product. These kinds of queries will be discussed in the following sections. A more straightforward way to retrieve a product is to query them by their external ID:

```
{
  commerce {
    product(externalId: "sfcc:///catalog/product/25592211M",
            siteId: "sfra-en-gb") {
      name
      listPrice
      offerPrice
      locale
    }
  }
}
```

Products may also be queried by their SEO segment or technical ID.

Client side Caching

Some existing graphql frameworks like Apollo do support client side caching. As caching usually requires a unique cache key, the 'externalId' of commerce objects is aliased additionally to the field 'id'.

6.2 Categories

Similar to products, categories can be retrieved as a part of a catalog query, or by navigating from a `CMExternalChannel` content query to the corresponding category. And, as with products, categories may be queried by their external ID. The following query fetches a root category together with the child categories up to the second level:

```
{
  commerce {
    category(categoryId: "ibm:///catalog/category/techId:ROOT",
             siteId: "99c8ef576f385bc322564d5694df6fc2") {
      name
      externalId
      ...CategoryFragment
      children {
        ...CategoryFragment
        children {
          ...CategoryFragment
        }
      }
    }
  }
}

fragment CategoryFragment on Category {
  name
  externalId
  products {
    name
    externalId
    listPrice
  }
}
```

Categories may also be queried by their SEO segment.

6.3 Catalogs

As with categories and products, catalogs can be retrieved by their external ID. From a catalog, you may query the root category as well as child categories, up to a fixed depth. For example, the following query retrieves the root category and the first level of child categories:

```
{
  commerce {
    catalog(catalogId: "sfcc:///catalog/catalog/storefront-catalog-en",
            siteId: "sfra-en-gb") {
      name
      externalId
      rootCategory {
        ...CategoryFragment
        children {
          ...CategoryFragment
        }
      }
    }
  }
}

fragment CategoryFragment on Category {
  name
  externalId
  products {
    name
    listPrice
  }
}
```

It is also possible to query a list of all available catalogs for a given site:

```
{
  commerce {
    catalogs(siteId: "sfra-en-gb") {
      name
      externalId
    }
  }
}
```

6.4 Augmentation

Categories, Products and Pages from the eCommerce system may be augmented with content from the CoreMedia CMS. This includes mapping media content such as pictures and videos to categories and products, as well as augmenting pages, categories and products with specific content objects.

6.4.1 Categories and Products Mapped to Media Content

CMS media content can be associated with products and categories by adding the product or category to the `Associated Catalog Items` form field in the `Metadata` tab within Studio (see Section 6.2.3, “Adding CMS Content to Your Shop” in *Studio User Manual*).

To query this media content, the GraphQL types `Category` and `Product` contain the fields `picture`, `pictures`, `video`, `videos`, and `media`, where the singular forms just retrieve the first picture or video in the list.

For example, pictures associated with a product may be queried as follows:

```
{
  commerce {
    product(externalId: "ibm:///catalog/product/AuroraWMDRS-1",
            siteId: "99c8ef576f385bc322564d5694df6fc2") {
      name
      listPrice
      offerPrice
      pictures {
        name
        uriTemplate
        crops {
          name
          aspectRatio {
            width
            height
          }
          sizes {
            width
            height
          }
        }
      }
    }
  }
}
```

The `picture` and `pictures` fields have the types `CMPicture` and `[CMPicture]!` types, respectively. This way, the full functionality of CMS pictures may be used to enrich the catalog or product presentation, such as picture variants with responsive image URI templates.

As an alternative, the more general `visuals` field may be used to query for pictures, videos and other visual content as a single list.

Moreover, both categories and products have fields for catalog images. In the following example, the `defaultImageUrl` and `catalogPicture.url` fields are queried for a product:

```
{
  commerce {
    product(externalId: "sfcc:///catalog/product/25592211M",
            siteId: "sfra-en-gb") {
      name
      listPrice
      offerPrice
      locale
      defaultImageUrl
      catalogPicture {
        url
      }
    }
  }
}
```

If any picture is associated with the given product, whether within the eCommerce system or the CMS (by the aforementioned mapping in Studio), the returned URLs point to the corresponding picture, where annotated pictures from the CMS take precedence over pictures from the eCommerce system.

Note that a running CAE instance is required for actual delivery of catalog pictures (see Section 6.5, "eCommerce Setup and Configuration" [85]). The URLs to catalog pictures are generated using a configured `URLPrefixResolver`. In the default configuration, in the `CommerceConfig` class, this is an instance of `AbsoluteUrlPrefixRuleProvider`. This way, the URL prefix is generated in the same way as in the CAE. Therefore, a proper site mapping needs to be configured for the *Headless Server* as well as for the CAE. See Section 6.4.2, "Augmented Categories and Products" [82] on how to set up such a site mapping.

6.4.2 Augmented Categories and Products

Categories and products can be augmented with content of type `CMExternalChannel` and `CMExternalProduct`, respectively. These content objects are created by Studio if you choose the menu item `Augment Category` for categories or `Augment Product` for products. See Section 6.2.3, "Adding CMS Content to Your Shop" in *Studio User Manual* for more details.

If a product is augmented, the augmenting content is available in the `externalProduct`. Note that this `externalProduct` field is only non-null if this product is actually augmented. The same is true for the `externalChannel` in categories - that field is only non-null if exactly this category is augmented, the field value is not inherited from the parent category.

In contrast, page grid placements **are** inherited along the navigation hierarchy. For example, a product variant inherits placements from the parent product, a product inherits placements from its category, which in turn inherits placements from its parent category or channel, all up the navigation hierarchy.

Page grid placements of categories, products and product variants can be retrieved with the `grid` field, just the same way as for `CMChannel` content objects. For categories, the placements of the ordinary page grid are retrieved, while for products the Product Detail Page (PDP) page grid is used. Product variants simply inherit all placements from their parent product.

The placements within a page grid might be retrieved as a whole, including the whole grid structure with grid rows. Alternatively, a plain list of placements can be retrieved, optionally filtered by placement names. In the following example, only the placements `"header"` and `"additional"` are retrieved for a product:

```
{
  commerce {
    product(siteId: "sfra-en-gb",
            externalId: "sfcc:///catalog/product/25448070M") {
      externalId
      name
      externalProduct {
        name
        teaserTitle
      }
      grid {
        placements(names: ["header", "additional"]) {
          name
          items {
            name
            type
            ... on CMTeasable {
              teaserTitle
              teaserText
              picture {
                uriTemplate
              }
            }
          }
        }
      }
    }
  }
}
```

In this example, you also query the `name` and `teaserTitle` fields of an associated `externalProduct`. Note that this `externalProduct` field is only non-null if this product is actually augmented. The same is true for the `externalChannel` in categories - that field is only non-null if exactly this category is augmented, the field value is not inherited from the parent category.

6.4.3 Augmented Pages

Pages within the eCommerce system can be augmented with `CMExternalPage` content objects. The `commerce` root object offers a field `externalPage` which allows to query the CMS page content given a page ID and a site ID. The following example query retrieves the `header` and `main` placements from the `CMExternalPage` associated with the `about-us` page:

```
{
  commerce {
    externalPage(externalId: "about-us",
                 siteId: "sfra-en-gb") {
      externalId
      name
      grid {
        placements(names: ["header", "main"]) {
          name
          items {
            name
            type
          }
        }
      }
    }
  }
}
```

6.5 eCommerce Setup and Configuration

A running Commerce Hub is required. In addition, at least one properly configured Commerce Adapter is required in the *Headless Server* app.

Depending on your system setup, this may be any combination of

```
commerce.hub.data.endpoints.sfcc
commerce.hub.data.endpoints.hybris
commerce.hub.data.endpoints.wcs
```

For catalog image URLs, a site mapping has to be configured in the same way as for the CAE, for instance

- For a local CAE: `blueprint.site.mapping.calista=http://localhost:49080`
- for Docker deployment: `BLUEPRINT_SITE_MAPPING_CALISTA:  //preview.${ENVIRONMENT_FQDN:-docker.localhost}`

A running CAE instance is then required for the actual delivery of catalog images. Within the CAE, the `CategoryCatalogPictureHandler` and the `ProductCatalogPictureHandler` take care for this.

