CoreMedia Content Cloud

Connector for SAP Commerce Cloud Manual

CoreMedia

CoreMedia GmbH

Altes Klöpperhaus, 5. OG

Rödingsmarkt 9

20459 Hamburg

March 14, 2024 (Release 2310)

List of Figures

List of Tables

List of Examples

1. Preface

This manual describes how the CoreMedia system integrates with *SAP Hybris*.

- Chapter 2, *Overview* [6] gives a short overview of the integration.
- Chapter 3, *Customizing SAP Hybris Commerce* [12] describes how you have to configure the commerce system to work with *CoreMedia Content Cloud*.
- Chapter 5, *Commerce-led Integration Scenario* [33] describes the commerce-led scenario and shows how you extend commerce pages with CMS fragments.
- Chapter 4, *Connecting with an SAP Hybris Commerce System* [23] describes how you connect a CoreMedia web application with a *Hybris Commerce* system.
- Section 5.7, "Link Building for Fragments" [64] describes deep links from fragments of the CMS system to pages of the *Hybris* system.
- Section 6.2, "Enabling Preview in Shop Context" [72] describes how you activate the preview of *Hybris Commerce* pages in *Studio*.
- Chapter 6, *Studio Integration of Commerce Content* [67] shows the eCommerce features integrated into *CoreMedia Studio*.
- Chapter 7, *Commerce Caching* [88] describes the CoreMedia cache for eCommerce entities.
- Chapter 8, *The eCommerce API* [96] describes the basics of the eCommerce API.

1.1 Audience

This manual is intended for architects and developers who want to connect *CoreMedia Content Cloud* with an eCommerce system and who want to learn about the concepts of the product. The reader should be familiar with *CoreMedia CMS*, , *SAP Hybris Commerce, Spring*, *Maven* , *Chef* and *Docker*.

1.2 Typographic Conventions

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

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

Table 1.1. Typographic conventions

In addition, these symbols can mark single paragraphs:

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

Table 1.2. Pictographs

1.3 Change Record

This section includes a table with all major changes that have been made after the initial publication of this manual.

Section	Version	Description

Table 1.3. Changes

2. Overview

This manual describes how the CoreMedia system integrates with *SAP Hybris Commerce*. You will learn how to add fragments from the CoreMedia system into a *Hybris* generated site, how to access the *Hybris* catalog from the CoreMedia system and how to develop with the *eCommerce API*. The configuration of your *Hybris* system is described in Chapter 3, *Customizing SAP Hybris Commerce* [12]

Integration scenarios

CoreMedia Content Cloud offers the commerce-led integration scenario with *SAP Hybris Commerce* [see Chapter 5, *Commerce-led Integration Scenario* [33]]. In the commerce-led scenario, pages are delivered by the *SAP Hybris Commerce* system. The page navigation is determined by the catalog category structure and cannot be changed in the CMS. You can augment the categories and product detail pages with content from the CMS. Content and settings are also inherited along the catalog category structure.

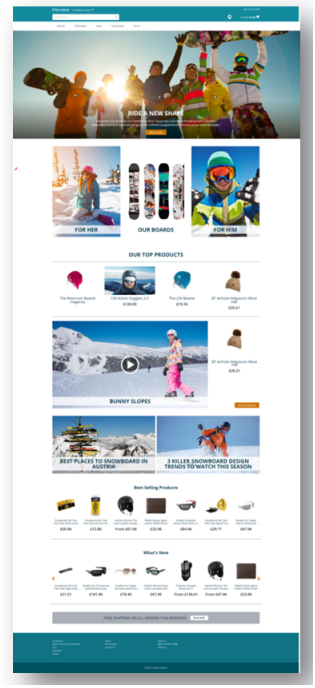

Figure 2.1. Hybris Homepage enriched with CMS Content

2.1 Commerce Hub Architecture

Commerce Hub is the name for the CoreMedia concept which allows integrating different eCommerce systems against a stable API.

Figure 2.2, " Architectural overview of the Commerce Hub " [8] gives a rough overview of the architecture.

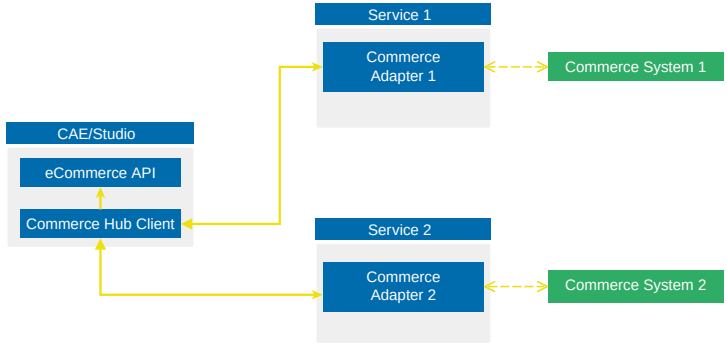

Figure 2.2. Architectural overview of the Commerce Hub

All CoreMedia components (CAE, Studio) that need access to the commerce system include a generic Commerce Hub Client. The client implements the CoreMedia eCommerce API. Therefore, you have a single, manufacturer independent API on CoreMedia side, for access to the commerce system.

The commerce system specific part exists in a service with the commerce system specific connector. The connector uses the API of the commerce system (often REST) to get the commerce data. In contrast, the generic Commerce Hub client and the Commerce Connector use gRPC for communication (see https://grpc.io/) for details.

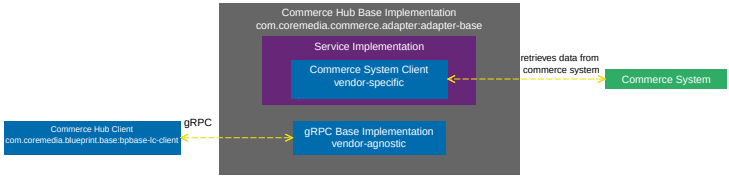

Figure 2.3. More detailed architecture view

Figure 2.3, “ More detailed architecture view ” [8] shows the architecture in more detail. At the Commerce Hub Client, you only have to configure the URL of the service and some other options, while at the Commerce System Client, you have to configure the commerce system endpoints, cache sizes and some more features.

2.2 Commerce Hub API

The *Commerce Hub* API consists of a gRPC API used by the *generic client*, and a Java API which consists of the Entities API as a wrapper around the gRPC messages, and a Java Feature API, used by the specific *adapter services*.

The gRPC API

The gRPC API defines the messages and services used for the gRPC communication between *generic client* and *adapter service*. It is not necessary to access this API from any custom code. Access should be encapsulated, using the provided Java APIs, described below. In case the existing feature set does not fulfill all needs for a custom commerce integration, the gRPC API may be extended. CoreMedia provides two sample modules, showing a gRPC API extension in the *Commerce Adapter Mock*. Please have a look at the Section 3.2, "CoreMedia Commerce Adapter Mock" in *Custom Commerce Adapter Developer Manual*.

> NOTE
>
> By Default the *base adapter* exposes the gRPC `ServerReflection` service. It is used by the *CoreMedia Commerce Hub Client* to obtain available features.

The Java API

The Java API consists of two parts. The first part defines Java Entities as a wrapper around gRPC. It is used by the *generic client* and the server in the *base adapter*.

The second part is meant for server side only. It defines the Java Interfaces, called Repositories, the *adapter services* may implement for any needed feature. This API should be used as an entry point for commerce adapter development.

Request flow

The request flow, using the above described APIs, starting from the generic client is as follows. Please have a look at Figure 2.3, " More detailed architecture view " [8] first.

1. The generic client sends a gRPC request to the vendor agnostic *base adapter*. The Entities API is used to convert the Java entity to the corresponding gRPC message.
2. The gRPC service implementation in the *base adapter* receives the gRPC request and invokes the corresponding repository methods.

While the API definition of the repositories is placed in the *base adapter*, the implementation which is called here is part of a specific commerce adapter.

The commerce adapter uses its vendor specific implementation to obtain the requested data from the commerce system. The data is then mapped to a CoreMedia commerce entity as defined by the base adapter.

Finally, the service implementation in the *base adapter* converts the given entity back to a gRPC response and sends it back to the *generic client*.

3. The *generic client* receives the gRPC response and uses the Entities API to obtain and process the requested entity.

3. Customizing SAP Hybris Commerce

> NOTE
>
> Only required when you want to use the eCommerce Blueprint for Hybris

This section describes how you have to adapt your Hybris project workspace in order to integrate with *CoreMedia Content Cloud*.

In general, certain configuration files need to be adapted for your *Hybris Project Workspace*.

> NOTE
>
> Deployment topics are not part of this manual. Please refer to appropriate Hybris documentation at https://help.hybris.com

Scope of delivery

In order to connect *CoreMedia Content Cloud* with your *SAP Hybris Commerce* system CoreMedia provides the *Workspace for SAP Commerce Cloud* archive (*Hybris workspace archive*, for short).

You will find the *Workspace for SAP Commerce Cloud* on the CoreMedia releases download page at http://releases.coremedia.com/cmcc-11 in the Commerce Integration section.

The customization involves the following aspects: *Installation steps*

- Section 3.1, “Adding the CoreMedia Extensions to your *Hybris Project Workspace*” [14] describes how to add the required CoreMedia Extensions to your *Hybris Project Workspace*
- Section 3.2, “Apply global JSPs” [16] describes how to apply customizations to *SAP Hybris Commerce* JSPs outside the CoreMedia Extensions.

- Section 3.3, "Configuring the CoreMedia Fragment Connector" [17] describes configuration of the fragment connector, which renders content from *CoreMedia Content Cloud* as fragments to *SAP Hybris Commerce* pages.
- Section 3.4, "Load Essential Data and Demo Data" [21] describes how to initialize essential data and demo data. This also implies the *CoreMedia Content Widget*, which is used to add content or assets from *CoreMedia Content Cloud* to *SAP Hybris Commerce* pages using the fragment connector.

NOTE

In the following sections `$HYBRIS_HOME` stands for the Hybris installation directory of your *SAP Hybris Commerce* installation and `$HYBRIS_WORKSPACE` stands for the path of your *Hybris Project Workspace*.

3.1 Adding the CoreMedia Extensions to your *Hybris Project Workspace*

CoreMedia Content Cloud comes with the *Hybris workspace archive* Zip file, which has to be applied to your *Hybris Project Workspace*. The *Hybris workspace archive* includes the following sub folders:

- The folder `cmlivecontext` contains an yacceleratorstorefront add-on. It provides the *CoreMedia Content Widget*, the *CoreMedia Fragment Connector* and essential and demo data.
- The folder `cmoccaddon` contains an yacceleratorstorefront add-on. It provides additional custom OCC controllers to read product and category data, optimized for use with CoreMedia Commerce Hub. The addon will be used for SAP Hybris 1905.
- The folder `cmocc` contains a commercewebservices extension. It replaces the previously necessary cmoccaddon and complies with the new SAP Hybris standard since version 2005. It provides additional custom OCC controllers to read product and category data, optimized for use with CoreMedia Commerce Hub.
- The folder `versions` contains for each supported *Hybris* version a dedicated folder `$YOUR_VERSION/custom` which contains configuration and JSP tags, which need to be applied to other extensions of the *Hybris Project Workspace*. Use the `$YOUR_VERSION` folder corresponding to the version you are using. In the following the folder will be referred to as the "global files folder". If your concrete minor version is not included, there is a chance to adapt the affected files for yourself. Start with the vanilla version of each file and find the right place to add the CoreMedia extensions. They are all marked as "CoreMedia extensions" in the included examples.

Steps for your SAP Hybris 2105 workspace:

1. Copy the `cmlivecontext` and `cmocc` folders to your *Hybris Project Workspace* below `$HYBRIS_HOME/bin/custom`
2. Take the file `versions/YOUR_VERSION/custom/hybris/configlocalextensions.xml` from the *Workspace for SAP Commerce Cloud* and copy all the extensions marked with a comment of the format `<!-- CoreMedia required extensions-->` into your Hybris `localextensions.xml` file.
3. Register the add-ons provided by CoreMedia and rebuild the workspace with the following commands:

```
# cd $HYBRIS_HOME/bin/platform
# . ./setantenv.sh
```

```
        # ant addoninstall -Daddonnames="cmlivecontext"
-DaddonStorefront.yacceleratorstorefront="yacceleratorstorefront"
        # ant clean all
```

3.2 Apply global JSPs

The global files folder of the *Hybris workspace archive* contains JSPs, which are not part of the CoreMedia extensions. They need to be applied to the yacceleratorstorefront. The folder layout underneath the global files folder reflects the layout of the *Hybris Project Workspace*.

For example you can find a customized version of the `master.tag` in the *Workspace for SAP Commerce Cloud* below `$HYBRIS_WORKSPACE/versions/YOUR_VERSION/custom/hybris/bin/modules/base-accelerator/yacceleratorstorefront/web/webroot/WEB-INF/tags/responsive/template/master.tag` whereas the path to the original `master.tag` in your *Hybris Project Workspace* is `$HYBRIS_HOME/bin/modules/base-accelerator/yacceleratorstorefront/web/webroot/WEB-INF/tags/responsive/template/master.tag`.

NOTE

In principle, you can copy the contained content on top of your *Hybris Project Workspace*, but CoreMedia recommends merging the changes manually with the original files. If you have done customizations before, you have to merge manually.

The customized CoreMedia JSP files reflect the CoreMedia default setup. If your own setup is different, you have to adapt the slots to your needs. For example, add additional slots at other locations as it is done in the examples.

3.3 Configuring the CoreMedia Fragment Connector

The *CoreMedia Fragment Connector* is the central component in the commerce-led integration scenario (see Chapter 5, *Commerce-led Integration Scenario* [33]).

The *CoreMedia Fragment Connector* is the component that connects with CoreMedia *CAE* to load CoreMedia content fragments into store pages. Configure the connector in the configuration file `$HYBRIS_HOME/bin/custom/cmlivecontext/project.properties`, as described below:

Configure at least the parameter *`com.coremedia.fragmentConnector.liveCaeHost`* with the host URL of your *Content Application Engine*. If you use a single *SAP Hybris Commerce* Server that should be able to connect to both, preview and production CAE, you also need to set *`com.coremedia.fragmentConnector.previewCaeHost`* with the host URL of the preview CAE. In case you have a dedicated staging server with separate production system, you only need to configure one CAE host, for each.

Find the meaning of all parameters in the configuration file in Table 3.1, "CoreMedia Connector Properties" [17].

`com.coremedia.fragmentConnector.liveCaeHost`	
Description	The `liveCaeHost` identifies the Live CAE, to be precise, the Varnish, Apache or any other proxy in front of the Live CAE. Each request made by the fragment connector will be prefixed with the `urlPrefix`.
Default	http://preview-apparel.192.168.252.100.xip.io/

`com.coremedia.fragmentConnector.previewCaeHost`	
Description	The `previewCaeHost` identifies the Preview CAE, to be precise, the Varnish, Apache or any other proxy in front of the Preview CAE. Each request made by the fragment connector will be prefixed with the `urlPrefix`. The `previewCaeHost` is only required if you want a single Commerce instance being able to access the preview CAE in case of Commerce preview against the stage catalog and the live CAE in all other cases. Additionally, the preview mode can be invoked through an HTTP header. If you have a dedicated Commerce instances for staging and separate production Commerce systems, you do not need to set this property. If this parameter is not set, the parameter `liveCaeHost` will be used instead.
Default	http://preview-apparel.192.168.252.100.xip.io/

`com.coremedia.fragmentConnector.urlPrefix`	
Description	This prefix identifies the web application, the servlet context and the fragment handler to handle fragment requests. The default request mapping of all the handlers within *CoreMedia Blueprint* that are able to handle fragment requests start with `service/fragment`.
Default	service/fragment

`com.coremedia.widget.templates`	
Description	Configures the template lookup path that is used when rendering CoreMedia Widget includes.
Default	/WEB-INF/views/addons/cmlivecontext/responsive/cms/templates/

`com.coremedia.fragmentConnector.defaultLocale`	
Description	Every fragment request needs to contain the tuple `(storeId, locale)` because it is needed to map a request to the correct site. Using `defaultLocale` you can set a default that is used for every request that does not contain a custom locale. You will see how it is used later, when you see the `IncludeTag` in action.
Default	en-US

`com.coremedia.fragmentConnector.contextProvidersCSV`	
Description	Every fragment request can be enriched with shop context specific data. It will be most likely user session related info, that is available in the commerce system and can be provided to the backend CAE via a `ContextProvider` implementation. See Section 5.3, “Extending the Shop Context” [48] for details.
Default	com.coremedia.livecontext.hybris.addon.contextproviders.UserContextProvider,com.coremedia.livecontext.hybris.addon.contextproviders.PreviewContextProvider

`com.coremedia.fragmentConnector.isDevelopment`	
Description	The fragment connector will return error messages that occur in the CAE while rendering a fragment if the `isDevelopment` parameter is set to true. For production environments you should set this option to `false`. Errors are logged than but do not appear on the commerce page so that the end user will not recognize the errors.

Default	true

`com.coremedia.fragmentConnector.disabled`

Description	Turn this flag to true if you want to disable the fragment connector. Disabled means that the fragment connector always delivers an empty fragment. This property is not mandatory. If this property is not set, the default is false.
Default	false

`com.coremedia.fragmentConnector.connectionTimeout`

Description	The connection timeout in milliseconds used by the fragment connector; that is the time to establish a connection. A value of "0" means "infinite".
Default	10000

`com.coremedia.fragmentConnector.socketTimeout`

Description	The socket read timeout in milliseconds used by the fragment connector; that is the time to wait for a response after a connection has successfully been established. A value of "0" means "infinite".
Default	30000

`com.coremedia.fragmentConnector.connectionPoolSize`

Description	Maximum number of connections used by the fragment connector.
Default	200

`com.coremedia.fragmentConnector.previewCaeAccessTokenHeader`

Description	An optional access token that is sent along with all HTTP requests towards the CoreMedia preview CAE. Can be used by the CAE to authorize the access.
Default	

`com.coremedia.fragmentConnector.liveCaeAccessTokenHeader`

Description	An optional access token that is sent along with all HTTP requests towards the CoreMedia live CAE. Can be used by the CAE to authorize the access.

