



Headless Server Manual

CoreMedia Content Cloud – v13

Copyright CoreMedia GmbH © 2026

CoreMedia GmbH

Altes Klöpperhaus, 5. OG

Rödingsmarkt 9

20459 Hamburg

International

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

Germany

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

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

Licenses and Trademarks

All trademarks acknowledged.
July 22, 2026 (Release 2512.1)

1. Preface	1
1.1. Audience	2
1.2. CoreMedia Services	3
1.2.1. Registration	3
1.2.2. CoreMedia Releases	4
1.2.3. Documentation	5
1.2.4. CoreMedia Training	8
1.2.5. CoreMedia Support	8
1.3. Typographic Conventions	10
1.4. Changelog	12
2. Overview	13
3. Configuration and Operation	15
3.1. Configuration of the <i>Headless Server</i>	16
3.2. Endpoints of the <i>Headless Server</i>	17
3.3. Caching	19
3.3.1. Unified API Cache	19
3.3.2. Cache Keys	19
3.3.3. Caffeine Cache	19
3.3.4. HTTP Cache-Control	20
3.4. Preview	21
3.4.1. JSON Preview Client	21
3.4.2. Custom Preview Client	22
3.5. Security	24
3.5.1. Query Allow List for GraphQL Queries	25
3.5.2. Limiting the Size of a Search Result	25
3.5.3. Limiting the Depth of a GraphQL Query	26
3.5.4. Limiting the Complexity of a GraphQL Query	26
3.5.5. Enforcing an Execution Timeout for GraphQL Queries	26
3.5.6. MediaType Content Negotiation	27
4. Development	29
4.1. Defining the GraphQL Schema	30
4.2. Headless Server Implementation with GraphQL-Java	32
4.2.1. Bootstrapping an Executable Schema	32
4.2.2. TypeDefinitionRegistry	32
4.2.3. RuntimeWiring	32
4.2.4. Invoking Queries	34
4.3. The @fetch Directive	37
4.4. The @inherit Directive	39
4.5. Model Mapper	40
4.6. Filter Predicates	41
4.7. Conversion Service	42
4.8. Adapter	43
4.9. Building Links	47
4.9.1. Link Composer for ID links	47
4.9.2. Link Composer for hyperlinks	47
4.9.3. Implementing Custom Link Composer	48
4.10. Content Schema	49
4.10.1. Simple Article Query	49

4.10.2. Article Query with Fragments and Parameters	50
4.10.3. Querying all available Sites	51
4.10.4. Site Query	52
4.10.5. Querying derived Sites	53
4.10.6. Page Query	54
4.10.7. Download Query	57
4.10.8. External Link Query	57
4.10.9. Querying Localized Variants	58
4.10.10. Querying Personalized Variants	58
4.11. Adding Custom Content Types	64
4.11.1. Default name convention	64
4.11.2. Adding a Custom Type Name Resolver	64
4.12. Using Time Dependent Visibility	66
4.13. Pagination	67
4.14. Remote Links	70
4.15. Taxonomies	73
4.16. Viewtypes	78
4.17. Plugin Support	80
4.17.1. Extension Points	81
4.17.2. Beans For Plugins	86
4.17.3. Resource file loading	87
4.18. Apollo Federation support	89
5. Rich Text	90
5.1. Rich Text Output	91
5.1.1. The Include Directive	93
5.1.2. YAML Anchors and Aliases	93
5.1.3. Code Comments	94
5.1.4. Name Property	94
5.1.5. Elements Property	94
5.1.6. Classes Property	95
5.1.7. Contexts and InitialContext Property	95
5.1.8. Handlers	97
5.1.9. HandlerSets Property	105
5.1.10. Internal Links	105
5.1.11. External Links	107
5.2. Using RichTextAdapters for Different Rich Text Gram- mars	108
5.2.1. Rich Text Adapters	108
5.2.2. Developing Custom RichTextAdapters	109
5.2.3. CoreMedia Grammar RichTextAdapter	111
6. Search	112
6.1. Generic Search	113
6.2. Dynamic Query Lists	120
6.3. Custom Filter Queries	122
7. eCommerce Extension	125
7.1. Headless Commerce Integration Architecture	126
7.2. Augmentation	128
7.2.1. Categories and Products Mapped to Media Con- tent	128

7.2.2. Augmented Categories and Products	129
7.2.3. Augmented Pages	131
7.3. Product Lists	133
7.4. References to Products and Categories	134
7.5. eCommerce Setup and Configuration	136
8. Persisted Queries	137
8.1. Loading Persisted Queries at Server Startup	138
8.1.1. Defining Persisted Queries in Plain GraphQL	138
8.1.2. Defining Persisted Query Maps in Apollo Format	139
8.1.3. Defining Persisted Query Maps in Relay Format	140
8.2. Query Allow Listing	141
8.3. Apollo Automatic Persisted Queries	142
9. REST Access to GraphQL	143
9.1. Mapping REST Access to Persisted Queries	145
9.2. JSLT Transformation	147
10. Site Filter	148
11. Media Endpoint	150
11.1. Media Endpoint URLs	152
11.2. Configuration of Media Endpoints	155
11.3. Customization of Media Endpoints	156
12. Metadata Root	158
12.1. PDE Mapping as Metadata	159
13. Frontend Client Development	161
13.1. Getting Started	162
13.1.1. Prerequisites	162
13.1.2. Setting up a React App	162
13.1.3. Setup Apollo for GraphQL	163
13.1.4. Developer Tools	163
13.2. Basic Guides	165
13.2.1. Retrieving All Sites from CoreMedia Headless Serv- er	165
13.2.2. Configuring Apollo Cache	166
13.2.3. Rendering the Homepage of a Site	167
13.2.4. Navigation and Routing	170
13.2.5. Rendering an Article	171
13.3. Standalone Component	174
13.3.1. Usage	174
13.3.2. Caching and rendering the requested place- ment	174
14. Configuration Property Reference	176
Glossary	177
Index	185

List of Figures

- 2.1. Headless Server overview 13
- 4.1. Remote Links 70
- 5.1. Conversion flow from Markup to a Map of scalars 109
- 7.1. Headless Commerce Integration Example 126
- 9.1. Headless server request/response flow using REST 144
- 13.1. Screenshot of the example homepage 169
- 13.2. Screenshot of the article detail page 173

List of Tables

- 1.1. CoreMedia manuals 5
- 1.2. Typographic conventions 10
- 1.3. Pictographs 11
- 1.4. Changes 12
- 4.1. Available Beans in HeadlessBlueprintBaseBeansForPluginsConfigura-
tion 87
- 5.1. Available context types for the contexts section. 96
- 5.2. Available properties for !Context and !RootContext. 97
- 5.3. Available properties for !Matcher. 97
- 5.4. Available properties for !Push and !ReplacePush. 98
- 5.5. Available properties for !ElementWriter. 99
- 5.6. Available properties for !ImageWriter. 101
- 5.7. Available properties for !LinkWriter. 101
- 5.8. Available properties for !PassStyles. 104
- 5.9. Available properties for !PassAttribute. 104

List of Examples

- 3.1. Example Cache-Control Configuration 20
- 3.2. Configuring Content Type Resolution for PDF and EPS Files 27
- 4.1. Creating a ModelMapper for Calendar objects 40
- 4.2. Creating a filter predicate 41
- 4.3. Retrieve a value from a struct with the StructAdapter 43
- 4.4. Different ways to pass the paths parameter to the settings field from the GraphQL perspective 44
- 4.5. Define SettingsAdapter as bean 44
- 4.6. Retrieve settings with the SettingsAdapter 45
- 4.7. Accessing the DataFetchingEnvironment. 45
- 4.8. Example of a new http request header to be copied to the graphql context. 81
- 4.9. Example of a filter predicate using the new context parameter. 82
- 4.10. Example of a custom SuggestionSearchServiceProvider. 85
- 4.11. Using a bean for plugin in a plugin configuration 86
- 6.1. Example implementation of a custom filter query. 122
- 11.1. Retrieving the URI template of a picture 150
- 11.2. Retrieving the URI template of a picture with an alternative image format 150
- 11.3. Retrieving the URI or the fully qualified URL of the original file of a picture 151
- 13.1. Example for Hello World App 163
- 13.2. Example Component rendering all available sites as a list 165
- 13.3. Configuring the Apollo Cache 166
- 13.4. Page query with siteID 167
- 13.5. Page Component render function 168
- 13.6. Iterating over all rows of the PageGrid 168
- 13.7. The PageGridPlacement Component 169
- 13.8. Installing React Router 170
- 13.9. The App.jsx rendering with routing 170
- 13.10. The PageGridPlacement.jsx rendering links around article banner 171
- 13.11. Identify id of article 171
- 13.12. Generating the full image URL 172
- 13.13. Detailview of an article component 172
- 13.14. Fragment Integration with a separate DOM Placeholder 174
- 13.15. Fragment Integration of DOM element with custom data attribute 174
- 13.16. fetching the wanted placement 175
- 13.17. rendering the PageGridPlacement 175

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 \[29\]](#) describes how to extend the *Headless Server*.
- [Section 5.1, “Rich Text Output” \[91\]](#) describes how to process Rich Text with the *Headless Server*.
- [Chapter 7, eCommerce Extension \[125\]](#) describes how to use eCommerce with the *Headless Server*.
- [Chapter 8, Persisted Queries \[137\]](#) describes how to configure and use Persisted Queries with the *Headless Server*.
- [Chapter 9, REST Access to GraphQL \[143\]](#) describes how to map HTTP GET requests to persisted GraphQL queries and how to transform the result.
- [Chapter 10, Site Filter \[148\]](#) describes the usage of site filters to get only content, belonging to one site.
- [Chapter 12, Metadata Root \[158\]](#) describes how you can get metadata for fields for preview driven editing functionality.
- [Chapter 13, Frontend Client Development \[161\]](#) describes how you can create a progressive web app for the *Headless Server* using React and describes some main concepts.
- [Chapter 14, Configuration Property Reference \[176\]](#) 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” \[5\]](#) 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-13>

Refer to our [Blueprint Github mirror repository](#) for recommendations to upgrade the workspace either via Git or patch files.

NOTE

If you encounter a 404 error then you are probably not logged in at GitHub or do not have sufficient permissions yet. See [Section 1.2.1, "Registration" \[3\]](#) for details about the registration process. If the problems persist, try clearing your browser cache and cookies.



Maven artifacts

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

<https://repository.coremedia.com>

You have to add your CoreMedia credentials to your Maven settings file as described in [section 3.1, "Prerequisites"](#) in *Blueprint Developer Manual*.

npm packages

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

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

The `.npmrc` file must be configured to be able to use the registry (see [Section 3.1, "Prerequisites"](#) in *Blueprint Developer Manual*).

License files

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

1.2.3 Documentation

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

<https://documentation.coremedia.com>

The manuals have the following content and use cases:

Manual	Audience	Content
Blueprint Developer Manual	Developers, architects, administrators	This manual gives an overview over the structure and features of <i>CoreMedia Content Cloud</i>. It describes the content type model, the <i>Studio</i> extensions, folder and user rights concept and many more details. It also describes administrative tasks for the features. 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 <i>Content Application Engine (CAE)</i>. You will learn how to write Freemarker templates that access the other CoreMedia modules and use the sophisticated caching mechanisms of the CAE.
Content Server Manual	Developers, architects, administrators	This manual describes the concepts and administration of the main CoreMedia component, the <i>Content Server</i>. You will learn about the content type model which lies at the heart of a CoreMedia system, about user and rights management, database configuration, and more.

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 <i>Elastic Social</i> module and how you can integrate it into your websites.
Frontend Developer Manual	Frontend Developers	This manual describes the concepts and usage of the Frontend Workspace. You will learn about the structure of this workspace, the CoreMedia themes and bricks concept, the CoreMedia Freemarker facade API, how to develop your own themes and how to upload your themes to the CoreMedia system.
Headless Server Developer Manual	Frontend Developers, administrators	This manual describes the concepts and usage of the <i>Headless Server</i> . You will learn how to deploy the Headless Server and how to use its endpoints for your sites.
Multi-Site Manual	Developers, Multi-Site Administrators, Editors	This manual describes different options to design your site hierarchy with several languages. It also gives guidance to avoid common pitfalls during your work with the multi-site feature.
Operations Basics Manual	Developers, administrators	This manual describes some overall concepts such as the communication between the components, how to set up secure connections, how to start application.
Search Manual	Developers, architects, administrators	This manual describes the configuration and customization of the <i>CoreMedia Search Engine</i> and the two feeder applications: the <i>Content Feeder</i> and the <i>CAE Feeder</i> .
Studio Developer Manual	Developers, architects	This manual describes the concepts and extension of <i>CoreMedia Studio</i> . You will learn about the underlying concepts, how to use the devel-

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

Table 1.1. 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 live online, in the CoreMedia training center or at your own location.

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

<https://www.coremedia.com/training>

Contact the training department at the following email address:

Email: training@coremedia.com

1.2.5 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:

<https://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

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

Support request

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

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

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

To put it in a nutshell, CoreMedia needs:

Support checklist

1. a person in charge (ideally, the CoreMedia system administrator)
2. extensive and sufficient system specifications
3. detailed error description
4. log files for the affected component(s)
5. if required, system files

An essential feature for the CoreMedia system administration is the output log of Java processes and CoreMedia components. They're often the only source of information for error tracking and solving. All protocolling services should run at the highest log level that is possible in the system context. For a fast breakdown, you should be logging at debug level. See [Section 4.7, "Logging"](#) in *Operations Basics* for details.

Log files

Which Log File?

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

Where do I Find the Log Files?

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

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

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

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

```
kubectrl logs --timestamps <pod>
```

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

Table 1.2. 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.
	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, smart-watches/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:

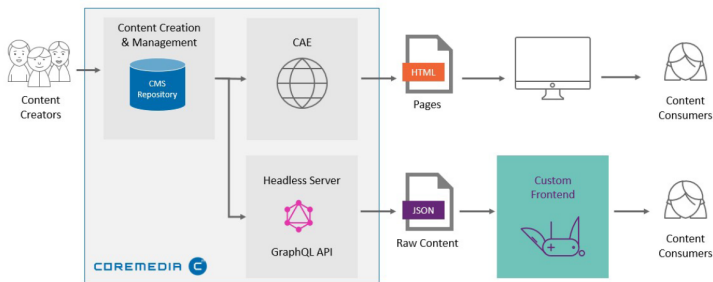


Figure 2.1. Headless Server overview

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 business logic**
- 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

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 as a container or as a standalone Spring Boot jar. See the [Deployment Manual](#) for details.

Plugin support

The *Headless Server* offers the ability to integrate custom code and resources via a plugin mechanism, using so called extension points. Adding code and resources to the *Headless Server* by a plugin has the advantage, that a plugin may have its very own build and deployment cycle, making this approach independent from the build and deployment cycle of the *Headless Server* itself.

For general details about plugins please see [Section 4.1.6, “Application Plugins”](#) in *Blueprint Developer Manual*. For details about the headless specific extension points and resource types please see [Section 4.17, “Plugin Support”](#) [80].

Apollo Federation support

Headless Server comes with Apollo Federation Support for GraphQL. For details how to enable federation support please see [Section 4.18, “Apollo Federation support”](#) [89]

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 3.3, “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/index.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 3.4, "Preview" \[21\]](#) for details.

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

REST

Persisted queries (see [Chapter 8, Persisted Queries \[137\]](#)) 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 \[143\]](#) 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 \[148\]](#) for details.

Media Endpoint

The media endpoint serves all managed media files (BLOBs). It is available at `/caas/v1/media`. See [Chapter 11, Media Endpoint \[150\]](#) for details.

3.3 Caching

This section describes the caching mechanisms that are used by the *Headless Server*.

3.3.1 Unified API Cache

The Unified API cache caches all content properties and metadata on access. The cache size can be configured via the property `repository.heap-cache-size`.

See [Section 3.12.1, “Unified API Spring Boot Client Properties”](#) in *Deployment Manual* for more information.

3.3.2 Cache Keys

Custom computations can be cached via `CacheKeys`, which use the `CoreMedia Cache` and are dependency tracked. For configuration of `CacheKeys` see [Section 3.13, “Cache Properties”](#) in *Deployment Manual*.

For the *Headless Server* several `CacheKeys` are implemented, e.g. the `SolrQueryCacheKey`. See [Section 3.3.6, “Headless Server Cache Key Properties”](#) in *Deployment Manual* for details.

Additionally, some adapters use `CacheKeys` internally, which are not exposed as public API, e.g. `ByPathAdapter`, `ViewController`, `PersonalizationRulesAdapter`, responsive media adapters.

For more cache related configuration see [Section 3.3.1, “Headless Server Spring Boot Properties”](#) in *Deployment Manual*.

3.3.3 Caffeine Cache

A Caffeine Cache is used for

- remote-links
- automatic-persisted-queries
- richtext
- preparsed-documents

See [Section 3.3.1, “Headless Server Spring Boot Properties”](#) in *Deployment Manual* for configuration options.

3.3.4 HTTP Cache-Control

HTTP Caching improves the website performance by instructing CDNs and clients to reuse previously fetched resources. The Cache-Control HTTP header offers fine-grained instructions for CDNs and HTTP clients on how to cache. With the CoreMedia Cache Control API and default implementation, projects have full control over caching behavior of content delivered by *CoreMedia Content Cloud*.

Aim of caching

HTTP Cache-Control headers can be configured for GET requests by URL pattern. The configuration options are those defined by [Cachecontrol](#). The most important property in this context is the `max-age` property.

The value of the Cache-Control header's `max-age` directive is the minimum of the values of the `validFrom/validTo` properties of the requested contents and the configured `max-age` value for the given request URL. If no cache control configuration exists and the content does not contain a value for its `validFrom/validTo` then no Cache-Control header is sent. A negative `max-age` value indicates that no Cache-Control header should be sent even in the presence of configured `validFrom/validTo` dates.

```
# articles should be cached for at most four hours
caas.cache-control.for-url-pattern[/caas/v1/article/**].max-age = 4h
# disable cache control headers for raw content requests
caas.cache-control.for-url-pattern[/caas/v1/content/**].max-age = -1
```

Example 3.1. Example Cache-Control Configuration

See [Section 3.3.5, “Headless Server Cache Control Properties”](#) in *Deployment Manual* for all configuration options.

3.4 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 as part of the Blueprint 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 9.34, "Multiple Previews Configuration"](#) in *Studio Developer Manual*.

3.4.1 JSON Preview Client

The JSON Preview Client is available in the Maven module `json-preview-client` of the Blueprint workspace.