7. Personalization Extension

Personalization functionality of the *Headless Server* is available within the Blueprint module `headless-server-p13n`. It contains a GraphQL schema extension within the file `p13n-schema.graphql`, Java code and Spring configuration to implement this schema extension.

NOTE

The `p13n` Blueprint Extension must be active as a prerequisite, which is the default. If needed, the extension can be activated with the CoreMedia Extension Tool.

With the Personalization extension, documents of type CMSelectionRules can be retrieved with the headless-server. The rules of the CMSelectionRule content are not automatically evaluated. You have to implement your rules processing in the client.

7.1 Retrieve CMSelectionRules Documents

For a CMSelectionRules document the following properties can be retrieved:

- **defaultContent**: The default content
- **rules**: The rules, each rule consists of
  - **rule**: The parsed rule as String.

    Referenced content is resolved as [type.]content:1234, for example, locationTaxonomies.content:1144.

    Referenced CMSegment documents are resolved inline by applying the conditions. E.g. segment.content:11612=true is replaced with (subjectTaxonomies.content:1374>0.85 and socialuser.gender=female)=true.

    The "and" operator has a higher precedence that the "or" operator.
  - **target**: The target content
  - **referencedContent**: A list of content that is referenced in the rule.

Query to retrieve a CMSelectionRules document:

```
{
  content {
    content(id: "1234") {
      ... on CMSelectionRules {
        id
        name
        rules {
          rule
          target {
            id
          }
          referencedContent {
            id
          }
        }
        defaultContent {
          id
        }
      }
    }
  }
}
```

7.2 Rules

Personalization rules are defined in the Blueprint extension *p13n-studio* (CMSelection-RulesForm).

Key	Key Value	Operator	Value	Example
location.city	-	=, !=	Hamburg, SanFrancisco, London, Singapore	location.city=\"Hamburg\"
keyword	String	<, <=, =, >=, >	per cent [0, ..., 1]	keyword.abc<0.10
referrer.url	-	!=, =, # (contains)	String	referrer.url#\"abc\"
referrer.searchengine	-	=, !=	google,bing,yahoo	referrer.searchengine=google
referrer.query	-	# (contains)	String	referrer.query#\"test\"
(resolved segment)	-	=, !=	true	(referrer.query#\"test\" and socialuser.gender=male)=true

Table 7.1. Generic Personalization rules

Taxonomies

Key	Key Value	Operator	Value	Example
locationTaxonomies	Location tag content id	<, <=, =, >=, >	per cent [0, ..., 1]	locationTaxonomies.content:1002=0.04
subjectTaxonomies	Subject tag content id	<, <=, =, >=, >	per cent [0, ..., 1]	subjectTaxonomies.content:1214>=0.01

Key	Key Value	Operator	Value	Example
explicit.content	Subject tag content id	=, !=	1	explicit.content:1198!=1
explicit.numberOfExplicitInterests	-	<, <=, =, >=, >	Number >= 0	explicit.numberOfExplicitInterests>=2

Table 7.2. Taxonomy Personalization rules

Date/Time

Rules are configured without a timezone in Studio. A reference timezone should be defined for a project, e.g. CET, and evaluated client-side.

Key	Operator	Value	Example
system.date	<, =, >	Date	system.date=2020-10-22T00:00:00
system.dateTime	<, >	Date Time	system.dateTime>2020-10-22T17:17:00
system.dayOfWeek	<, =, >	1 (Sunday), ..., 7 (Saturday)	system.dayOfWeek=7
system.timeOfDay	<, >	Timestamp	system.timeOfDay>14:21:59

Table 7.3. Date/Time Personalization rules

Elastic Social

Key	Operator	Value	Example
socialuser.gender	=, !=	male, female	socialuser.gender=female
es_check.numberOfComments	<, <=, =, >=, >	Number >= 0	es_check.numberOfComments<=1

Key	Operator	Value	Example
es_check.number-OfLikes	<, <=, =, >=, >	Number >= 0	es_check.numberOfLikes<=3
es_check.number-OfRatings	<, <=, =, >=, >	Number >= 0	es_check.numberOfRatings>=4
es_check.userLog-gedIn	=, !=	true	es_check.userLoggedIn=true

Table 7.4. Elastic Social Personalization rules

Commerce

Key	Operator	Value	Example
com-merce.userseg-ments	# (contains)	E-Commerce User segment	commerce.usersegments#\"ibm:///cata-log/segment/8000000000000001004\"

Table 7.5. Commerce Personalization rules

SFMC

Key	Operator	Value	Example
sfmc.journeys	# (contains)	SFMC journey refer-ence	sfmc.journeys#[sfmcJourneyReference]

Table 7.6. SFMC Personalization rules

8. Persisted Queries

Persisted Queries allow clients to issue GraphQL queries without transferring the whole (potentially long) query string at each request. Instead, clients pass a short ID or hash of the query string. The actual query string is stored on the server side, either by loading it at server startup, or by a client upload as part of an *Automatic Persisted Query*.

Persisted Queries have the following advantages:

- Reduced bandwith

  The payload of the request is generally reduced.

- Better CDN cacheability

  Clients can use HTTP GET requests even for large queries.

- Reduced latency

  Using HTTP GET makes it easy to avoid CORS preflight requests issued by a browser client (HTTP OPTIONS requests).

- Query whitelisting

  Client queries may be restricted to the queries already known to the server, blocking potentially malicious queries.

Several GraphQL client frameworks support persisted queries, including Apollo Client and Relay. The CoreMedia *Headless Server* allows you to leverage this advanced GraphQL feature.

- Section 8.1, “Loading Persisted Queries at Server Startup” [92] describes how to set up the *Headless Server* to load persisted queries at startup time. This allows for query whitelisting if the set of queries issued by clients is known in advance.
- Section 8.2, “Query Whitelisting” [95] describes whitelisting of queries. That is, only queries loaded in the server during startup can be executed.
- Section 8.3, “Apollo Automatic Persisted Queries” [96] describes a more flexible approach called Automatic Persisted Queries. Automatic Persisted Queries allow clients to upload persisted queries to the server at runtime.

8.1 Loading Persisted Queries at Server Startup

Resource files containing GraphQL queries can be loaded into the Headless Server at server start up time, turning these queries into persisted queries.

Currently, three different resource file formats are supported for persisted queries, namely plain GraphQL files and JSON maps in Apollo and Relay format.

8.1.1 Defining Persisted Queries in Plain GraphQL

All resources matching the pattern configured with the property `caas.persisted-queries.query-resources-pattern` are loaded as persisted queries, one query per resource file. The filename without extension serves as the query ID. The pattern must be suitable for a Spring `PathMatchingResourcePatternResolver` which is used to load these resources.

The default pattern is `classpath:graphql/queries/*.graphql`, which means that all resource files within the `graphql/queries` directory are loaded if they have the `graphql` file extension.

Actually, not all resource files matching this pattern might be loaded - there is a configuration property `caas.persisted-queries.exclude-file-name-pattern` that specifies a regular expression for resource files to be ignored.

This pattern defaults to `.*Fragment(s)?.graphql` which is useful to skip resource files holding reusable query fragments. These fragments may then be included into a query file by means of the `#import` directive. The following is an example query including fragments from the resource `referenceFragments.graphql`:

```
query ArticleQuery($id: String!) {
  content {
    article(id: $id) {
      ... Reference
      title
      detailText
      teaserTitle
      teaserText
  }
}
#import "./referenceFragments.graphql"
```

If this query is saved in a resource file with the name `article.graphql`, the query will have the ID `article`. Therefore, you may now send a HTTP GET request with just this ID instead of the query string:

```
wget -q -O -
'http://myheadlessserver:41180/graphql?id=article&variables={"id":"1556"}'
```

8.1.2 Defining Persisted Query Maps in Apollo Format

The Apollo client tool extracts GraphQL queries from your client code and generates a JSON file in the following form:

```
{
  "version": 2,
  "operations": [
    {
      "signature":
"88a2611edf717d47e91712e57f652aed0efb8ffa3190466aa05ce448468203c5",
      "document": "query ArticleQuery(....) {...}}",
      ...
    }, {
      "signature":
"64cff55bc1c8bfc2e6f8522aa4481bebee33eb7f1d9d9a3c8af12fc2e2aa2a9b",
      "document": "query PageQuery(....) {...}}",
      ...
    },
    ...
  ]
}
```

This JSON file can then be used by your client and the *Headless Server* for query whitelisting (see Section 8.2, “Query Whitelisting” [95]).