Default	
`com.coremedia.fragmentConnector.parameterIncludeList`	
Description	Comma separated list of parameter names. If set, these parameters will be copied from the shop request to the CAE fragment request. All other parameter will be ignored. If set, this list has precedence over `com.coremedia.fragmentConnector.parameterExcludeList`.
Default	
`com.coremedia.fragmentConnector.parameterExcludeList`	
Description	Comma separated list of parameter names. If set, all parameters but the configured ones will be copied from the shop request to the CAE fragment request. The property `com.coremedia.fragmentConnector.parameterIncludeList` has precedence.
Default	

Table 3.1. CoreMedia Connector Properties

3.4 Load Essential Data and Demo Data

The *cmlivecontext* extension comes with impex data to prepare the *Hybris* content catalog of Apparel Site UK to work together with the demo data of the *CoreMedia Blueprint* workspace. The impex data can be found in the *Workspace for SAP Commerce Cloud* below `$HYBRIS_HOME\bin\custom\cmlivecontext\resources\cmlivecontext\import\contentCatalogs\apparel-ukContentCatalog\cms-content.impex`

WARNING

Before importing the data you should understand, what data is added and especially what changes will be done to existing pages. Feel free to edit the impex file or prepare the content manually via the *Hybris* administration cockpits.

Out of the box the impex import will apply the following changes:

- Add a dedicated OAuth Client for the Commerce Adapter to receive cmsTickets via OAuth.
- Add CoreMedia LiveContextContentComponent to ComponentTypeGroups `narrow` and `wide`
- Add CoreMedia LiveContext Page Template to be used for CoreMedia Content Pages
- Add Page CoreMedia CMContentPage
- Modifying existing Pages to add the *CoreMedia Content Widget* to their Page Grids. The following pages are affected:
  - HomePage
  - ProductDetail Page
  - Product Grid Page (Category Landing Page)

To add essential data and *CoreMedia Content Cloud* demo data to your *Hybris* Content Catalog, open *Hybris SAP Administration Cockpit* and navigate to *Platform* > *Update*. The list should contain the extension "cmlivecontext". Check "cmlivecontext" and update the system.

To verify if the update was successful open the *SAP Administration Cockpit*. Select *WCMS* > *Page* in the left hand menu. You should find the CoreMedia-ContentPage, a page to display channels and articles managed in CoreMedia.

You should also find *CoreMedia Content Widget* in the page grid of the homepage. For further details how to work with the *CoreMedia Content Widget* see Section 5.2.1, "CoreMedia Content Widget" [37]

4. Connecting with an *SAP Hybris Commerce* System

The connection of your *Blueprint* web applications (*Studio* or *CAE*) to a *SAP Hybris Commerce* system is configured on the Commerce Adapter side and on the CMS side. The configuration consists of two parts:

- Configuration of the Commerce Adapter to connect to a *SAP Hybris Commerce* system (see Section 4.1, "Configuring the Commerce Adapter" [24]).
- Settings configuration in *Studio*. It references the Commerce Adapter endpoint and the catalog and store configuration *Studio* and *CAE* uses for commerce integration (see Section 4.2, "Shop Configuration in Content Settings" [26]).

NOTE

Prerequisite

Before you can connect the CoreMedia system with the *SAP Hybris Commerce* system you need to deploy the CoreMedia extensions into your *Hybris* system as described in Chapter 3, *Customizing SAP Hybris Commerce* [12].

4.1 Configuring the Commerce Adapter

Configuring the Commerce Adapter

The physical connection to the *Hybris Commerce* system is configured in the Commerce Adapter. The Commerce Adapter itself communicates via SAP REST API calls with the *Hybris* system.

The Commerce Adapter comes along with a set of configuration properties. For detailed documentation and defaults see Chapter 9, *Commerce Adapter Properties* [98].

The `commerce-adapter-hybris` expects to be connected to the latest supported *SAP Commerce Cloud* version by default. To connect to an older *SAP Commerce Cloud* version, the adapter must be started with the Spring profile for the target version activated, e.g. `hybris-2105`.

Starting the Commerce Adapter

This guide describes how to build and run the `commerce-adapter-hybris` Docker container.

Prerequisites to be installed:

- Maven
- Docker
- Docker Compose (optional)

CoreMedia provides a Docker setup for the *SAP Commerce Cloud Connector*. It is part of a dedicated CoreMedia SAP Commerce Cloud Connector Contributions Repository.

After cloning the workspace, a `coremedia/commerce-adapter-hybris` Docker image can be build via `mvn clean install` command.

To run the `commerce-adapter-hybris` Docker container, the configuration properties for the adapter must be set (see above). Spring Boot offers several ways to set the configuration properties, see Spring Boot Reference Guide - 24. Externalized Configuration. When starting the Docker container, this will probably lead to setting either environment variables (using the Docker option `--env` or `--env-file`) or mounting a configuration file (using the Docker option `--volume`).

The Docker container can be started with the command

```
docker run \
  --detach \
  --rm \
  --name commerce-adapter-hybris \
  --publish 44265:6565 \
  --publish 44281:8081 \
  [--env ...|--env-file ...|--volume] \
  coremedia/commerce-adapter-hybris:${ADAPTER_VERSION}
```

To run the `commerce-adapter-hybris` Docker container with the CoreMedia CMCC Docker environment, add the `commerce-adapter-hybris.yml` compose file that is provided with the CoreMedia Blueprint Workspace to the `COMPOSE_FILE` variable in the Docker Compose `.env` file. Ensure that the environment variables that are passed to the Docker container are also defined in the `.env` file:

```
COMPOSE_FILE=compose/default.yml:compose/commerce-adapter-hybris.yml
HYBRIS_HOST=...
...
```

The `commerce-adapter-hybris` container is started with the CoreMedia CMCC Docker environment when running

```
docker compose up --detach
```

Detailed information about how to set up the CoreMedia CMCC Docker environment can be found in Chapter 2, *Docker Setup* in *Deployment Manual*.

4.2 Shop Configuration in Content Settings

The store specific properties that logically define a shop instance are part of the content settings. They configure the Commerce Adapter endpoint, which storeId should be used, which catalog, the currency and other shop related settings.

Refer to the Javadoc of the class `com.coremedia.blueprint.base.livecontext.client.settings.CommerceSettings` for further details.

Each site can have one single shop configuration (see the Blueprint site concept to learn what a site is). That means only shop items from exactly that shop instance (with a particular view to the product catalog) can be interwoven to the content elements of that site. In the example settings there is a `LiveContext` settings content linked with the root channel. This is the perfect place to make these settings.

The following store specific settings must be configured below the struct property named `commerce`:

Name	Type	Description	Example	Required
`endpoint`	String Property	Host and Port of the Commerce Adapter.	hybris-commerce-adapter:6565	true (if endpointName is not set)
`endpointName`	String Property	The endpoint name to lookup the Spring gRPC service configuration .	hybris	true (if endpoint is not set)
`locale`	String Property	The ISO locale code for the connected Catalog. This overwrites the Site locale. It is only needed if the CoreMedia Site locale differs from the Shop locale and if you need the exact Shop locale to access the catalog.	en-GB	false
`currency`	String Property	The displayed currency for all product prices.	GBP	false. If not set, the currency will be retrieved

Name	Type	Description	Example	Required
				from the site locale.
`storeConfig`	Struct Property	Struct property containing store configuration		true
`storeConfig.id`	String Property	The ID of the store.	apparel-uk	true
`storeConfig.name`	String Property	The name of the store as it is set in the commerce system.	Apparel-Catalog	true
`catalogConfig`	Struct Property	Struct property containing catalog configuration.		true
`catalogConfig.id`	String Property	The ID of the catalog.	apparelProductCatalog	true
`catalogConfig.name`	String Property	The name of the catalog.	apparelProductCatalog	true
`catalogConfig.alias`	String Property	The alias of the catalog.	catalog	false. If not set, 'catalog' will be used as default alias.
`customEntityParams`	Struct Property	Site specific custom entity parameters, which are attached to the communication with the commerce adapter. See Section 4.4, "Configuring Custom Entity Parameters" [31] for more information.		false. If not set, no site specific custom entities will be used.

Table 4.1. livecontext settings

NOTE

Be aware, that the locale is also part of each shop context. It is defined by the locale of the site. That means all localized product texts and descriptions have the same language as the site in which they are included and one specific currency.

4.3 Check if everything is working

Prerequisites

- The *CoreMedia Content Cloud* infrastructure has been deployed and is running.
- The *Hybris workspace archive* has been applied to the *Hybris Project Workspace* and the *SAP Hybris Commerce* server is running.
- The *SAP Hybris Commerce* server is accessible from CoreMedia Studio and the Commerce Adapter servers.
- The CoreMedia Preview *CAE* and Live *CAE* are accessible from the *SAP Hybris Commerce* server.

Check the Studio - Hybris REST Connection

1. Open *Studio*, select the "Hybris Apparel - English" site, open the Library. If necessary, switch the Library to browse Mode.
2. In the repository tree view, locate a node named *Apparel-Catalog*. This is the entry point to browse the connected *Hybris* product catalog.
3. Browse the catalog in studio and check if everything works as expected. Section 6.1, “Catalog View in CoreMedia Studio Library” [68] describes what it looks like.

If errors occur:

- Check the Studio log and the Commerce Adapter log for errors.
- Check in *CoreMedia Studio* if the "LiveContextSettings" are configured correctly, see Section 4.2, “Shop Configuration in Content Settings” [26].
- Check if the REST connector is configured correctly (see Section 4.1, “Configuring the Commerce Adapter” [24]). Check for example, if the deployment property `hybris.host` is configured correctly.

Check Studio - Hybris Preview Integration

1. Open the Homepage of the "Hybris Apparel - English" site in Studio

   The *Hybris* shop page should be displayed in the preview panel.
2. Repeat step 1 for Products and Categories.

If errors occur:

- Check the Studio log, the Preview *CAE* log and the Commerce Adapter log for errors.
- Check if `hybris.link.storefront-url` is configured correctly for the Commerce Adapter.
- Check if the "coremedia_preview" OAuth client has been imported via impex correctly. This is required to request a cmsTicket from *Hybris* previewwebservices extension.
- Check if, `StudioPreviewUrlService` is accessible. Call `https://hybrishost:9002/yacceleratorstorefront/cmpreview`. The given URL is incomplete, but the controller should be dispatched and raise an error like "HTTP Status 400 - Required String parameter 'type' is not present".

Check Fragment Connector

1. Open the Apparel-UK site and check if CoreMedia Demo content is displayed.

   The *Hybris* homepage should be displayed and CoreMedia is embedded.

If errors occurred or no CoreMedia Content is displayed

- Check for errors in the Hybris log and the Preview *CAE* log and the Commerce Adapter log.
- Check, if `$HYBRIS_HOME/bin/custom/cmlivecontext/project.properties` is configured correctly.
- Check in *SAP SmartEdit*, if the homepage has content slots containing *CoreMedia Content Widgets*. These slots are named "LiveContext HP Slot XX". If not, something went wrong while importing impex data.

4.4 Configuring Custom Entity Parameters

Custom entity parameters can be used to transport additional information from the client to the commerce adapter.

Let's say you want to transmit the environment type (Dev, UAT, Prod) of your client with every request. This way you want to resolve certain host names on the adapter side for different environments. Out of the box there is no dedicated field "environment" available in the `EntityParams`, which are sent along with every request from the client to the commerce system. The custom entity parameters enable you to provide this information to the adapter side without API changes. You can do this by simple configuration.

Example:

This example shows a configuration for an *`environment`* entity parameter:

Adapter Configuration

Configure on the adapter side `metadata.custom-entity-param-names=environment` to tell the connected clients, to send the custom parameter named "environment" alongside with every client request.

Client Configuration

Configure a global variable on the client side, using the property `commerce.hub.data.customEntityParams`. Simply add the name of the variable to the property name:

`commerce.hub.data.customEntityParams.environment=UAT`

You can also configure custom entity params in *Studio* via commerce settings. This way, it is possible to transmit site specific environment parameters to the commerce adapter.

```
commerce (Struct)
  customEntityParams (Struct)
    environment=UAT (String)
```

NOTE

If the same parameter is defined via property and via *Studio* commerce settings, the site specific commerce settings configuration has precedence over the global property based configuration.

5. Commerce-led Integration Scenario

In the commerce-led integration scenario the commerce system delivers content to the customer. The shop pages are augmented with fragment content from the CoreMedia system.

This chapter describes how you include the content from the CMS into shop pages. Have also a look into Section 6.4, "Augmenting Commerce Content" [77] and Chapter 6, *Working with Product Catalogs* in *Studio User Manual* for more details about the *Studio* usage for eCommerce.

- Section 5.1, "Commerce-led Scenario Overview" [34] gives an overview over the request flow in the commerce-led integration scenario.
- Section 5.2, "Adding CMS Fragments to Shop Pages" [36] describes how you can add fragments to the commerce system via the CoreMedia widgets and the `lc:include` tag and how you can augment shop pages in *Studio*.
- Section 5.3, "Extending the Shop Context" [48] describes how you extend the shop context that is delivered to the CMS.
- Section 5.4, "Solutions for the Same-Origin Policy Problem" [51] describes how the same-origin policy problem has been solved for the CoreMedia solution.
- Section 5.5, "Caching In Commerce-Led Scenario" [54] describes the caching in the commerce-led scenario.
- Section 5.6, "Prefetch Fragments to Minimize CMS Requests" [59] describes how to prefetch fragments in the commerce-led scenario.

5.1 Commerce-led Scenario Overview

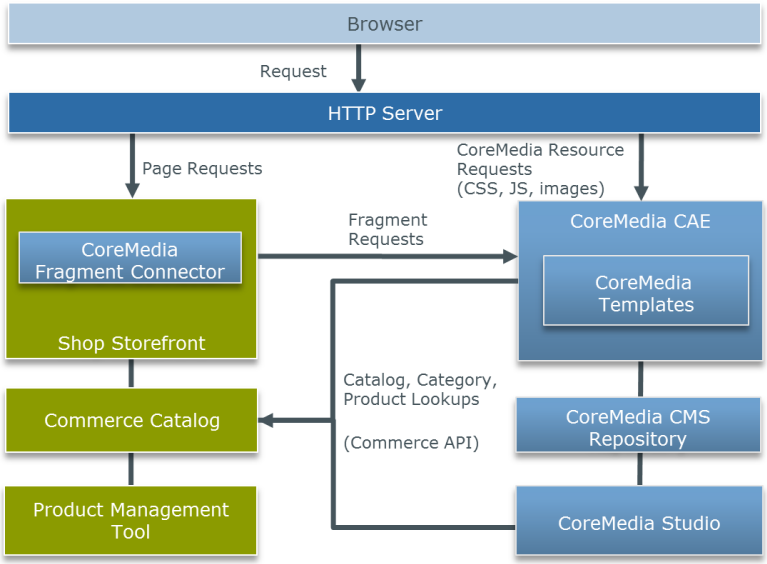

Figure 5.1. Commerce-led Architecture Overview

Figure 5.1, "Commerce-led Architecture Overview" [34] shows the commerce-led integration scenario where the CoreMedia *CAE* operates behind the commerce server for all page request. Moreover, you can see two kinds of requests. While the left side shows HTTP page requests to the commerce server, that include fragments delivered by the *CAE*, the right side shows resource or Ajax requests directly redirected by the one virtual host in front of both servers to the *CAE*.

A typical flow of requests through a commerce-led system is as follows:

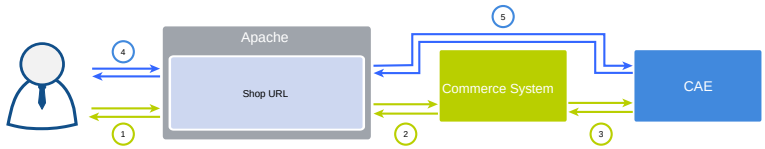

Figure 5.2. Commerce-led Request Flow

1. A user requests a product detail page that is received by the virtual host.
2. The virtual host identifies the request as a commerce request and forwards it to the commerce server.
3. Part of the requested Product Detail Page (PDP) is a CMS content fragment. Hence, the commerce system requests the fragment from the *CAE*.
4. The resulting HTML page flows back to the user's browsers. Because the page contains dynamic *CAE* fragments which have to be fetched via Ajax, the browser triggers the corresponding request against the virtual host.
5. As this is a *CAE* request, the virtual host forwards it directly to the *CAE*.

From the point of view of the user all requests are sent to exactly one system, represented by the one virtual host that forwards the requests accordingly. That leads to the same-origin policy problem. Solutions for this are presented in section Section 5.4, “Solutions for the Same-Origin Policy Problem” [51].

5.2 Adding CMS Fragments to Shop Pages

A pure eCommerce system is focused on the more transactional aspects of the buying process. To create a more engaging user experience you can augment the catalog pages with editorial content from the CMS. This includes, articles, images or videos.

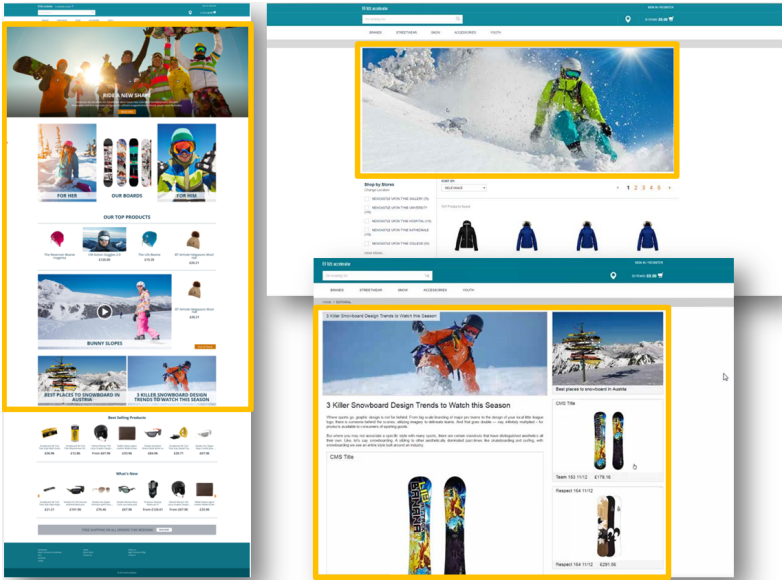

Figure 5.3. Various Shop Pages with CMS Fragments

Types of augmentable pages

There are two types of shop pages that can be extended by *CoreMedia Content Cloud*:

- **Catalog Pages** that are part of the catalog hierarchy, like a Category Overview or Landing Page and a Product Detail Page (PDP). They are extended by `Augmented Categories` and `Augmented Products` in the CMS.
- **Other Pages** that are not located in the catalog hierarchy. For example, all subordinate shop pages like "Contact Us", "Log On", "Checkout", "Register" or "Search Result", which also belong to a shop but don't have a category or a product connected with.