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, the dependency to `json-preview-client` has to be removed from `pom.xml` of module `headless-server-app`.

JSON Preview Client Configuration

A JSON Preview is available for all content types that have a preview configured. To support specific content type 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
```

3.4.2 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.

Adding coremedia.preview.js

The preview client needs to include `coremedia.preview.js` to enable communication with Studio. The file is provided by the published npm package `@coremedia/preview-bootstrap`; add it as a dependency of your custom preview client. The bundled `json-preview-client` serves it at </static/preview/coremedia.preview.js>. As an alternative for older tech stacks that cannot consume npm packages, the same file is published as the `com.coremedia.preview:preview-bootstrap` Maven artifact, from which it can be added as a runtime dependency or copied.

Preview Driven Editing

To enable Preview Driven Editing (PDE), preview metadata tags need to be set (`data-cm-metadata`). This attribute can be set in the preview HTML, either via a static HTML tag or asynchronously. This means you can also fetch the available device types asynchronously and set metadata attributes later. An example for setting the metadata attributes is shown below:

```
const metadata = [
  {
    cm_responsiveDevices: {
      mobile_portrait: { width: 414, height: 736, order: 1, isDefault:
"true" },
      mobile_landscape: { width: 736, height: 414, order: 2 },
      tablet_portrait: { width: 768, height: 1024, order: 3 },
      tablet_landscape: { width: 1024, height: 768, order: 4 },
    },
    cm_preferredWidth: 1200,
  },
];

const div = document.createElement("div");
```

```
div.setAttribute("data-cm-metadata", JSON.stringify(metadata));
document.body.appendChild(div);
```

Preview Refresh Mode

The preview client can support different refresh modes. By default, the Studio preview iframe is reloaded each time a content change takes place. For modern web frameworks, like React, Angular, Vue, etc., this is not the desired behavior, as a full page reload is not necessary and expensive. Instead, the preview client can support a "soft" refresh mode, where only the changed data is fetched and a refresh is made of the affected components. To enable this mode, the preview client needs to call the method `studioAddRefreshListener` on the JavaScript `window` object and is provided by the `coremedia.preview.js` file. The method takes a callback function as parameter, which is called by Studio whenever a content change takes place. The callback itself is called with a `CustomEvent` parameter which contains additional information about the preview settings (e.g. preview date and time) in the `detail` field. An example implementation in React could look like this:

```
import { useState, useEffect } from 'react'
import './App.css'

function App() {
  const [previewTs, setPreviewTS] = useState(Date.now())

  const handleRefresh = (event) => {
    setPreviewTS(Date.now());
  };

  useEffect(() => {
    // attach listener
    window.studioAddRefreshListener(handleRefresh);
    // cleanup
    return () => {
      window.studioRemoveRefreshListener(handleRefresh);
    };
  }, []);

  return (
    <>
    <h1>Preview Refresh Sample Page</h1>
    <div className="card">
      <h3>Last Preview Timestamp: {previewTs}</h3>
    </div>
    </>
  )
}

export default App
```

The given example updates the state of the React component by re-rendering a timestamp. In a real-world application, the changed data would be fetched from the headless server.

3.5 Security

Depending on the frontend approach, the *Headless Server* may be fully or partially exposed to public access. Therefore, the *Headless Server* needs an effective 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.

WARNING

Because of the database character of any GraphQL endpoint, the publicly accessible content items should never contain any confidential data, like access credentials or user data.



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

- Externally, before the *Headless Server* is actually invoked, by using hardware (load balancers, firewalls), a web server (gateway) or a so-called backend-for-frontend approach.
- 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 accessing 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:

- Allowing only listed persisted GraphQL queries described in [Section 3.5.1, "Query Allow List for GraphQL Queries"](#) [25].
- Blocking of content items, especially `CMSSettings` content items, from delivery, using their repository path. See the deployment manual [Section 3.3, "Headless Server Properties"](#) in *Deployment Manual* for details about the configuration property `caas.graphql.repository-path-exclude-patterns`.

- Limiting the size of a search result described in [Section 3.5.2, “Limiting the Size of a Search Result”](#) [25].
- Limiting the depth of a GraphQL query described in [Section 3.5.3, “Limiting the Depth of a GraphQL Query”](#) [26].
- Limiting the complexity of a GraphQL query described in [Section 3.5.4, “Limiting the Complexity of a GraphQL Query”](#) [26].
- Enforce an execution timeout for GraphQL queries described in [Section 3.5.5, “Enforcing an Execution Timeout for GraphQL Queries”](#) [26].

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.



3.5.1 Query Allow List for GraphQL Queries

A query allow list means that only the persisted queries that reside on the server are allowed to execute. All other GraphQL queries are denied.

The allow list may be enabled by setting the configuration property `caas.persisted-queries.allow-list` to `true`. See [Section 8.2, “Query Allow Listing”](#) [141] for more details.

3.5.2 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.search.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. If the configured maximum search limit is smaller than the default limit, it overwrites the default limit.

3.5.3 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.

3.5.4 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.

3.5.5 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.

3.5.6 MediaType Content Negotiation

The `MediaController` is responsible for the delivery of binary contents like images and other content types. For security reasons, the Spring framework sets the HTTP Content-Disposition response header to the static value `inline; filename=f.txt` for potentially insecure content types, for example, PDF files, unless it was specifically set previously.

This behaviour may produce undesirable results when downloading files via the `MediaController`, as the filename is anonymous and the content type is forced to the suffix `txt`, no matter what the real content type might be.

It is however possible to configure Spring to suppress this default behaviour for specific content types, using `CaasConfig`.

```
/**
 * Code example to suppress the default Content-Disposition header for
 * potentially insecure content types. Add to CaasConfig if necessary.
 */
@Override
public void configureContentNegotiation(
    ContentNegotiationConfigurer configurer
) {
    configurer.mediaType("pdf", MediaType.APPLICATION_PDF);
    configurer.mediaType("eps", new MediaType("application", "postscript"));
}
```

Example 3.2. Configuring Content Type Resolution for PDF and EPS Files

Please see the original Spring Web MVC Documentation about Content Types for a more detailed insight about the security aspects and about so called reflected file download attacks (RFD).

Also refer to [Chapter 11, *Media Endpoint* \[150\]](#) about how the `MediaController` sets the Content-Disposition response header.

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*.

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.

Define the data to be queried

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 content 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 the GraphQL type `ContentRoot`. This root object supports CoreMedia CMS content repository access. It is implemented by the Spring bean `content` of the equally named Java class `ContentRoot`. Further content fields can be added to the GraphQL schema. These must then be implemented either by a `@fetch` directive (see section [Section 4.3, “The @fetch Directive” \[37\]](#)), 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 GraphQL type `ContentRoot` is bound to the Spring controller class `ContentRootController` by the Spring-GraphQL annotation `@QueryMapping`.

The query root type is extensible by adding a Spring controller with the Spring-GraphQL annotation `@QueryMapping` and a method named correspondingly to the query name. These controllers are also used for the eCommerce integration and the metadata root.

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 to the classpath. The preferred way of adding schema extensions is a Spring bean of type `Resource` along with the qualifier annotation `@Qualifier("graphqlSchemaResource")`. During startup, the *Headless Server* collects all beans with that qualifier and merges them together, yielding the complete schema.

Extending the schema, additional files

```
@Bean
@Qualifier("graphqlSchemaResource")
public Resource mySchemaResource() throws IOException {
```

```
PathMatchingResourcePatternResolver loader = new
PathMatchingResourcePatternResolver();
return
Arrays.stream(loader.getResources("classpath*:graphql/my-path/my-schema-extension.graphql"))
    .findFirst()
    .orElseThrow(() -> new IOException("GraphQL schema resource
graphql/my-path/my-schema-extension.graphql' not found."));
}
```

Further [adapters](#) ([Section 4.8, “Adapter” \[43\]](#)), [model mappers](#) ([Section 4.5, “Model Mapper” \[40\]](#)), [filter predicates](#) ([Section 4.6, “Filter Predicates” \[41\]](#)), 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 as a Spring bean. This separates your extensions from future changes within the `headless-server-base` module. The eCommerce integration module described in [Chapter 7, *eCommerce Extension* \[125\]](#) may serve as an example.

4.2 Headless Server Implementation with GraphQL-Java

The *Headless Server* is based on the Java implementation of GraphQL. For information and details, please see the original documentation at the [GraphQL Java Homepage](#).

This chapter summarizes the basics of GraphQL-Java and describes, how CoreMedia has used and extended the underlying framework.

4.2.1 Bootstrapping an Executable Schema

In order to process GraphQL queries, the GraphQL runtime needs an executable version of the GraphQL schema definitions. The final executable GraphQL schema models the more static part of the GraphQL runtime. A `GraphQLSource` binds all components like the `TypeDefinitionRegistry`, the `RuntimeWiring` and all schema resources into the final executable GraphQL schema.

4.2.2 TypeDefinitionRegistry

The `TypeDefinitionRegistry` class contains all objects such as interfaces, types, enums or scalars contained in one or more GraphQL schemes. The basic GraphQL scheme in *Headless Server* is defined in the file `content-schema.graphql`.

4.2.3 RuntimeWiring

The `RuntimeWiring` class wires the different components, like instrumentation and schema directives, which may manipulate the behavior of the GraphQL runtime. `RuntimeWiring` and `TypeDefinitionRegistry` are used by the `GraphQLSource` class to generate the final, executable `GraphQLSchema`.

4.2.3.1 SchemaDirectiveWiring

A `SchemaDirectiveWiring` class implements a GraphQL schema directive, like the `@fetch` directive. All schema directives are added to the `RuntimeWiring` class.

4.2.3.2 WiringFactory

A `WiringFactory` is used by the GraphQL runtime to gain information about, for example, the types and scalars necessary to process a query. To do this, a `WiringFactory` consists of several `provides*` methods and corresponding factory methods, for example, to create `DataFetcher` instances.

The *Headless Server* employs a custom implementation of a `WiringFactory` class, the `ModelMappingWiringFactory`. Basically, the `ModelMappingWiringFactory` class applies a suitable `ModelMapper` on the result of a `DataFetcher`.

4.2.3.3 ModelMapper

Implementations of `ModelMapper` are used as an additional conversion layer to convert the types delivered by a `DataFetcher` into a type, which ideally can be processed directly by the GraphQL runtime. Technically a `ModelMapper` is a Java Function with the signature `Function<T, Optional<R>>`.

All `ModelMapper` instances are created as regular Spring beans which are then consumed and invoked by the `CompositeModelMapper`. During runtime, the `CompositeModelMapper` tries to find an appropriate `ModelMapper` for a resolved property and applies it for type conversion. *Headless Server* features two `ModelMapper` implementations out of the box, the `richTextModelMapper` and the `dateModelMapper`. Both are created in `CaasConfig`.

The `FilteringModelMapper` is the so called `rootModelMapper`. As the name `rootModelMapper` implies, `FilteringModelMapper` acts as the first `ModelMapper` in the invocation chain of `ModelMapper` instances. `FilteringModelMapper` has two tasks. It invokes a list of `Predicates` to filter the resolved content, then it delegates the filter result to the `CompositeModelMapper` which in turn invokes the type conversion with a suitable `ModelMapper`.

4.2.3.4 DataFetcher

`DataFetcher` instances are the central objects to resolve the value of a property of a query. They are also created by the `WiringFactory` and feature a functional interface.

4.2.4 Invoking Queries

Headless Server uses the Spring-GraphQL library to invoke GraphQL queries that are received via HTTP requests. Spring-GraphQL operates on Springs functional web framework. At the beginning of an invocation chain stands a `RouterFunction` which declares its responsibility for a certain path and eventually restricts to specific HTTP methods. If the `RouterFunction` takes an incoming request, it delegates the further handling to a handler class. To process GraphQL queries a special `GraphQLHandler` is necessary. `GraphQLHandler` is an implementation by Spring-GraphQL and handles the runtime aspects of a query invocation.

Headless-Server comes with its own GraphQL handler which subclasses the original `GraphQLHandler`. The handler extends the original abilities by allowing also HTTP-Get requests to send GraphQL queries and enabling persisted queries via HTTP-Get.

4.2.4.1 The Query Root: ContentRoot

A query root is the primary object necessary to resolve a GraphQL query. The query root for all content queries is mapped to the property name `content` and is of type `ContentRoot`. It is created as a Spring bean in the `CaasConfig` class and provides access to the content repository. The GraphQL property `content` in the query root of the content-schema is bound to an equally named method, annotated with `@QueryMapping` in a standard Spring controller class. The mapped method delivers the `ContentRoot` object.

Binding any root type GraphQL property to an annotated Spring controller is effectively done by the Spring-GraphQL library. Using this approach, it's easy to extend the query root with custom extensions. The integration of the eCommerce extension and the metadata root is implemented in the same way.

```
# GraphQL-Schema extension:
extend type Query {
  customProperty: CustomRoot
}

type CustomRoot {
  types: [CustomType]
}

type CustomType {
  name: String
}
```

```
// Java class
@Controller
public class CustomRootController {
```

```
private final CustomRoot customRoot;

public CustomRootController (CustomRoot customRoot) {
    this.customRoot = customRoot;
}

@QueryMapping
public CustomRoot customProperty() {
    return customRoot;
}
}
```

The Java class `ContentRoot` reflects the GraphQL root type in the query root of the same name. All fields, defined in the GraphQL type correspond to a getter of the same name, for example the `page` query, which corresponds to the getter `public getPage(DataFetchingEnvironment environment)`. The result of the getter method is the so called root object (not identical to the query root) on which the following resolving process relies.

On top of the reflection based invocation of the getter methods, `CoreMedia` added the `@fetch` directive, which allows to express the data fetching for a property in the GraphQL scheme using the Spring Expression Language (SpEL). The Spring EL allows a less restrictive approach to use the query root or even to invoke completely different objects instead of the `ContentRoot`, namely most of the adapters, like the `SettingsAdapter` or the `NavigationAdapter`.

4.2.4.2 Default Invocation Chain

The GraphQL runtime tries to resolve any property in a query using a `DataFetcher`. By default, the built in `PropertyDataFetcher` is used to resolve a property. The `PropertyDataFetcher` operates on the query root using reflection. The object returned by the query root is then processed reversely by the invocation chain of `DataFetcher` instances.

1. `CapStructPropertyDataFetcher`
2. `ViewBySiteFilterDataFetcher`
3. `FilteringDataFetcher`
4. `ConvertingDataFetcher`

4.2.4.3 Fetch Directive Invocation Chain

The `@fetch` directive alters the default invocation chain by invoking the `SpelDataFetcher` class instead of the default `PropertyDataFetcher` class. The `SpelDataFetcher` uses SpEL to operate either on the query root

or invokes instead other Spring Beans. Either way, the invoked object has to return an object or a `DataFetcherResult` instance. The slightly altered invocation chain of `DataFetcher` is:

1. `ViewBySiteFilterDataFetcher`
2. `FilteringDataFetcher`
3. `ConvertingDataFetcher`

4.2.4.4 Resolving Custom Scalars

After the invocation chain, all resolved properties need to be resolved into basic data types. In all cases where the runtime does not know how to handle certain data types, a custom scalar is usually part of the schema. Custom scalars are part of the bootstrapping process. Each custom scalar declared in a schema must have a corresponding `GraphQLScalarType`. The `GraphQLScalarType` is responsible along with a coercing class to resolve any custom scalar into primitives.

4.2.4.5 Resolving Types

The task of a type resolver is to resolve an input type, which is delivered by the invocation chain, into a known GraphQL type defined in the schema, for example, resolve a `Content` object of the content type `CMArticle` into the string `CMArticleImpl`, which is pointing to the implementing GraphQL type in the schema. Now, that the GraphQL runtime "knows" how to use the content object by knowing the implementing type and its properties, the properties can be resolved, using the above described invocation chain.

4.3 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 4.8, “Adapter” \[43\]](#)) 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}}}}`

4.4 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 content 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.

4.5 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.

4.6 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(QUALIFIER_CAAS_FILTER_PREDICATE)
public FilterPredicate
validityDateFilterPredicate(ContextVariableValueService<Object>
contextVariableValueService) {
    return new ValidityDateFilterPredicate(contextVariableValueService);
}
```

Example 4.2. Creating a filter predicate

Beans with Qualifier `PluginSupport#QUALIFIER_CAAS_FILTER_PREDICATE` are picked up automatically and applied to `ModelMappers`.

4.7 Conversion Service

The *Headless Server* uses a configured instance of the Spring `ConversionService` at several places.

The `ConvertingDataFetcher` uses the conversion service to convert any resolved field value, e.g. a `RichTextAdapter` object, into an object type which can be consumed by the GraphQL runtime, e.g. a string representation of a rich text.

The Spring Expression Language uses the `ConversionService` to convert any argument into the necessary type required to be invoked on a function, e.g. `@localizedVariantsAdapterFactory.to().getLocalizationForLocale(#root, #language, #country, #variant)`.

In this example, the function `getLocalizationForLocale(...)` is called on an instance of a `LocalizedVariantsAdapter`. The `ConversionService` will try to convert all parameters of the function into appropriate types defined by the function itself.

To add additional converters to the conversion service, one simply has to implement the `org.springframework.core.convert.converter.Converter` and create it as a regular Spring Bean.

Additional Spring Converters are especially useful when using GraphQL input type. See [section “Automatic conversion of GraphQL input types to Java Objects” \[46\]](#) for details.

4.8 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.

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

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

StructAdapter

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 CMSSettingsImpl implements CMSSettings ... {
  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

SettingsAdapter

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.

FilenameBlobAdapter and DownloadFilenameBlobAdapter

The `FilenameBlobAdapter` and the `DownloadFilenameBlobAdapter` provide access to the filename of a blob. The `DownloadFilenameBlobAdapter` additionally provides a download filename, which is either the same as the filename or a forced download name, if specified. To control the download behavior of a blob, the `DownloadFilenameBlobAdapter` uses the setting `useCMDDownloadFilename`. If this setting is set to `true` in the `CMDDownload` document or somewhere up the corresponding navigation tree relation, the property `filename` is used instead of the standard filename calculation.