For the *Headless Server*, the JSON file must be accessible at server startup time as a resource resolvable by a Spring PathMatchingResourcePatternResolver. One way to do this is to transfer the JSON file to the *Headless Server* workspace for inclusion at build time as a Java resource file.

By default, the *Headless Server* looks for Apollo query maps at locations specified by the configuration property `caas.persisted-queries.apollo-query-map-resources-pattern`, which defaults to `classpath:graphql/queries/apollo*.json`.

8.1.3 Defining Persisted Query Maps in Relay Format

The Relay Compiler may be asked to extract GraphQL queries from your client code and to generate a JSON file containing a map from query IDs (which are MD5 hashes) to query strings, for example:

```
{
  "33c07385fca167d81c2906b4f2ada3ac": "query AppArticleQuery(....) {...}}",
  "d614bb0396056705ef5a00815b828076": "query AppPageQuery(....) {...}}",
  ...
}
```

This map can then be used by your client and the *Headless Server* for query whitelisting (see next section).

For the *Headless Server*, the JSON map must be acessible at server startup time as a resource resolvable by a Spring PathMatchingResourcePatternResolver. One way to do this is to transfer the JSON file to the *Headless Server* workspace for inclusion at build time as a Java resource file. By default, for the *Headless Server*, the JSON map must be transferred to the *Headless Server* workspace to be included at build time. The *Headless Server* looks for Apollo query maps at locations specified by the configuration property `caas.persisted-queries.relay-query-map-resources-pattern`, which defaults to `classpath:graphql/queries/relay*.json`.

8.2 Query Whitelisting

Query whitelisting is a way to make the *Headless Server* more robust against potentially malicious (for example, expensive) queries. When whitelisting is turned on, the *Headless Server* will execute only the queries loaded into the server during startup (the whitelisted queries). All other queries will be rejected with a HTTP `403 Forbidden` response.

Query whitelisting in the *Headless Server* may be turned on by setting the configuration property `caas.persisted-queries.whitelist` to `true`.

Queries issued by clients do not need to match exactly the whitelisted ones. It suffices if their *normal form* is equal to the normal form of a whitelisted query. The GraphQL controller is configured with a `QueryNormalizer` which transforms a GraphQL query string into a normal form, where definitions and fields follow a specific order (for example, lexicographically) and whitespace is minimized.

Query whitelisting is recommended for projects which expose a GraphQL service for some dedicated clients for which the set of queries issued by the clients is known in advance. Usually, you will want to turn whitelisting off for your development environment so that front end developers can utilize the full flexibility of GraphQL. Once client development has finished, the queries can be extracted from the client code and transferred to the production environment where whitelisting is turned on.

8.3 Apollo Automatic Persisted Queries

Query whitelisting is a good and recommended option for services where the exact set of queries that clients may issue is known in advance (see Section 8.2, “Query Whitelisting” [95] . It is not an option for services which expose a generic API in GraphQL terms, such as the Github API. For such a service, allowing only a predefined set of queries would be far too restrictive, so potentially malicious queries must be detected by other means than simple whitelisting.

The Automatic Persisted Queries protocol proposed by Apollo has been designed for such services. It provides a way to take advantage of persisted queries (but without whitelisting) without losing the flexibility of the original GraphQL service.

The main idea of Automatic Persisted Queries is an optimistic request passing the SHA256 hash of the query instead of the query string itself. If the query is already known to the server, the server executes the query as normal. If the query is not known to the server, it answers with a `PersistedQueryNotFound` error. The client then reissues the request, this time passing the query string along with the hash. The next time, if the same or another client issues an optimistic request with the same hash, the server can process the query and respond with the result right away.

Automatic Persisted Queries in the *Headless Server* are turned on by default. They may be turned off by setting the configuration property `caas.persisted-queries.automatic` to `false`. However, uploading arbitrary queries is disabled anyway if whitelisting is turned on. Then, uploading queries is still supported for queries with a normal form equal to the normal form of some whitelisted query.

9. REST Access to GraphQL

Although CoreMedia recommends using the GraphQL endpoint to develop modern client applications, it may be desirable to run a client application using a REST API, for different reasons:

- A REST based client application already exists and can or should not be changed.
- Reduce network traffic.
- Limit the type and amount of queries.

While the latter two objectives could be addressed by persisted queries, query whitelisting and other security means (see Section 3.4, "Security" [21]), the first objective is the most common one why you would want to add a REST layer on top of GraphQL.

The rest controller is enabled by default. If REST access is not desired, the controller can be disabled by setting the configuration property `caas.graphql-restmapping-controller.enabled = false`.

A new REST mapping layer for the *Headless Server* now allows for issuing REST requests instead of GraphQL queries. A list of REST endpoints can now be configured which map the request to a corresponding Chapter 8, *Persisted Queries* [91]. Moreover, the query result can optionally be transformed using JSLT in order to meet the client requirements.

All REST endpoints and their corresponding persisted queries are listed and visualized in the Swagger-UI.

CoreMedia delivers the following examples of persisted queries with the *Headless Server*:

article , page , picture , site	Executes a '... by Id' GraphQL query.
search	Executes a generic 'Search' GraphQL query.

The response of persisted queries using the GraphQL endpoint is JSON as specified by graphql.org. However, it is possible to invoke a JSLT transformation on the result transparently when using the REST endpoint to a persisted query. The files specifying the JSLT transformation must have the same name as the persisted query ID for which they are intended for. These files are stored in the folder `resources/transformations`. In addition to that, it is possible to define a default or fallback transformation by creating a file called `default.jslt` (see next Section 9.2, "JSLT Transformation" [100] for details).

The corresponding REST endpoints to the example persisted queries are:

- https://<your-host>/caas/v1/article/<id>

- https://<your-host>/caas/v1/page/<id>
- https://<your-host>/caas/v1/picture/<id>
- https://<your-host>/caas/v1/site/<siteId>
- https://<your-host>/caas/v1/search/

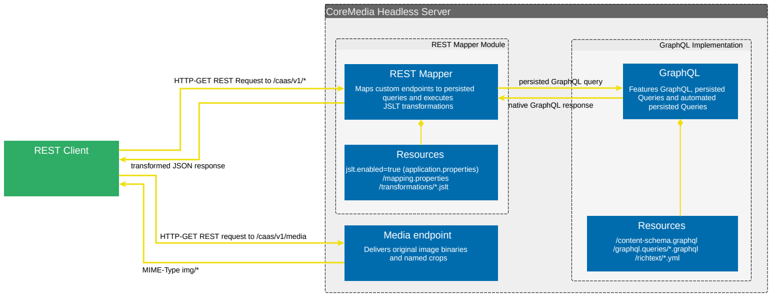

CoreMedia Headless Server
REST Mapper Module
GraphQL Implementation
REST Mapper
Maps custom endpoints to persisted queries and executes JSLT transformations
GraphQL
Features GraphQL, persisted Queries and automated persisted Queries
Resources
jslt.enabled=true (application.properties) /mapping.properties /transformations/*.jslt
Resources
/content-schema.graphql /graphql.queries/*.graphql /richtext/*.yml
Media endpoint
Delivers original image binaries and named crops
REST Client
HTTP-GET REST Request to /caas/v1/*
transformed JSON response
HTTP-GET REST request to /caas/v1/media
MIME-Type img/*
persisted GraphQL query
native GraphQL response

Figure 9.1. Headless server request/response flow using REST

9.1 Mapping REST Access to Persisted Queries

Every persisted query may be accessed via REST. To enable access via REST, it is necessary, to add a mapping of the persisted query to the intended endpoint. By default, the mapping is defined in the file `resources/graphql/rest-mapping/simple-mapping.properties`.