Even the homepage and other special topic pages belong to this type. These pages are extended by `Augmented Pages` in the CMS.

In addition, you can show complete CMS pages in the context of the commerce system. That page type is called **Content Pages**.

The augmentation process

The basis for augmentation is the use of the *CoreMedia Content Widget* or the `lc:include` tag in the commerce system.

On the commerce side, add the *CoreMedia Content Widget* to the commerce page layouts or write the `lc:include` tag directly into a shop template. The value of the `placement` property corresponds to the `placement` name within a CMS-side page layout. Technically, the *CoreMedia Content Widget* uses also the `lc:include` tag internally. See Section 5.2.1, “CoreMedia Content Widget” [37] and Section 5.2.2, “The CoreMedia Include Tag” [40] for details.

When you have prepared the shop-side with such content slots (either as *CoreMedia Content Widget* or directly with `lc:include` tags in shop templates), and the commerce system is properly connected with the CMS systems, you can now start augmenting shop pages in *Studio*.

Section 6.4, “Augmenting Commerce Content” [77] describes the procedure.

5.2.1 CoreMedia Content Widget

Adding the CoreMedia Content Widget

On the *Hybris Commerce* side it is necessary to define slots where the CMS content can be displayed. This is normally done by adding the *CoreMedia Content Widget* to a *Hybris Commerce* page layout. The tool with which this can be done is the *SAP SmartEdit*.

Take the Apparel-UK homepage page as an example. As you can see in the screenshot below, there is one *CoreMedia Content Widget* placed.

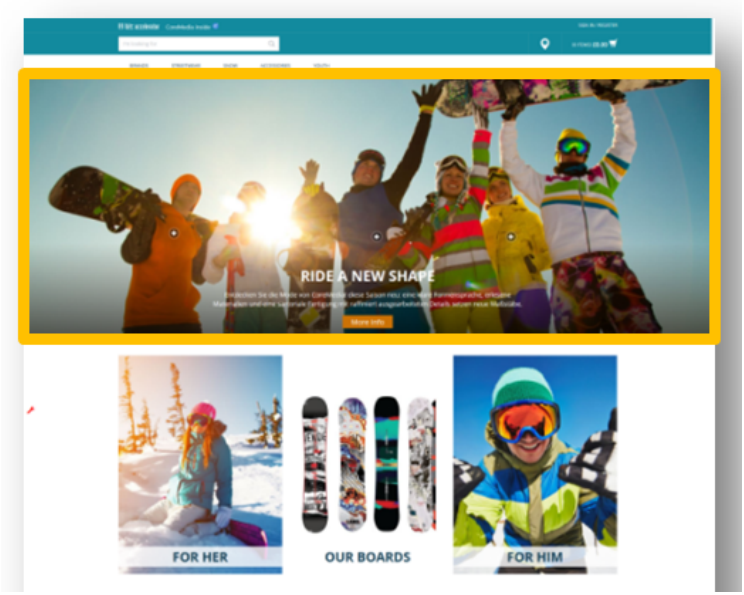

Figure 5.4. Using the CoreMedia Content Widget - A Homepage Fragment

The content that is shown in the *CoreMedia Content Widget* is taken from a placement of an augmenting CMS page. The name of the placement in the CMS page needs to correspond to the name configured in the *CoreMedia Content Widget*.

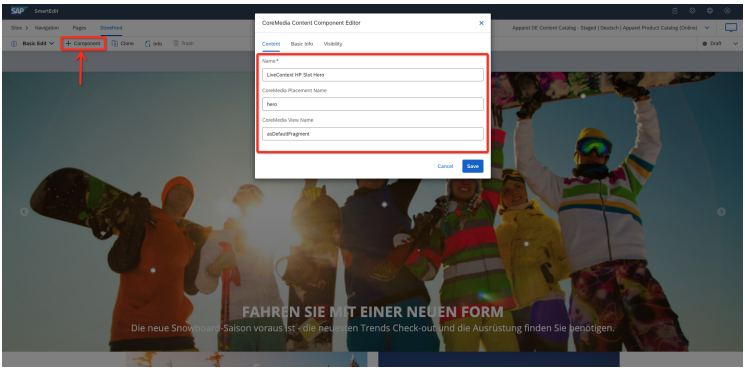

Figure 5.5. Using the CoreMedia Content Widget - Connection to CMS Content via placement name

NOTE

The name of the placement shown in the Studio form is the localized label. The name of the `placement` attribute in the *CoreMedia Content Widget* must match the technical name in the page grid definition.

CoreMedia Content Widget Configuration Options

Option	Description
CoreMedia Placement Name	The name of the placement as defined in CoreMedia CMS. Content on page grids in CoreMedia are defined through so called placements. Each placement is associated with a specific position of the page grid through its name. Using CoreMedia Studio the editor can add content to the placement which will be shown at the associated position of the page grid and subsequently in the layout of this CoreMedia Content Widget. If the placement is empty, the full page grid is taken.
CoreMedia View Name	The view name of the template that will be used for rendering on the CMS side. Each placement or page can be rendered with a specific view. A template with that name must exist in the CAE.

Table 5.1. CoreMedia Content Widget Configuration Options

The *CoreMedia Content Widget* is preconfigured for the Apparel-UK site to be available for the most common page grid slots. It has been added to the component type groups `wide` and `narrow` via initial impex import. Feel free to adjust this to your needs.

Using the lc:include tag

If the *CoreMedia Content Widget* cannot be used, like in the HTML head section or within an existing component, it is still possible to plug in a fragment rendered by the CMS into the output HTML. This can be achieved by using the `lc:include` tag directly within a JSP. This is a development task and is typically done during the project phase. Later, editors will only deal with `Augmented Categories` and `Augmented Pages` that they can edit and preview via *CoreMedia Studio*.

Technically, the *CoreMedia Content Widget* is using the `lc:include` tag as well. See Section 5.2.2, “The CoreMedia Include Tag” [40] for a description.

5.2.2 The CoreMedia Include Tag

Behind the scenes of the *CoreMedia Content Widget* works the CoreMedia `lc:include` tag. You may also use it in your own `JSP` templates to embed CoreMedia content on the commerce side. In general it is used like this:

```
<%@ taglib prefix="lc"
           uri="http://www.coremedia.com/2014/livecontext-2" %>
<c:if test="${not empty param.content}">
  <c:set var="lc_externalRef" value="cm-seosegment:${param.content}"/>
</c:if>

<lc:include
        storeId="${cmsSite.uid}"
        locale="${lc:toLocale(cmsSite.locale)}"
        productId="${product.code}"
        categoryId="${searchPageData.categoryCode}"
        placement="${placement}"
        view="${param.view}"
        externalRef="${lc_externalRef}"
        parameter="${param.parameter}"
        pageId="${cmsPage.uid}"/>
```

All parameters are described in the next two sections.

Include Tag Reference

The tag attributes have the following meaning:

Parameter	Description
storeId, *locale*	These attributes are mandatory. They are used in the CAE to identify the site that provides the requested fragment.
catalogId	In a multi-catalog scenario this attribute is mandatory. It is used in the CAE to identify the catalog context for rendering the requested fragment.
productId, *categoryId*	These attributes are used in the CAE to find the context which will be used for rendering the requested fragment. Both parameters should not be set at the same time since depending on the attributes set for the include tag, different handlers are invoked: If the *categoryId* is set, `CategoryFragmentHandler` will be used to generate the fragment HTML. If the *productId* is set, `ProductFragmentHandler` will be used to generate the fragment HTML.
pageId	This parameter is optional. Usually, the page ID is computed from the requested URL (the last token in the URL path without a file extension). If you set the parameter, the automatically generated value is overwritten. On the

Parameter	Description
	Blueprint side an *Augmented Page* will be retrieved to serve the fragment HTML. The transmitted page ID parameter must match the *External Page ID* of the *Augmented Page*. You might use the parameter, for example, in order to have one CoreMedia page to deliver the same content to different shop pages.
placement	This attribute defines the name of a placement in the page grid of the requested context. In the example for the header fragment, the "header" placement was used. If you do not want to render a certain placement but a view of the whole context (generally a CMChannel), you may omit it. If the view attribute isn't set, the "main" placement will be used as default instead. This attribute can be combined with the *externalRef* attribute. In this case the placement will be rendered for a specific CMChannel, so the external reference must point to a CMChannel instance.
view	The attribute "view" defines the name of the CMS view which will render the fragment. Such view templates must exist on the CMS side. There are several views prepared in the Blueprint: `metadata` (to render the HTML title and metadata), *externalHead* (to render parts of the HTML header like CSS and JavaScripts that are needed in CMS fragments), *externalFooter* (is also mostly used for loading scripts) and *asAssets* (that can render the *CoreMedia Product Asset Widget*). If you omit the view, the default view will be used. In such cases you have either the *placement* or the whole page grid of a CoreMedia page is rendered.
externalRef	This attribute is used in the CAE to find content. Several formats are supported here as described in the next section. The attribute can be used in combination with the *view* and/or *parameter* attribute.
parameter	This attribute is optional and may be used to apply a request attribute to the CAE request. The request attribute is stored using the constant `FragmentPageHandler.PARAMETER_REQUEST_ATTRIBUTE`. The value may be read from a triggered web flow, for example, to pass a redirect URL back to the commerce system once the flow is finished. The attribute also supports values to be passed in JSON format (using single quotes only), for example `parameter="{'test':'some value','value':123}"`. The key/values pairs are available in the `FragmentParameters` object and may be accessed using the `getParameterValue(String key)` method. Other additional values, like information about the current user that should be passed for every request, may be added to the request context that is build when the commerce system requests the fragment information from the CAE (see next section).

Parameter	Description
var	This attribute is optional. If set, the parsed output of the CAE is not written in the parsed output stream but in a page attribute named like the *var* parameter value. This allows you, for example, to replace or transform parts of the CAE result or, if empty, to render a different output.
exposeErrors	This attribute is optional. If set to true, the tag will expose any errors that occur during the interaction with the CMS. These errors are then directly written to the response. Thus, the commerce system has the ability to handle the errors, to show an error page, for instance.
httpStatusVar	This attribute is optional. If set, the HTTTP status code of the fragment request is set into a page attribute named like the *httpStatusVar* parameter value. This allows you, for example, to react on the result code, for example, set the fragment as uncacheable in the caching layer of your commerce system.

Table 5.2. Attributes of the Include tag

External References

Any linkable CoreMedia content can be included as a fragment by specifying a value for the `externalRef` attribute. The value of the attribute is applied to the first `ExternalReferenceResolver` predicate that is applicable for the `externalRef` value. The Spring list `externalReferenceResolvers` which contains the supported `ExternalReferenceResolvers` is injected to the `ExternalRefFragmentHandler`. This section shows the supported formats that are applicable for the existing resolvers.

The following table shows an overview about the possible values for the `externalRef` attribute.

Value Type	Example	Description
Content ID	cm-coremedia:///cap/content/4712	Includes the content with the given cap id as fragment. The root channel of the corresponding site will be used as context.
Numeric Content ID	cm-4712	Works the same way like the content ID include, only with the numeric content ID.
Absolute Content Path	cm-path!!Themes!basic!img!icons!ico_rte_link.png	Includes the content with the given absolute path. All exclamation marks ['!'] after

Value Type	Example	Description
		the prefix 'cm-path!' will be mapped to slashes ['/'] to provide a valid absolute CMS path. The given path may not contain 'Sites' [referencing content of a different site is not allowed]. The *`storeId`* and *`locale`* parameter are still mandatory for this case.
Relative Content Path	cm-path!actions!Login	Includes the content with the given path treated as a relative path from the site's root folder. All exclamation marks ['!'] after the prefix 'cm-path!' will be mapped to slashes ['/'] to provide a valid relative CMS path. The given path may not contain '..' [going up in the hierarchy]. The site is determined through the *`storeId`* and *`locale`* parameter.
Numeric Context and Content ID	cm-3456-6780	The prefix is the numeric content ID of the context to be rendered. The suffix is the numeric content ID of the content to be rendered with the given context.
Segment Path	cm-segmentpath:!corporate!on-the-table	The actual value [excl. the format prefix `cm-segmentpath:`] denotes a segment sequence, separated by exclamation marks. The segments are matched against the values of the `segment` properties of the content. The very last segment denotes the actual content. The other segments denote the navigation hierarchy which determines the context of the content. The example value references a linkable content with the segment `on-the-table` in the context of a channel `corporate` [which is apparently the root channel, since it consists of a single segment]. The context and the content must fulfill the Blueprint's context relationship, otherwise the request is handled as invalid.

Value Type	Example	Description
		Segment Path external references are resolved by querying the Solr search engine. The *CAE Feeder* must be running for up-to-date results.
Search Term	cm-searchterm:summer	Includes the content that contains the given search term (specified after the prefix `cm-searchterm:`). This resolver is typically used to resolve search landing pages. By default, contents of type `CMChannel` below the segment path `<root segment>/livecontext-search-landing-pages` are checked if their `keywords` search engine index field contains the term. Matching is case-insensitive by default and can be customized by using a different search engine field or field type. The value of the segment path which is used to identify the SLP channel is configured with the property `livecontext.slp.segmentPath`. Content type and search engine field can be configured with Spring properties `searchTermExternalReferenceResolver.contentType` and `searchTermExternalReferenceResolver.field`, respectively. The segment path is configured as relative path after the root segment. The configured segment path value must not start with a slash. Search term lookup is cached, by default for 60 seconds. You can configure the cache time in seconds with Spring property `cache.timeout-seconds.com.coremedia.livecontext.fragment.resolver.SearchTermExternalReferenceResolver` and the maximum number of cached search term lookups with `cache.capacit`

Value Type	Example	Description
		`ies.com.coremedia.livecontext.fragment.resolver.SearchTermExternalReferenceResolver` (defaults to 10000). Search Term external references are resolved by querying the Solr search engine. The *CAE Feeder* must be running for up-to-date results.

Table 5.3. Supported usages of the externalRef attribute

Finding Handlers

You can control the behavior of the `include` tag by providing different sets of attributes. Depending on the used attributes, different handlers are invoked to generate the HTML.

The CoreMedia `lc:include` tag requests data from the CAE via HTTP. Each attribute value of the include tag is passed as path or matrix parameter to the `FragmentPageHandler`. In order to find the matching handler, the `FragmentPageHandler` class calls the `include` method of all fragment handler classes defined in the file `livecontext-fragment.xml`. The first handler that returns "true" generates the HTML. Example 5.1, “Default fragment handler order” [45] shows the default order:

```
<util:list id="fragmentHandlers"
value-type="com.coremedia.livecontext.fragment.FragmentHandler">
  <description>This list contains all handlers that are used for fragment
calls.</description>
  <ref bean="externalRefFragmentHandler" />
  <ref bean="externalPageFragmentHandler" />
  <ref bean="productFragmentHandler" />
  <ref bean="categoryFragmentHandler" />
</util:list>
```

Example 5.1. Default fragment handler order

If the handlers are in the default order, then the table shows which handler is used depending on the attributes set. An "x" means that the attribute is set, a "-" means that the attribute is not allowed to be set and no entry means that it does not matter if something is set. For more details, have a look into the handler classes.

External Reference	Page ID	Category ID	Product ID	Used Handler
x				`ExternalRefFragmentHandler`

External Reference	Page ID	Category ID	Product ID	Used Handler
-	x	-	-	`ExternalPageFragmentHandler`
-			x	`ProductFragmentHandler`
-		x	-	`CategoryFragmentHandler`

Table 5.4. Fragment handler usage

NOTE

The parameters category id and product id may be treated as technical id or as external id. It is recommended to work with external ids if possible. If the commerce system cannot pass external ids into the fragment parameters because only technical ids are available, this behaviour must be configured on the commerce adapter side. The property *`metadata.additional-metadata.allow-tech-ids=true`* has to be set for the commerce adapter, if you want to use technical ids in the fragment connector.

Fragment Request Context

In addition to the passed request parameters, a context is build by the registered `ContextProvider` implementations that are part of the commerce workspace. The context provider passes context information as header attributes to the CAE. For more details see Section 5.3, “Extending the Shop Context” [48].

CMS Error Handling

Since the CoreMedia `include` tag requests data from the CAE via HTTP, errors can occur. The error handling can be controlled by different parameters. If the *`com.coremedia.fragmentConnector.isDevelopment`* property (see Section 3.3, “Configuring the CoreMedia Fragment Connector” [17]) is set to `true`, the `include` tag will embed occurring error messages as strings into the page output. You may not want to see such information on the live side, thus the flag can be set to `false` and all output will be suppressed (the errors are only visible in the log).

This behavior is sufficient for providing additional (possibly optional) information on a page, a banner or teaser, for instance. But if the requested content is the major content

of a page, then it is not desirable to deliver a mainly empty page. In such a case the commerce system should be able to handle the error situation and answer in an appropriate form. That could be, for example, a `404` error page.

For this purpose the *`exposeErrors`* parameter was introduced to the `include` tag. If this parameter is set to `true`, the tag will expose any error that occurs during the interaction with the CMS. These errors are directly written to the response. Sending a response with an error status code (`404`, for instance) requires that still nothing has been written to the `Response` object. Therefore, this flag should only be set on the `include` tag if rendered early enough before any other response code has been set.

In the *Hybris workspace archive* the usage of the *`exposeErrors`* parameter is demonstrated in the `main.tag` JSP. The template is executed on every page request and renders, among other things, the HTML `head` section of a page. The first occurrence of the `include` tag is used to do the error handling.