Although the schema demands a nested list structure as an argument, the underlying GraphQL framework accepts an incomplete list structure, even a single string. GraphQL-Java enhances the missing lists automatically, which might lead to an unwanted or unexpected behaviour. Therefore it is recommended, to always specify an unambiguous, full list structure as demanded by the schema. This is true for the usages of the `SettingsAdapter` in the schema as well as for the `StructAdapter`, whenever a setting or a struct is retrieved by its path.

```
// a single string is interpreted as a single path, as expected.
settings(paths: "commerce")

// same behaviour as above
settings(paths: ["commerce"])
settings(paths: [["commerce"]])

// two elements list: EACH entry is handled as an individual path! (potentially
// unexpected behaviour)
settings(paths: ["commerce", "endpoint"])

// recommended: fully qualified list structure specifying two settings paths
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

Additionally to those Adapters explicitly mentioned above, there are several other Adapters available, for example:

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.

DataFetchingEnvironment Support

Similar like a `DataFetcher`, adapters can access the GraphQL `DataFetchingEnvironment`. The access is possible in two flavours. First, the `DataFetchingEnvironment` is available in the SpEL evaluation context under the name `#dataFetchingEnvironment`.

Second, if an adapter extends the class `DataFetchingEnvironmentAware`, the current `DataFetchingEnvironment` is automatically injected after(!) the instantiation of the adapter via its factory. The `DataFetchingEnvironment` can be accessed via a getter. Additionally, `DataFetchingEnvironmentAware` offers a convenience method to read any variable from the GraphQL context, e.g. the preview date.

Due to the fact, that the `DataFetchingEnvironment` is injected after the adapters instantiation, the factory method itself cannot access the `DataFetchingEnvironment` via the getter. If access is necessary during instantiation, the first approach via an explicit SpEL expression is inevitable.

```
# Example: Passing the DataFetchingEnvironment explicitly via SpEL.
type CMNavigationImpl implements CMNavigation {
    ...
    grid: PageGrid @fetch(from: "@pageGridAdapter.to(#root, 'placement',
#dataFetchingEnvironment)")
    ...
}

# Example: Transparent access to the DataFetchingEnvironment.
```

```
# - 'byPathAdapter' extends DataFetchingEnvironmentAware.
# - DataFetchingEnvironment is injected after the factory method 'to()!!'
# - 'getPageByPath(#path)' accesses the DataFetchingEnvironment internally.

type ContentRoot {
    ...
    pageByPath(path: String!): CMChannel @fetch(from:
"@byPathAdapter.to().getPageByPath(#path) ")
    ...
}
```

Example 4.7. Accessing the DataFetchingEnvironment.

Automatic conversion of GraphQL input types to Java Objects

In most cases, scalar types like strings or integers are sufficient as call parameters for the execution method of an adapter. It is however also possible to use GraphQL input types as call parameters as well. Since GraphQL does not offer an out-of-the-box conversion of GraphQL input types to Java Objects, the given value of an input type will be a deserialized composition of collection classes, similar to a deserialized JSON string.

A Spring Converter offers the possibility to convert the values of an input type into a typed Java object, which in turn can be used as input parameter of an adapter. If a suitable converter exists, the underlying Spring Expression Language will invoke it implicitly. A Converter simply has to be created as a regular SpringBean. It will then be automatically added to the EvaluationContext of the Spring Expression Language. See also [Section 4.7, “Conversion Service”](#) [42].

4.9 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.

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 4.3.2.2, "Writing Link Schemes"](#) in *Content Application Developer Manual* and [Section 4.3.2.3, "Post Processing Links"](#) in *Content Application Developer Manual*).

Link Composer

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.

4.9.1 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.

4.9.2 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*.

4.9.3 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 5.1.10, “Internal Links” \[105\]](#). 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:/contenttype/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.

4.10 Content Schema

The types and interfaces in the schema file `content-schema.graphql` define a subset of the CoreMedia Blueprint content model in GraphQL SDL terms. The Blueprint content 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 content items 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: "@byPathAdapter.to().getPageByPath(#path,
#context['caasViewName'])")
  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.

4.10.1 Simple Article Query

The following GraphQL query is a simple example for fetching data from a `CMArticle` content item. 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(imageFormat: JPG)
      }
    }
  }
}
```

```
}
}
```

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 `crop-Name` and `width` parameters. The file name section at the end of `uriTemplate` is optional and only for SEO purpose.

The parameter `imageFormat` is optional. By providing a supported image format (jpeg, png or gif), the url is calculated to trigger the corresponding image transformation on the media endpoint. Note, that in this case the trailing part of the url which contains the filename, is of course not optional anymore.

4.10.2 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 GraphQL, 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.

4.10.3 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  
  }  
}
```

```
}
}
```

4.10.4 Site Query

To query a specific site, issue a query containing the content/site field, either with the root segment of the associated homepage or 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
                }
              }
            }
          }
        }
      }
    }
  }
}

```

Alternatively you may query a site using the root segment parameter instead (example shortened):

```

query SiteQueryByRootSegment($rootSegment: String!) {
  content {
    site(rootSegment: $rootSegment) {
      name
      id
      locale
    }
  }
}

```

4.10.5 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
      }
    }
  }
}

```

```

    }
  }
}

```

4.10.6 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 content 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, for example, simply 'for-professionals'.

Navigation

Especially when rendering pages, showing some kind of navigation components is usually an important task. Some of these components may be the current navigation level, the homepage and the main navigation or the next navigation level of the currently displayed page.

The graphql type CMNavigation offers everything necessary to render any type of navigation component.

```
{
  content {
    pageByPath(path: "corporate/for-consumers/aurora-b2c") {
      id
      name
      segment
      mainNavigation: root {
        children {
```

```

        id
        name
        segment
    }
    currentNavigationLevel: parent {
      children {
        id
        name
        segment
      }
    }
    subNavigationLevel: children {
      id
      name
      segment
    }
  }
}
}

```

4.10.7 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
          sizeLong
          eTag
        }
      }
    }
  }
}

```

4.10.8 External Link Query

For a CMExternalLink that is linked to for example a CMTeaser, an external URL can be queried as follows:

```

{
  content {
    content(id: "12216") {
      ... on CMTeaser {
        teaserTarget {
          ... on CMExternalLink {
            url
            openInNewTab
          }
        }
      }
    }
  }
}

```

```

    }
  }
}

```

4.10.9 Querying Localized Variants

Localized variants of any content object can be obtained using either the field `localizedVariants` 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.

4.10.10 Querying Personalized Variants

The Headless Server allows you to retrieve information about variants. To this end, each content object defines a field `contentVariants` that contains a list of possible variants, if any. Each variant has a field `content` for the variant defining the content and a field `conditionStrings` for the conditions that need to apply for the variant to become active. A field `conditions` provides access to parsed forms of the conditions. For each variant's content, you can either query further fields directly or you can make follow-up requests. The latter approach is particularly relevant if a single-page application accesses the Headless Server directly, allowing you to check the conditions first and retrieve only those content items that are actually needed. However, the increased

number of round trips can outweigh the benefits of a reduced size of the initial response.

During delivery, the first variant for which at least one condition is active is used. If no variant with an active condition can be found, the baseline is used. As a special case, the list of variants contains variants that define only conditions, but no content. These are test control variants that allow you to track a test event. For rendering, these variants should be replaced by the next variant with an active condition *and* a defined content.

4.10.10.1 Retrieving Variants and Conditions

This is how you retrieve variants using GraphQL:

```
{
  content {
    page(id: "123456") {
      id
      name
      contentVariants {
        content {
          id
        }
        conditionStrings
      }
    }
  }
}
```

The output would look like this:

```
{
  "data": {
    "content": {
      "page": {
        "id": "123456",
        "name": "The Page",
        "contentVariants": [
          {
            "content": {
              "id": "345678"
            },
            "conditionStrings": [
              "cmec:///pl3n/segment/12345678901234567890;cmecType=segment"
            ]
          }
        ]
      }
    }
  }
}
```

The `contentVariants` attribute contains all variants in order of descending priority as per `ContentVariantService.getVariants()`. You should render the first variant for which one of its conditions is met and for which the `content` attribute is defined. If no variant with a matching condition is found, the baseline should be rendered.

Some of the variants may be test control variants, for which the `content` field is not set. In such a case, the GraphQL result might look like this:

```
{
  "data": {
    "content": {
      "page": {
        "id": "123456",
        "name": "The Page",
        "contentVariants": [
          {
            "content": null,
            "conditionStrings": [
              "cmec:///pl3n/variant/567890"
            ]
          },
          {
            "content": {
              "id": "345678"
            },
            "conditionStrings": [
              "cmec:///pl3n/variant/567891"
            ]
          }
        ]
      }
    }
  }
}
```

If the conditions of a test control variant apply, the choice of the test control should be tracked, but you should continue to search for less prioritized variants with a non-null `content` attribute and satisfied conditions for rendering. If there is no such variant, use the baseline.

4.10.10.2 Retrieving Variants and Parsed Conditions

Especially when interfacing with a personalization provider directly, it can be beneficial to retrieve the conditions in parsed form:

```
{
  content {
    page(id: "123456") {
      id
      name
      contentVariants {
        content {
          id
        }
        conditions {
          condition
          vendor
          type
          externalId
          vendorAttributes
        }
      }
    }
  }
}
```

```
}
}
```

The output would look like this:

```
{
  "data": {
    "content": {
      "page": {
        "id": "123456",
        "name": "The Page",
        "contentVariants": [
          {
            "content": {
              "id": "345678"
            },
            "conditions": [
              {
                "condition":
"cmec:///p13n/segment/12345678901234567890;cmecType=segment",
                "vendor": "cmec",
                "type": "segment",
                "externalId": "12345678901234567890",
                "vendorAttributes": "cmecType=segment"
              }
            ]
          }
        ]
      }
    }
  }
}
```

See the notes about the interpretation of the variant list in the previous section.

4.10.10.3 Retrieving Properties Uniformly

Retrieving properties for both the baseline and all variants at the same time can result in complex queries when the previous approach is used, because property names have to be specified twice. This is particularly verbose, if multiple accesses to variants are nested.

```
{
  content {
    page(id: "123456") {
      id
      name
      teaserTitle
      teaserText {
        text
      }
      contentVariants {
        content {
          id
          name
          ... on CMChannel {
            teaserTitle
            teaserText {
              text
            }
          }
        }
      }
    }
  }
}
```

```
        conditionStrings
      }
    }
  }
}
```

To simplify such situations, both the variants and the baseline itself can be requested in a single collection `contentVariantsAndBaseline`:

```
{
  content {
    page(id: "123456") {
      contentVariantsAndBaseline {
        content {
          id
          name
          ... on CMChannel {
            teaserTitle
            teaserText {
              text
            }
          }
        }
      }
      conditionStrings
    }
  }
}
```

This results in:

```
{
  "data": {
    "content": {
      "page": {
        "contentVariantsAndBaseline": [
          {
            "content": {
              "id": "345678",
              "name": "Variant Page",
              "teaserTitle": "Variant Home",
              "teaserText": "Home Variant Home"
            },
            "conditionStrings": [
              "cmec:///p13n/segment/12345678901234567890;cmecType=segment"
            ]
          },
          {
            "content": {
              "id": "123456",
              "name": "The Page",
              "teaserTitle": "Home",
              "teaserText": "Home Sweet Home"
            },
            "conditionStrings": null
          }
        ]
      }
    }
  }
}
```

For `contentVariantsAndBaseline` the baseline always comes last in the list, because it has a lower priority than all variants, which are ordered by descending priority. Unlike the variants, the baseline has null values for the properties

`conditions` and `conditionStrings`, indicating that no conditions have to be checked.

4.10.10.4 Retrieving the Baseline

The following query allows you to retrieve the baseline for a content item. Note that this is rarely needed, as you typically obtain a link to a baseline first and then navigate to its variants.

```
{
  content {
    page(id: "345678") {
      id
      name
      base {
        id
        name
      }
    }
  }
}
```

This yields:

```
{
  "data": {
    "content": {
      "page": {
        "id": "345678",
        "name": "Variant 1",
        "base": {
          "id": "123456",
          "name": "The Page"
        }
      }
    }
  }
}
```

4.11 Adding Custom Content Types

The following approaches are available for mapping a custom CMS content type to a GraphQL type in the schema:

- **Follow the default name convention** — define a GraphQL interface named after the CMS content type and a GraphQL type with the `Impl` suffix. No additional configuration is required.
- **Implement a custom resolver** — choose a custom GraphQL type name and optionally also a custom GraphQL interface name, and implement resolvers to map CMS content types to GraphQL type/interface names programmatically.

4.11.1 Default name convention

If the GraphQL interface name matches the CMS content type name exactly, the default resolver automatically resolves content of that type to the GraphQL object type whose name is the interface name with an `Impl` suffix appended. If no such GraphQL type is found, it walks up the CMS content type parent hierarchy until a matching GraphQL `*Impl` type is found.

To add a custom GraphQL type and interface that maps to the content type `CustomArticle`, add the following to the GraphQL schema:

```
interface CustomArticle @inherit(from: ["CMTeasable"]) {
  text: String
}

type CustomArticleImpl implements CustomArticle @inherit(from:
["CMTeasableImpl"]) {
  text: String @fetch(from: "text")
}
```

The built-in `contentTypeNameResolver` resolves content of type `CustomArticle` to `__typename: "CustomArticleImpl"`.

4.11.2 Adding a Custom Type Name Resolver

If the GraphQL type name does not follow the convention, you can implement a custom resolver. Add a `TypeNameResolver<Content>` Spring bean and return the GraphQL type name wrapped in an `Optional`, or an empty `Optional` to pass control to the next resolver in the chain — including the built-in

convention resolver. Use `@Order` to ensure the custom resolver is executed before the default one.

To add a custom GraphQL type named `CustomType` which maps to the content type `CMExampleDocument`, add the type and the resolver:

```
type CustomType implements CMTeasable @inherit(from: ["CMTeasableImpl"]) {
  text: String @fetch(from: "text")
}
```

```
@Bean
@Order(0) // run before the built-in resolver
public TypeNameResolver<Content> myTypeNameResolver() {
  return content -> {
    if ("CMExampleDocument".equals(content.getType().getName())) {
      return Optional.of("CustomType");
    }
    return Optional.empty(); // fall through to the next resolver
  };
}
```

If the GraphQL type implements only standard interfaces (whose names match a standard existing content type), no further configuration is needed.

If it also implements a custom interface whose name does not match any content type name, also declare and register that interface.

```
type CustomType implements CustomTeasable @inherit(from: ["CMTeasableImpl"])
{
  text: String @fetch(from: "text")
}

interface CustomTeasable @inherit(from: ["CMTeasable"]) {
  text: String
}
```

```
@Bean
public ProvidesTypeNameResolver myProvidesTypeNameResolver() {
  return typeName -> "CustomTeasable".equals(typeName) ? Optional.of(true)
    : Optional.empty();
}
```

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 graphql invocation utilizes `PreviewDateContextParameter` to forward the **view date** delivered by the header or, if not available, the current date as a fallback value to the `GraphQLContext`. The view date can be retrieved via the `DataFetchingEnvironment` in a data fetcher.

```
import static
com.coremedia.caas.headless_server.plugin_support.PluginSupport.CONTEXT_PARAMETER_NAME_PREVIEW_DATE;
ZonedDateTime viewDate =
dataFetchingEnvironment.getGraphQLContext().get(CONTEXT_PARAMETER_NAME_PREVIEW_DATE);
```

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 Pagination

Overview

To optimize communication between client and server and to reduce overfetching, data can be retrieved paginated from the *Headless Server*.

There are two possibilities to retrieve paged data in the *Headless Server*:

- Retrieve data completely from the data sources, paging is applied afterwards.
- Retrieve only a subset of data, pagination is applied to data sources.

The *Headless Server* exposes both possibilities in the same way to the client. The different paging mechanisms are applied in the GraphQL schema.

Fields that can be retrieved paged are suffixed with "Paged", e.g. `items` and `itemsPaged`. The optional parameters are `offset` and `limit` and the return value is a `*PaginationResult` depending on the result type of the collection.

A paged field is defined with the following pattern:

```
fieldPaged(offset: Int, limit: Int): *PaginationResult
```

A pagination result type is defined with a specific list, for example `Content_`:

```
type ContentPaginationResult {
  totalCount: Double
  result: [Content_]
}
```

Apply paging after data retrieval

Not all data source accesses allow a paged query. For example, a content has a list property. This list property is always retrieved completely together with all other content properties from the *Content Server*, it cannot be loaded partially. The *Headless Server* provides the possibility to apply paging, after the data was retrieved by invoking a `pagingHelper`. The `pagingHelper` gets the offset, limit and the original collection as parameters and returns a paged result.

Example how to apply the `pagingHelper` to a list property:

```
itemsPaged(offset: Int, limit: Int): CollectionItemPaginationResult
@fetch(from: "@pagingHelper.apply(#offset, #limit, #this.items)")
```

Apply paging in Adapter

If data should be retrieved paged from a data source, the corresponding `Adapter` can implement the `PagingAware` interface. This interface provides methods to apply offset and limit and to return a `Paging` result containing metadata and the actual result.

Example how to get a paged result from a `PagingAware` Adapter:

```
itemsPaged(offset: Int, limit: Int): CollectionItemPaginationResult
@fetch(from: "@queryListAdapter.to(#root).getPagingResult(#offset, #limit)")
```

Return types

Paged fields return a `*PaginationResult` type, that contains metadata and the result. For each result type, either a specific `*PaginationResult` type needs to be defined or the generic `ContentPaginationResult` type can be used. See `content-schema.graphql` for predefined `*PaginationResult` types.

A paged result looks like:

```
"itemsPaged": {
  "totalCount": 4,
  "result": [
    {
      "id": "7326"
    },
    {
      "id": "7334"
    }
  ]
}
```

Add a custom paged field

To add a custom paged field, first it needs to be decided, if the paging should be applied on data retrieval (`PagingAware` Adapter) or after data retrieval (`pagingHelper`).

`PagingAware` Adapter:

- Let the custom Adapter implement the interface `PagingAware` and implement the corresponding functions.
- Add a new field to the schema with parameters `offset` and `limit` and a `fetch` directive that calls the Adapter's `getPagingResult` method.

PagingHelper:

- Add a new field to the schema with parameters offset and limit and a fetch directive that calls the `pagingHelper` with offset, limit and the original list (e.g. `#this.items`).

The return type definition applies to both variants:

- If a custom return type is required, define a new `*PaginationResult` return type with fields `totalResult` and `result`. Define a custom type for the result field.
- Alternatively use one of the predefined `*PaginationResult` return types.

4.14 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. A fresh batch loader is created for each GraphQL request. Resolved URLs are cached across requests using a Spring Cache backed by Caffeine (configured via `caas.cache-specs[remote-links]`), so repeated requests for the same remote link return the cached URL without a remote call. The cache is keyed by the combination of content ID, siteId and context, so the same content requested for different sites or contexts resolves to its own URL.