The name of the mapping file is configurable with the property `caas-rest.query-mapping-pattern`. The pattern must be suitable for a Spring `PathMatchingResourcePatternResolver` which is used to load these resources.

```
  #  Spring PathMatchingResourcePattern to file or files
  #  Defaults to 'graphql/rest-mapping/*.properties'
  #  Example:
 caas-rest.query-mapping-pattern = graphql/rest-mapping/my-mappings.properties
```

Any persisted query which should be made accessible via REST must be mapped with the filename of the file where the query is defined, without the extension (.graphql), followed by an equal sign and the intended mapping URI fragment.

```
      article = /article/{id}
      page = /page/{id}
      picture = /picture/{id}
      search = /search/
      site = /site/{siteId}
```

The mapping file allows commenting lines via a `#` in the beginning of a line. Empty lines are also ignored, so using them for grouping is no problem. The mapped URI fragment is always relative to the endpoint `/caas/v1`.

In addition to a plain URI fragment, it is allowed, adding REST path parameters to the mapped URI fragment using the URI template pattern: `{myPathVariable}`. The path parameters are automatically dispatched to the persisted query as GraphQL variables, as well as any query parameters.

9.2 JSLT Transformation

Depending on the requirements of a REST client, it may be desirable to transform the rather generic GraphQL JSON response into a custom JSON structure. You can do this, using JSLT transformations.

JSLT is a transformation language for JSON, inspired by jq, XPath, and XQuery. For more information and reference about it, please refer to the JSTL documentation.

JSLT transformation templates must be stored in this path: `resources/transformations`. Example transformation templates for all persisted queries are delivered:

```
      article.jslt
      _default.jslt
      errors.jslt
      page.jslt
      picture.jslt
      site.jslt
      search.jslt
```

The delivered default transformations are very basic. They simply unwrap the outer two elements of the standard GraphQL response to the pure result data. Furthermore, they showcase how to include a centralized error handling using the JSLT import directive.

A JSLT transformation file is invoked transparently using the name of the invoked persisted query. Whenever a corresponding transformation file is not found, a fallback transformation defined in `default.jslt` is invoked instead, if it exists. CoreMedia provides a fallback transformation template in the file `_default.jslt`, which simply returns the input as the output (= no transformation). To enable this fallback mechanism, rename `_default.jslt` to `default.jslt`. If the fallback template is missing, the JSLT processor is not invoked at all.

Developing more complex transformations may be time consuming as the transformations are read only once when invoked for the first time. Changes on the transformation files only take place after a restart of the *Headless Server*. To overcome this, the online JSLT evaluator is very useful. Just copy the original GraphQL response to the 'input' textarea and use the 'JSLT' textarea to develop any JSLT transformation and see result directly by clicking the `Run!` button.

10. Site Filter

Many relational database systems offer a "view" feature. A view provides an easy way to "see" only data, which is relevant for a certain use case. The *Headless Server* adopts this concept, to provide a filter to a specific site. Therefore, a site filter restricts the access of a GraphQL query to content objects of only one site.

In a scenario where CoreMedia is used to host a multitude of sites, like a site for each brand, prefiltered content might make it easier for frontend developers to develop a frontend client for one specific brand. Furthermore potential copyright problems for media content like pictures etc or an unintentional mixup of contents belonging to different sites, are prevented effectively in the first place.

A site filter is invoked simply by putting the homepage segment in front of the standard graphql endpoint or any of the REST endpoints mapped to persisted graphql queries.

Given a site with a homepage segment of 'corporate-de-de', a site filter would result in these additional endpoints:

```
#  generic access pattern to graphql with a site filter prefix
#  http://[hostname]/[homepage-segment]/graphql
http://[hostname]/corporate-de-de/graphql

#  generic access pattern to a REST endpoint with a site filter prefix
#  http://[hostname]/[homepage-segment]/caas/v1/[restendpoint]
#
#  given, there is a defined REST endpoint to /article,
#  incl a correspondingly named persisted query
http://[hostname]/corporate-de-de/caas/v1/article/[id]
```

Limitations

A site filter restricts the access to contents which belong to one site. This is accomplished without the use of users, groups or access rights. Using the standard endpoints (/graphql) without a site filter, it is still possible to access any data of any site! If you want to prevent the full access, please consider a corresponding access rule in your gateway webserver.

11. Metadata Root

The Metadata Root provides custom metadata for fields. It is configured via a GraphQL schema extension within the file `metadata-schema.graphql` and implemented in the class MetadataRoot.

The Metadata Root delivers type definitions retrieved via introspection together with their fields. The fields are enriched with metadata information. The following type definitions are supported:

- InterfaceTypesDefinition
- ObjectTypesDefinition

Query to retrieve metadata:

```
{
  metadata {
    types {
      name
      fields {
        name
        metadata
      }
    }
  }
}
```

Customization

Custom metadata can be added by adding a bean of type MetadataProvider to the Spring context.

Configuration

The Metadata Root can be disabled by setting the property `caas.metadata.enabled` to false.

11.1 PDE Mapping as Metadata

To integrate PDE (preview driven editing) functionality to a client, a mapping from the field name in the GraphQL schema to the content type property is required. This mapping is defined on the Headless Server and delivered via MetadataProvider as metadata on fields.

Configuration

The mapping is configured in file `propertyMapping.json`. For an interface type, a field name and the corresponding document type property name is listed.

Property mapping configuration (propertyMapping.json):

```
{
  "CMCollection": {
    "teasableItems": "properties.items",
    "bannerItems": "properties.items",
    "detailItems": "properties.items"
  },
  ...
```

The configured mapping applies also to types that implement the interface.

Configuration is only required for fields whose name differs from the document type property name and for implied content properties.

The default is `properties.<fieldname>`.

Implied content properties like `id`, `type` etc. are suffixed with "_" and need to be configured explicitly in the mapping file. A default configuration is provided.

Example response of metadata request with PDE mapping:

```
{
"data": {
  "metadata": {
    "types": [
      {
        "name": "CMCollectionImpl",
        "fields": [
        {
          "name": "id",
          "metadata": {
            "mapping": "id_"
          }
        },
        {
          "name": "teasableItems",
          "metadata": {
            "mapping": "properties.items"
          }
        },
          ...
        }
      ]
    }
}
```

Scope

PDE mapping metadata is provided for ObjectTypeDefinitions that implement an interface, for example `CMArticleImpl`.

The MetadataProvider for PDE Mapping is configured for preview.

12. Frontend Client Development

Web apps, created with the React Javascript library, are a great way to present content from the CMS to consumers via the headless server. This section provides general information and a guide to set up and develop a React app with the Apollo framework. Apollo connects to the GraphQL endpoint of a CoreMedia headless server and fetches the data to display a CoreMedia page, for which Apollo fits best. This setup and its structure are a recommendation to get started quickly and efficiently. Of course other frameworks or different approaches are possible.

The following sections describe how to set up a new React app, which prerequisites are needed, and how to fetch and render some CoreMedia content in the app.

NOTE

The Github repository https://github.com/CoreMedia/coremedia-headless-client-react includes an example app written in TypeScript including routing, view dispatching, preview integration and more.

12.1 Getting Started

To get started quickly, this chapter will show you how to get a React app up and running with Apollo in a basic setup. This app will seamlessly connect to a CoreMedia headless server, showcasing some CoreMedia specific solutions.

12.1.1 Prerequisites

First, you need an up-to-date version of Node.js (latest LTS) and additionally the package manager alternative yarn.

Recommended versions:

- Node: 12.x
- Yarn: 1.22.x

12.1.2 Setting up a React App

Create React App will be used for this example since it offers a fast and powerful setup to start with. It comes with pre-configured webpack, a development server and tools for testing. For more information on Create React App, see the official documentation. Other configurations, bundlers or tools that help developing with React are available too.