Since the template is executed for all shop pages the flag must be set depending on the target page. If it's a content centered page (it has, for example, a *`cm`* parameter), then the parameter would be set to true, in case of a category or product detail page probably not.

```
exposeErrors="${not empty param.content && empty product.code && empty
searchPageData.categoryCode}"
```

Another possibility to handle failed fragment requests is the usage of the *`httpStatusVar`* parameter. If this parameter is set, the include tag will write the HTTP status code of the fragment request into a JSP attribute/variable. You can then add JSP code to react on specific result codes and for example disable caching of this fragment in the commerce cache.

```
<lc:include ...
  httpStatusVar="status"/>
...
<c:if test="${not empty status && status >= 400}">
  ... // error handling
</c:if>
```

5.3 Extending the Shop Context

To render personalized or contextualized info in content areas it is important to have relevant shop context info available during CAE rendering. It will be most likely user session related info, that is available in the Commerce system only and must now be provided to the backend CAE. Examples are the user id of a logged in user, gender, the date the user was logged in the last time or the names of the customer segment groups the user belongs to, up to the info which campaign should be applied. Of course these are just examples and you can imagine much more. So it is important to have a framework in order to extend the transferred shop context information flexibly.

The relevant shop context will be transmitted to the CoreMedia CAE automatically as HTTP header parameters and can there be accessed for using it as "personalization filter". It is a big advantage of the dynamic rendering of a CoreMedia CAE that you can easily process this information at rendering time.

The transmission of the context will be done automatically. You do not have to take care of it. On the one end, at the commerce system, there is a context provider framework where the context info is gathered, packaged and then automatically transferred to the backend CAE. A default context provider is active and can be replaced or supplemented by your own `ContextProvider` implementation.

Implement a custom ContextProvider

To extend the shop context you have to supply implementations of the `ContextProvider` interface. The `ContextProvider` interface demands the implementation of a single method.

```
package com.coremedia.livecontext.connector.context;

import javax.servlet.http.HttpServletRequest;

public interface ContextProvider {

  /**
   * Add values to the given context.
   * @param contextBuilder the contextBuilder - the means to add entries to
 the entry
   * @param request - the current request, from which e.g. the session can
be retrieved
   * @param environment - an environment, not further specified
   */
  void addToContext(ContextBuilder contextBuilder, HttpServletRequest request,
 Object environment);
}
```

Example 5.2. ContextProvider interface method

Such implementations of the `ContextProvider` interface must be provided with the *Hybris Commerce* workspace. That is typically be done below the `$HYBRIS_HOME\bin\custom\cmlivecontext\acceleratoraddon\web\src` directory of your *Hybris* project extension provided by CoreMedia (`cmlivecontext`). Such context provider implementations will use the *Hybris* API to gather information from the current shop session. The current user ID or all segment names the current user is member of are prominent examples of such context data.

There can be multiple `ContextProvider` instances chained. Each `ContextProvider` enriches the `Context` via the `ContextBuilder`. The resulting `Context` wraps a map of key value pairs. Both, keys and values have to be strings. That means if you have a more complex value, like a list, it is up to you to encode and decode it on the backend CAE side. Be aware that the parameter length can not be unlimited. Technically it is transferred via HTML headers and the size of HTML headers is limited by most HTTP servers.

CAUTION

As a rough upper limit you should not exceed 4k bytes for all parameters, as they will be transmitted via HTTP headers. You should also note that this data must be transmitted with each backend call.

All `ContextProvider` implementations are configured via the property `com.coremedia.fragmentConnector.contextProvidersCSV` in the file `coremedia-connector.properties` as a comma separated list. The configured `ContextProvider` instances are called each time a CMS fragment is requested from the *CAE* backend.

Read shop context values on the CAE side

On the backend CAE side the shop context values will be automatically provided via a `Context` API. You can access the context values during rendering via a Java API call.

All fragment requests are processed by the `FragmentCommerceContextInterceptor` in the *CAE*. This interceptor calls `LiveContextContextAccessor.openAccessToContext(HttpServletRequest request)` to create and store a `Context` object in the request. You can access the `Context` object via `LiveContextContextHelper.fetchContext(HttpServletRequest request)`.

```
import com.coremedia.livecontext.fragment.links.context.Context;
import
com.coremedia.livecontext.fragment.links.context.LiveContextContextHelper;

import javax.servlet.http.HttpServletRequest;

public class FragmentAccessExample {
...
  private LiveContextContextAccessor fragmentContextAccessor;
```

```
  public void buildContextHttpServletRequest request(){
    fragmentContextAccessor.openAccessToContext(request);
  }

  public String getUserIdFromRequest(HttpServletRequest request){
    Context context = LiveContextContextHelper.fetchContext(request);
    return (String) context.get("wc.user.id");
  }
...
}
```

Example 5.3. Access the Shop Context in CAE via Context API

5.4 Solutions for the Same-Origin Policy Problem

When the commerce system has to deliver the end user's web pages, *CoreMedia Content Cloud* offers a way to enrich those web pages with content from the CoreMedia CMS; the fragment connector.

Integrating content from the CoreMedia system into the shop pages presents a challenge due to the same-origin policy:

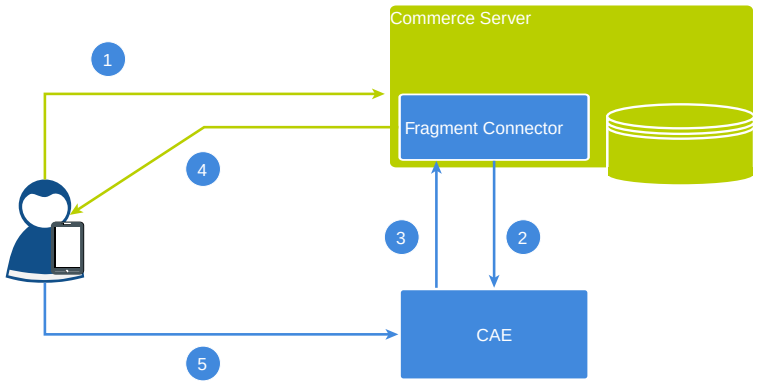

Figure 5.6. Cross Domain Scripting with Fragments

The image above shows a typical situation when a user requests a shop page that includes CoreMedia fragments.

1. The page request from the end user is sent to the commerce server.
2. While rendering the page, the commerce server requests a fragment from the CAE.
3. The returned fragment contains itself parts that must be delivered dynamically. Take the login button. It is user specific, hence it must not be cached. The CoreMedia Blueprint may include such parts via Ajax requests or as ESI tags, depending on the capabilities of the component which sent the request.
4. The commerce server returns the complete page, including the fragment that was rendered by the CAE.
5. Because it is assumed that the CoreMedia eCommerce fragment contains a dynamic part, which must not be cached, the browser tries to trigger an Ajax request to the CAE. But this breaks the same-origin policy and will not succeed.

Solution 1: Access-Control-Allow-Origin

The first solution is built into the CoreMedia Blueprint workspace, so you may use it out of the box. The idea is to customize the same origin policy by setting the `Access-Control-Allow-Origin` HTTP header accordingly. The allowed origins can be configured via the properties `cae.cors.allowed-origins-for-url-pattern[*]`.

```
cae.cors.allowed-origins-for-url-pattern[{path\:.*}]= \
    http://my.site.domain1,https://my.site.domain2
```

To fine-tune the configuration for Cross-Origin Resource Sharing (CORS), use the provided `cae.cors` configuration properties. See Section 3.1.4, "CORS Properties" in *Deployment Manual* and Section 4.3.1.8, "Solution for the Same-Origin Policy Problem" in *Content Application Developer Manual*.

Solution 2: The Proxy

To solve this problem the classical way, the Ajax request needs to be sent to the same origin than the whole page request in step 1 was. The next image shows the solution to this problem: A reverse proxy needs to be put in front of both the CAE and the commerce server.

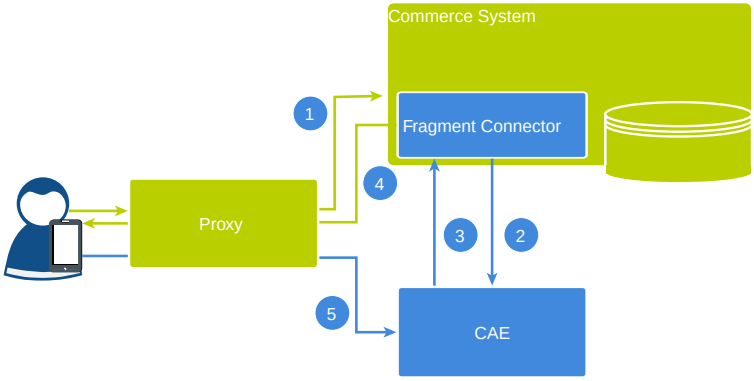

Figure 5.7. Cross Site Scripting with fragments

Actually, you may use any proxy you feel comfortable with. The following snippet shows the configuration for a Varnish. Two back ends were defined, one for the CoreMedia eCommerce CAE named `blueprint` and another one for the commerce server named `commerce`.

The `vcl_recv` subroutine is called for every request that reaches the Varnish instance. Inside of it the request object `req` is examined that represents the current request. If its `url` property starts with `/blueprint/`, it will be sent to the CoreMedia eCommerce CAE. Any other request will be sent to the commerce system. [`~` means "contains" and the argument is a regular expression]

Now, if you request a shop URL through Varnish and the resulting page contains a CoreMedia eCommerce fragment including a dynamic part that must not be cached, like the sign in button, the Ajax request will work as expected.

```
backend commerce {
    .host = "ham-its0484-v";
    .port = "80";
}

backend blueprint {
    .host = "ham-its0484";
    .port = "40081";
}

sub vcl_recv {
    if (req.url ~ "^/blueprint/") {
      set req.backend = blueprint;
    } else {
      set req.backend = commerce;
    }
}
```

5.5 Caching In Commerce-Led Scenario

This section discusses the ability of using a caching proxy between the shop system and the *CAE* in the commerce-led scenario. That could be, for example, a CDN or a *Varnish Cache*. This increases the reliability of the CMS system: Fragments can be served from the cache even if the CMS is unreachable.

For this purpose, fragment requests with only static data have to be distinguished from those with dynamic personalized data. Static fragments are cacheable, but dynamic fragments are not. When the fragment delivered by the *CAE* contains personalized content, the fragment can still be cached as the `DynamicInclude` mechanism is used as specified in Section 6.2.1, "Using Dynamic Fragments in HTML Responses" in *Blueprint Developer Manual* for such dynamic fragments. This means the fragment with the dynamic content is fetched in a separate call with a different URL pattern. These can be handled by the proxy differently.

To enable the usage of `DynamicInclude` for personalized content add a Boolean property `p13n-dynamic-includes-enabled` to your page setting and set it to `true`.

You can also control how the `DynamicInclude` is handled. Per default if you just enable dynamic include a placement containing any personalized content (even if nested inside linked collections) will be loaded via dynamic include as a whole. In contrast to this you can add and enable the Boolean property `p13n-dynamic-includes-per-item` to achieve a more fine granular dynamic include. So in case the aforementioned placement contains personalized content only this content is loaded via dynamic include, making the non-personalized parts of the placement cacheable.

CAUTION

Please note that using dynamic include per item has some limitations:

It will only work as expected if the container of the personalized content (CMSelection-Rules or CMP13NSearch) is part of the rendering (more precisely: part of a render node, for example, being used as parameter `self` in a `cm.include` call). Any mechanism that simplifies / flattens nested container structures may prevent this from happening and can cause that the personalized content might be cached.

This especially means that using the (now deprecated) `getFlattenedItems` method of the `com.coremedia.blueprint.layout.Container` interface should be avoided. Please check Section 5.16, "Rendering Container Layouts" in *Frontend Developer Manual* for a possible approach which is used in CoreMedia's example themes.

In addition to this, the dynamic include mechanism does not preserve parameters passed to the template which is being loaded via dynamic include at the moment (for example, the `params` parameter of the `cm.include` call) so you need to work around this limitation for now.

Example Request Flow

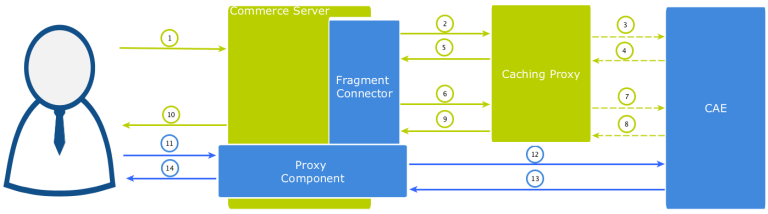

Figure 5.8. Example request flow

Figure 5.8, "Example request flow" [55] shows the commerce-led integration scenario the user requests a page with a static and a potentially dynamic CoreMedia fragment delivered by *CAE*. Note that the green arrows symbolize the flow of static content (cacheable) and the blue the flow of dynamic content. A dotted line means that the symbolized flow is optional and is omitted when the (cacheable) content is already cached.

1. A user requests a shop page from the commerce server. Let's assume the shop page consists of a static and a potentially dynamic fragment. The commerce server asks the fragment connector to collect the fragments.

2. The connector requests *CAE* for the static fragment.
3. The Caching Proxy intercepts the request and delivers the static fragment if already cached. Let's assume it is not or the TTL has expired, the request is forwarded to *CAE*.
4. *CAE* delivers the static fragment to the Caching Proxy.
5. The Caching Proxy caches the static fragment and delivers it to the fragment connector.
6. In case of another fragment include on the commerce page the connector requests *CAE* for the potentially dynamic fragment.
7. Again the Caching Proxy intercepts the request and delivers the fragment if already cached. Assuming it is not or the TTL has expired, the request is forwarded to *CAE*.
8. Assume that the *CAE* detects a personalized piece of content within the fragment (that cannot be cached), then it decides to deliver the fragment as `DynamicInclude`. The result is still a cacheable HTML fragment but contains a link from where the dynamic fragment can be loaded. This link points to a proxy component that is part of the CoreMedia package installed in the commerce server. Such a fragment is then later retrieved via AJAX (see step 11).
9. The Caching Proxy caches the result even if it contains only the stub with a link to retrieve a dynamic fragment and delivers it to the fragment connector.

   The HTML fragment is then post-processed by the Commerce server.
10. If the connector has all fragments together, the Commerce server can deliver the complete page to the requesting browser. In this case the result will contain a static CMS fragment inline and an AJAX stub with dynamic include URL that point to the Proxy Component.
11. The user's browser triggers a AJAX call to the Proxy Component to load the dynamic fragment.
12. The Commerce server enriches the dynamic request with the user context information and the Proxy Component forwards it to the *CAE*. This time the dynamic request is not intercepted by the Caching Proxy. Such dynamic include URLs are always passed to the *CAE*. The proxy is configured accordingly.
13. The *CAE* delivers the content of the personalized dynamic fragment back to the Proxy Component.
14. The Proxy Component forwards the dynamic content to the user's browser after it was post-processed by the Commerce server.

The *CAE* renders the fragment adaptively. That means if no personalized content is used in a fragment, no dynamic include will be triggered. For instance, several fragments of the kind from step 2 to 5 would then be delivered.

The CoreMedia Proxy Component

The CoreMedia Proxy Component is part of *Hybris Project Workspace* and will be installed with all other CoreMedia customizations. Technically it is a Spring controller that uses the request mapping `/cmdynamic` with a single *`url`* parameter. This parameter contains an encoded *CAE* URL that is then be called by the Proxy Component, post-processed (all containing links will be generated) and the result is finally sent to the browser.

The post-processing of the received fragment payload is an important step carried out by both the Proxy Component and the *CoreMedia Fragment Connector*. At this point, their processing is similar. Links to other shop pages which may be contained in a fragment coming from the CAE must be post-processed in the Commerce system. This is because the knowledge about the final link format is in the Commerce system. In addition, other server side includes can also be done, for example, the rendering of a price info.

See the section Section 5.7.1, “How fragment links are build” [64] for more information about link building on the commerce site.

```
<div class="cm-fragment"
data-cm-fragment="/yacceleratorstorefront/en/cmdynamic/?url=%2Fblueprint%2Fservlet%2F
dynamic%2Fplacement%2Fp13n%2Fapparelhomepage%2F104%2Fplacement%2Fhero%3FtargetView%3DasDefaultFragment%255Bhero%255D%26
fragmentContext%3D%2Fapparel-uk%2Fen-GB%2Fparams%3Bview%253DasDefaultFragment%3Bplacement%253Dhero%3BpageId%253Dhomepage'>
</div>
```

Example 5.4. AJAX Stub

The contained URL will be decoded by the Proxy Component and called on the *CAE*.

```
/blueprint/servlet/dynamic/placement/p13n/apparelhomepage/104/placement/hero?
targetView=asDefaultFragment%5Bhero%5D&
fragmentContext=/apparel-uk/en-GB/params;view%3DasDefaultFragment;placement%3Dhero;pageId%3Dhomepage
```

Example 5.5. Effective Dynamic Include URL

Altogether there are also a few variants of these URLs which differ slightly in their path components. The identifying segment path can be filtered by the regular expression `/dynamic/.+?/p13n/`. A Caching Proxy in between should ignore these kinds of URLs.

Adding Context Information to Dynamic Calls

Fragments calls to the *CAE* can carry context information as request headers. For example that can be a membership of a customer segment or the current user id. Such

information will be transmitted as HTTP request headers. Should personalized content be used, along with caching between Commerce server and *CAE* please make sure all relevant context data are provided in the *CoreMedia Fragment Connector*. Please see the Section 5.3, “Extending the Shop Context” [48]. for details.

CAUTION

If the feature "Dynamic Includes in Content Fragments" stays off but personalized content is still used, the generated fragments must not be cached. Otherwise, the first user who generates such a fragment would determine the cached content.

5.6 Prefetch Fragments to Minimize CMS Requests

A shop page in the commerce-led scenario can contain multiple CMS fragments (placements and views). Normally, each CMS fragment would cause an external HTTP call to the CAE which can lead to performance loss and, depending on the commerce system, reach a limit of outgoing requests on the commerce side (see Figure 5.9, “Multiple Fragment Requests without Prefetching” [59]). Furthermore, each request is processed consecutively. As a result, the response times for each individual CAE request add up to the total pageview time. Therefore, CAE offers a mechanism to lower the amount of CAE requests by prefetching all expected fragments in advance in a single call.