Configuration

The following configuration options are available, see [Section 3.3.4, “Remote Service Adapter Properties”](#) in *Deployment Manual* for details:

<code>caas.remote.baseUrl</code>	Base URL of the remote handler.
<code>caas.remote.httpClientConfig.*</code>	Configuration options of the HttpClient used by the RestTemplate.
<code>caas.cache-specs[remote-links]</code>	(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.

4.15 Taxonomies

Overview

In the *Headless Server*, taxonomies can be retrieved via id or path, see [section “Retrieve a taxonomy” \[73\]](#). How to retrieve content items tagged with specific taxonomies is described in [section “Retrieve content tagged with a taxonomy” \[76\]](#).

Retrieve a taxonomy

Taxonomies are handled with the bean `taxonomyAdapter` defined in `Caas Config.java`. The following functionality is supported:

- Retrieve a taxonomy by id
- Retrieve a localized taxonomy by id
- Retrieve a taxonomy by path segments

Retrieve a taxonomy by id

This GraphQL query is a simple example for fetching a `CMTaxonomy` content item by `id`.

```
{
  content {
    taxonomy(id: "coremedia:///cap/content/1234") {
      id
      name
      value
    }
  }
}
```

Retrieve a localized taxonomy by id (and locale)

This GraphQL query is a simple example for fetching a localized `CMTaxonomy` document by `id` and `locale`. If the `locale` parameter is `null` or skipped completely, the target locale will be determined by the `TaxonomyLocalizationStrategy` that is passed to the `TaxonomyAdapter`.

```
{
  content {
    localizedTaxonomy(id: "coremedia:///cap/content/1234", locale: "en-US")
  }
}
```

```
{
  id
  value
}
```

To fetch a list of the supported locales, the query `supportedTaxonomyLocales` can be executed:

```
{
  content {
    supportedTaxonomyLocales
  }
}
```

The result contains the list `translations` which describes the target locales and the `defaultLocale`, that describes the locale of the `value` field of every taxonomy.

Retrieve taxonomy localizations for CMTaxonomy

Localizations can also be queried directly on the `CMTaxonomy` type via field `localization(locale:String!)` or via field `localizations`, which provides all available localizations:

```
{
  content {
    content(id: "12345") {
      ... on CMTaxonomy {
        id
        name
        value
        localizations {
          id
          locale
          value
        }
        localization(locale: "de") {
          id
          locale
          value
        }
      }
    }
  }
}
```

Retrieve a taxonomy by path segments

To retrieve a taxonomy via path segments, these parameters can be provided:

- **pathSegments:** A String containing only the taxonomy `value`, or all path segments up to the root, separated by '/'. The path segment lookup is performed via linked parents, i.e. the `value` of the linked parent up to the root.
- **type:** `CMTaxonomy` for Subject Taxonomies (default), `CMLocTaxonomy` for Location Taxonomies. Will be matched exactly.
- **siteId:** The `siteId` of the site to look up taxonomies. If empty, taxonomies will be looked up globally.

This GraphQL query is a simple example for fetching a `CMLocTaxonomy` content item by a single segment path:

```
{
  content {
    taxonomyByPath(pathSegments: "Tokyo") {
      id
      name
      value
    }
  }
}
```

This GraphQL query is an example for fetching a `CMTaxonomy` content item by the complete segment path up to the root:

```
{
  content {
    taxonomyByPath(pathSegments: "Asia/Japan/Tokyo") {
      id
      name
      value
    }
  }
}
```

Global and site specific taxonomies

If a `siteId` is provided, taxonomies are retrieved for the corresponding site, else globally.

- Global Taxonomies are retrieved from:
global configuration path + `/Taxonomies`, e.g. `/Settings/Taxonomies`.
- Site specific Taxonomies are retrieved from:
site root folder + site specific configuration path + `/Taxonomies`,
e.g. `/Sites/Aurora Augmentation/United States/English/Options/Settings/Taxonomies`.

The site specific and global configuration paths are defined via configuration properties and can be overridden:

- `content.globalConfigurationPath`

- `content.siteConfigurationPath`

The configuration paths are passed to the constructor of the class `TaxonomyAdapterFactory.java` and can also be changed explicitly in `CaasConfig.java`.

NOTE

It is assumed, that taxonomy paths are unique. If multiple taxonomies are found for a path, only the first one is returned. Also, localized path segments are not supported. Every segment must match the actual `value` field of a node.



Retrieve content tagged with a taxonomy

Content items can be tagged with subject and/or location taxonomies. For faster lookup, the tags are stored for each content item in the Solr index of the CAE. To retrieve a content item tagged with a specific taxonomy, a search is executed on the CAE index. Therefore, the *Headless Server* search extension is required, to use this functionality.

Search query with custom filter

To search for a content item that is tagged with a given taxonomy, a provided custom filter query needs to be applied. See [Section 6.3, "Custom Filter Queries" \[122\]](#) for details of custom filter queries.

The custom filter queries identified by keys `SUBJ_TAXONOMY_OR` and `LOCATION_TAXONOMY_OR` provide capabilities to create a Solr query containing filter queries for given taxonomies.

The custom filter queries take either the taxonomy ids or the paths as arguments:

- List of numeric ids, e.g. `["1234", "5678"]`
- List of content ids, e.g. `["coremedia:///cap/content/1234", "coremedia:///cap/content/5678"]`
- List of complete path segments or the taxonomy value, e.g. `["Blog/Kitchen", "Cooking"]`. The lookup is performed via `taxonomyAdapter`.

Example query to retrieve articles that are tagged with the subject taxonomies 'Cooking' OR 'Kitchen':

```
{
  content {
    search(query: "*", docTypes: ["CMArticle"],
      customFilterQueries: [
        {SUBJ_TAXONOMY_OR: ["Cooking", "Kitchen"]}
      ]
    ) {
```

```
    numFound
    result {
      id
    }
  }
}
```

To combine taxonomies via **AND**, multiple custom filter queries with the same key can be provided.

Example query to retrieve articles that are tagged with the subject taxonomies 'Cooking' **AND** 'Kitchen':

```
{
  content {
    search(query: "*", docTypes: ["CMArticle"],
      customFilterQueries: [
        {SUBJ_TAXONOMY_OR: ["Cooking"]},
        {SUBJ_TAXONOMY_OR: ["Kitchen"]}
      ]
    )
  }
  {
    numFound
    result {
      id
    }
  }
}
```

Global and site specific taxonomies

The taxonomy lookup is performed globally by default. For a site specific lookup the siteld can be given as first list item, e.g. ["siteId:corporate", "1234", "5678"]

Example query to retrieve articles that are tagged with the site specific location taxonomies 'Europe' **OR** 'Asia' of the site with siteld 'corporate':

```
{
  content {
    search(query: "*", docTypes: ["CMArticle"],
      customFilterQueries: [
        {LOC_TAXONOMY_OR: ["siteId:corporate", "Europe", "Asia"]}
      ]
    )
  }
  {
    numFound
    result {
      id
    }
  }
}
```

4.16 Viewtypes

Overview

When rendering a content item, different information may be used to display. For example, a collection could be displayed as a simple list, or as teasers with picture and details. To control the different variants, several content types have a `viewtype` property containing a `layout variant`.

To define a query for a subset of fields, that are needed for rendering, the `viewtype` property of a content item needs to be considered in the query. This applies not only for a subset of the top level fields, but also for fields of linked contents.

Clients can already formulate conditional queries depending on the viewtype, but these queries can become overly complex and hard to handle. The default approach, to always retrieve all fields, leads to overfetching.

To be able to pose precise queries and only retrieve the required fields, a viewtype specific type is needed. This viewtype specific type can be defined in the GraphQL schema by adding a new type, that is marked with the annotation `@viewtype`.

Example: A collection that is viewtype specific for viewtype `hero`:

```
type ViewTypeHeroCollection implements CMCollection @inherit(from:
["CMCollectionImpl"]) @viewtype(name: "hero") {}
```

The `@viewtype` annotation takes the name of the viewtype as argument and can be defined for object types:

```
directive @viewtype(
  name: String!
) on OBJECT
```

Now the client can pose viewtype specific queries:

```
{
  content {
    content(id: "1234") {
      ... on CMCollection {
        name
      }
      ... on ViewTypeHeroCollection {
        items {
          ... on CMTeasable {
            pictures {
              uriTemplate
            }
          }
        }
      }
    }
  }
}
```

```
}  
  }  
} }  
}
```

Serverside, the viewtype specific types are resolved by a `PostProcessor`, that evaluates the annotation and returns the specific type instead of the default type.

Supported types

The viewtype annotation is supported for the types

- `CMCollection`
- `PageGridPlacement`

The viewtype specific types `ViewTypeHeroPageGridPlacement` and `ViewTypeHeroCollection` for layout variant `hero` are already available in the Blueprint

To define a new viewtype specific type, follow these steps:

- Add a new type, that implements one of the supported interfaces `PageGridPlacement` or `CMCollection`.
- Add the annotation `@viewtype` to that type with the name of the layout variant to resolve.

To support further types with a viewtype specific type resolution, a custom `PostProcessor` can be provided as bean.

4.17 Plugin Support

Overview

Headless Server supports the usage of plugins by offering headless specific extension points and service beans. For details about how to develop and deploy plugins, please see [Section 4.1.6, “Application Plugins”](#) in *Blueprint Developer Manual*.

Three types of plugin support are offered:

- Extension points

Extension points are concrete implementations of certain interfaces or classes in a plugin, which are annotated with `@ExtensionPoint`. These extension points are then consumed by the *Headless Server*.

- Beans for plugins

Beans for plugins are service beans especially designed to be used by a plugin. The service beans are provided to the plugin as a spring configuration class, which should be imported by the plugins own spring configuration class.

- Resource file loading

Similar to extension points, a plugin may provide resource files to *Headless Server*, which are then additionally consumed at different points within *Headless Server*.

To develop a plugin for *Headless Server*, you need to add these maven dependencies to your project:

```
<!-- headless specific extension points -->
<dependency>
  <groupId>com.coremedia.caas</groupId>
  <artifactId>headless-server.plugin-support</artifactId>
  <version>${cms.version}</version>
  <scope>provided</scope>
</dependency>

<!-- optional: common beans for plugins -->
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>common.beans-for-plugins-container</artifactId>
  <version>${cms.version}</version>
  <scope>provided</scope>
</dependency>

<!-- optional: headless blueprint base beans for plugins -->
<dependency>
  <groupId>com.coremedia.blueprint.base</groupId>
  <artifactId>bpbse-headless-server-core</artifactId>
  <version>${cms.version}</version>
```

```
<scope>provided</scope>
</dependency>
```

4.17.1 Extension Points

CopyToContextParameter

The extension point `CopyToContextParameter` offers the ability to declare additional parameters, for example, HTTP headers, which then will be added automatically to the GraphQL context during graphql queries.

```
public class SecurityTokenContextParameter implements
CopyToContextParameter<String, String> {

    public static final String HEADER_NAME = "X-SECURITY-TOKEN";
    public static final String NAME_IN_CONTEXT = "securityToken";

    @Override
    public String getName() {
        return HEADER_NAME;
    }

    @Override
    public String getNameInContext() {
        return NAME_IN_CONTEXT;
    }

    @Override
    public ContextValueOrigin getValueOrigin() {
        return ContextValueOrigin.REQUEST_HEADER;
    }

    @Override
    public boolean previewOnly() {
        return false;
    }
}

// Bean factory in the plugin configuration class
@Bean
public SecurityTokenContextParameter securityTokenContextParameter() {
    return new SecurityTokenContextParameter(settingsService);
}
```

Example 4.8. Example of a new http request header to be copied to the graphql context.

Implementations of this extension point can also be provided within `CaasConfig`, using the qualifier `PluginSupport#QUALIFIER_CAAS_COPY_TO_CONTEXT_PARAMETER` at bean creation.

FilterPredicate

Implementations of `FilterPredicate` are also fed to the `ModelMapper` during the server start. They can also be provided within `CaasConfig`, using the qualifier `PluginSupport#QUALIFIER_CAAS_FILTER_PREDICATE` at bean creation. See [Section 4.6, “Filter Predicates” \[41\]](#) for details.

```
public class SecurityTokenFilterPredicate implements FilterPredicate<Object>
{
    private static final String SERVER_SECRET_TOKEN =
"secret-hash-set-by-environment";

    public boolean test(DataFetchingEnvironment dataFetchingEnvironment, Object
o) {
        String securityToken = (String) ((Map<String, Object>)
dataFetchingEnvironment.getContext()).get(SecurityTokenContextParameter.NAME_IN_CONTEXT);

        return (securityToken != null &&
SERVER_SECRET_TOKEN.equalsIgnoreCase(securityToken));
    }
}

// Bean factory in the plugin configuration class
@Bean
public SecurityTokenFilterPredicate securityTokenFilterPredicate(
ContextVariableValueService contextVariableValueService
) {
    return new SecurityTokenFilterPredicate(contextVariableValueService);
}
```

Example 4.9. Example of a filter predicate using the new context parameter.

PluginSchemaAdapterFactory

In conjunction with the plugin resource loading feature for GraphQL schema extensions, so called schema adapters can be invoked to resolve schema properties via the `fetch` directive using the Spring Expression Language (SpEL). While the out of the box schema adapters can be used without any problems, named schema adapters from within a plugin have to implement the extension point `PluginSchemaAdapterFactory`, in order to define the adapters name from within the plugin.

Note, that plugin schema adapters can only be used within the SpEL context of the GraphQL schema. For more information about adapters, please see [Section 4.8, “Adapter” \[43\]](#). An example can be found in the JavaDocs at `PluginSchemaAdapterFactory`.

CustomScalarType

The extension point `CustomScalarType` allows the definition of custom scalar types in plugins. Note, that to use a custom scalar type, you need to define

it in the GraphQL schema as well as instantiate an instance of `CustomScalarType` as a Spring bean in the `PluginConfiguration`.

The custom scalar types of plugins are provided as a Spring bean by the `Headless Server` using the qualifier `PluginSupport#QUALIFIER_PLUGIN_CUSTOM_SCALARS`.

CaasWiringFactory

This extension point allows to define additional `WiringFactory` implementations in a plugin by implementing `CaasWiringFactory`.

Implementations of this extension point are provided as a Spring bean using the qualifier `PluginSupport#QUALIFIER_PLUGIN_WIRING_FACTORIES`.

All `WiringFactory` implementations, which are not part of a plugin must be marked with the qualifier `PluginSupport#QUALIFIER_CAAS_WIRING_FACTORIES` in order to distinguish them from the predicates created inside of `Headless Server` and merge them with the ones implemented in plugins.

PluginSchemaGenerator

This extension point allows to define an alternative `SchemaGenerator` by implementing the interface `PluginSchemaGenerator`.

In case a plugin defines a `PluginSchemaGenerator`, it replaces the default schema generator. Only one schema generator may be active at a time. In case multiple plugins try to register its own `PluginSchemaGenerator`, it cannot be assured, which one will be active. The active schema generator is printed into the log during startup of the *Headless Server*.

LinkComposer

This extension point allows to define additional `LinkComposer` implementations in a plugin by implementing the interface `UriLinkComposer` for URI links or `GraphQLLinkComposer` for GraphQL links.

Implementations of this extension point can be accessed using the qualifier `PluginSupport#QUALIFIER_PLUGIN_LINK_COMPOSERS_URI` and `PluginSupport#QUALIFIER_PLUGIN_LINK_COMPOSERS_GRAPHQL` at bean creation.

The default implementations of `LinkComposer` are then merged with the ones implemented in plugins.

CustomFilterQuery

A `CustomFilterQuery` provides the ability to add additional filter queries to the Solr query, using the `customFilterQueries` parameter of the GraphQL `search` query.

Implementations of this type are provided as a Spring bean via a Spring configuration class, e.g. `CaasConfig` or via the means of a plugin.

For details about the implementation please see [Section 6.3, “Custom Filter Queries” \[122\]](#).

SearchServiceProvider

All search related adapters are using an SPI (Service Provider Interface) architecture, which makes it very easy to implement and provide an alternative service provider. The corresponding service provider defines method signatures for all important aspects of a search, like query creation, parameter validation, execution and result transformation. The corresponding adapter then invokes these aspects but it only acts as kind of a runtime environment for the service provider, not implementing any relevant business logic itself.

The search related SPI extension points are providing default implementations for the latter three aspects, while the creation of the solr query is usually part of a custom implementation.

The regular search is based on the `SearchServiceProvider`. The provider is invoked by an instance of the `SearchAdapter`. The default service provider is implemented by `DefaultSearchServiceProvider`. Implementations of `SearchServiceProvider` provided via a plugin will replace the `DefaultSearchServiceProvider`.

FacetedSearchServiceProvider

The faceted search is based on the `FacetedSearchServiceProvider`. The provider is invoked by an instance of the `FacetedSearchAdapter`. The default service provider is implemented by `DefaultFacetedSearchServiceProvider`. Implementations of `FacetedSearchServiceProvider` provided via a plugin will replace the `DefaultFacetedSearchServiceProvider`.

SuggestionSearchServiceProvider

Search suggestions are based on the `SuggestionSearchServiceProvider`. The provider is invoked by an instance of the `SuggestionAdapter`. The default service provider is implemented by `DefaultSuggestionSearchServiceProvider`. Implementations of `SuggestionSearchServiceProvider` provided via a plugin will replace the `DefaultSuggestionSearchServiceProvider`.

```
public class CustomSuggestionSearchSPI extends SuggestionSearchServiceProvider
{
    private final SolrQueryBuilder suggestionsSolrQueryBuilder;

    public CustomSuggestionSearchSPI(ContentRepository contentRepository,
                                     SolrQueryBuilder
suggestionsSolrQueryBuilder) {
        super(contentRepository);
        this.suggestionsSolrQueryBuilder = suggestionsSolrQueryBuilder;
    }

    @Override
    public SolrQuery createSearchQuery(String searchExpression,
                                       Site site,
                                       List<String> docTypes,
                                       boolean includeSubTypes,
                                       Content siteRootDocument,
                                       DataFetchingEnvironment
dataFetchingEnvironment,
                                       List<FilterQueryArg>
customDynamicFilterQueries,
                                       List<FilterQueryArg>
customStaticFilterQueries,
                                       Map<String, Function<List<String>,
String>> filterQueryDefinitions) {
        List<String> filterQueries = new ArrayList<>();

        ZonedDateTime viewDate = dataFetchingEnvironment
            .getGraphQLContext()
            .get(PluginSupport.CONTEXT_PARAMETER_NAME_PREVIEW_DATE);

        filterQueries.add(
            SearchQueryHelper.validFromPastToValueQuery(
                suggestionsSolrQueryBuilder.getValidFromFieldName(),
                viewDate));

        filterQueries.add(
            SearchQueryHelper.validFromValueToFutureQuery(
                suggestionsSolrQueryBuilder.getValidToFieldName(),
                viewDate));

        filterQueries.addAll(
            SearchHelper.getExpandedCustomFilterQueries(
                customStaticFilterQueries,
                customDynamicFilterQueries,
                filterQueryDefinitions));

        return suggestionsSolrQueryBuilder.createSearchQuery(
            searchExpression,
            siteRootDocument,
            -1,
            -1,
            filterQueries,
            Collections.emptyMap(),

```

```

        false);
    }
}

```

Example 4.10. Example of a custom *SuggestionSearchServiceProvider*.