It is recommended to use TypeScript in your project. This guide is using JavaScript to keep the examples simple. It offers some information on how to configure and develop together with React, Apollo and CoreMedia Headless.

To install Create React App, simply enter the following code in a command line interface:

```
yarn create react-app headless-example-app
```

This will download the files into a new folder, named `headless-example-app`.

After navigating into the new folder, Apollo and Graphql can be installed as a dependency using yarn like this:

```
yarn add @apollo/client graphql
```

This will install the most recent beta version of Apollo 3. It offers improvements on caching, performance and more.

Now the app is complete, and the development server can be started with:

```
yarn start
```

12.1.3 Setup Apollo for GraphQL

The first step will be to configure the Apollo client and cache in a basic way. The more in-depth setup will be done in the Section 12.2.2, "Configuring Apollo Cache" [110]. For more information on Apollo, see the Apollo documentation.

To get Apollo running in the app, the Apollo Client needs to be imported in the App.jsx file. HttpLink and InMemoryCache will be needed for configuration.

The next step is to initialise it. A new instance of ApolloClient, named `client`, is created with two options. `cache` is an `InMemoryCache` object and `link` provides the `uri` address to the CoreMedia headless server the client will be connected to.

```
import { ApolloClient, ApolloProvider, HttpLink, InMemoryCache } from
'@apollo/client';

const client = new ApolloClient({
    cache: new InMemoryCache(),
    link: new HttpLink({
        uri: 'https://headless.example.com/graphql',
    })
});
```

In a final step, the `ApolloProvider` is wrapped around the app in the render method to be accessible to all inner components.

```
function App() {
  return (
    <ApolloProvider client={client}>
      <h1>Hello World</h1>
    </ApolloProvider>
  );
}

export default App;
```

Example 12.1. Example for Hello World App

Now the app works with Apollo and is connected to the CoreMedia Headless Server.

12.1.4 Developer Tools

For debugging, running GraphQL queries or checking the Apollo Cache we recommend following browser extensions, available for Chrome and Firefox:

- Apollo Client Devtools
- React Devtools

12.2 Basic Guides

After setting up a basic React app with an Apollo client, the next step is to fetch some data from CoreMedia Headless server. The next sections are describing, how to get some basic data and how to render content as React components.

12.2.1 Retrieving All Sites from CoreMedia Headless Server

A first simple step to display data from CoreMedia is to get a list of all available sites. For this create a new file `SitesList.jsx` which includes a React component `SitesList` and the GraphQL query.

```
import React from 'react';
import {gql, useQuery} from "@apollo/client";

const ALL_SITES_QUERY = gql`
query GetAllSites {
  content {
    sites {
      id
      name
      locale
    }
  }
}
`;

function SitesList() {
  const {loading, error, data} = useQuery(ALL_SITES_QUERY);

  if (loading) {
    return <p>Loading...</p>;
  }
  if (error) {
    return <p>Error :(</p>;
  }

  return (
    <div>
      <h1>{data.content.sites.length} Sites available</h1>
      <ul>
        {data.content.sites.map((site =>
          <li id={site.id}>{site.name} ({site.locale})</li>
        ))}
      </ul>
    </div>
  );
}

export default SitesList;
```

Example 12.2. Example Component rendering all available sites as a list

Add this component to your `App.jsx` inside the `ApolloProvider` and you should see the list of all available sites with the name and locale.

```
import SitesList from "./SitesList";
...
  return (
    <ApolloProvider client={client}>
      <SitesList/>
    </ApolloProvider>
  );
```

12.2.2 Configuring Apollo Cache

It is necessary to configure the InMemoryCache for the caching to work correctly and to successfully map every item to an ID.

Since CoreMedia content types are more complex than just boolean, string or number, the Apollo cache needs to know what kind of supertypes to expect and what types they consist of. This helps to identify cacheable content types like banner, CMArticle or CM-Collection. Therefore, the possible types need to be generated from the schema and included in the cache configuration.

The easiest way is to create a separate script to download them as JSON and save it as `possibleTypes.json` in your app. More information on this and a complete code example can be found in the documentation for "generating possible types automatic-ally".

```
import possibleTypes from './possibleTypes.json';

const client = new ApolloClient({
  cache: new InMemoryCache({
    possibleTypes
  })
  ...
});
```

Example 12.3. Configuring the Apollo Cache

CAUTION

If you don't add the generated list of possible types to the ApolloClient, the following components do not include and render any other property than the id.

12.2.3 Rendering the Homepage of a Site

This chapter goes through all necessary steps to render a site's homepage, it's PageGrid and Placements. All starting from the `path` of the page. For cleaner, smaller files, a better overview and to have GraphQL queries separated, this app uses one component for each content item like page or pageGrid etc.

Page Component and Query

The page is the entry point for the site and is loading essential data for the homepage like the PageGrid, PageGridPlacements and the banners or collections. So the query in the `Page.jsx` loads this content and passes it down to all other view components. The Query looks like this:

```
const PAGE_QUERY = gql`
  query PageQuery($pagePath: String!) {
    content {
      pageByPath(path: $pagePath) {
        id
        title
        grid {
          rows {
            placements {
              name
              items {
                ... on CMTeasable {
                  id
                  teaserText
                  teaserTitle
                }
              }
            }
          }
        }
      }
    }
  }
`;
```

Example 12.4. Page query with siteID

The `pagePath` is passed to the `useQuery` hook as an `variables` option, so it is available to the query. The path in our example is `"corporate"`.

From the received data, the rows are now passed on as an array to the PageGrid component by applying the spread operator on `data.content.page.grid`. But only if `grid` has any content. To test this it can be written as boolean equation with the ´&&´ operator, as shown in the example.

The page itself is a good place to start layouting the app, since it is the first component to render to the DOM. So a header and footer component for example could be added here too.

```
function Page(props) {
  const pagePath = "corporate";
  const { loading, error, data } = useQuery(PAGE_QUERY, {
    variables: { pagePath },
  });

  if (loading) {
    return <p>Loading...</p>;
  }
  if (error) {
    return <p>Error :(</p>;
  }

  return (
    <div className="page">
      {data.content.pageByPath.grid && <PageGrid
{...data.content.pageByPath.grid} />}
    </div>
  );
}
```

Example 12.5. Page Component render function

PageGrid Component

The PageGrid component now iterates over the rows and their containing placements, to structure the content into several PageGridPlacement components. The key parameter is required by React to have a unique identifier for rendering multiple of the same component at once.

```
function PageGrid(props) {
  const rows = props.rows || [];
  return (
    <>
      {rows.map((row) =>
        row.placements.map(
          (placement) =>
          placement && <PageGridPlacement key={placement.name} {...placement}
 />
        )
      )}
    </>
  );
}
```

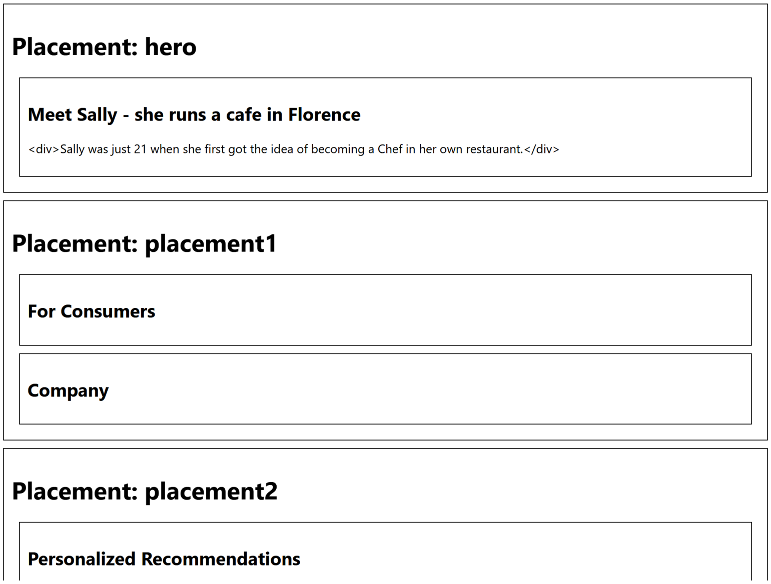
Example 12.6. Iterating over all rows of the PageGrid