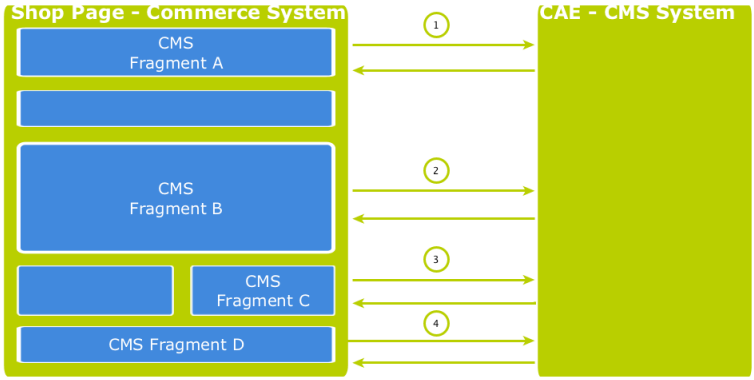

Figure 5.9. Multiple Fragment Requests without Prefetching

How to configure which fragments to prefetch

If the "prefetching feature" is enabled in the *CoreMedia Fragment Connector* on the commerce side, a dedicated `prefetchFragments` call is made to the CAE. The result is a JSON structure that consists of all fragments that are pre-rendered by the CAE. To predict the fragment calls that would normally follow, the CAE follows a twofold strategy.

- Each CMS fragment call of a single shop page should conceptually go to the "same" CMS page. Which means technically, that all the parameters that identify a CMS page should be the same in all CMS fragment calls of a single shop page (these are: `ex-`

ternalRef, *productId*, *categoryId* and *pageId*). The CAE therefore uses these parameters to predict the required fragments. Every placement in the assigned page layout can be considered as "potentially to be requested". Therefore, every placement is contained as a separate fragment in the JSON result. To identify the view that should be used to render the placement a configuration is read from the *LiveContext* `Settings` content. The Figure 5.10, "LiveContext Settings: Prefetch Views per Placement" [61] shows an example configuration. If no setting can be found, it is assumed that the default view should be rendered for a placement.

- Additionally, every shop page requests a few more, mostly technical fragments from the CAE. These fragments are requested as different "views" of the same page. Examples of such views are *metadata*, *externalHead* and *externalFooter* that are likely to be included on every shop page. These "additional views" are also read from the *LiveContext* `Settings` content and they are also included in the JSON result. The Figure 5.11, "LiveContext Settings: Prefetching Additional Views" [62] shows an example of such a configuration.

If all required fragments are already included in the prefetch result, then only one CAE fragment request is needed per shop page. All subsequent fragment calls are then served from the local fragment cache within the *CoreMedia Fragment Connector*. Thus, the configuration should be complete for each shop page type. The configuration is placed in the *LiveContext* `Settings` content, to be found in the `Options/Settings` folder of the corresponding site and linked in the root channel. In the following sections the configuration is explained in detail.

Prefetch Configuration: View per Placement

The first configuration option is to define a view name for a certain placement. You can add this view name to the prefetch result, otherwise the default view would be rendered for this placement. Within the *livecontext-fragments* struct the *placementViews* sub-struct is used to store this information.

livecontext-fragments		Struct
prefetchedViews	Struct with 3 properties	Struct
placementViews		Struct
defaults		Struct List
#1		Struct
section	header	Link to Symbol
view	asDefaultFragment	String
#2		Struct
section	banner	Link to Symbol
view	asDefaultFragment	String
#3		Struct
section	footer	Link to Symbol
view	asDefaultFragment	String
layouts		Struct List
#1		Struct
layout	Fragment PDP	Link to Settings
placementViews		Struct List
#1		Struct
view	asHeaderFragment	String
section	header	Link to Symbol

Figure 5.10. LiveContext Settings: Prefetch Views per Placement

NOTE

The configuration needs only to be done, if there are placements that should be rendered with a different view than the default view.

Below the *placementViews* struct, two sub-elements are used:

`defaults`	Defines the view, a placement will be prefetched with, for all layouts. It overrides the default view and is itself overwritten by a layout specific configuration in the `layouts` struct element.
`layouts`	Defines a layout-specific view with which a placement will be prefetched. It overrides the view defined in the `defaults` struct element for this specific placement.

Prefetch Configuration: Additional Views

The second configuration option is the definition of additional views which should also be included into the prefetch result. Within the *livecontext-fragments* struct the *prefetchedViews* sub-struct is used for these settings.

▼ livecontext-fragments		Struct
▼ prefetchedViews		Struct
▼ defaults		String List
#1	metadata	String
#2	externalHead	String
#3	externalFooter	String
▼ contentTypes		Struct List
▼ #1		Struct
type	CMLinkable	String
▼ prefetchedViews		String List
#1	metadata	String
#2	asFragment	String
#3	asBreadcrumb	String
#4	externalHead	String
#5	externalFooter	String
#6	DEFAULT	String
▼ layouts		Struct List
▼ #1		Struct
layout	🛠 Fragment PDP	Link to 🛠 Settings
▼ prefetchedViews		String List
#1	metadata	String
#2	asBreadcrumb	String
#3	externalHead	String
#4	externalFooter	String
▶ placementViews	Struct with 1 property	Struct

Figure 5.11. LiveContext Settings: Prefetching Additional Views

Below the `prefetchedViews` struct three sub-elements are used:

`defaults` — Defines the views that should be additionally prefetched for all layouts. It is overwritten by a layout specific configuration in the `layouts` element.

`layouts` — Defines the views that should be additionally prefetched for a specific layout. It overwrites the configuration in the `defaults` struct element.

`contentTypes` — Defines the views that should be prefetched for a specific content type on Content Pages (see Section 5.2, "Adding CMS Fragments to Shop Pages" [36] for a definition of Content Page) (for example, a page that has a CMS article as main content).

Content Pages can contain CMS content of different types. For each type you can configure a struct with views that will be prefetched. You can use abstract or parent content

types to combine multiple types (`CMLinkable`, for instance).

If more than one configured content type can be applied to a given content, the configuration for the most specific content type will prevail. For example when `CMLinkable` and `CMChannel` are configured, then for a `CMChannel` content item only the configuration for `CMChannel` will be taken into account.

To define the default view to be additionally prefetched, use the `DEFAULT` identifier.

Configuration in *SAP Hybris*

The prefetch functionality is enabled by default. It can be enabled or disabled via property *`com.coremedia.fragmentConnector.isPrefetchEnabled`* in `coremedia-connector.properties`.

5.7 Link Building for Fragments

Overview

If you include CoreMedia fragments into shop pages, these fragments might also contain links to shop pages; a link to an Augmented Category, for example. In the commerce-led scenario all pages are rendered by the commerce system. The link generation is also done on the commerce system.

The *eCommerce Blueprint* contains `@Link` annotated methods to create links to the commerce system, for example to a Category Page. These links can be retrieved from the `LinkService`, which can be accessed via the `CommerceConnection`. The `LinkService` itself requests URL templates from the Commerce Adapter. Later these URL templates are post-processed by the `LiveContextLinkTransformer`. The result is a JSON snippet in HTML comments that is finally converted into a link on the commerce site (see Section 5.7.1, "How fragment links are build" [64] for details). Since these links point to the commerce system there is no need for a matching `@RequestMapping` method. See also the Section 4.3, "The CAE Web Application" in *Content Application Developer Manual* for more information regarding link building.

The templates which finally generate the commerce URLs can be found in *Hybris Project Workspace* below path `$HYBRIS_HOME/bin/custom/cmlivecontext/acceleratoraddon/web/webroot/WEB-INF/views/responsive/cms/templates`.

5.7.1 How fragment links are build

Each `lc:include` tag requests an HTML fragment via HTTP from the CAE. Every link within a fragment that is requested by the commerce system from the CAE is processed by the `LiveContextLinkTransformer` class. The transformer only applies for fragment requests and finally requests URL templates from the `LinkRepository` on the Commerce Adapter side. For fragment request the Commerce Adapter returns JSON strings to the CAE. Each of these JSON objects contains at least the values of the constants `objectType` and `renderType` and the ID of the content or commerce object.

Assume the HTML fragment contains a link to a `CMArticle` content item. Instead of rendering the regular link, for example

```
http://cae-host/blueprint/servlet/page/mySite/mySegment/mySeoContent-4712
```

the corresponding Link generated by the `LiveContextLinkResolver` would look like:

```
a href="<!--CM {
    "id":"cm-1696-4712",
    "renderType":"url",
    "externalSeoSegment":"mySeoContent-4712",
    "objectType":"content"}
    CM-->" ...
```

The `CoreMedia Fragment Connector` on the commerce side parses the JSON, identifies the object type and rendering type and applies a template to render a commerce link. For the given example, the template `Content.url.jsp` is used, applied by the pattern "<OBJECT_TYPE>.<RENDER_TYPE>.jsp".

The JSP file on the commerce side finally generates the resulting URL.

5.7.2 Commerce Links for CoreMedia Content

Links to CoreMedia Contents like articles and channels look like this:

```
https://hybris-host/yacceleratorstorefront/en/cm/best-picture-contest/
```

Example 5.6. Commerce URL

The request path `"/cm"` is mapped to `CmContentPageController` on the commerce side.

If you want to change the predefined URL prefix "/cm" for CoreMedia Content Pages, you need to customize the controller mapping for `CmContentPageController` and link generation in `Content.url.jsp`, `StudioPreviewUrlService#setCmContentUrlPrefix` and `UrlTag#buildContentUrl`.

5.7.3 Commerce Links for Studio Preview

Studio and the Preview-CAE do not know the *SAP Hybris Commerce* URL-Schema of shop pages. Therefore, the CoreMedia service `StudioPreviewUrlService` deployed in the *SAP Hybris Commerce* system generates the commerce URLs in order to preview commerce items as shop pages in *CoreMedia Studio*. The class `CommerceLinkScheme` wraps the corresponding @Link methods in the *CoreMedia Blueprint* workspace. It retrieves the commerce links via the `PreviewUrlService` from the Commerce Adapter.

The request flow is quite complicated. The example below represents the request flow to preview a *Hybris* product from within *CoreMedia Studio*:

1. *Studio* generates this preview URL for the product with the given ID.

```
https://preview-cae-host/preview?id=hybris:///catalog/product/104176
&site=Hybris-Apparel-UK-Site-ID&contentTimestamp=54539
&p13n_test=true&p13n_testcontext=0 > 302
```

2. The Preview-CAE receives the preview URL, internally dispatches it to the `CommerceLinkScheme` and sends a redirect to the `StudioPreviewUrlService` deployed in the *SAP Hybris Commerce* System.

```
https://hybris-host/yacceleratorstorefront/cmpreview?site=apparel-uk&id=104176
&type=product&cmsTicketId={ticket-id} > 302
```

3. The *SAP Hybris Commerce* System receives the request, generates a `CMSPreviewTicket` with the given parameters and redirects to the *Hybris* PreviewServlet.

```
https://hybris-host/yacceleratorstorefront/cx-preview
?site=apparel-uk&cmsTicketId={ticket-id} > 302
```

4. The *SAP Hybris Commerce* System receives the previewServlet request again and redirects to the resulting shop URL:

```
https://hybris-host/yacceleratorstorefront/c/
Nightlife-T-Shirt-Women/p/104176?cmsTicketId={ticket-id}> 200
```

6. Studio Integration of Commerce Content

CoreMedia Content Cloud integrates with *SAP Hybris Commerce*. In the following it is simply called the "commerce system" or "the shop".

From classical shop pages, like a product catalog ordered by categories or product detail pages up to landing pages or homepages, all grades of mixing content with catalog items are conceivable. The approach followed in this chapter, assumes that items from the catalog will be linked or embedded without having stored these items in the CMS system. Catalog items will be linked typically and not imported.

- Section 6.1, "Catalog View in CoreMedia Studio Library" [68] gives a short overview over the Catalog Integration in the *Studio* Library.
- Section 6.3, "Commerce related Preview Support Features" [74] gives a short overview over the commerce related preview functions that are supported in *CoreMedia Studio*.
- Section 6.4, "Augmenting Commerce Content" [77] describes how you augment commerce content in the commerce-led scenario in *CoreMedia Studio*.

6.1 Catalog View in CoreMedia Studio Library

When the connection to a *Hybris Commerce* system and a concrete shop for a content site are configured as described in Chapter 4, *Connecting with an SAP Hybris Commerce System* [23] the *Studio* Library shows the commerce catalog to browse product categories and products in the commerce catalog and to search for products and product variants. After the editor has selected a preferred site with a valid store configuration the catalog view will be enabled and the catalog will be shown in the Library:

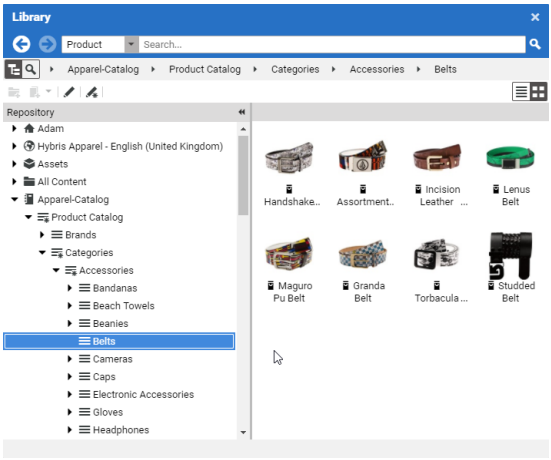

Figure 6.1. Library with catalog in the tree view

In some catalogs it is possible to put a category on multiple places within the catalog tree. But the Commerce Hub ensures that a category can only have one home (a unique parent category). All additional occurrences of a category are shown as a link in the tree. If you click on such a link node you will automatically end up at the place in the tree where the category is actually at home.

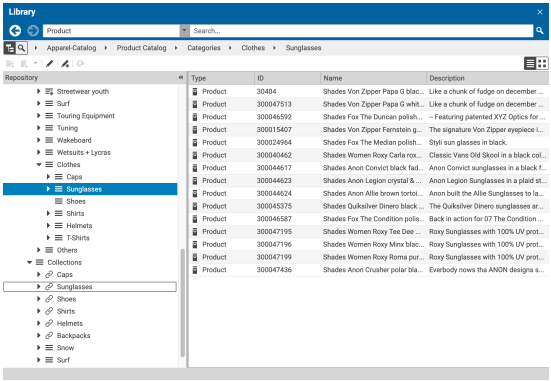

Figure 6.2. Library tree with multiple occurrences of the same category

These catalog items can be accessed and assigned to various places within your content. For example, an *eCommerce Product Teaser* content item can link to a product or product variant from the catalog. The product link field (in *eCommerce Product Teaser* content item) can be filled by drag and drop from the library in catalog mode.

Linking a content (like the *eCommerce Product Teaser*) to a catalog item leads to a link that is stored in the CMS content item and references the external element. Apart from the external reference (in the case of the commerce system it is typically a persistent identifier like the product code for products) no further data will be imported (importless integration).

While browsing through the catalog tree you can also open a preview of a category or a product from the library. Simply double-click on a product in the product list or use the context menu on a product or a category and choose the entry **Open in Tab** from the context menu as shown in the pictures below.

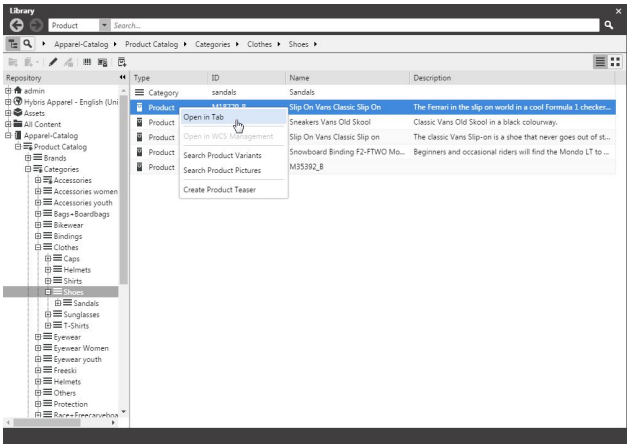

Figure 6.3. Open Product in tab

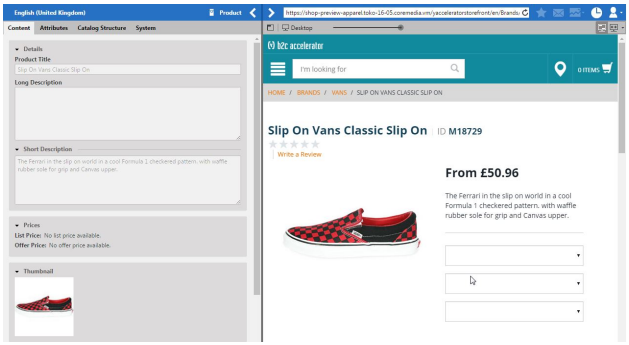

Figure 6.4. Product in tab preview

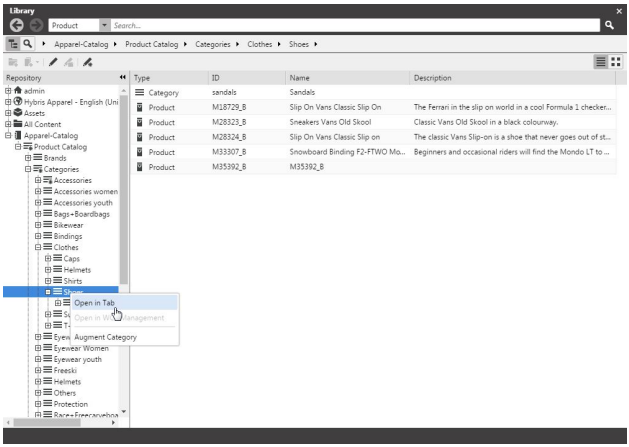

Figure 6.5. Open Category in tab

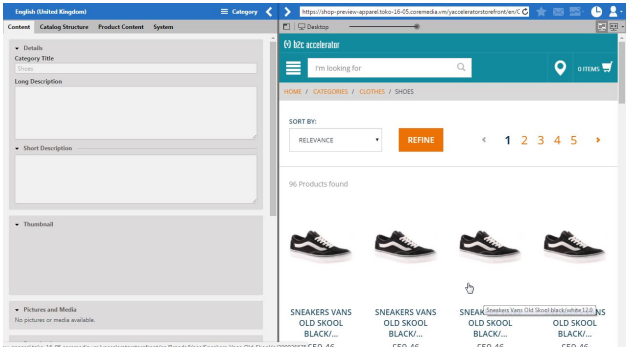

Figure 6.6. Category in tab preview

In addition to the ability to browse through the commerce catalog in an explorer-like view it is also possible to search for products and variants from catalog. As for the content search if you are in the catalog mode and you type a search keyword into the search field and press **Enter**, the search in the commerce system will be triggered and a search result displayed.