Extension Points for the Media Endpoints

The media endpoints of *Headless Server* can be extended by implementing extension points also. For details about these extension points, please see [Section 11.3, “Customization of Media Endpoints” \[156\]](#).

4.17.2 Beans For Plugins

Beans for plugins are regular Spring Beans provided by the main application context to the plugin context. The beans are meant to provide certain services of the *Headless Server* as part of the public Plugin API.

WARNING

Many beans for plugins are regular service beans, also used in the main application context. As most of these beans are designed as singletons, they must never be used by a plugin via the configuration classes used in the main application context. Doing so will result in unpredictable side effects, as the beans, designed as singletons, are created in the plugin context also!



To use beans for plugins the correct way, import the intended beans for plugins Configuration to your plugin configuration class.

Headless Server offers these beans for plugins configuration classes to provide beans for plugins:

- `com.coremedia.cms.common.plugins.beans_for_plugins2.CommonBeansForPluginsConfiguration`
- `com.coremedia.blueprint.base.caas.beans_for_plugins.HeadlessBlueprintBaseBeansForPluginsConfiguration`

```

@Configuration(proxyBeanMethods = false)
@Import({
    HeadlessBlueprintBaseBeansForPluginsConfiguration.class,
})
public class MyPluginConfiguration {
    @Bean
    public MyBean getMyBean(
        @Qualifier("caeSolrQueryBuilder") SolrQueryBuilder solrQueryBuilder
    ) {

```

```
    ) {  
      // create the bean  
      ...  
      return myBean;  
    }  
  }  
}
```

Example 4.11. Using a bean for plugin in a plugin configuration

CommonBeansForPluginsConfiguration

Provides shared beans which are not especially offered by *Headless Server*. For details about the provided beans, see the original [JavaDoc](#).

HeadlessBlueprintBaseBeansForPluginsConfiguration

Provides *Headless Server* specific beans originating from Blueprint Base.

Bean Name	Type	Description
caeSolrQueryBuilder	SolrQueryBuilder	Provides a SolrQuery for the 'cmdismax' endpoint.
dynamicContent-SolrQueryBuilder	SolrQueryBuilder	Provides a SolrQuery for the 'select' endpoint.
suggestionsSolrQuery-Builder	SolrQueryBuilder	Provides a SolrQuery for the 'suggest' endpoint.

Table 4.1. Available Beans in HeadlessBlueprintBaseBeansForPluginsConfiguration

4.17.3 Resource file loading

Supported resource types in plugins are expected at predefined, non configurable paths. To provide any of the supported resource file types, simply add your resources at the predefined resource paths.

The paths are defined in the class [PluginSupport](#) as static constants.

GraphQL schema extensions

Plugins may add their own graphql schema extensions by adding resource files. The path pattern is defined at [PluginSupport#GRAPHQL_SCHEMA_RE-](#)

`SOURCE_PATTERN`. For details about graphql schema definition, see [Section 4.1, “Defining the GraphQL Schema” \[30\]](#).

Schema metadata property mapping

To support preview based editing in Studio, it is also possible to add appropriate metadata property mappings for the graphql extensions. The path pattern is defined at `PluginSupport#METADATA_PROPERTY_MAPPING_RESOURCE_PATTERN`. For details about metadata property mapping, see [Chapter 12, *Metadata Root* \[158\]](#).

Richtext transformations

To add individual richtext transformations, plugins may provide additional YAML configurations. The path pattern is defined at `PluginSupport#RICHTEXT_RESOURCE_PATTERN`. For details about richtext transformations, see [Section 5.1, “Rich Text Output” \[91\]](#).

Persisted queries

Serverside persisted queries may be provided using the path pattern defined at `PluginSupport#PERSISTED_QUERY_RESOURCE_PATTERN`. For details about persisted queries, see [Section 8.1, “Loading Persisted Queries at Server Startup” \[138\]](#).

Rest mappings to persisted queries

The optional, corresponding rest mappings for persisted queries may be provided using the path pattern defined at `PluginSupport#REST_MAPPING_RESOURCE_PATTERN`. For details about REST Access, see [Section 9.1, “Mapping REST Access to Persisted Queries” \[145\]](#).

JSLT transformations

Additional JSLT transformations for REST requests of persisted queries may be provided using the path pattern defined at `PluginSupport#JSLT_RESOURCE_PATTERN`. For details about JSLT transformations, see [Section 9.2, “JSLT Transformation” \[147\]](#).

4.18 Apollo Federation support

Headless Server offers optional Apollo Federation Support for GraphQL. To enable Federation Support add this library to your maven dependencies.

```
<dependency>
  <groupId>com.apollographql.federation</groupId>
  <artifactId>federation-graphql-java-support</artifactId>
  <scope>runtime</scope>
</dependency>
```

During server start the library will be detected and federation support will be enabled. The log will show an additional info level message at server start, that federation support is active.

To test that federation support is in fact working, the enhanced introspection can be queried like this:

```
query {
  _service {
    sdl
  }
}
```

For more details about the federation support please see the original documentation about the integration on the net at [Spring-GraphQL](#). About GraphQL federation in general please refer to the original documentation at [Apollo GraphQL](#).

5. Rich Text

Processing rich text content is a complex issue. The following two chapters describe, how the *Headless Server* handles different aspects of rich text processing.

- [Section 5.1, “Rich Text Output” \[91\]](#) describes, how the *Headless Server* processes CoreMedia rich text grammar out of the box and how the output can be adapted to your needs by means of a YAML based configuration.
- [Section 5.2, “Using RichTextAdapters for Different Rich Text Grammars” \[108\]](#) describes, how the *Headless Server* can be extended in order to handle any type of custom XML grammar.

5.1 Rich Text Output

Delivering CoreMedia RichText 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.
- **[CMLocalized!]**: A list of all embedded content objects within 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:
richtext-field-name {
  graphql-field-name(view: "transformation-name")
}

Example:
detailText{
  text: (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 'initialContext', which are all included in this basic example file.

When writing a configuration in YAML style, the indentation is most important. For a reference about YAML you may refer to <https://yaml.org/>.

5.1.1 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
```

5.1.2 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:
anyOtherProperty: *nameAnchor
```

```
# which is equivalent to:  
anyOtherProperty: 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
```

5.1.3 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!
```

5.1.4 Name Property

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

```
name: myTransformerName
```

5.1.5 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" ]
```

5.1.6 Classes Property

A list of CSS classes to be considered when parsing the raw markup. Usually included by the include directive (see [Section 5.1.1, “The Include Directive” \[93\]](#)) 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_style !!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
...
```

5.1.7 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
```

5.1.7.1 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.

5.1.8 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

5.1.8.1 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.

Property	Description
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
    ...
```

5.1.8.2 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 }
      ...
```

5.1.8.3 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:

```

```

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
    ...
```

Custom Output Handler

New handlers can be added or existing handlers can be replaced. To do this, extend the class `AbstractOutputHandler` and develop a customized handler according to your individual needs. Then register the new handler by adding a `TypeDescription` with the new handler as a regular Spring Bean to `CaasConfig.java` with the Qualifier `'caasRichtextTypeDescriptions'`. After this, the new handler can be used under its tag name in a transformation definition, like `default.yml`.

```
public class MyOutputHandler extends AbstractOutputHandler {
    ...
}

@Bean
@Qualifier(QUALIFIER_CAAS_RICHTEXT_TYPE_DESCRIPTIONS)
public TypeDescription customTypeDescription() {
    return new TypeDescription(MyOutputHandler.class, new
    Tag("!MyOutputHandler"));
}
```

Replacing an existing handler is done the same way but in this case using the existing tag name, e.g. `LinkWriter`. If an existing handler is replaced, the revised handler should implement the same properties, e.g. `attributeTransformers`. Not doing so will most likely raise an exception at startup because, the transformation `yml` tries to set properties, that do not exist anymore.

```
public class CustomLinkWriter extends AbstractOutputHandler {
    ...
}

@Bean
@Qualifier(QUALIFIER_CAAS_RICHTEXT_TYPE_DESCRIPTIONS)
public TypeDescription customLinkWriterTypeDescription() {
```

```
return new TypeDescription(CustomLinkWriter.class, new Tag("!Linkwriter"));
}
```

5.1.8.4 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.

Custom attribute and element transformers can be added by implementing the interface `AttributeTransformer` or `ElementTransformer` and passing them as type description to the `RichtextTransformerReader`.

Element Transformer

An `Element Transformer` allows generating 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 adding/removing/modifying attributes of an element node. Available attribute transformers are:

```
!PassStyles
!PassAttribute
```

Attribute transformers can be added to all output writers.

PassStyles Attribute Transformer

The PassStyles attribute transformer filters the styles contained in the class attribute. In YAML it is introduced by:

```
!PassStyles
```

Property	Description
styles	List of style classes to be passed through to delivery.

Table 5.8. Available properties for !PassStyles.

Example: Passing only the declared styles of the class attribute to the output.

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

PassAttribute Attribute Transformer

The PassAttribute attribute transformer passes the declared attribute and its value to the delivery. In YAML it is introduced by:

```
!PassAttribute
```

Property	Description
name	Name of the attribute to be passed through with its value.

Table 5.9. Available properties for !PassAttribute.

Example: Passing the declared attribute to the output.

```
...
  outputHandler:
    !ElementWriter
    attributeTransformers:
      - !PassAttribute { name: colspan }
```

5.1.9 HandlerSets Property

Using handler sets allows grouping and reusing 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
    ...
```

5.1.10 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 4.9, “Building Links” \[47\]](#).

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

data-href Contains the generated link.

data-show Contains the link behavior.

Possible values for link behavior 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 `CMedia` objects (or subtypes).

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

```
<p>ChefSupply RGB LED Strip</p>
<p></p>
```

Note that an example link composer `contentUriLinkComposer` for content objects is configured in `CaasConfig.java` and may need customization as described in [Section 4.9, “Building Links” \[47\]](#). This example link composer generates links that contains the content id:

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

5.1.11 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 external link.
- data-show** Contains the link behavior.
- data-role** Contains the target frame identifier, if available.

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

- new
- replace
- embed
- other
- none

5.2 Using RichTextAdapters for Different Rich Text Grammars

The *Headless Server* comes with an architecture to parse different flavors of rich text, including an out of the box RichTextAdapter to parse and transform the well known CoreMedia rich text grammar. The architecture allows customizing both, the grammar to be parsed and the underlying parsing technology, using standard Spring Boot beans.

The content repository delivers rich text as objects of type Markup, whereas the content schema declares a custom scalar type RichTextTree on all fields of the type Markup. The underlying architecture of graphql-java requires registering an implementation of the Coercing interface for a declared custom scalar type (RichTextTree). This is done in the config class CaasConfig by adding the scalar type and its conversion type Map and creating a bean of type GraphQLScalarType, which takes the Coercing implementation. By doing this, graphql-java now always expects a Map<String, Object> object when resolving fields of the scalar type RichTextTree.

With this kind of registration, only one Coercing class per scalar is possible. To overcome this limitation, CoreMedia has added a mechanism to invoke custom classes to handle different grammar types and to use any type of parsing/transformation technology.

5.2.1 Rich Text Adapters

An implementation class of the interface RichTextAdapter is used to parse and eventually transform markup of a specific grammar and provide it as an object structure, representing the transformed markup.

In order to create a grammar specific RichTextAdapter, a ModelMapper to map Markup to RichTextAdapter is registered using the graphql-java instrumentation in CaasConfig. The ModelMapper then creates for every Markup object an instance of the appropriate RichTextAdapter implementation. The RichTextAdapter processes the given Markup into a custom representation of the rich text. Finally, an additional converter is responsible to convert the custom representation of the RichTextAdapter into the common markup representation of type Map<String, Object>.

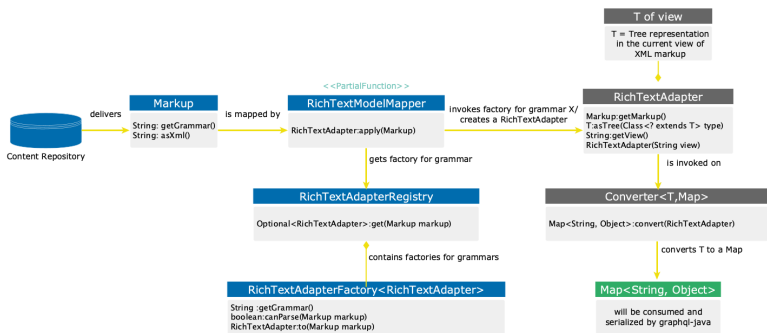


Figure 5.1. Conversion flow from Markup to a Map of scalars

NOTE

The diagram shows the general conversion flow of markup objects. It hides the configuration details, which are considered implementation specific, for example, the YAML based configuration of the transformation of the out-of-the-box RichTextAdapter for the CoreMedia grammar, as described at [Section 5.1, "Rich Text Output"](#) [91].



5.2.2 Developing Custom RichTextAdapters

Please note, that the following example code is abbreviated for demonstration purposes.

To develop a custom RichTextAdapter, these three basic steps must be made:

1. Implement your own RichTextAdapter
2. Implement a RichTextAdapterFactory for your RichTextAdapter
3. Implement a "To-Map" Converter for your RichTextAdapter

First step:

Implement a custom RichTextAdapter which is able to parse the intended XML grammar and provide it as an object structure which represents the XML tree. Optionally also support the transformation of the parsed grammar into any other "view", for example, transform the former markup into XML, which is bare of any XML tags but the surrounding root tag.

```
public class ExampleGrammarRichTextAdapter extends AbstractRichTextAdapter
{
    public ExampleGrammarRichTextAdapter(Markup markup) {
        super(markup);
    }

    @Override
    public <T> T asTree(Class<? extends T> type) {
        return null;
    }

    @Override
    public Set<Content> getReferencedContent() {
        return null;
    }

    @Override
    public String asString() {
        return null;
    }
}
```

Second step:

Create a factory class for your adapter and provide it as a bean in CaasConfig.

```
public class ExampleRichTextAdapterFactory
implements RichTextAdapterFactory<ExampleGrammarRichTextAdapter> {
    @Override
    public String getGrammar() {
        return "my-xml-grammar-1.0";
    }

    @Override
    public ExampleGrammarRichTextAdapter to(Markup markup) {
        return new ExampleGrammarRichTextAdapter(markup);
    }
}
```

```
@Bean
public ExampleRichTextAdapterFactory exampleGrammarRichTextFactory() {
    return new ExampleRichTextAdapterFactory();
}
```

Third step:

Implement a "To-Map" converter, which is responsible to convert your transformed XML tree representation into a common, Map based tree structure, which is easy to digest for graphql-java. Also provide it as a bean in CaasConfig.

```
public class ExampleRichTextToMapConverter
implements RichTextToMapConverter<ExampleGrammarRichTextAdapter> {
    @Override
    public Map<String, Object> convert(ExampleGrammarRichTextAdapter source)
    {
        return null;
    }
}
```

```
@Bean
public ExampleRichTextToMapConverter exampleRichTextToMapConverter() {
}
```

```
return new ExampleRichTextToMapConverter();  
}
```

5.2.3 CoreMedia Grammar RichTextAdapter

Headless Server delivers a ready to use implementation to parse and transform the CoreMedia rich text grammar 1.0, using the same techniques as described above. The implementation classes are

- CMGrammarRichTextAdapter
- CMGrammarRichTextAdapterFactory
- CMGrammarRichTextToMapConverter

Though the `CMGrammarRichTextAdapterFactory` defines the grammar, which can be parsed by its corresponding `RichTextAdapter`, `CMGrammarRichTextAdapter` is grammar agnostic. `CMGrammarRichTextAdapter` implements a stax based parsing technology and can be configured using YAML files (see [Section 5.1, “Rich Text Output” \[91\]](#) for details).

It is possible to reuse `CMGrammarRichTextAdapter` in a custom factory responsible for a custom grammar. This is an alternative of the first development step. If you consider reusing `CMGrammarRichTextAdapter`, keep in mind, that a grammar specific YAML configuration is necessary on top.

6. Search

The `headless-search` for the *Headless Server* encapsulates search related functionality like faceted and generic search, suggestions, dynamic query lists and their corresponding types.

It is part of the Blueprint Base module and contains a GraphQL schema extension within the file `search-schema.graphql`, Java code and Spring configuration.

If necessary, the `headless-search` can be deactivated by configuration properties. Note that it is possible, to deactivate the search schema extension explicitly, without deactivating the related code. This provides the possibility to add a customized version of the search schema. See the [Section 3.3.1, “Headless Server Spring Boot Properties”](#) in *Deployment Manual* for details.

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

6.1 Generic Search

Search related features are handled by the adapters `searchAdapter`, `facetedSearchAdapter` and `suggestionsAdapter`. 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
- Faceted search results
- Search suggestions

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 `SearchAdapter` 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 and invalid content types will cause a graphql error in the response. When passing an abstract content type, the subtypes are retrieved, if the parameter `includeSubTypes = true`. Passing an abstract content type with `includeSubTypes = false` will also cause a graphql error. The search result does not contain abstract content types, only the concrete sub types.

- `sortFields`: List of sort field with order, separated by '_', in upper case, for example, `ID_ASC`.

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

PLAYED_DATE. This enum can be extended in the schema by adding an available field with a sort order.

The available fields are defined in `SearchConstants#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

To configure custom fields, a specific bean can be configured, see section below.

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, for example, 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 `SearchAdapter` employs `DefaultSearchServiceProvider`, which in turn uses the `caeSolrQueryBuilder` Spring bean. `caeSolrQueryBuilder` invokes searches on Solr on the `cmdismax` endpoint. For details, see [Section 3.8.1, "Details of Language Processing Steps"](#) in *Search Manual*.