PageGridPlacement Component

For this example app the resulting web page will look very basic. So for any banner, it renders only the `teaserTitle` and `teaserText`. How to render an image is described in the following section.

```
const divStyle = {
  border: '1px solid black',
  margin: '10px',
  padding: '10px'
};

function PageGridPlacement(props) {
  const name = props.name;
  const items = props.items || [];
  return (
    (items.length > 0 &&
    <div className={name} style={divStyle}>
      <h1>Placement: {name}</h1>
      {items.map((item) => (
        ((item.teaserTitle || item.teaserText) && <div style={divStyle}>
          <h2>{item.teaserTitle}</h2>
          <p>{item.teaserText}</p>
        </div>)
      ))}
    </div>)
  );
}
```

Example 12.7. The PageGridPlacement Component


Placement: hero

Meet Sally - she runs a cafe in Florence

<div>Sally was just 21 when she first got the idea of becoming a Chef in her own restaurant.</div>

Placement: placement1

For Consumers

Company

Placement: placement2

Personalized Recommendations

Figure 12.1. Screenshot of the example homepage

NOTE

Writing everything in one component can quickly lead to large and messy files. To prevent this, the query can be imported from a separate file in the components folder.

12.2.4 Navigation and Routing

Now, that the app can render a whole page, the next step is to add basic navigation. For this example, the banner from the homepage will link to an article and a link in the head of the page will lead back to the homepage.

The navigation relies on the node module "react-router-dom" and needs to be installed first:

```
yarn add react-router-dom
```

Example 12.8. Installing React Router

This will offer all capabilities of react-router, but bound to DOM elements. So in the app it provides components to create links and they will change the browser location on click. But instead of reloading the page or sending this request to the server, the router identifies the changes and matches the new URL path against different patterns, which can be provided in the `App.jsx` via routes and that link to the components defined here. For more information on react-router see their official documentation. The `App.jsx` has now a route switch, a header element linking to the homepage and a switch with two routes, matching the URL without a path to the `site` component and with path `/article/:id`, where `id` will be the content id of the article, to the `Article` component, which will be added in the next section.

```
import { BrowserRouter, Link, Route, Switch } from "react-router-dom";
...
  return (
    <ApolloProvider client={client}>
      <BrowserRouter>
        <Link to="/">
          <header>
            <h3>Home</h3>
          </header>
        </Link>

        <Switch>
          <Route path="/" exact component={Page}/>
          <Route path="/article/:id" component={Article}/>
        </Switch>
      </BrowserRouter>
    </ApolloProvider>
  );
```

Example 12.9. The App.jsx rendering with routing

The banner on the homepage need links to the article detail component. Therefor the `PageGridPlacement` should render `link` elements around each placement item and add it's `id` to the URL path:

```
import { Link } from "react-router-dom";

function PageGridPlacement(props) {
  const name = props.name;
  const items = props.items || [];
  return (
    items.length > 0 && (
      <div className={name} style={divStyle}>
        <h1>Placement: {name}</h1>
        {items.map(
          (item, index) =>
            (item.teaserTitle || item.teaserText) && (
              <div key={index} style={divStyle}>
                {item.__typename === "CMArticleImpl" && (
                  <Link to={`/article/${item.id}`}>
                    <h2>{item.teaserTitle}</h2>
                  </Link>
                )}
                <p>{item.teaserText}</p>
              </div>
            )
        )}
      </div>
    )
  );
}
```

Example 12.10. The PageGridPlacement.jsx rendering links around article banner

12.2.5 Rendering an Article

The content for an article will not be loaded via the page query for the homepage, since it only needed the banner information. So as a detail view, the article component fetches the required data with its own query using its content ID. You find the query in the completed component below. The `articleId` can have two different sources. Either the component was called by the router and it is found in the `props.match` object, or it was directly passed into the component, for example by the fragment preview and is a direct property:

```
const idFromLink = props.match.params.id;
const articleId = idFromLink ? idFromLink : props.id;
```

Example 12.11. Identify id of article

The title can immediately be used and rendered as a `<h1>` tag for example. But the URI of the picture and the detail text need further processing to work. This is done in the next two sections.

Rendering an Image

For this basic example, the original image is used. To use the address in an `<img/>` tag, it needs to be absolute. So the missing domain URL is combined with the string of the relative URI and written into the tag. In this example app the URL is already used for configuring the Apollo Client and so it is a good approach to save it in an environmental file to be accessed app wide. For example as `REACT_APP_URL`:

```
const article = data.content.article;

const serverUrl = process.env.REACT_APP_URL || "";
const imageUrl = serverUrl + article.picture.data.uri;
```

Example 12.12. Generating the full image URL

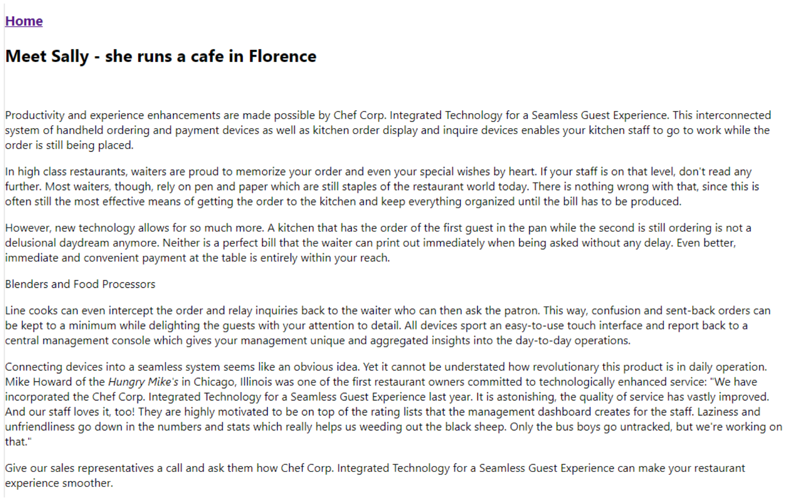
Rendering Markup as Richtext

The detail text is markup and there are multiple npm modules that help with rendering it correctly. But for now it is sufficient to use dangerouslysetinnerhtml, a way of React to set the inner HTML for DOM nodes. Since it is not secure and open for cross-site scripting it is not advised to use it in a real world scenario.

Image and Richtext ready, the article component looks as follows:

```
const ARTICLE_QUERY = gql`
  query ArticleQuery($articleId: String!) {
    content {
      article(id: $articleId) {
        id
        title
        detailText
        picture {
          data {
            uri
          }
        }
      }
    }
  }
`;

function Article(props) {
  const idFromLink = props.match.params.id;
  const articleId = idFromLink ? idFromLink : props.id;

  const { loading, error, data } = useQuery(ARTICLE_QUERY, { variables: {
articleId } });

  if (loading) {
    return <p>Loading...</p>;
  }
  if (error) {
    return <p>Error {error}</p>;
  }

  const article = data.content.article;
  const imageUrl = serverUrl + article.picture.data.uri;

  return (
    <div className="article-container">
      <h2>{article.title}</h2>
      <img src={imageUrl} alt="" />
      <p dangerouslySetInnerHTML={{ __html: article.detailText }} />
    </div>
  );
}

export default Article;
```

Example 12.13. Detailview of an article component



Home

Meet Sally - she runs a cafe in Florence

Productivity and experience enhancements are made possible by Chef Corp. Integrated Technology for a Seamless Guest Experience. This interconnected system of handheld ordering and payment devices as well as kitchen order display and inquire devices enables your kitchen staff to go to work while the order is still being placed.