6.2 Enabling Preview in Shop Context

CoreMedia Content Cloud enables you to directly preview pages for not augmented or augmented products, not augmented or augmented categories and CoreMedia channels in *CoreMedia Studio* within the shop context (as a shop page with the shop frame around it). Otherwise, you would get a CoreMedia-typical fragment preview that shows a content item with multiple views.

To enable the preview of Category Pages in the shop context, add a Boolean property `livecontext.policy.commerce-category-links` to your LiveContext settings and set the value "true".

To enable the preview of Product Pages in the shop context, add a Boolean property `livecontext.policy.commerce-product-links` to your LiveContext settings and set the value "true".

To enable the preview of CoreMedia Channels in the shop context, add a Boolean property `livecontext.policy.commerce-page-links` to your LiveContext settings and set the value "true".

In order to enable the preview of Commerce shop pages in Studio, proceed as follows:

1. Make sure the customization coming with the *Workspace for SAP Commerce Cloud* has been applied to your *SAP Hybris Commerce* installation (see Chapter 3, *Customizing SAP Hybris Commerce* [12]).

Configure in the CoreMedia system

2. In the `studio-server` app, the `studio.previewUrlWhitelist` property must contain the commerce URL (including the port, for example `*coremedia.com` or `http://localhost:40080`). The default CAE preview URL must remain in the `studio.previewUrlWhitelist` property too.

   You can find more information regarding link building for commerce items here: Section 5.7, “Link Building for Fragments” [64].

NOTE

If your *SAP Hybris Commerce* shop storefront uses any clickjacking prevention features (for example, X-Frame-Options), make sure to allow the shop preview being embedded as an iframe within *CoreMedia Studio*.

To do so uncomment or adjust the property `xss.filter.header.X-Frame-Options` in `$HYBRIS_HOME/hybris/bin/platform/project.properties`. For more information refer to the *Hybris* documentation.

6.3 Commerce related Preview Support Features

CoreMedia Studio supports a variety of commerce preview functions directly:

- Time based preview (time travel)

  When a preview date is set in *CoreMedia Studio*, it sets the virtual render time to a time in the future. If the currently previewed page contains content from *Hybris Commerce*, it is desirable that also these content reflects the given preview time. That could be a certain validity period of a product or another display rule that influences the displayed catalog items.

  If such preview is requested from *Hybris Commerce* the preview date is also sent to *Hybris Commerce* as part of the `cmsTicket` parameter. The *Hybris Commerce* recognizes the transmitted preview date and renders the shop content accordingly.

- Customer segment based preview

  The feature segment based preview supports the creation of personalized content. In this case, content is shown depending on the membership in specific customer segments. In addition to the existing rules, you can define rules that are based on the belonging to customer segments that are maintained by the commerce system.

  These commerce segments will be automatically integrated and appear in the chooser if you create a new rule in a personalized content. For a preview, editors can use test personas which are associated with specific customer segments.

  Figure 6.7, “Test Customer Persona with Commerce Customer Segments” [75] shows an example where the test persona is female and has already been registered.

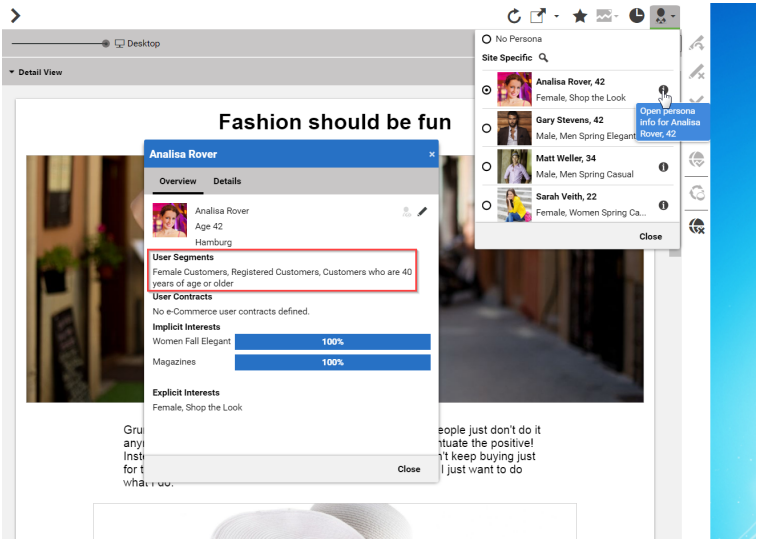

Figure 6.7. Test Customer Persona with Commerce Customer Segments

Such preview settings apply as long as they are not reset by the editor.

The test persona content can be created and edited in *CoreMedia Studio*. The customer segments available for selection will be automatically read from the commerce system. By default, all user segments available in the eCommerce system are displayed for selection. Under some circumstances it may be desirable to restrict the shown user segments, for instance for studio performance reasons or for better clarity for the editor. See Section 3.2.4, "Configuring The PersonaSelector" in *Personalization Hub Manual*.

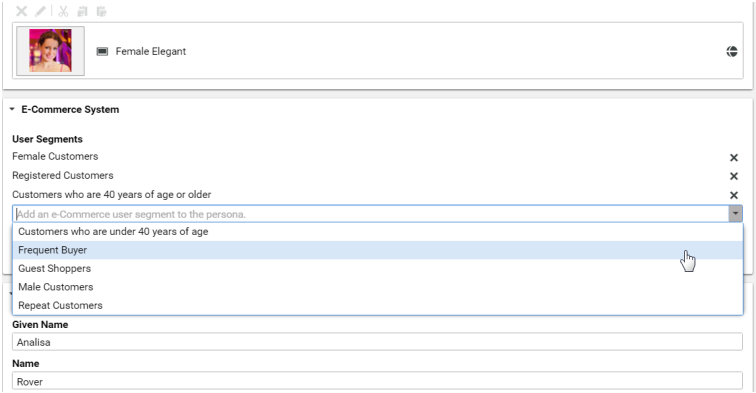

Figure 6.8. Edit Commerce Segments in Test Customer Persona

The commerce segments that the current user belongs to are available during the rendering process within a *CoreMedia CAE*. Thus, content from the CoreMedia system can also be filtered based on the current commerce segments.

In the other direction, if the personalized content is integrated within a content fragment on a shop page, the current commerce user is also transmitted as a parameter. Thus, the CoreMedia system can retrieve the connected customer segments from the commerce system in order to perform commerce segment personalization within the supplied content fragments.

6.4 Augmenting Commerce Content

In the commerce-led scenario you can augment pages from the Commerce System, such as products (Product Detail Pages), categories (Category Overview/Landing Pages) and other shop pages (like the Contact-Us Page linked from the Homepage Footer). The following sections describe the steps required in *Studio*.

Extending a shop page with CMS content comprises the following steps, which will be explained in the corresponding sections.

1. In the CMS create a content item of type `Augmented Category`, `Augmented Product` or `Augmented Page`.
2. Augment the root nodes of the catalogs as described in Section 6.4.1, "Augmenting the Root Nodes" [77].
3. When you augment a category or product, the connection between the category/product and the `Augmented Category`/`Augmented Product` content is automatically created. For the `Augmented Page` you have to create this connection manually via an external page id property
4. In the `Augmented Category`, `Augmented Product` or `Augmented Page` choose a page layout that corresponds to the shop page layout. It should contain all the placements that are referenced in the *CoreMedia Content Widgets* defined on the Commerce side.
5. Drop the augmenting content into the right placements of the augmented content item. That is, into a placement whose name corresponds with the name defined in the *CoreMedia Content Widget*.

6.4.1 Augmenting the Root Nodes

Catalog view in Studio

If the shop connection is properly configured, you will see an additional top level entry in the *Studio* library that is named after your store (for example, *Hybris, Apparel*). Below this node you can open the *Product Catalog* with categories and products. The *Product Catalog* node also represents the root category of a catalog.

Augmented catalog roots

To have a common ancestor for all augmented catalog pages, the root node of the configured catalog must be augmented. You can augment the root category by clicking *Augment Category* in the context menu of the root category. An augmented category content opens up, where you can start to define the default elements of your catalog pages, like the page layouts for the Category Overview Pages (CLP) and Product Detail Pages (PDP) and first content elements. All sub categories, augmented or not, will inherit

these settings. See Section 6.2.3, "Adding CMS Content to Your Shop" in *Studio User Manual* for more information.

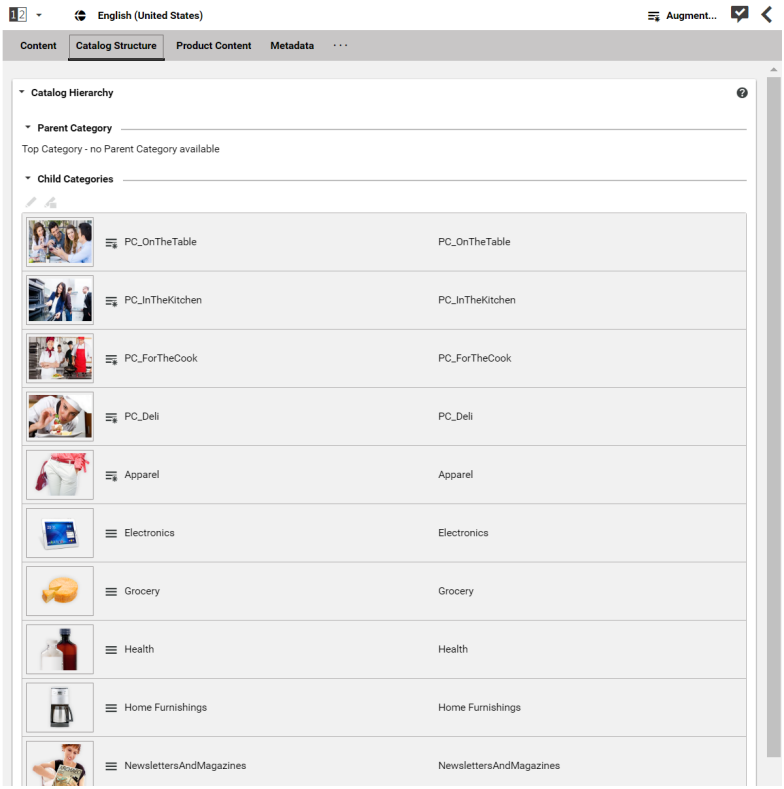

Figure 6.9. Catalog structure in the catalog root content item

Now, you can start augmenting sub categories of the catalog. All content and settings are inherited down in this hierarchy.

6.4.2 Selecting a Layout for an Augmented Page

CoreMedia Content Cloud comes with a predefined set of page layouts. Typically, this selection will be adapted to your needs in a project. By selecting a layout an editor specifies which placements the new page will have, which of them can be edited and

how the placements are arranged generally. It should correspond to the actual shop page layout. All usable placements should be addressed. The placement names must match the placement names used in the slot definition on the shop side.

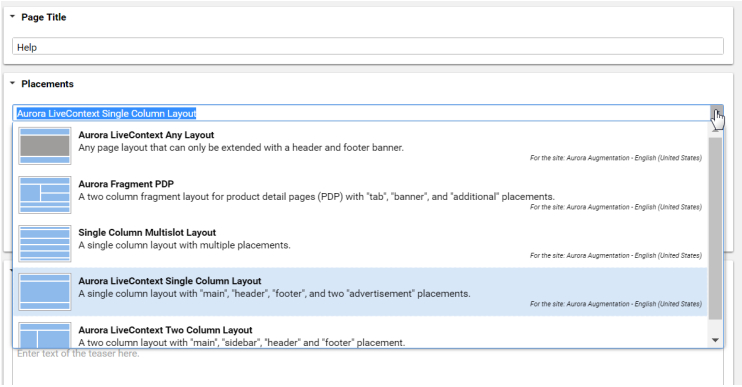

Figure 6.10. Choosing a page layout for a shop page

If you augment a category, the corresponding `Augmented Category` content item contains two page layouts: the one in the *Content* tab is applied to the Category Overview Page and the other in the *Product Content* tab is used for all Product Detail Pages. Both layouts are taken from the root category. The layouts that are set there form the default layouts for a site. Hence, they should be the most commonly used layouts. If you want something different, you can choose another layout from the list.

6.4.3 Finding CMS Content for Category Overview Pages

Category overview pages

A category overview page is a kind of landing page for a product category. If a user clicks on a category without specifying a certain product, then a page will be rendered that introduces a whole product category with its subcategories. Category overview pages contain a mix of product lists with and promotional content like product teasers, marketing content (that can also be product teasers but of better quality) or other editorial content.

You can use the *CoreMedia Content Widget* in the commerce-led scenario in order to add content from the CoreMedia CMS to the category overview page.

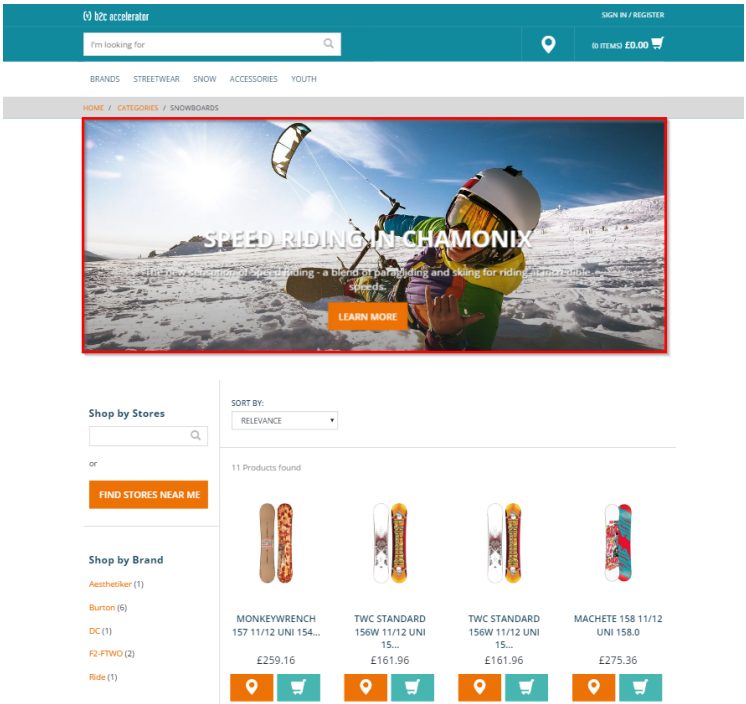

Figure 6.11. Category Overview Page with CMS Content

Information passed to the CoreMedia system

When a category page contains the *CoreMedia Content Widget*, then on request, the current category ID and the name of the placement configured in the *CoreMedia Content Widget* are passed to the CoreMedia system. The CoreMedia system uses this information to locate the content in the CoreMedia repository that should be shown on the category overview page.

Locating the content in the CoreMedia system

CoreMedia Content Cloud tries to find the required content with a hierarchical lookup using the category ID and placement name information. The lookup involves the following steps:

CoreMedia Content Cloud tries to find the required content with a hierarchical lookup, performing the following steps:

1. Select the `Augmented Page` that is connected with the shop.
2. Search in the catalog hierarchy for an `Augmented Category` content item that references the catalog category page that should be augmented and that contains a placement with the name defined in the *CoreMedia Content Widget*.

a. If there is no *Augmented Category* for the category, search the category hierarchy upwards until you find an *Augmented Category* that references one of the parent categories.

b. If there is no *Augmented Category* at all, take the site root *Augmented Page*.

3. From the `Augmented Category` content found take the content from the placement which matches the placement name defined in the *CoreMedia Content Widget*.

Figure 6.12, "Decision diagram" [81] shows the complete decision tree for the determination of the content for the category overview page or the product detail page [see below for the product detail page].

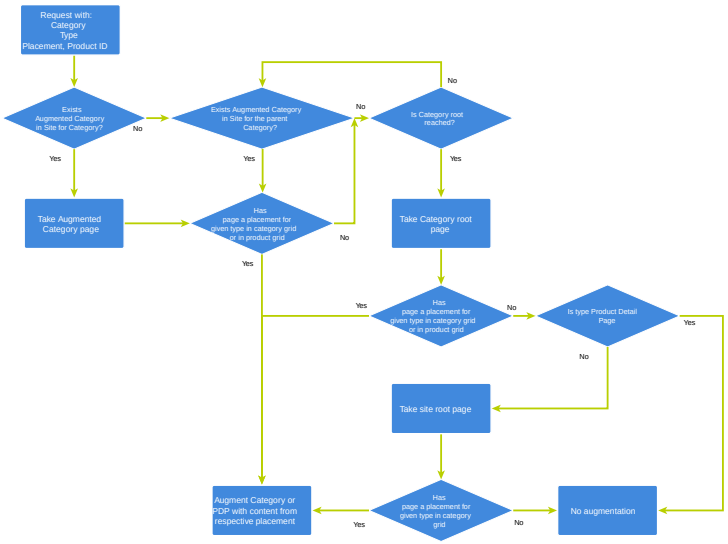

Figure 6.12. Decision diagram

Keep the following rules in mind when you define content for category overview pages:

- You do not have to create an *Augmented Category* for each category. It's enough to create such a page for a parent category. It is also quite common to create pages only for the top level categories especially when all pages have the same structure.
- You can even use the site root's `Augmented Page` to define a placement that is inherited by all categories of the site.
- If you want to use a completely different layout on a distinct page (a landing page's layout, for example, differs typically from other page's layouts), you should use different placement names for the "Landing Page Layout", for example with a `landing-`

`page` prefix (as part of the technical identifier in the struct of the layout content item). This way, pages below the intermediate landing page, which use the default layout again, can still inherit the elements from pages above the intermediate page (from the root category, for instance), because the elements are not concealed by the intermediate page.

6.4.4 Finding CMS Content for Product Detail Pages

Product Detail Pages

Product detail pages give you detailed information concerning a specific product. That includes price, technical details and many more. You can enhance these pages with content from the CoreMedia system by adding the *CoreMedia Content Widget* similar to the category overview page.