The used `SearchServiceProvider` is at the same time an `Extension Point`, which can be implemented and provided by a plugin. See [Section 4.17, "Plugin Support" \[80\]](#) 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 [Section 5.4.22, "Website Search"](#) in *Blueprint Developer Manual*.

- `searchDoctypeSelect`, `search.doctypeselect`: content 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.

Faceted Search Results

The *Headless Server Search* is able to do a faceted search on configured facets on the Solr search index. *Headless Server* comes with preconfigured facets, for example, on the content type. See [Section 5.4.22, "Website Search"](#) in *Blueprint Developer Manual* on how to configure facets on the search index.

In contrast to the regular search without facets, the `facetedSearch` query requires the parameter `siteId` mandatorily. To issue a faceted search request, the search query has to define the desired facets using the parameter `facetFilters`:

```
{
  content {
    facetedSearch(
      query: "*"
      siteId: "abffe57734feeee"
      facetLimit: 10
      facetFilters: [
        { facetAlias: "type", args: ["CMArticle"], excludeInFacet: false }
        { facetAlias: "subject", args: ["1234", "5678"] }
      ]
    ) {
      numFound
      facets {
        alias
        field
        values {
          query
          value
          hitCount
          facetContent {
            id
          }
        }
      }
    }
  }
}
```

```
result {  
  id  
  type  
  ... on CMArticle {  
    detailText {  
      text  
    }  
  }  
}
```

The facets can be found in the `facets` property of the search result. They provide information about the requested facets, the corresponding facet values and their count of content items where the facet occurred. If the facet value can be resolved to a content item, the content is provided as `Content_` in the property `facetContent`.

Use these parameters to issue a faceted search

- `query`: The search query.
- `offset`: The offset.
- `limit`: The limit of search result.
- `facetLimit`: Limits the size of facet values per facet. Defaults to studio config if set or 5 if not.
- `sortFields`: List of sort field with order, separated by '_', in upper case, for example, `ID_ASC`.

The set of available sort fields in the schema is limited to the enum `SortFieldWithOrder` defined in the content schema: `ID`, `DOCUMENTTYPE`, `TITLE`, `TEASER_TITLE`, `MODIFICATION_DATE`, `CREATION_DATE`, `EXTERNALLY_DISPLAYED_DATE`. This enum can be extended in the schema by adding an available field with a sort order.

The available fields are defined in `SearchConstants#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`

To configure custom fields, a specific bean can be configured, see section below.

Possible order field values: `ASC`, `DESC`

- `siteld`: The siteld. The siteld is mandatory in order to retrieve the configured facets per site.
- `facetFilters`: List of `FacetFilter` input objects with one or more configured facets. Optionally with facet values to be excluded from the faceted search

result. If no `FacetFilter` is given, all configured facets are calculated automatically.

The input type `FacetFilter` consists of these parameters:

- `facetAlias`: Mandatory name of a facet as configured.
- `filterValues`: Optional list of filter values for the given facet. The filter values are effectively a filter query on the configured field of the facet, e.g. `type` on the standard field `documenttype`.
- `excludeInFacet`: Defaults to `true`. If set false, the query clause with filter values is not excluded from facet calculation, resulting in a facet result with the given filter values only.
- `customFilterQueries`: Like the generic search, the facet search can also be extended by custom filter queries. See [Section 6.3, “Custom Filter Queries” \[122\]](#) for details.

NOTE

A faceted search query is a non trivial and complex query. Be aware, that additional queries using the custom filter queries, might affect the search result and the facet calculation in unexpected manners.



By default, the `FacetedSearchAdapter` employs `DefaultFacetedSearchServiceProvider`, which in turn uses the `caeSolrQueryBuilder` Spring bean. `caeSolrQueryBuilder` invokes searches on Solr on the `cm` `dismax` endpoint. For details, see [Section 3.8.1, “Details of Language Processing Steps”](#) in *Search Manual*.

The used `FacetedSearchServiceProvider` is at the same time an `ExtensionPoint`, which can be implemented and provided by a plugin. See [Section 4.17, “Plugin Support” \[80\]](#) for details.

Search Suggestions

Suggestions are a very popular feature for any search on a website. Suggestions are calculated simultaneously and then provided as an optional list to choose from, thus relieving the user from typing the full search expression.

The *Headless Server* is able to provide suggestions for search query expressions.

```
{
  content {
    suggest (
      query: "sal"
    ) {
      value
      count
    }
  }
}
```

```
}  
}
```

Use these parameters to issue a search suggestion query.

- `query`: The search query.
- `docTypes`: Content types to restrict the search result.

Misspelled content types and invalid content types will cause a graphql error in the response. When passing an abstract content type, the subtypes are retrieved, if the parameter `includeSubTypes = true`. Passing an abstract content type with `includeSubTypes = false` will also cause a graphql error. The search result does not contain abstract content types, only the concrete sub types.

- `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'.
- `customFilterQueries`: Like the generic search, the search suggestions can also be extended by custom filter queries. See [Section 6.3, "Custom Filter Queries" \[122\]](#) for details.

By default, the `SuggestionAdapter` employs `DefaultSuggestionSearchServiceProvider`, which in turn uses the `suggestionsSolrQueryBuilder` Spring bean. `suggestionsSolrQueryBuilder` invokes searches on Solr on the `suggest` endpoint. For details, see [Section 3.8.1, "Details of Language Processing Steps" in Search Manual](#).

The used `SuggestionSearchServiceProvider` is at the same time an `ExtensionPoint`, which can be implemented and provided by a plugin. See [Section 4.17, "Plugin Support" \[80\]](#) for details.

Configuration of custom SOLR fields

To configure a custom field of the SOLR index, a bean with qualifier `customSolrFields` can be added to the Spring context.

This bean of type `Map<String, String>` contains the custom field's name as a constant accessor and the field name in the SOLR index, e.g. `TITLE`, `title`.

This custom field can then be used, e.g. to apply a sort order.

The `customSolrFields` are applied to the `SolrQueryBuilder`.

The default SOLR fields are defined in the class `SearchConstants`, these are the default fields of the SOLR CAE index.

Generic configuration

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

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

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

Configuration of a custom index

If search should be performed on a custom SOLR index, the `SolrQueryBuilder` must be extended and configured. The following constructor arguments can be passed:

- `searchHandler`: the search handler, e.g. `/cmdismax`
- `filterQueryDefinitionMap`: a map containing filter query definitions to be used by custom filter queries
- `customFields`: custom fields of the SOLR index as map containing the field name and the SOLR field name, e.g. `TITLE, title`

6.2 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 CM-QueryList content.

```
{
  content {
    queryList(id: "7692") {
      title
      items {
        ... on CMLinkable {
          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 CMQueryList content.

```
{
  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.search.cache.querylist-search-cache-for-seconds`

6.3 Custom Filter Queries

Generic search and dynamic query lists can be extended with custom filter queries, that are applied to the `fq` parameter of the Solr query.

Custom filter queries must be predefined in as an implementation of `CustomFilterQuery` and provided as a Spring bean, before they can be used. As `CustomFilterQuery` is an extension point also, custom filter queries may be provided as part of a plugin or directly, e.g. by `CaasConfig`.

Definition of custom filter queries

The definition of a custom filter query consists of a query identifier and a function, that maps the field values to a Solr query.

- **Query Identifier:** a String value to identify the query. The graphql input type `FilterQueryArg` needs to be extended with the query identifier.
- **Mapping Function:** a function which takes a `List<String>` as argument and returns a String, that contains the Solr query in Solr syntax.

For example, a filter query definition could be defined with a query identifier `EXCLUDE_IDS` and a (here simplified) mapping function:

```
// Extension of input type FilterQueryArg in a graphql schema:
extend input FilterQueryArg {
  EXCLUDE_IDS: [String!]
}

// Bean factory in a configuration class
@Bean
public CustomFilterQuery excludeIdsQuery() {
  return new CustomFilterQuery() {

    /**
     * The query identifier.
     */
    @Override
    public String getName() {
      return "EXCLUDE_IDS";
    }

    /**
     * The mapping function.
     */
    @Override
    public String apply(List<String> values) {
      return SearchQueryHelper
        .negatedQuery(
          SearchQueryHelper
            .orQuery(SearchConstants.FIELDS.ID.toString(), values)
        );
    }
  }
}
```

```
};  
}
```

Example 6.1. Example implementation of a custom filter query.

In order to demonstrate the usage and possibilities, *Headless Server* comes with some out-of-the-box custom filter queries, namely:

- **TITLE_OR**: Query for one or more exact search expressions on the title field of the index.
- **EXCLUDE_IDS**: Exclude one or more content ids from the search result.
- **FRESHNESS**: Query for contents newer than the given date on the modification date field of the index.
- **LOC_TAXONOMY_OR**: Query for location taxonomy values.
- **SUBJ_TAXONOMY_OR**: Query for subject taxonomy values.

There are some help utilities in `SearchQueryHelper` to generate the Solr query in Solr syntax. Alternatively, the Solr query can also be given in direct Solr syntax.

Apply custom filter queries

A custom filter query can be applied statically for all queries or dynamically for each graphql query.

Static custom filter queries

Static filter queries, that shall be applied to all Solr queries, can be passed to the corresponding `*AdapterFactories`, e.g. `SearchServiceAdapterFactory` or `QueryListAdapterFactory`. They are then added to all Solr queries automatically.

Dynamic custom filter queries

Dynamic filter queries, that are applied to a specific GraphQL query, can be added as query argument for generic search, faceted search, suggestions or dynamic query lists. The input format is defined via the built-in type `FilterQueryArg`

All custom filter queries are applied as `fq` (filter query) fragments to the Solr query.

This GraphQL query is an example for fetching a search result using the pre-defined custom filter queries `EXCLUDE_IDS` and `TITLE_OR`.

```
{  
  {  
    content {  
      search(query: "*", docTypes: ["CMArticle"], customFilterQueries:
```

```
[{EXCLUDE_IDS: ["1234", "5678"]}, {TITLE_OR: ["Make your dream come true",  
"Eveningwear Trends"]}]}) {  
  numFound  
  result {  
    id  
    ... on CMArticle {  
      title  
    }  
  }  
}  
}  
}
```

This GraphQL query is an example for fetching query list items using the pre-defined custom filter queries EXCLUDE_IDS.

```
{  
  {  
    content {  
      queryList(id: "10") {  
        ... on CMQueryList {  
          id  
          filteredItems(customFilterQueries: {EXCLUDE_IDS: ["1234", "5678"]})  
        {  
          ... on CMLinkable {  
            id  
          }  
        }  
      }  
    }  
  }  
}
```

7. eCommerce Extension

All eCommerce functionality of the *Headless Server* is bundled within the Blueprint module `headless-server-ec-augmentation`. It contains GraphQL schema extension files, Java code and Spring configuration to implement this schema extension. The extension allows clients to issue GraphQL queries for augmentation data for categories, products, external commerce pages and product lists.

The GraphQL schema extension contains commerce specific types and support for product and category augmentations.

The schema extension uses the GraphQL extension mechanism to add a new field `commerce` of type `CommerceRoot` to the query root object. This API may use an underlying Commerce Hub connection to the commerce system. Some of the commerce related calls can also be found below `content` as long as they do not need an underlying Commerce Hub connection.

No Commerce Data

The eCommerce extension does not provide access to pure eCommerce related data like catalogs, categories and products. Instead the *Headless Server* provides augmentation data for categories, products, external commerce pages, product lists and navigation. Pure eCommerce data should be retrieved from the eCommerce system itself. In order to use the *Headless Server* in ecommerce projects with GraphQL, projects should use a schema gateway to combine both schemas (CoreMedia Headless Server and commerce system) to one combined graph. It is also possible to let a client talk to both backends in parallel, depending on the degree of integration needed.



7.1 Headless Commerce Integration Architecture

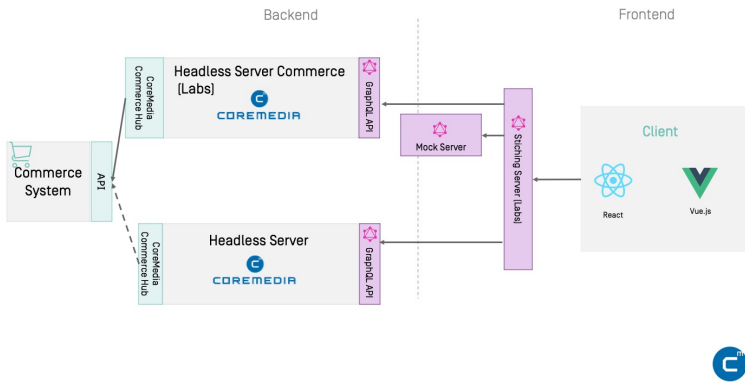


Figure 7.1. Headless Commerce Integration Example

The diagram shows an example architecture of a commerce integration with the CoreMedia Headless Server. In addition to the CoreMedia Headless Server, other CoreMedia labs components are used in the example setup. These labs components cannot be used in real projects without customization.

- *Client [Labs]*

Spark is a CoreMedia example application based on React, TypeScript and the Headless Server of CoreMedia Content Cloud. It uses the stitching server as single data endpoint for commerce and content data. The CoreMedia Spark example application is no official CoreMedia product, but is available as a CoreMedia labs project. See <https://github.com/CoreMedia/coremedia-headless-client-react>.

- *Stitching Server [Labs]*

The Stitching Server merges the GraphQL-Endpoints of the Headless Commerce Server and Headless Content Server and dispatches incoming GraphQL-Queries to the corresponding endpoints. The Stitching Server is no official CoreMedia product, but is part of the Spark Workspace and available as a CoreMedia labs project.

- *Mock Server [Labs]*

If there is no commerce system available for frontend development, the Mock Server can be used to provide commerce data to the client. The data can be recorded and replayed and it is stored in the file system. The Mock Server is no official CoreMedia product, but is part of the Spark Workspace and available as a CoreMedia labs project.

- *Headless Server Commerce [Labs]*

The Headless Commerce Server is an example GraphQL endpoint for the commerce data. The server establishes a UAPI connection to a Content Server and gRPC connections to configured CoreMedia Commerce Adapters (Commerce Hub). Headless Server Commerce is no official CoreMedia product, but is available as a CoreMedia labs project. A component that corresponds to the Headless Commerce Server component is obsolete if your commerce system offers a GraphQL endpoint on its own. See <https://github.com/Core-Media/coremedia-headless-commerce>

- *CoreMedia Headless Server*

The *Headless Server* serves as GraphQL endpoint for pure content data. It also provides access to content, which augments commerce products and categories. The server establishes a UAPI connection to a Content Server. Although it is not the endpoint for commerce data, the *Headless Server* still uses an underlying Commerce Hub connection to load hierarchical catalog information from a connected commerce system (see [Section 7.2, “Augmentation” \[128\]](#)).

- *Commerce System*

The commerce system provides access to commerce data. If the commerce system offers its own GraphQL API it should be used directly.

7.2 Augmentation

Categories, products and pages from the eCommerce system can be augmented with content from the CoreMedia CMS. This includes mapping media content such as pictures, videos and downloads to categories and products, as well as augmenting pages, categories and products with specific content objects (see [Section 6.2.3, “Adding CMS Content to Your Shop”](#) in *Studio User Manual*).

7.2.1 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 type `Augmentation` contains the fields `picture`, `pictures`, `video`, `videos`, `media` and `downloads`, 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:

```
{
  content {
    productAugmentationBySite(externalId: "PC_ORANGE_TEA", breadcrumb:
["PC_DELI", "PC_ToDrink"], siteId: "99c8ef576f385bc322564d5694df6fc2") {
      commerceRef {
        externalId
        siteId
        locale
      }
      pictures {
        name
        uriTemplate
        crops {
          name
          aspectRatio {
            width
            height
          }
          sizes {
            width
            height
          }
        }
      }
    }
  }
}
```

If any picture is associated with the given product in the CMS (by the aforementioned mapping in Studio), the returned URLs point to the corresponding picture.

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 product presentation, such as picture variants with responsive image URI templates (see [Chapter 11, Media Endpoint \[150\]](#)).

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

Any pictures or thumbnails defined on the commerce side should be retrieved from the commerce system endpoint.

7.2.2 Augmented Categories and Products

Categories and products can be augmented with content of type `CMExternalChannel` and `CMExternalProduct`, respectively. These content objects are created in 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, an augmenting content is created and the product/category is linked internally via the `externalId` field. If you query the augmentation for the product/category from the *Headless Server*, you receive a `ProductAugmentation` or `CategoryAugmentation` respectively. An `Augmentation` type provides access to page grid placements, linked assets or the augmenting content itself. Note that not every product/category is augmented and therefore the `content` field can be null.

In contrast to plain content related page grid placements, page grids for augmentations are inherited along the commerce navigation hierarchy. For example, a product variant cannot be augmented itself, instead it 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 commerce navigation hierarchy.

There are two ways to do augmentation queries:

- `CommerceRoot`
- `ContentRoot`

Query for augmenting content with CommerceRoot

To retrieve the hierarchy information of the category tree the *Headless Server* uses a connection to a commerce adapter under the hood. These augmentation queries can be found below the `CommerceRoot` (see [section Section 7.1, “Headless Commerce Integration Architecture” \[126\]](#)).

```
{
  commerce {
    productAugmentationBySite(externalId: "PC_BRITISH_TEA", siteId:
"99c8ef576f385bc322564d5694df6fc2") {
      pdpPagegrid {
        placements(names: ["header", "additional"]) {
          ...
        }
      }
    }
  }
}
```

Query for augmenting content with ContentRoot

In contrast, the *Headless Server* also offers augmentation queries below the `ContentRoot`. These queries do not rely on an underlying commerce connection, but need to receive `hierarchy` parameter from the client. In case the commerce connection is sometimes slow, it can also slow down the augmentation queries of the *Headless Server*.

```
{
  content {
    productAugmentationBySite(externalId: "PC_BRITISH_TEA", breadcrumb:
["PC_DELI", "PC_ToDrink"], siteId: "99c8ef576f385bc322564d5694df6fc2") {
      pdpPagegrid {
        placements(names: ["header", "additional"]) {
          ...
        }
      }
    }
  }
}
```

You might have noticed the difference between the call below `content` and `commerce`. The call below `content` needs an additional `breadcrumb` parameter, as this query cannot use an underlying Commerce Hub connection to automatically resolve the category hierarchy of the requested product. The breadcrumb information is used to search for augmented categories in the content repository.