In high class restaurants, waiters are proud to memorize your order and even your special wishes by heart. If your staff is on that level, don't read any further. Most waiters, though, rely on pen and paper which are still staples of the restaurant world today. There is nothing wrong with that, since this is often still the most effective means of getting the order to the kitchen and keep everything organized until the bill has to be produced.

However, new technology allows for so much more. A kitchen that has the order of the first guest in the pan while the second is still ordering is not a delusional daydream anymore. Neither is a perfect bill that the waiter can print out immediately when being asked without any delay. Even better, immediate and convenient payment at the table is entirely within your reach.

Blenders and Food Processors

Line cooks can even intercept the order and relay inquiries back to the waiter who can then ask the patron. This way, confusion and sent-back orders can be kept to a minimum while delighting the guests with your attention to detail. All devices sport an easy-to-use touch interface and report back to a central management console which gives your management unique and aggregated insights into the day-to-day operations.

Connecting devices into a seamless system seems like an obvious idea. Yet it cannot be understated how revolutionary this product is in daily operation. Mike Howard of the *Hungry Mike's* in Chicago, Illinois was one of the first restaurant owners committed to technologically enhanced service: "We have incorporated the Chef Corp. Integrated Technology for a Seamless Guest Experience last year. It is astonishing, the quality of service has vastly improved. And our staff loves it, too! They are highly motivated to be on top of the rating lists that the management dashboard creates for the staff. Laziness and unfriendliness go down in the numbers and stats which really helps us weeding out the black sheep. Only the bus boys go untracked, but we're working on that."

Give our sales representatives a call and ask them how Chef Corp. Integrated Technology for a Seamless Guest Experience can make your restaurant experience smoother.

Figure 12.2. Screenshot of the article detail page

NOTE

If you like to dive into more details and to understand some core concepts, please go to The Github repository https://github.com/CoreMedia/coremedia-headless-client-react. It includes an example app written in TypeScript with routing, view dispatching, preview integration and more.

12.3 Stand Alone Component

Instead of rendering a whole page, showing only a fragment is a common and versatile use case and can easily be done with CoreMedia Headless Server, React and Apollo. For example, one placement with a slideshow of banners should be included into a Wordpress blog.

This segment describes the most important parts of the stand alone fragment. A React App, loading specific data via Apollo, that is compiled into one single Javascript file and only needs a DOM element as anchor to be rendered into.

12.3.1 Usage

```
    <script src="dist/full/js/fragment-integration.js"></script>
    <script>
      document.addEventListener("DOMContentLoaded", () => {
        fragmentIntegration.render(
          "calista",
          "placement1",
          document.querySelector("#here"),
          ""
        );
      });
    </script>
    <div id="here"></div>
```

Example 12.14. Fragment Integration with a separate DOM Placeholder

```
    <script src="dist/full/js/fragment-integration.js"></script>
   <div data-cm-react-fragment='{"path":"calista","placement":"placement1",
 "url": "}'></div>
```

Example 12.15. Fragment Integration of DOM element with custom data attribute

12.3.2 Caching and rendering the requested placement

The `Fragment.tsx` handles the request to the Headless Server, requests the wanted data and calls a component to pass it into.

With the CoreMedia Headless Server a query can ask for a specific placement. Like in the example below, the page is set via the `$path` variable and the placement by `$placement`.

```
const PLACEMENT_OF_PATH_QUERY = gql`
  query PlacementOfPathQuery($path: String!, $placement: String! ) {
    content {
      pageByPath(path: $path) {
        grid {
          rows {
            placements(names:[$placement]) {
              name
              items {
                ...Teasable
              }
            }
          }
        }
        id
        title
      }
    }
  }
  ${teasableFragment}
`;
```

Example 12.16. fetching the wanted placement

Afterwards, the items and the name of the placement are passed to the `PageGrid Placement` component of the app and it handles the rendering from here. Since it is used in both, the stand alone fragment and the complete app, creating a shared module for the required components becomes handy.

```
const placementName = data.content.pageByPath?.grid?.placements[0].name;
const placementItems = data.content.pageByPath?.grid?.placements[0].items;

return (
  <PageGridPlacement name={placementName} items={placementItems} />
);
```

Example 12.17. rendering the PageGridPlacement

13. Configuration Property Reference

Different aspects of the *Headless Server* can be configured with different properties. All configuration properties are bundled in the Deployment Manual (Chapter 4, *CoreMedia Properties Overview* in *Deployment Manual*). The following links contain the properties that are relevant for the *Headless Server*:

- Section 4.3.1, "Headless Server Spring Boot Properties" in *Deployment Manual* contains properties for the general configuration of the *Headless Server*.
- Section 4.3.2, "Persisted Query Properties" in *Deployment Manual* contains properties for persisted queries.
- Section 4.3.3, "Remote Service Adapter Properties" in *Deployment Manual* contains properties for the configuration of the remote service of *Headless Server*.
- Section 4.3.4, "Properties of External Frameworks" in *Deployment Manual* contains properties for the configuration of GraphiQL.
- Section 4.3.5, "Renamed Properties" in *Deployment Manual* contains an overview of old and new names of renamed *Headless Server* properties.
- Section 4.10, "UAPI Client Properties" in *Deployment Manual* contains properties for UAPI clients which can also be used by the *Headless Server*.

Glossary

Blob	Binary Large Object or short blob, a property type for binary objects, such as graphics.
CAE Feeder	Content applications often require search functionality not only for single content items but for content beans. The *CAE Feeder* makes content beans searchable by sending their data to the *Search Engine*, which adds it to the index.
Content Application Engine (CAE)	The *Content Application Engine* (*CAE*) is a framework for developing content applications with *CoreMedia CMS*. While it focuses on web applications, the core frameworks remain usable in other environments such as standalone clients, portal containers or web service implementations. The CAE uses the Spring Framework for application setup and web request processing.
Content Bean	A content bean defines a business oriented access layer to the content, that is managed in *CoreMedia CMS* and third-party systems. Technically, a content bean is a Java object that encapsulates access to any content, either to CoreMedia CMS content items or to any other kind of third-party systems. Various CoreMedia components like the CAE Feeder or the data view cache are built on this layer. For these components the content beans act as a facade that hides the underlying technology.
Content Delivery Environment	The *Content Delivery Environment* is the environment in which the content is delivered to the end-user. It may contain any of the following modules: • *CoreMedia Master Live Server* • *CoreMedia Replication Live Server* • *CoreMedia Content Application Engine* • *CoreMedia Search Engine* • *Elastic Social* • *CoreMedia Adaptive Personalization*
Content Feeder	The *Content Feeder* is a separate web application that feeds content items of the CoreMedia repository into the *CoreMedia Search Engine*. Editors can use the *Search Engine* to make a full text search for these fed items.

Content item	In *CoreMedia CMS*, content is stored as self-defined content items. Content items are specified by their properties or fields. Typical content properties are, for example, title, author, image and text content.
Content Management Environment	The *Content Management Environment* is the environment for editors. The content is not visible to the end user. It may consist of the following modules: • *CoreMedia Content Management Server* • *CoreMedia Workflow Server* • *CoreMedia Importer* • *CoreMedia Site Manager* • *CoreMedia Studio* • *CoreMedia Search Engine* • *CoreMedia Adaptive Personalization* • *CoreMedia Preview CAE*
Content Management Server	Server on which the content is edited. Edited content is published to the Master Live Server.
Content Repository	*CoreMedia CMS* manages content in the Content Repository. Using the Content Server or the UAPI you can access this content. Physically, the content is stored in a relational database.
Content Server	*Content Server* is the umbrella term for all servers that directly access the CoreMedia repository: *Content Servers* are web applications running in a servlet container. • *Content Management Server* • *Master Live Server* • *Replication Live Server*
Content type	A content type describes the properties of a certain type of content. Such properties are for example title, text content, author, ...
Contributions	Contributions are tools or extensions that can be used to improve the work with *CoreMedia CMS*. They are written by CoreMedia developers - be it clients, partners or CoreMedia employees. CoreMedia contributions are hosted on Github at https://github.com/coremedia-contributions.
Controm Room	*Controm Room* is a *Studio* plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other *Studio* users.
CORBA (Common Object Request Broker Architecture)	The term *CORBA* refers to a language- and platform-independent distributed object standard which enables interoperation between heterogenous applications over a network. It was created and is currently controlled by the Object Management Group (OMG), a standards consortium for distributed object-oriented systems. CORBA programs communicate using the standard IIOP protocol.