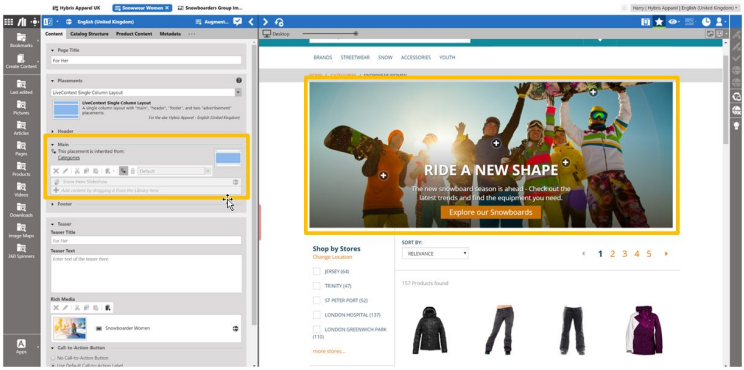

Figure 6.13. Product detail page with CMS content highlighted by borders

Information passed to the CoreMedia system

Similar to the category overview pages, the Category ID and placement name are passed to *CoreMedia Content Cloud* in order to locate the content.

Locating the content in the CoreMedia system

For product detail pages, the page can be directly augmented with an `Augmented Product` content type. If this is not the case, *CoreMedia Content Cloud* uses the same lookup as described for the category overview page. The only slight difference that the site root `Augmented Page` content item is not considered as a default for the product detail page.

The content to augment is taken from a separate page grid of the `Augmented Category`, called *Product Content* or from the *Content* tab of the `Augmented Product`.

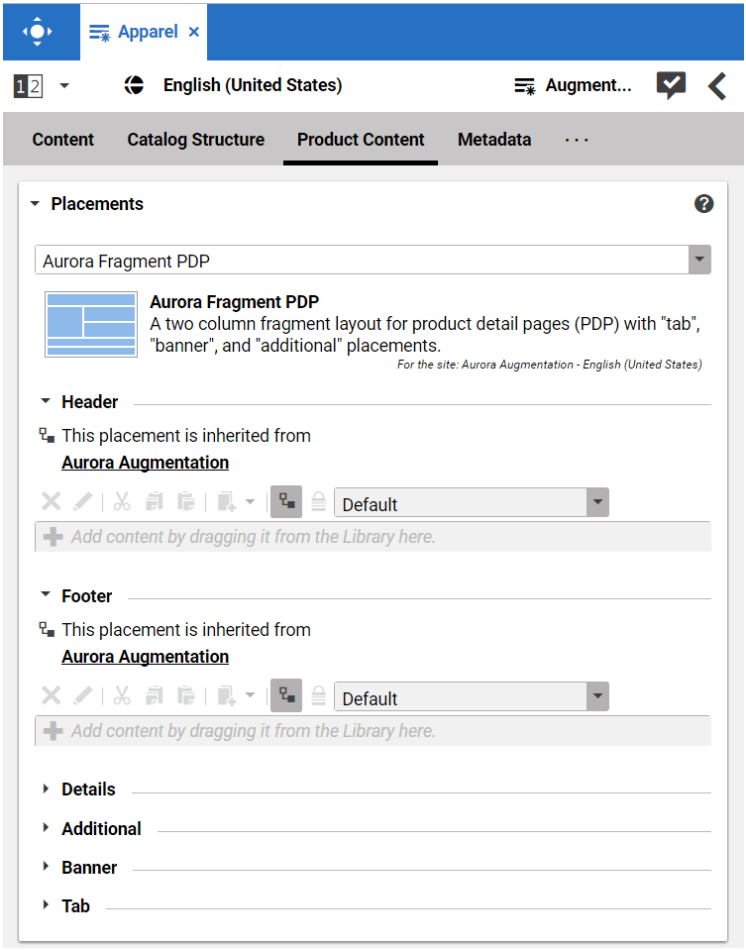

Figure 6.14. Page grid for PDPs in augmented category

Adding CMS Assets to Product Detail Pages

Product detail pages

You can enhance product detail pages with assets from the CoreMedia system by adding the *CoreMedia Product Asset Widget*. Since this area is by default not managed via CMS Cockpit, the *CoreMedia Product Asset Widget* is added directly to the `productDetailsPanel.tag`.

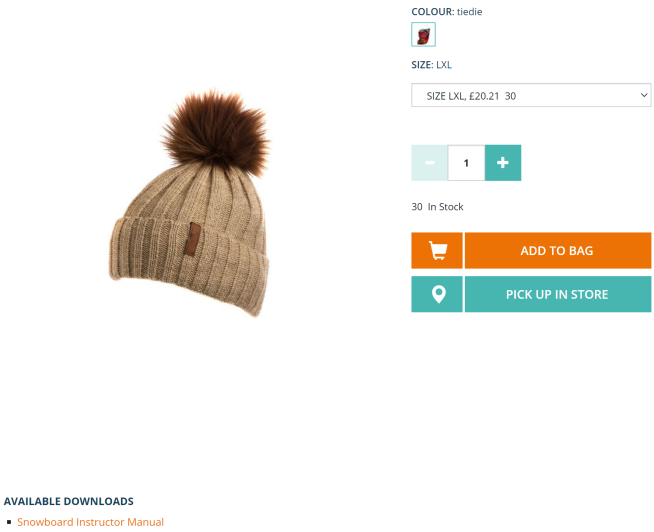

Figure 6.15. Product detail page with CMS assets

Information passed to the CoreMedia system.

The Product ID and orientation are passed to *CoreMedia Content Cloud* in order to locate and layout the assets.

Locating the assets in the CoreMedia system

To find assets for product detail pages, *CoreMedia Content Cloud* searches for the picture content items which are assigned to the given product. These items are then sorted in alphabetical order. See Section 6.6, "Advanced Asset Management" in *Blueprint Developer Manual* for details.

6.4.5 Adding CMS Content to Non-Catalog Pages (Other Pages)

Non Catalog Pages (Other Pages)

Non-catalog pages (Augmented Pages) like 'Contact Us', 'Log On' or even the homepage are shop pages, which can also be extended with CMS content. The homepage case is quite obvious. The need to enrich the homepage with a custom layout and a mix of promotional and editorial content is very clear. However, the less prominent pages can also profit from extending with CMS content. For example, context-sensitive hotline teasers, banners or personalized promotions could be displayed on those pages.

You can augment a non-catalog page with *Studio* using the preview's context menu. In the *Studio* preview, navigate to the non-catalog page that should be augmented, right-click its page title and select *Augment page* from the context menu.

You can also perform the following steps using the common content creation dialog:

1. Make sure, that the layout of the page in the commerce system contains the *CoreMedia Content Widget*.
2. Create a content item of type *Augmented Page* and add it to the *Navigation Children* property of the site root content.
3. Enter the ID of the other page below the navigation tab into the *External Page ID* field of the `Augmented Page`.
4. Optional: Set the *External URI Path* if special URL building is needed.

In the following example a banner picture was added to an existing "Contact Us" shop page. To do so, you have to create an *Augmented Page*, select a corresponding page layout and put a picture to the *Header* placement.

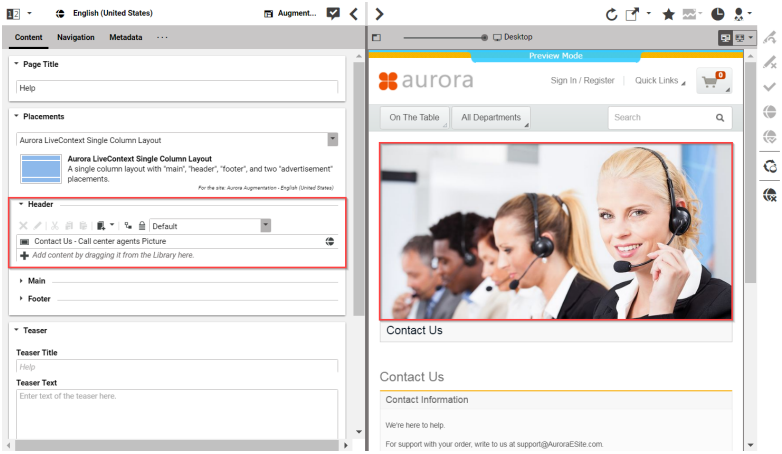

Figure 6.16. Example: Contact Us Pagegrid

Difference between the augmentation of catalog and other pages

The case to augment a non-catalog page with *CoreMedia Studio* differs only slightly from augmenting a catalog page. You use `Augmented Page` instead of `Augmented Category` and instead of linking to a category content, you have to enter a page ID in the *External Page ID* field. The page ID identifies the page unambiguously. Typically, it is the last part of the shop URL path without any parameters.

```
https://<shop-host>/<some-path>/contact-us
```

The URL above would have the page id `contact-us` that will be inserted into the *External Page ID* on the *Navigation* tab. In case of a standard "SEO" URL without the need of any parameters the *External URI Path* field can be left empty.

Figure 6.17. Example: Navigation Settings for a simple SEO Page

NOTE

Be aware that the property *External Page ID* must be unique within all other "Other Pages" of that site. Otherwise, the rendering logic is not able to resolve the matching page correctly. A validator in *CoreMedia Studio* displays an error message, if a collision of duplicate *External Page ID* values occurs. Your navigation hierarchy can differ from the "real" shop hierarchy. There is also no need to gather all pages below the root page. You can completely use your custom hierarchy with additional pages in between, that are set *Hidden in Navigation* but can be used to define default content for are group pages.

Special Case: Homepage

Special Case: Homepage

The home page of the site is the main entry point, when you want to augment a commerce catalog. In the commerce-led scenario, it is a content item of type `Augmented Page`. While in a content-led scenario, it would be of type `Page`.

The *External Page ID* field can be left empty. The homepage is anyway the last instance that will be chosen if no other page can be found to serve a fragment request.

The *External URI Path* field is also likely to remain empty, unless the shop site is to be accessible with an URL, which still has a path component (for example, `../en/aurora/home.html`). But in most cases you wouldn't want that.

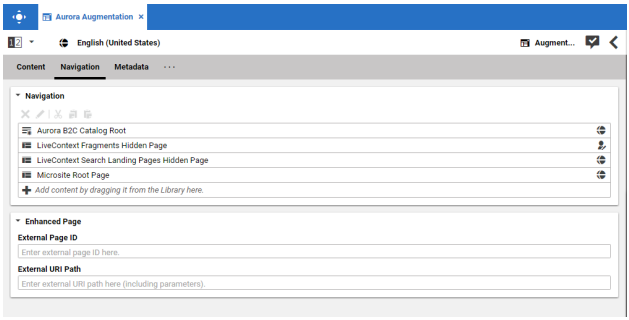

Figure 6.18. Special Case: Navigation Settings for the Homepage

7. Commerce Caching

The CoreMedia system uses caching to speed-up access to various eCommerce entities (e.g. catalogs, categories, products, segments etc.). These entities are cached when they are requested by the CoreMedia system.

Commerce-Hub Cache Infrastructure

Caching of commerce entities is implemented in different layers of the Commerce Hub infrastructure:

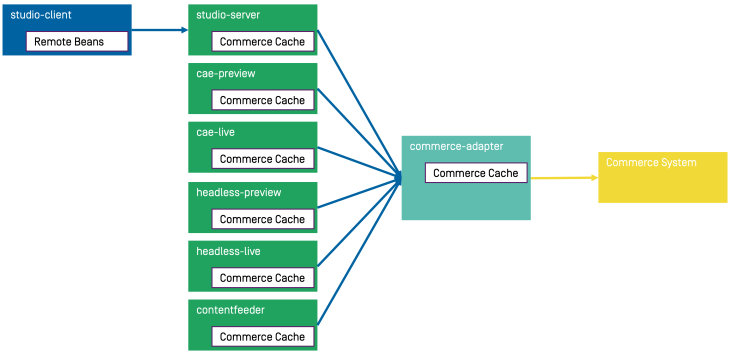

Figure 7.1. Multiple levels of caching

- Caching is implemented in the Commerce Adapter to accelerate access to commerce entities and to avoid heavy traffic on the *SAP Hybris* system due to multiple clients connected to the same system.
- Caching is implemented in the Commerce Adapter client library which is used in Studio, Content Application Engine, *Headless Server* and Content Feeder. This avoids redundant network communication with the Commerce Adapter when accessing commerce entities.
- Caching is implemented in the Studio Client. Commerce entities are loaded as `RemoteBeans` and take part in the *Studio* invalidation mechanism. Updates can be displayed directly if they are recognized.

Java based apps like the Commerce Adapter and Commerce Adapter clients, e.g., Studio, Content Application Engine, *Headless Server*, and Content Feeder, use the `CoreMedia Cache` to cache commerce entities.

NOTE

It is recommended to cache as many commerce entities as possible in the Commerce Adapter for a rather long time and to enable both immediate recomputation and persistent caching of messages as described further down in this chapter. Commerce client apps may then be configured to use rather small caching times and small capacities for commerce entities.

Cache Invalidation by Actuator

Commerce entities are cached for a configurable time span. Changes made to commerce items on the *SAP Hybris* won't be visible until this cache time expires. Two issues arise when only relying on the expiry of cache keys.

First, a proper adjustment of the cache times compromises between two requirements: On the one hand cache times should be short in order to provide an up-to-date system. On the other hand cache times should be long in order to reduce the traffic on the *SAP Hybris*. Second, updating a cache entry requires a controlled invalidation across all relevant caches of the Commerce Hub infrastructure. It is not sufficient to have a cache entry expire in one cache if other caches are still returning the old value.

The Commerce Adapter is the central component that addresses both issues. It allows for a proactive invalidation of cache entries via the `invalidate` actuator and it informs all connected caches about this invalidation. Each client connects as an invalidation observer to the adapter and is notified when a cache entry is to be invalidated. The propagation of the invalidation event ensures that all connected client caches are also updated.

The actuator can be triggered manually or via custom scripts depending on the workflow of the connected *SAP Hybris*. If the update cycles of the *SAP Hybris* are known or if changes can be detected automatically and be used to trigger a script invoking the `invalidate` actuator, then long cache times can be configured to hold commerce entities in the cache as long as possible.

The following figure shows the actuator component in the Commerce Adapter and the direction of events propagating the invalidation.

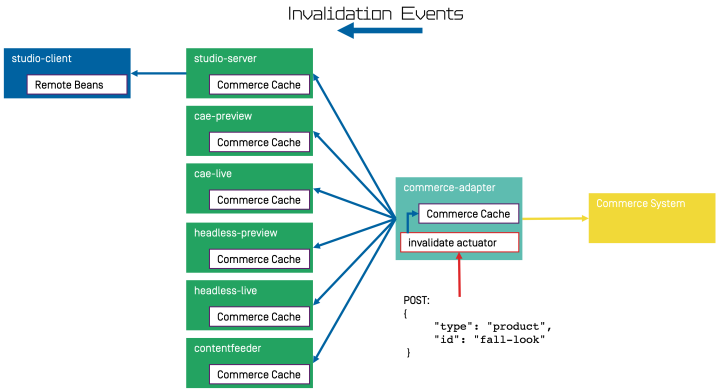

Figure 7.2. Commerce Cache Invalidation

The actuator can be called by using a `POST` request.

```
http://<adapter-host>:<adapter-port>/actuator/invalidate
```

The body is of `JSON` code with 2 mandatory parameters; all must be present but can also be left empty.

`type` The entity type. Can be one of the following values: `catalog`, `category`, `product`, `segment`, `marketing_spot`. Further values can be registered in a project customization. If it is empty, the value remains unspecified and, for example, all items with the given *`type`* are invalidated.

`id` The entity ID. If it is empty, all items of an entity type are invalidated.

Examples:

```
{
  "type": "product",
  "id": "dress-3"
}
```

Invalidate product *`dress-3`* in the Commerce Adapter and in all connected clients.

```
{
  "type": "category",
```

Invalidate category *`dresses`* in the Commerce Adapter and in all connected clients.

```
  "id": "dresses"
}
```

```
{
  "type": "category",
  "id": ""
}
```

Invalidate all categories in the Commerce Adapter and in all connected clients.

```
{
  "type": "",
  "id": ""
}
```

Invalidate all commerce items in the Commerce Adapter and in all connected clients (invalidate all).

> **NOTE**
>
> If a client misses a notification, for example because it is unavailable, it would continue to deliver the old value until the next invalidation comes in, either via actuator or timeout. If there is any suspicion that a cache is out-of-sync, the actuator can be called again.

Invalidation messages from Commerce Adapter to the connected clients can also be turned off using the following configuration property. Then the cache items in the clients disappear only after they have expired. Invalidation messages are turned on by default.

```
entities.send-invalidations=true
```

> **NOTE**
>
> Please note, there is no automatic mechanism involved that is able to trigger the invalidation when a commerce item is changed in the *SAP Hybris*. Such a mechanism can be provided in projects.

Immediate Recomputation of Cache Keys

Commerce entities can be recomputed immediately if they are invalidated in the Commerce Adapter using the following configuration property. This feature is useful to keep the cache of the Commerce Adapter filled with the most frequently used commerce entities. The feature is turned off by default.

```
entities.recompute-on-invalidation=true
```

NOTE

Recomputation is triggered no matter if the invalidation was send from the cache timer or the `invalidate` actuator. Cache keys that are evicted due to space considerations of the cache are not recomputed.

Persisted Caching of gRPC Messages

Incoming and outgoing gRPC messages can be saved to disk to speed-up the Commerce Adapter. This feature allows the Commerce Adapter to read messages from disk when started and to use the restored messages for the following two purposes:

- Immedately respond to requests with the restored response.
- Replay the restored requests so that the cache fills with up-to-date values served by the *SAP Hybris*.

When all requests have been replayed the restored messages are discarded so that responses are only taken from the commerce cache. New incoming requests and their responses are saved to disk using the allowed maximum number of files configured via `entities.message-store.files`. The allowed number of files default to the configured cache capacities as described in the next section. The feature is turned off by default but can be enabled by setting the following configuration property so that it points to an existing directory.

```
entities.message-store.root=file://<PATH_TO_DIRECTORY>
```

WARNING

The directory configured via `entities.message-store.root` must not be a shared directory.

NOTE

The contents of the directory configured via `entities.message-store.root` may be copied so that new Commerce Adapter instances read messages written by another Commerce Adapter.