An `Augmentation` type provides access to page grid placements of categories, products and product variants. For categories, the placements of the ordinary page grid are retrieved, while for products the Product Detail Page (PDP) and the corresponding `pdpPagegrid` is used. Product variants simply inherit all placements from their parent product.

NOTE

It is recommended to use the augmentation API below the `ContentRoot` because it is the future-proof solution with less calls and better decoupling.



The placements within a page grid can be retrieved in whole, including the complete 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:

```
{
  content {
    productAugmentationBySite(externalId: "PC_BRITISH_TEA", breadcrumb:
["PC_DELI", "PC_ToDrink"], siteId: "99c8ef576f385bc322564d5694df6fc2") {
      commerceRef {
        externalId
        siteId
        locale
      }
      content {
        repositoryPath
        ... on CMTeasable {
          title
          teaserText
        }
      }
    }
    pdpPagegrid {
      placements(names: ["header", "additional"]) {
        name
        items {
          name
          type
          ... on CMTeasable {
            teaserTitle
            teaserText
            picture {
              uriTemplate
            }
          }
        }
      }
    }
  }
}
```

In this example, you also query the `title` and `teaserText` fields of an associated `content`. Note that this `content` field is only non-null if this product is actually augmented. The same is true for the `content` in category augmentations – that field is only non-null if exactly this category is augmented, the field value is not inherited from the parent category.

7.2.3 Augmented Pages

Pages within the eCommerce system can be augmented with `CMExternalPage` content objects (see [Section 6.2.3.6, "Adding Content to Other Pages"](#) in *Studio User Manual*). The `commerce` root object offers a field `externalPage` which allows querying 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
          }
        }
      }
    }
  }
}
```

7.3 Product Lists

Product lists are handled with the `productListAdapter` (see [Section 6.2.2.2, "Adding a Product List"](#) in *Studio User Manual*). 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 CM-ProductList content.

```
{
  content {
    productList(id: "856") {
      items {
        ... on CMTeasable {
          teaserTitle
          teaserText
        }
        ... on ProductRef {
          externalId
        }
      }
    }
  }
}
```

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

The results of the query are automatically filtered for Valid from and valid to conditions.

Product List Cache Configuration

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

7.4 References to Products and Categories

The *Headless Server* does not provide access to purely commerce data directly. Instead the schema includes the types `CategoryRef` and `ProductRef`, which represent a link to a category or a product respectively. Links from CMS contents to commerce objects can be accessed via a `productRef` for content of type `CMExternalProduct` or `categoryRef` for content of type `CMExternalChannel`.

```
{
  content {
    content(id: "3240") {
      ...on CMExternalProduct {
        repositoryPath
        productRef {
          externalId
          locale
          storeId
        }
      }
    }
  }
}
```

The `CategoryRef` can be used to be resolved externally into a category, the `ProductRef` can be resolved into a product. This can be done via schema stitching or directly within a headless client application.

For example a product list query would look like this:

```
{
  content {
    content(id: "850") {
      ... on CMProductList {
        items {
          ... on CMTeasable {
            teaserTitle
          }
          ... on CommerceRef {
            externalId
            storeId
            locale
          }
        }
      }
    }
  }
}
```

And here the data retrieved:

```
{
  "data": {
    "content": {
      "content": {
        "items": [
          {
```

```
        "externalId": "AuroraWMDRS-1",
        "storeId": "1",
        "locale": "en-US"
    },
    {
        "externalId": "AuroraWMDRS-4",
        "storeId": "1",
        "locale": "en-US"
    },
    {
        "teaserTitle": "Find your personal style"
    },
    {
        "teaserTitle": "Editorial Blog"
    },
    {
        "externalId": "AuroraWMDRS-23",
        "storeId": "1",
        "locale": "en-US"
    },
    {
        "externalId": "AuroraWMDRS-24",
        "storeId": "1",
        "locale": "en-US"
    }
  ]
}
```

A `CommerceRef` includes the data needed to load the product itself from the commerce system again.

7.5 eCommerce Setup and Configuration

Although the *Headless Server* does not deliver catalog data, it still needs an underlying commerce connection to resolve page grids inherited along the commerce category hierarchy, extend the commerce navigation and provide dynamic product lists managed in Studio. Therefore 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

```
spring.grpc.client.channel.sfcc.target  
spring.grpc.client.channel.hybris.target  
spring.grpc.client.channel.commercetools.target  
spring.grpc.client.channel.wcs.target
```

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}`

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 bandwidth

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 allow list

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” \[138\]](#) describes how to set up the *Headless Server* to load persisted queries at startup time. This allows for the query allow list if the set of queries issued by clients is known in advance.
- [Section 8.2, “Query Allow Listing” \[141\]](#) describes the query allow list. That is, only queries loaded in the server during startup can be executed.
- [Section 8.3, “Apollo Automatic Persisted Queries” \[142\]](#) 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 an HTTP GET request with just this ID instead of the query string (mind URL encoding):

```
wget -q -O - 'http://myheadlessserver:41180/graphql \
?query=PersistedQueryMarker \
&extensions={"persistedQuery": {"version": 1,"queryId": "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":
"64cff55bcl8bfc2e6f8522aa4481bebee33eb7f1d9d9a3c8af12fc2e2aa2a9b",
      "document": "query PageQuery(...) {...}",
      ...
    },
    ...
  ]
}
```

This JSON file can then be used by your client and the *Headless Server* for query allow list (see [Section 8.2, “Query Allow Listing”](#) [141]).

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 allow list (see next section).

For the *Headless Server*, the JSON map 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, 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 Allow Listing

Registering queries in an allow list is a way to make the *Headless Server* more robust against potentially malicious (for example, expensive) queries. When allow-list is turned on, the *Headless Server* will execute only the queries loaded into the allow list of the server during startup. All other queries will be rejected with an error message in the JSON response.

The allow list in the *Headless Server* may be turned on by setting the configuration property `caas.persisted-queries.allow-list` to `true`.

Queries issued by clients do not need to match exactly those in the allow list. It suffices if their *normal form* is equal to the normal form of an allowed query. This is realized by means of the `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.

The allow list 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 allow-list 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 allow-list is turned on.

8.3 Apollo Automatic Persisted Queries

The allow list 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 Allow Listing” \[141\]](#)). 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 pre-defined set of queries would be far too restrictive, so potentially malicious queries must be detected by other means than a simple allow list.

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 an allow list) 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 `allow-list` is turned on. Then, uploading queries is still supported for queries with a normal form equal to the normal form of those in the allow list.

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.
- Easier to cache, using GET request.
- Security: Limit access to a well-defined list of queries and effectively prevent access to other contents.

The REST access to persisted queries is enabled by default. If REST access is not desired, the feature can be disabled by adding its autoconfiguration class to SpringBoots exclude list, e.g. in an environment variable.

```
SPRING_AUTOCONFIGURE_EXCLUDE=com.coremedia.caas.web.rest.RestMappingAutoConfiguration
```

The REST mapping of persisted queries allows for issuing REST requests instead of GraphQL queries. A list of REST endpoints can be configured which map the request to a corresponding [Chapter 8, Persisted Queries \[137\]](#). 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” [147] 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`

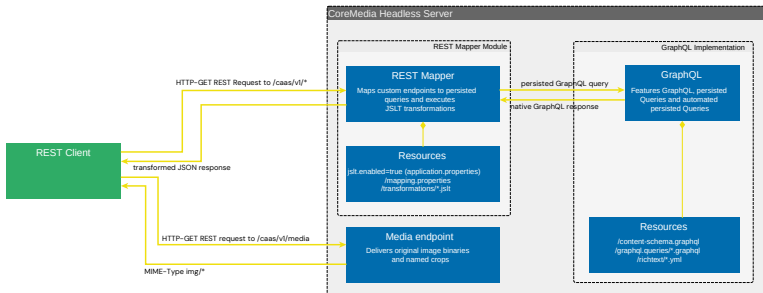


Figure 9.1. Headless server request/response flow using REST

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. The mapping file expects one mapping per line. The format of a mapping may be one of the following two. Both possibilities are equivalent.

```
# 1. query-id = uri-template
article = /article/{id}

# 2. query-id = JSON Object containing at least the key 'uriTemplate' with
the template as its value.
# e.g. article = {"uriTemplate": "/article/{id}"}
```

Additionally, the JSON object must contain type mappings for all query variables, which are not of type "string". Due to the fact that http parameters are strings by nature, the stricter validation of GraphQL query parameters requires a conversion of http parameters into the correct type. Supported conversion types are these basic scalars:

- boolean
- integer
- float

The mapping then may look similar to this:

```
# query-id = JSON Object containing type mapping additionally.  
search = {"uriTemplate": "/search", "limit": "integer", "offset": "integer",  
  "includeSubTypes": "boolean"}
```

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`.

The mapped URI fragment should not end with a slash. However, if it does, the trailing slash will be automatically removed upon system startup.

In addition to a plain URI fragment, it is allowed to add 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 [JSLT 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, for example, or an unintentional mixup of contents belonging to different sites, are prevented effectively.

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]
```

A complete listing of all existing site specific endpoints and its site ids can be acquired via the additional custom actuator endpoint at `/actuator/siteRestrictedEndpoints` or via the Swagger UI. The list via the Swagger UI only reflects the state at server start. As the list of site specific endpoints may change during runtime of the headless server, those changes are only available via the custom actuator endpoint.

The site filter access is enabled by default. If the site filter access is not desired, the feature can be disabled by adding its autoconfiguration class to SpringBoots exclude list, e.g. in an environment variable.

```
SPRING_AUTOCONFIGURE_EXCLUDE=com.coremedia.caas.web.view.impl.ViewAutoConfiguration
```

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 web server.



11. Media Endpoint

The media endpoint provides access to all media files (blobs), managed by the CMS. The endpoint supports image transformation in terms of precalculated crop sizes and supported image formats (see [Section 9.5.3, “Image Cropping and Image Transformation”](#) in *Studio Developer Manual* for details about crops). The URL to a managed media file is usually retrieved by means of a GraphQL query.

The following examples show, how you retrieve the URL of images and media files via a GraphQL query.

```
{
  content {
    picture(id: "1904") {
      id
      name
      uriTemplate
      crops {
        name
        sizes {
          width
        }
      }
    }
  }
}
```

Example 11.1. Retrieving the URI template of a picture

```
{
  content {
    picture(id: "1904") {
      id
      name
      uriTemplate(imageFormat: PNG)
      crops {
        name
        sizes {
          width
        }
      }
    }
  }
}
```

Example 11.2. Retrieving the URI template of a picture with an alternative image format

```
{
  content {
    picture(id: "1904") {
      id
      name
      data {
        uri
      }
      fullyQualifiedUrl
    }
  }
}
```

Example 11.3. Retrieving the URI or the fully qualified URL of the original file of a picture

It is important to note that the URLs accepted by the media endpoint (blob delivery) should always match the urls retrieved via GraphQL (link building). The link building is performed by Link Composers and Adapters and then woven into the GraphQL schema using the `fetch` directive to invoke the named bean `uriLinkComposer`. For details about Link Composers and Adapters please refer to the corresponding sections [???](#) and [Section 4.8, “Adapter” \[43\]](#).

11.1 Media Endpoint URLs

The media endpoint consists of the following distinct endpoints:

- Endpoint for images with crops and width.
- Endpoint for images with crops and width, format transformation and file name.
- Endpoint for generic media files.

Endpoint for images with crops and width

The first endpoint requests an image by means of the name of the crop and the desired width. The structure of the URI template is as follows:

```
/caas/v1/media/{mediaId}/{propertyName}/{hash}/{cropName}/{width}
```

The supported crop names and widths can be retrieved as part of the query for the `uriTemplate` (see [Chapter 11, Media Endpoint \[150\]](#)). The placeholders 'cropName' and 'width' must be replaced by a valid combination of the supported values. Trying to request an invalid 'cropName' or 'width' will result in an `HTTP 404 Not Found` error.

Supported combinations of crop names and widths

The names of the crops and the widths are defined in the content repository as part of the `Responsive Image Settings`. For more information read [Section 5.4.14, "Images" in *Blueprint Developer Manual*](#).



Endpoint for images with crops and width, format transformation and file name

The second endpoint to request images additionally supports on-the-fly image format transformation and the original file name. The structure of the URI template is as follows:

```
/caas/v1/media/{mediaId}/{propertyName}/{hash}/{cropName}/{width}/{filename}
```

The image format is specified by the file extension in the 'filename'. The format transformation is triggered by replacing the file extension with one of the supported image formats 'jpg', 'jpeg', 'png' or 'gif', or by directly requesting the respective `uriTemplate` like shown in the examples in [Chapter 11, Media End-](#)

point [150]. Requesting an unsupported format will result in an HTTP 400 Bad Request error.

Supported Formats of Cloud Installations

Self-Managed and new Cloud installations since CMCC 11 (2307) differ in terms of supported image formats, image sizes, and image editing capabilities. The features of image editing in a Cloud installation are described in <https://documentation.coremedia.com/coremedia-services/image-transformation/>.



Endpoint for generic media files

The third endpoint is the most generic. It provides access to any media file which is managed by the CMS. The structure of the URI template is as follows:

```
/caas/v1/media/{mediaId}/{propertyPath}/{hash}/{filename}
```

Note that the `filename` is optional.

You can comfortably explore the described endpoints using the Swagger UI provided with the overview page.

Content Disposition Header

Whenever a media file is requested with its correct filename (including the file extension), the HTTP header `Content-Disposition` will be set to `inline; filename=<the-original-file-name>`.

Placeholders of the media endpoints

mediaId	The content ID of the media/picture.
propertyName	The name of the property, where the blob is stored, usually <code>data</code> .
propertyPath	The name of the property where the blob is stored or the full property path, in case the blob ist stored in a struct.
hash	The hash of the blob. Usually queried via GraphQL.
cropName	The name of an existing crop of an image. Usually queried via GraphQL.

width	An existing width belonging to the crop name. Usually queried via GraphQL.
filename	The file name of a media file, including its file extension. Usually queried via GraphQL.

11.2 Configuration of Media Endpoints

The media endpoint offers configuration options for cache header control (properties `caas.media*`). Image transformation is controlled by the configuration options of the 'transform image service'.

See [Section 3.3, "Headless Server Properties"](#) in *Deployment Manual* and [Section 3.17, "Image Transformation Properties"](#) in *Deployment Manual* for details.

11.3 Customization of Media Endpoints

The media endpoints are implemented by means of an internal REST controller with the url patterns statically defined. These patterns can't be changed by configuration. Other aspects of the media endpoints however can be customized by implementing certain interfaces which are provided as Extension Points. The following interfaces are available for customization:

- `ContentResolver` - Allows to customize the resolution of content IDs to media content objects.
- `FilenameResolver` - Allows to customize the resolution of valid media file names.
- `BlobResolver` - Allows to customize the resolution of media blobs.
- `MediaUrlValidator` - Allows to customize the validation of media URLs.
- `MediaUrlValidationExceptionHandler` - Allows to customize the handling of exceptions thrown by a `MediaUrlValidator`.

Content Resolving and Media URL Validation

The resolvers are invoked in the order listed above. Default implementations of all resolvers are provided out of the box. The default implementations can be replaced by custom implementations by providing a bean of the respective type. After invocation of the resolvers, an instance of `ResolvedBlobInfo` is created, which contains the results of the resolvers.

After the invocation of the resolvers, validation of the media URL is performed by several implementations of `MediaUrlValidator`. The validation is performed by the provided default validators in the order listed below:

- `defaultPathValidator` - Validates the given property path of the media URL. If the property path is invalid, an HTTP 404 Not Found is returned.
- `defaultValidityDateValidator` - Validates the validity dates of the resolved media content. If the validity dates are defined and not valid, an HTTP 404 Not Found is returned. If the validity dates are not defined, the validation is skipped.

- `defaultFilenameValidator` - Validates the filename from the request against the resolved valid filenames. If the given filename does not match one of the resolved filename, an HTTP 404 Not Found is returned.
- `defaultCropNameValidator` - Validates the crop name and width from the request against the existing crop names, available in the resolved blob. If the given crop name or width does not match, an HTTP 404 Not Found is returned.
- `defaultBlobValidator` - Validates if a resolved blob exists. If a blob is not found, an HTTP 404 Not Found is returned. If the media type is `application/x.coremedia.encrypted`, an HTTP 403 Forbidden is returned.
- `defaultHashValidator` - Validates the hash from the request against the hash from the resolved blob. If the given hash does not match, an HTTP 404 Not Found is returned. If the configuration property `caas.strict-blob-hash-validation` is set to false (default is true), and the given hash does not match but the rest of the URL is valid, an HTTP 301 Moved Permanently is invoked.

If necessary, all validators can be disabled by a property beginning with the prefix 'caas.media-controller'. Please see [Section 3.3, "Headless Server Properties"](#) in *Deployment Manual* for details.

Exception Handling

If a validation fails, an `MediaUrlValidationException` is thrown. The exception, which bears a message, an HTTP-Status-Code and an optional redirect location, is handled by the `MediaUrlValidationExceptionHandler`. The default implementation of the exception handler will return an HTTP response with the status code and the message from the exception. If you want to customize the handling of validation errors, you can provide your own implementation of the `MediaUrlValidationExceptionHandler`. The custom implementation will be invoked instead of the default implementation.

For details about any of the described interfaces, please also refer to the Javadoc.

12. 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 adding its autoconfiguration class to SpringBoots exclude list, e.g. in an environment variable.

```
SPRING_AUTOCONFIGURE_EXCLUDE=com.coremedia.caas.web.metadata.MetadataAutoConfiguration
```

12.1 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 field to property name mapping is configured in file(s) at a configurable location (`classpath*:graphql/metadata/propertyMapping*.json`) as part of Blueprint with a configurable default filename (`propertyMapping.json`), see [Section 3.3.3, "Metadata Properties"](#) in *Deployment Manual* for details.

To add a new custom property mapping file definition, either change the location or the default filename and add the custom property mapping file definition accordingly.

To merge the default property mapping with a custom mapping, add a custom file to the default location and choose a name that matches the given pattern but is different from the default filename, for example, `propertyMapping-custom.json`. The default file is then loaded first, so that subsequent files can override the values.

The entries in the property mapping file consist of interface types that wrap the mapping of field name to the content type property name.

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 content type property name and for implied content properties.

The default mapping for fields is `"<fieldname>": "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 in `propertyMapping.json`.