CoreMedia Studio	*CoreMedia Studio* is the working environment for business specialists. Its functionality covers all the stages in a web-based editing process, from content creation and management to preview, test and publication. As a modern web application, *CoreMedia Studio* is based on the latest standards like Ajax and is therefore as easy to use as a normal desktop application.
Dead Link	A link, whose target does not exist.
DTD	A Document Type Definition is a formal context-free grammar for describing the structure of XML entities. The particular DTD of a given Entity can be deduced by looking at the document prolog: `<!DOCTYPE coremedia SYSTEM "http://www.coremedia.com/dtd/coremedia.dtd"` There're two ways to indicate the DTD: Either by Public or by System Identifier. The System Identifier is just that: a URL to the DTD. The Public Identifier is an SGML Legacy Concept.
Elastic Social	*CoreMedia Elastic Social* is a component of *CoreMedia CMS* that lets users engage with your website. It supports features like comments, rating, likings on your website. *Elastic Social* is integrated into *CoreMedia Studio* so editors can moderate user generated content from their common workplace. *Elastic Social* bases on NoSQL technology and offers nearly unlimited scalability.
EXML	EXML is an XML dialect used in former CoreMedia Studio version for the declarative development of complex Ext JS components. EXML is Jangaroo 2's equivalent to Apache Flex (formerly Adobe Flex) MXML and compiles down to ActionScript. Starting with release 1701 / Jangaroo 4, standard MXML syntax is used instead of EXML.
Folder	A folder is a resource in the CoreMedia system which can contain other resources. Conceptually, a folder corresponds to a directory in a file system.
Headless Server	CoreMedia Headless Server is a CoreMedia component introduced with CoreMedia Content Cloud which allows access to CoreMedia content as JSON through a GraphQL endpoint. The generic API allows customers to use CoreMedia CMS for headless use cases, for example delivery of pure content to Native Mobile Applications, Smartwatches/Wearable Devices, Out-of-Home or In-Store Displays or Internet-of-Things use cases.
Home Page	The main entry point for all visitors of a site. Technically it is often referred to as root document and also serves as provider of the default layout for all subpages.
IETF BCP 47	Document series of *Best current practice* (BCP) defined by the Internet Engineering Task Force (IETF). It includes the definition of IETF language tags, which are an abbreviated language code such as en for English, pt-BR for Brazilian Portuguese,

	or nan-Hant-TW for Min Nan Chinese as spoken in Taiwan using traditional Han characters.
Importer	Component of the CoreMedia system for importing external content of varying format.
IOR (Interoperable Object Reference)	A CORBA term, *Interoperable Object Reference* refers to the name with which a CORBA object can be referenced.
Jangaroo	*Jangaroo* is a JavaScript framework developed by CoreMedia that supports ActionScript as an input language which is compiled down to JavaScript. You will find detailed descriptions on the Jangaroo webpage http://www.jangaroo.net.
Java Management Extensions (JMX)	The Java Management Extensions is an API for managing and monitoring applications and services in a Java environment. It is a standard, developed through the Java Community Process as JSR-3. Parts of the specification are already integrated with Java 5. JMX provides a tiered architecture with the instrumentation level, the agent level and the manager level. On the instrumentation level, MBeans are used as managed resources.
JSP	JSP (Java Server Pages) is a template technology based on Java for generating dynamic HTML pages. It consists of HTML code fragments in which Java code can be embedded.
Locale	Locale is a combination of country and language. Thus, it refers to translation as well as to localization. Locales used in translation processes are typically represented as IETF BCP 47 language tags.
Master Live Server	The *Master Live Server* is the heart of the *Content Delivery Environment*. It receives the published content from the *Content Management Server* and makes it available to the *CAE*. If you are using the *CoreMedia Multi-Master Management Extension* you may use multiple *Master Live Server* in a CoreMedia system.
Master Site	A master site is a site other localized sites are derived from. A localized site might itself take the role of a master site for other derived sites.
MIME	With Multipurpose Internet Mail Extensions (MIME), the format of multi-part, multimedia emails and of web documents is standardised.
MXML	MXML is an XML dialect used by Apache Flex (formerly Adobe Flex) for the declarative specification of UI components and other objects. CoreMedia Studio uses the Open Source compiler Jangaroo 4 to translate MXML and ActionScript sources to JavaScript that is compatible with Ext JS 6.
Personalisation	On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.
Projects	A project is a collection of content items in CoreMedia CMS created by a specific user. A project can be managed as a unit, published or put in a workflow, for example.

Property	In relation to CoreMedia, properties have two different meanings: In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content item depends on the content type. In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.
Replication Live Server	The aim of the *Replication Live Server* is to distribute load on different servers and to improve the robustness of the *Content Delivery Environment*. The *Replication Live Server* is a complete Content Server installation. Its content is an replicated image of the content of a *Master Live Server*. The *Replication Live Server* updates its database due to change events from the *Master Live Server*. You can connect an arbitrary number of *Replication Live Servers* to the *Master Live Server*.
Resource	A folder or a content item in the CoreMedia system.
ResourceURI	A ResourceUri uniquely identifies a page which has been or will be created by the *Active Delivery Server*. The ResourceUri consists of five components: Resource ID, Template ID, Version number, Property names and a number of key/value pairs as additional parameters.
Responsive Design	Responsive design is an approach to design a website that provides an optimal viewing experience on different devices, such as PC, tablet, mobile phone.
Site	A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In *CoreMedia CMS* a site especially consists of a site folder, a site indicator and a home page for a site. A typical site also has a master site it is derived from.
Site Folder	All contents of a site are bundled in one dedicated folder. The most prominent document in a site folder is the site indicator, which describes details of a site.
Site Indicator	A site indicator is the central configuration object for a site. It is an instance of a special content type, most likely `CMSite`.
Site Manager	Swing component of CoreMedia for editing content items, managing users and workflows. The Site Manager is deprecated for editorial use.
Site Manager Group	Members of a site manager group are typically responsible for one localized site. Responsible means that they take care of the contents of that site and that they accept translation tasks for that site.
Template	In CoreMedia, JSPs used for displaying content are known as Templates. OR

	In *Blueprint* a template is a predeveloped content structure for pages. Defined by typically an administrative user a content editor can use this template to quickly create a complete new page including, for example, navigation, predefined layout and even predefined content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes web application	The *User Changes* web application is a *Content Repository* listener, which collects all content, modified by *Studio* users. This content can then be managed in the *Control Room*, as a part of projects and workflows.
Version history	A newly created content item receives the version number 1. New versions are created when the content item is checked in; these are numbered in chronological order.
Weak Links	In general *CoreMedia CMS* always guarantees link consistency. But links can be declared with the *weak* attribute, so that they are not checked during publication or withdrawal. Caution! Weak links may cause dead links in the live environment.
Workflow	A workflow is the defined series of tasks within an organization to produce a final outcome. Sophisticated applications allow you to define different workflows for different types of jobs. So, for example, in a publishing setting, a document might be automatically routed from writer to editor to proofreader to production. At each stage in the workflow, one individual or group is responsible for a specific task. Once the task is complete, the workflow software ensures that the individuals responsible for the next task are notified and receive the data they need to execute their stage of the process.
Workflow Server	The *CoreMedia Workflow Server* is part of the Content Management Environment. It comes with predefined workflows for publication and global-search-and-replace but also executes freely definable workflows.
XLIFF	XLIFF is an XML-based format, standardized by OASIS for the exchange of localizable data. An XLIFF file contains not only the text to be translated but also metadata about the text. For example, the source and target language. *CoreMedia Studio* allows you to export content items in the XLIFF format and to import the files again after translation.

Index