The response of a metadata request containing PDE mapping looks like:

```
{
  "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 restriction is applied, because the PDE field mapping is not required for root types and custom object types. The mapping is also not available for `InterfaceTypeDefinitions`, for example `CMArticle`.

The `MetadataProvider` for PDE Mapping is configured for preview only, as PDE is only available in preview apps and typically used to preview data in Studio.

13. 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.

- [Section 13.1, “Getting Started”](#) [162]
- [Section 13.2, “Basic Guides”](#) [165]
- [Section 13.3, “Standalone Component”](#) [174]

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.



13.1 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.

13.1.1 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

13.1.2 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 preconfigured 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 to develop 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
```

13.1.3 Setup Apollo for GraphQL

The first step will be to basically configure the Apollo client and cache. The more in-depth setup will be done in the [Section 13.2.2, “Configuring Apollo Cache” \[166\]](#). 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 initialize 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 13.1. Example for Hello World App

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

13.1.4 Developer Tools

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

- [Apollo Client Devtools](#)
- [React Devtools](#)

13.2 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.

13.2.1 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 13.2. 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>
);
```

13.2.2 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 `CMCollection`. 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 automatically"](#).

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

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

Example 13.3. 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`.



13.2.3 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.

13.2.3.1 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 13.4. 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 13.5. Page Component render function

13.2.3.2 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 13.6. Iterating over all rows of the PageGrid

13.2.3.3 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 13.7. The PageGridPlacement Component

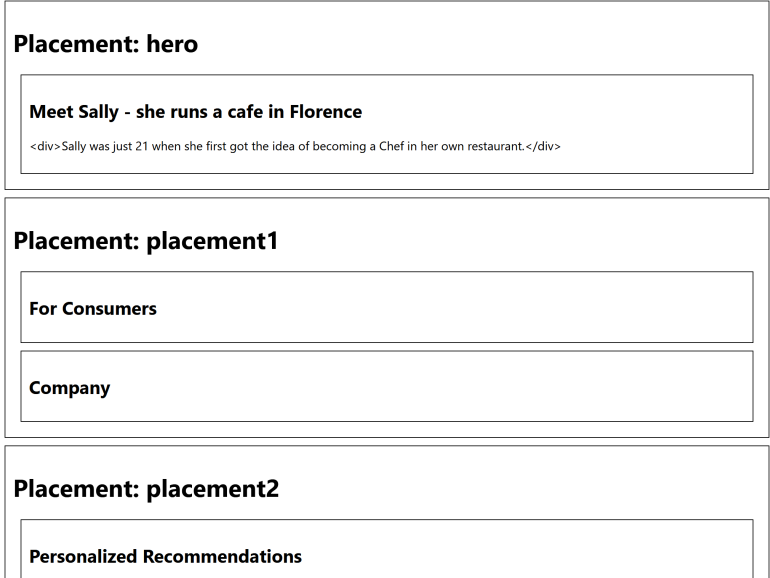


Figure 13.1. Screenshot of the example homepage

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.



13.2.4 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 13.8. 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>
);
```

```

    </ApolloProvider>
  );

```

Example 13.9. The App.jsx rendering with routing

The banner on the homepage need links to the article detail component. Therefore the `PageGridPlacement` should render `link` elements around each placement item and add its `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 13.10. The PageGridPlacement.jsx rendering links around article banner

13.2.5 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 13.11. 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.

13.2.5.1 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 13.12. Generating the full image URL

13.2.5.2 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 13.13. 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 13.2. 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.



13.3 Standalone 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 standalone 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.

13.3.1 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 13.14. 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 13.15. Fragment Integration of DOM element with custom data attribute

13.3.2 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`.

Additionally, it is also possible to exclude specific placements by passing an `excludeNames` argument. For example if you like to fetch all placements except "header" and "footer". Although both parameters can be used simultaneously, note that the `excludeNames` is applied independent of `names` and may remove some placements which are in the `names` list.

```
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 13.16. fetching the wanted placement

Afterwards, the items and the name of the placement are passed to the `PageGridPlacement` component of the app, and it handles the rendering from here. Since it is used in both, the standalone 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 13.17. rendering the PageGridPlacement

14. 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 3, CoreMedia Properties Overview](#) in *Deployment Manual*). The following links contain the properties that are relevant for the *Headless Server*:

- [Section 3.3.1, “Headless Server Spring Boot Properties”](#) in *Deployment Manual* contains properties for the general configuration of the *Headless Server*.
- [Section 3.3.2, “Persisted Query Properties”](#) in *Deployment Manual* contains properties for persisted queries.
- [Section 3.3.3, “Metadata Properties”](#) in *Deployment Manual* contains properties for the configuration of the metadata root of *Headless Server*.
- [Section 3.3.4, “Remote Service Adapter Properties”](#) in *Deployment Manual* contains properties for the configuration of the remote service of *Headless Server*.
- [Section 3.3.7, “Properties of External Frameworks”](#) in *Deployment Manual* contains properties for the configuration of GraphiQL.
- [Section 3.14.1, “CORS Properties”](#) in *Deployment Manual* contains properties for the configuration of Cross-Origin Resource Sharing (CORS) of the *Headless Server*.
- [Section 3.12, “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.
CaaS	Content as a Service or short caas, a synonym for the CoreMedia Headless Server.
CAE Feeder	Content applications often require search functionality not only for single content items but for content beans. The <i>CAE Feeder</i> makes content beans searchable by sending their data to the <i>Search Engine</i> , which adds it to the index.
Content Application Engine (CAE)	The <i>Content Application Engine</i> (CAE) is a framework for developing content applications with <i>CoreMedia CMS</i>. 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 <i>CoreMedia CMS</i> 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 <i>Content Delivery Environment</i> is the environment in which the content is delivered to the end-user. It may contain any of the following modules: • <i>CoreMedia Master Live Server</i> • <i>CoreMedia Replication Live Server</i> • <i>CoreMedia Content Application Engine</i> • <i>CoreMedia Search Engine</i> • <i>Elastic Social</i>

- *CoreMedia Native Personalization*

Content Feeder	The <i>Content Feeder</i> is a separate web application that feeds content items of the CoreMedia repository into the <i>CoreMedia Search Engine</i> . Editors can use the <i>Search Engine</i> to make a full text search for these fed items.
Content item	In <i>CoreMedia CMS</i> , 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 <i>Content Management Environment</i> is the environment for editors. The content is not visible to the end user. It may consist of the following modules: • <i>CoreMedia Content Management Server</i> • <i>CoreMedia Workflow Server</i> • <i>CoreMedia Studio</i> • <i>CoreMedia Search Engine</i> • <i>CoreMedia Native Personalization</i> • <i>CoreMedia Preview CAE</i>
Content Management Server	Server on which the content is edited. Edited content is published to the Master Live Server.
Content Repository	<i>CoreMedia CMS</i> 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	<i>Content Server</i> is the umbrella term for all servers that directly access the CoreMedia repository: <i>Content Servers</i> are web applications running in a servlet container. • <i>Content Management Server</i> • <i>Master Live Server</i> • <i>Replication Live Server</i>
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 <i>CoreMedia CMS</i> . They are written by CoreMedia developers – be it clients, partners or Core-

	Media employees. CoreMedia contributions are hosted on Github at https://github.com/coremedia-contributions .
Control Room	<i>Control Room</i> is a <i>Studio</i> plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other <i>Studio</i> users.
CORBA (Common Object Request Broker Architecture)	The term <i>CORBA</i> 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	<i>CoreMedia Studio</i> 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, <i>CoreMedia Studio</i> is based on the latest standards like Ajax and is therefore as easy to use as a normal desktop application.
Dead Link	A link, whose target does not exist.
Derived Site	A derived site is a site, which receives localizations from its master site. A derived site might itself take the role of a master site for other derived sites.
DTD	A Document Type Definition is a formal context-free grammar for describing the structure of XML entities. The particular DTD of a given Entity can be deduced by looking at the document prolog: <pre><!DOCTYPE coremedia SYSTEM "http://www.coremedia.com/dtd/coremedia.dtd"</pre> 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	<i>CoreMedia Elastic Social</i> is a component of <i>CoreMedia CMS</i> that lets users engage with your website. It supports features like comments, rating, likings on your website. <i>Elastic Social</i> is integrated into <i>CoreMedia Studio</i> so editors can moderate user generated content from their common

workplace. *Elastic Social* bases on NoSQL technology and offers nearly unlimited scalability.

EXML	EXML is an XML dialect used in former CoreMedia Studio version for the declarative development of complex Ext JS components. EXML is Jangaroo 2's equivalent to Apache Flex (formerly Adobe Flex) MXML and compiles down to ActionScript. Starting with release 1701 / Jangaroo 4, standard MXML syntax is used instead of EXML.
Folder	A folder is a resource in the CoreMedia system which can contain other resources. Conceptually, a folder corresponds to a directory in a file system.
FTL	FTL (FreeMarker Template Language) is a Java-based template technology for generating dynamic HTML pages.
gRPC	gRPC is an open source high performance Remote Procedure Call (RPC) framework.
Headless Server	CoreMedia Headless Server is a CoreMedia component introduced with CoreMedia Content Cloud which allows access to CoreMedia content as JSON through a GraphQL endpoint. The generic API allows customers to use CoreMedia CMS for headless use cases, for example delivery of pure content to Native Mobile Applications, Smartwatches/Wearable Devices, Out-of-Home or In-Store Displays or Internet-of-Things use cases.
Home Page	The main entry point for all visitors of a site. Technically it is often referred to as root document and also serves as provider of the default layout for all subpages.
IETF BCP 47	Document series of <i>Best current practice</i> (BCP) defined by the Internet Engineering Task Force (IETF). It includes the definition of IETF language tags, which are an abbreviated language code such as en for English, pt-BR for Brazilian Portuguese, or nan-Hant-TW for Min Nan Chinese as spoken in Taiwan using traditional Han characters.
IOR (Interoperable Object Reference)	A CORBA term, <i>Interoperable Object Reference</i> refers to the name with which a CORBA object can be referenced.
Jangaroo	<i>Jangaroo</i> is a JavaScript framework developed by CoreMedia that supports TypeScript (formerly MXML/ActionScript) as an input language which is compiled down to JavaScript compatible with Ext JS. You will find detailed descriptions on the Jangaroo webpage http://www.jangaroo.net . Jangaroo 4 is the Action-

		Script/MXML/Maven based version for CMCC 10. Since CMCC 11 (2110), Jangaroo uses TypeScript and is implemented as a <i>Node.js</i> and <i>npm</i> based set of tools.
Java Management Extensions (JMX)		The Java Management Extensions is an API for managing and monitoring applications and services in a Java environment. It is a standard, developed through the Java Community Process as JSR-3. Parts of the specification are already integrated with Java 5. JMX provides a tiered architecture with the instrumentation level, the agent level and the manager level. On the instrumentation level, MBeans are used as managed resources.
Locale		Locale is a combination of country and language. Thus, it refers to translation as well as to localization. Locales used in translation processes are typically represented as IETF BCP 47 language tags.
Master Live Server		The <i>Master Live Server</i> is the heart of the <i>Content Delivery Environment</i> . It receives the published content from the <i>Content Management Server</i> and makes it available to the CAE. If you are using the <i>CoreMedia Multi-Master Management Extension</i> you may use multiple <i>Master Live Server</i> in a CoreMedia system.
Master Site		A master site is a site other localized sites are derived from. A localized site might itself take the role of a master site for other derived sites.
MIME		With Multipurpose Internet Mail Extensions (MIME), the format of multi-part, multimedia emails and of web documents is standardised.
MXML		MXML is an XML dialect used by Apache Flex (formerly Adobe Flex) for the declarative specification of UI components and other objects. Up to CMCC 10 (2107), CoreMedia Studio used the Open Source compiler Jangaroo 4 to translate MXML and ActionScript sources to JavaScript that is compatible with Ext JS 7. Starting with CMCC 11 (2110), a new, Node.js and npm based version of Jangaroo is used that supports standard TypeScript syntax instead of MXML/ActionScript, still compiling to Ext JS 7 JavaScript.
OCI (Open Container Initiative)		The Open Container Initiative (OCI) is a Linux Foundation project that defines open industry standards for container formats and runtimes. OCI specifications ensure compatibility and interoperability between container tools, engines, and orchestration platforms like Docker and Kubernetes.

ORAS (OCI Registry As Storage)	ORAS (OCI Registry As Storage) is a tool and specification that extends OCI registries to store and distribute OCI artifacts beyond container images. It provides a standardized way for developers to push and pull arbitrary content types to and from container registries, enabling these registries to function as general artifact stores.
Personalisation	On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.
Projects	With projects you can group content and manage and edit it collaboratively, setting due dates and defining to-dos. Projects are created in the Control Room and managed in project tabs.
Property	In relation to CoreMedia, properties have two different meanings: In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content item depends on the content type. In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.
Replication Live Server	The aim of the <i>Replication Live Server</i> is to distribute load on different servers and to improve the robustness of the <i>Content Delivery Environment</i> . The <i>Replication Live Server</i> is a complete Content Server installation. Its content is an replicated image of the content of a <i>Master Live Server</i> . The <i>Replication Live Server</i> updates its database due to change events from the <i>Master Live Server</i> . You can connect an arbitrary number of <i>Replication Live Servers</i> to the <i>Master Live Server</i> .
Resource	A folder or a content item in the CoreMedia system.
ResourceURI	A <i>ResourceUri</i> uniquely identifies a page which has been or will be created by the <i>Active Delivery Server</i> . The <i>ResourceUri</i> 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 <i>CoreMedia CMS</i> 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 <code>CMSite</code>.
Site Manager Group	Members of a site manager group are typically responsible for one localized site. Responsible means that they take care of the contents of that site and that they accept translation tasks for that site.
Template	In CoreMedia, FreeMarker templates used for displaying content are known as Templates. OR In <i>Blueprint</i> 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, pre-defined layout and even predefined content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes Application	The <i>User Changes Application</i> is a <i>Content Repository</i> listener, which collects all content, modified by <i>Studio</i> users. This content can then be managed in the <i>Control Room</i>, as a part of projects and workflows.
Variants	The set of all content items in a multi-site hierarchy related to each other via master references. This includes the top-level master content items themselves.
Version history	A newly created content item receives the version number 1. New versions are created when the content item is checked in; these are numbered in chronological order.
Weak Links	In general <i>CoreMedia CMS</i> always guarantees link consistency. But links can be declared with the <i>weak</i> attribute, so that they are not checked during publication or withdrawal.

Caution! Weak links may cause dead links in the live environment.

Workflow

A workflow is the defined series of tasks within an organization to produce a final outcome. Sophisticated applications allow you to define different workflows for different types of jobs. So, for example, in a publishing setting, a document might be automatically routed from writer to editor to proofreader to production. At each stage in the workflow, one individual or group is responsible for a specific task. Once the task is complete, the workflow software ensures that the individuals responsible for the next task are notified and receive the data they need to execute their stage of the process.

Workflow Server

The *CoreMedia Workflow Server* is part of the Content Management Environment. It comes with predefined workflows for publication but also executes freely definable workflows.

XLIFF

XLIFF is an XML-based format, standardized by OASIS for the exchange of localizable data. An XLIFF file contains not only the text to be translated but also metadata about the text. For example, the source and target language. *CoreMedia Studio* allows you to export content items in the XLIFF format and to import the files again after translation.

Index

Symbols

!Context, 96
 !ElementFromClass, 103
 !ElementWriter, 99
 !EmptyElementWriter, 100
 !Handler, 97
 !ImgWriter, 100
 !LinkWriter, 101
 !Matcher, 97
 !PassAttribute, 104
 !PassStyles, 104
 !Push, 98
 !ReplacePush, 98
 !RootContext, 96
 @fetch directive, 37
 @inherit directive, 39

, 134

A

adapter, 43
 apollo federation support, 15, 89
 article query, 49
 Attribute Transformers, 103

B

beans for plugins, 86

C

cache, 19
 cache control, 20
 changes in manual, 12
 classes property, 95
 content root, 30
 content schema, 30, 49
 content type
 custom, 64

context handler, 95
 context handlers, 98
 contexts property, 95
 ConversionService, 42
 Converter, 42
 custom content type, 64
 custom filter query, 84
 Custom Output Handler, 102
 custom preview client, 22
 custom scalar type, 82

D

default view, 91
 derived sites, 53
 download query, 57
 dynamic query lists, 120

E

eCommerce augmentation, 128
 eCommerce configuration, 136
 eCommerce schema, 125
 element transformer, 103
 elements property, 94
 endpoints, 17
 Event Matcher, 97
 execution timeout, 26
 extension points, 81, 156
 external link query, 57
 external links, 107

F

filter predicate, 41, 82
 filter query, 122
 filter style classes, 103

G

GraphiQL, 17
 GraphQL, 17, 30
 GraphQL type name, 64

H

handlerSets property, 105
 Headless Server
 properties, 176

I

- include directive, 93
- initialContext property, 95
- internal links, 105

J

- JSLT transformation, 143, 147
- JSON preview, 17
- Json preview, 21

L

- links, 47
- localized variants, 58

M

- Media, 18
- Media Endpoint, 150
- mediatype content negotiation, 27
- metadata schema, 158
- model mapper, 40

N

- name property (transformer), 94

O

- Output Handlers, 99

P

- page query, 54
- pagination, 67
- Persisted Queries, 137
- personalized variants, 58
- plainFirstParagraph view, 91
- plugin faceted search service provider, 84
- plugin graphql schema generator, 83
- plugin linkcomposer, 83
- plugin schema adapter factory, 82
- plugin search service provider, 84
- plugin suggestion search service provider, 85
- plugin support, 15, 80
- plugin wiring factory, 83
- preview, 21
- Product lists, 133
- ProvidesTypeNameResolver, 64

Q

- Query Allow List, 25
- query complexity, 26
- query depth, 26
- query root, 30

R

- remote links, 70
- resource file loading, 87
- REST, 17, 143
- REST endpoints, 144
- REST Mapping, 145
- Rich Text, 91
- Rich Text Transformer, 91
- rich text views, 91
- RichText, 108
- RichTextAdapter, 108

S

- search, 112–113
- search configuration, 119
- search parameters, 113
- search result limit, 25
- Security, 24
- simplified view, 91
- Site Filter, 18, 148
- site query, 52
- sites query, 51
- Swagger UI, 17

T

- taxonomy, 73
- time travel, 66
- transformation mapping, 103
- transformation rules, 103
- TypeNameResolver, 64

U

- Unified API cache, 19
- URI template, 146

V

- viewtype, 78

Y

YAML Alias, 93

YAML Anchor, 93

YAML Comment, 94

YAML configuration, 92