Cache Configuration of the Commerce Adapter

> NOTE
>
> This chapter applies to the Commerce Adapter, but not to the generic clients like Studio, Content Application Engine, *Headless Server* and Content Feeder.

In order to adjust the cache configuration you can use the following properties for cache capacities and cache timeouts respectively:

- `cache.capacities.*`
- `cache.timeout-seconds.*`

The last part of the configuration property is the config key. Each cache key, e.g. for a product, is using its well known config key (e.g. `product`) to set the capacity and the cache time. The cache capacity denotes the number of commerce entities that the cache can hold of a specific cache class while the cache time specifies the duration that the cache can hold a commerce entity.

There are 2 types of config keys, those that are the same for all different commerce adapters and those that are specific to each vendor adapter. A wide part of the caching is already done within the base adapter library on `Service` level (e.g. the `ProductService`) and does not have to be done in each vendor specific adapter.

Common base adapter config keys:

catalogs	The list of all catalogs for a store referenced by ID and the definition of the default catalog.
catalog	A catalog with its properties and a reference to the root category.
category	A category with its properties. Sub-categories are referenced by ID, as well as products that belong directly to the category. Probably all categories should be cached. They are often used and often traversed. The memory consumption of each cache entry should be small, but can increase if custom attributes are used.
product	Products and variants/SKUs altogether. Please note, there is no distinction between base products and variants/SKUs. Keep this in mind when choosing a capacity value! The memory consumption of each cache entry should be small, but can increase if custom attributes are used.
segments	The list of all customer segments referenced by ID.
segment	A customer segment with its properties. The memory consumption of each cache entry is very small.

Vendor specific config keys:

accesstoken	API access tokens. There is no effect in setting the cache time. The cache time will be computed according to the expiration time of the requested token.
productdata	Used for hierarchical variant/SKU lookups and in services that are not covered by the base adapter caching, like `PriceService`, `LinkService` etc. Please note, there is no distinction between base products and variants/SKUs. Keep this in mind when choosing a capacity value! Each entry consumes ~100kB heap memory.

The default values for the capacity and cache time of each cache key can be found in the in the `application.properties` file in the adapter or consult the Spring Boot environment actuator of the app.

Commerce Cache Configuration of Commerce Adapter Clients

> NOTE
>
> This chapter applies to Commerce Adapter clients like Studio, Content Application Engine, *Headless Server* and Content Feeder.

Every commerce cache class has a default capacity and default cache time configured in the application. Each of the default values can be adapted to the needs of your system environment by overwriting the corresponding properties.

Refer to the Chapter 9, *Commerce Adapter Properties* [98] if you want to adjust the cache configuration for your Commerce Adapter

In order to adjust the cache configuration you can use the following properties [see Section 3.7, "Commerce Hub Properties" in *Deployment Manual* for details] for cache capacities and cache timeouts respectively:

- `cache.capacities.ecommerce.*`
- `cache.timeout-seconds.ecommerce.*`

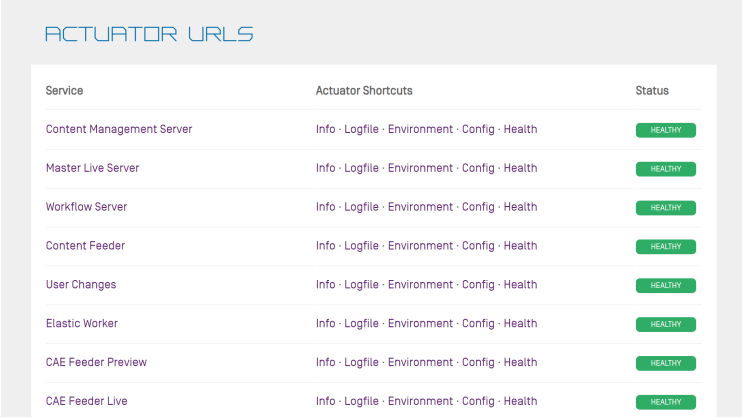

Figure 7.3. Actuator URLs in overview page

You have to replace the trailing "*" with the configuration key of the concrete cache key. You can find the keys and the default values using the Actuator URLs from the default overview page (**https://overview.docker.localhost**) in the default Blueprint Docker deployment. Click the *Config* link and search for the cache.capacities.ecommerce or cache.timeout-seconds.ecommerce prefix.


```
,,
"commerce.hub.cache-com.coremedia.blueprint.base.livecontext.client.config.CommerceAdapterClientCacheConfigurationProperties": {
  "prefix": "commerce.hub.cache",
  "properties": {
    "exposeProxy": false,
    "timeoutSeconds": {
      "product": 3600,
      "category": 3600,
      "catalogsforstore": 86400,
      "linkcategory": 60,
      "linkproduct": 60,
      "linkcontent": 60,
      "linkexternalpage": 60,
      "linkexternalpagenonseo": 60,
      "segment": 5000,
      "segments": 3600,
      "facetsforproductsearch": 300,
```


Figure 7.4. Actuator results for cache.timeout-seconds.ecommerce properties

8. The eCommerce API

The *eCommerce API* is a Java API provided by *CoreMedia Content Cloud* that can be used to build shop applications.

The *eCommerce API* is used internally to render catalog-specific information into standard templates. Furthermore, the Studio Library integration makes use of the API to browse and work with catalog items. If you develop your own shop application you will use the API in your templates and/or business logic (handlers and beans).

Various services allow you to access the eCommerce system for different tasks:

`CatalogService`	This service can be used to access the product catalog in many ways: traverse the category tree, products by category, various product and category searches.
`MarketingSpotService`	This service gives you access to Commerce e-Marketing Spots, a common method to use marketing content (product teasers, images, texts) depending on the customer segments.
`SegmentService`	This service lets you access customer segments, for example, the customer segments the current user is a member of.
`CartService`	This service lets you manage orders.
`AssetService`	This service lets you retrieve catalog assets, for example, product pictures or downloads, that are managed by the CMS. Unlike other services, this service only accesses the CMS.

The Commerce API includes some additional methods that denotes the vendor (the name, the version). In *CoreMedia Studio* there is an option to open a management application for a commerce item (product or category). The required base URL is also set through on the vendor specific connection.

The following key points will give you a short overview of the components that are also involved. They build up an infrastructure to bootstrap a connection to a commerce system and/or perform other supportive tasks.

`Commerce`	This class is the essential part of the bootstrap mechanism to access a commerce system. You

	can use it to create a connection to your commerce system.
`CommerceConnectionInitializer`	This class is used to initialize a request specific commerce connection. The resolved connection is stored in a thread local variable. The `CommerceConnection` class provides access to all vendor specific eCommerce service implementations.
`CommerceBeanFactory`	This class creates `CommerceBeans` whose implementation is defined via Spring. It is also used by the services to respond service calls, for example, instances of `Product` and/or `Category` beans. You can integrate your own commerce bean implementations via Spring (inheriting from the original bean implementation and place your own code would be a typical pattern).
`StoreContextProvider`	This class retrieves an applicable `StoreContext` (the shop configuration that contains information like the shop name, the shop ID, the locale and the currency).
`UserContextProvider`	This class is responsible to retrieve the current `UserContext`. Some operations, like requesting dynamic price information, demand a user login. These requests can be made on behalf of the requesting user. User name and user ID are then part of the user context.
`CommerceIdProvider`	The class `CommerceIdProvider` is used to create `CommerceId` instances. The class `CommerceId` is able to format and parse references to resources in the commerce items. References to commerce items will be possibly stored in content, like a product teaser stores a link to the commerce product.

Commerce beans are cached depending on time. Cache time and capacity can be configured via Spring.

Please refer to the Javadoc of the `Commerce` class as a good starting point on how to use the *eCommerce API*.

9. Commerce Adapter Properties

`hybris.base-path`	
Type	java.lang.String
Default	/ws410/rest
Description	The base path of the REST API ["/ws410/rest"]
`hybris.default-catalog-version-preview`	
Type	java.lang.String
Default	Staged
Description	Default catalog version. On preview-cae and studio the defaultCatalogVersion.preview value is used
`hybris.host`	
Type	java.lang.String
Default	
Description	The full qualified hostname of the Hybris system
`hybris.link.asset-url`	
Type	java.lang.String
Default	
Description	Base URL for assets (e.g. https://shop-hybris.yourdomain.com)
`hybris.link.link-templates`	

Type	java.util.Map<java.lang.String,java.lang.String>
Default	
Description	Map of link templates. Only lookup keys in lowercase and without "_" are valid. Known default lookup keys are defined in StorefrontRefKeysCommerceLed. These patterns can include tokens which will be replaced. These tokens must be well known. The following tokens are predefined: • {storefrontUrl} ... the current store front URL • {storeId} ... the current store id • {locale} ... the current locale in java format, eg. en_US • {language} ... the current language in java format, eg. en • {catalogId} ... the current catalog id • {categoryId} ... the current category id • {productId} ... the current product id • {seoSegment} ... the current seo segment path (can contain path delimiters) • {storefrontUrlPreview} ... the current store front URL • {previewTicket} ... the preview ticket id • {userGroup} ... the current user group, if available

`hybris.link.link-templates.categorylinkfragment`

Type	java.lang.String
Default	<!--CM {"parentCategoryId":"{parentCategoryId}","topCategoryId":"{topCategory-Id}","level":{level},"renderType":"url","categoryId":"{categoryId}","objectType":"category"} CM-->
Description	Used to generate category page links into CoreMedia fragments.

`hybris.link.link-templates.categorypreviewurl`

Type	java.lang.String
Default	{storefrontUrlPreview}/cmpreview?site={storeId}&catalogId={catalogId}&catalogVer-sion={catalogVersion}&ticketId={previewTicket}&userGroup={userGroup}&id={category-Id}&type=category
Description	Used to build the preview URL to a category page.

`hybris.link.link-templates.cmajaxlinkfragment`

Type	java.lang.String
Default	<!--CM {"url":"{url}""renderType":"url","objectType":"ajax"} CM-->
Description	Used to generate ajax urls to CoreMedia contents into CoreMedia fragments.

`hybris.link.link-templates.cmcontentlinkfragment`

Type	java.lang.String
Default	<!--CM {"externalSeoSegment":"{externalSeoSegment}","renderType":"url","objectType":"content"} CM-->
Description	Used to build links to shop pages displaying CoreMedia Articles and Channels into CoreMedia fragments.

`hybris.link.link-templates.cmcontentpreviewurl`

Type	java.lang.String
Default	{storefrontUrlPreview}/cmpreview?site={storeId}&catalogId={catalogId}&catalogVersion={catalogVersion}&ticketId={previewTicket}&userGroup={userGroup}&id={seoSegment}&type=content
Description	Used to build the preview URL to a shop page which displays a CoreMedia content.

`hybris.link.link-templates.externalpagepreviewurl`

Type	java.lang.String
Default	{storefrontUrlPreview}/cmpreview?site={storeId}&catalogId={catalogId}&catalogVersion={catalogVersion}&ticketId={previewTicket}&userGroup={userGroup}&id={pageId}&type=externalpage
Description	Used to build the preview URL to a shop page.

`hybris.link.link-templates.productlinkfragment`

Type	java.lang.String
Default	<!--CM {"productId":"{productId}","renderType":"url","categoryId":"{categoryId}","objectType":"product"} CM-->
Description	Used to build product detail page links into CoreMedia fragments.

`hybris.link.link-templates.productpreviewurl`	
Type	java.lang.String
Default	{storefrontUrlPreview}/cmpreview?site={storeId}&catalogId={catalogId}&catalogVersion={catalogVersion}&ticketId={previewTicket}&userGroup={userGroup}&id={productId}&type=product
Description	Used to build the preview URL to a product detail page.

`hybris.link.link-templates.shoppagelinkfragment`	
Type	java.lang.String
Default	<!--CM {"externalSeoSegment":"{externalSeoSegment}","renderType":"url","objectType":"page"} CM-->
Description	Used to build URLs to shop pages into CoreMedia fragments.

`hybris.link.storefront-url`	
Type	java.lang.String
Default	
Description	The storefront url

`hybris.oauth.client-id`	
Type	java.lang.String
Default	
Description	ClientID used for OAuth2 Authentication with SAP Commerce System. Used to get authorized to access protected OCC API calls.

`hybris.oauth.client-secret`	
Type	java.lang.String
Default	
Description	Password used together with the clientId.

`hybris.oauth.network-address-cache-ttl-in-millis`	
Type	java.lang.Integer
Default	-1
Description	Timeout for DNS cache entries in milliseconds
`hybris.oauth.path`	
Type	java.lang.String
Default	/authorizationserver/oauth/token
Description	Path used to request new OAuth Tokens
`hybris.oauth.port`	
Type	java.lang.Integer
Default	9002
Description	Port used for OAuth token requests
`hybris.oauth.protocol`	
Type	java.lang.String
Default	https
Description	Protocol used for OAuth token requests
`hybris.occ.base-path`	
Type	java.lang.String
Default	/occ/v2
Description	Base path of OCC Rest Services
`hybris.occ.custom-attributes-for`	
Type	java.util.Map<java.lang.String,java.util.List<java.lang.String>>

Default	
Description	Configure attribute names, which are transmitted to the client as customAttributes. The key corresponds to the prefix of the document for json mappgings in lowercase. For example for ProductDocument it is "product". The value is a comma separated list of attributes, which shall be available on the client side via com.coremedia.livecontext.ecommerce.common.CommerceBean#getCustomAttributes. The value is transmitted as String representation of the JSON Object. Example: `hybris.occ.custom-attributes-for.product=metaKeywords,metaDescription`

`hybris.password`

Type	java.lang.String
Default	
Description	The password belonging to the administrative user

`hybris.port`

Type	java.lang.Integer
Default	9001
Description	Port of SAP Commerce REST Services (9001)

`hybris.port-ssl`

Type	java.lang.Integer
Default	9002
Description	Secure port of SAP Commerce REST Services (9002)

`hybris.preview-token-user`

Type	java.lang.String

Default	anonymous
Description	The preview token user passed to the Preview Token Service
`hybris.preview-token-user-group`	
Type	java.lang.String
Default	
Description	The preview token usergroup passed to the Preview Token Service
`hybris.protocol`	
Type	java.lang.String
Default	http
Description	Protocol used for REST communication with SAP Commerce (http)
`hybris.protocol-secure`	
Type	java.lang.String
Default	https
Description	Secure protocol used for REST communication with SAP Commerce (https)
`hybris.user`	
Type	java.lang.String
Default	
Description	The administrative user used to access the SAP Hybris REST Services
`hybris.http-client.invalidation-chunk-size`	
Type	java.lang.Integer
Default	500
Description	Cache invalidation chunk size.

`hybris.http-client.invalidation-max-wait-in-milliseconds`	
Type	java.lang.Integer
Default	0
Description	Maximum wait time for cache invalidation.

`cache.capacities`	
Type	java.util.Map<java.lang.String,java.lang.Long>
Default	
Description	Number of cache entries per cache class until cache eviction takes place. The keys must match the cache classes as defined by the cache keys. Please refer to javadoc of com.coremedia.cache.CacheKey.

`cache.timeout-seconds`	
Type	java.util.Map<java.lang.String,java.lang.Long>
Default	
Description	TTL in seconds until certain cache entries are invalidated.

`entities.circuit-breaker-names`	
Type	java.util.Map<java.lang.String,java.lang.String>
Default	
Description	Mapping of data lookup keys (cache classes) to circuit breaker names. Mapping to 'none' disables circuit breakers for the mapped data lookup keys. Example: Mapping 'product' to 'products' will use a separate circuit breaker named 'products' for product calls. The new circuit breaker can have its own configuration via 'resilience4j.circuitbreaker.configs.products'. Mapping 'product' to 'none' will disable the circuit breaker for product requests.

`entities.default-circuit-breaker-name`	
Type	java.lang.String

Default	base
Description	The default breaker name.

`entities.disable-circuit-breakers`

Type	java.lang.Boolean
Default	false
Description	Disable circuit breakers and cache failed calls in cache class *failed*.

`entities.exponential-backoff.factor`

Type	java.lang.Double
Default	1.5
Description	The factor to be applied to the delay to compute the next delay.

`entities.exponential-backoff.initial-delay`

Type	java.time.Duration
Default	2s
Description	The initial delay of the backoff.

`entities.message-store.files`

Type	java.util.Map<java.lang.String,java.lang.Long>
Default	
Description	The number of request/response pairs to cache persistently. The keys must be valid cache classes as configured for the data lookup service, e.g., catalog, catalogs, category, categories, etc.

`entities.message-store.root`

Type	org.springframework.core.io.Resource
Default	

Description	Root resource to persistently store messages. If this property is not set, no messages will be persisted. Configure a value to enable persistent caching of messages.

`entities.products.register-parent-dependency`

Type	java.lang.Boolean
Default	true
Description	Controls if a parent dependency is registered for a non-base product so that it is invalidated together with its base product.

`entities.recompute-on-invalidation`

Type	java.lang.Boolean
Default	false
Description	Whether to recompute entities proactively on invalidation.

`entities.send-invalidations`

Type	java.lang.Boolean
Default	true
Description	Whether or not to propagate invalidations of entities to the clients.

`metadata.additional-metadata`

Type	java.util.Map<java.lang.String,java.lang.String>
Default	
Description	Map of additional metadata. Can be used as customization hook. All properties starting with "metadata.additional-metadata.*" are transmitted to the generic client on the CMS side.

`metadata.custom-attributes-format`

Type	com.coremedia.commerce.adapter.base.entities.CustomAttributesFormat
Default	

Description	Format of the custom attribute values. The keys are always plain strings. Used to identify the deserialization format on the CMS side.
`metadata.custom-entity-param-names`	
Type	java.util.Collection<java.lang.String>
Default	
Description	List of parameter names, which values need to be transmitted with every entity request from the CMS side.
`metadata.replacement-tokens`	
Type	java.util.Map<java.lang.String,java.lang.String>
Default	
Description	Map of key value pairs. Used as replacement map for example for link building in the generic client on the CMS side.
`metadata.vendor`	
Type	java.lang.String
Default	
Description	Name of the vendor. Used to identify the connected vendor on the CMS side.

Table 9.1. SAP Commerce Adapter related Properties

Glossary

Approve	*CoreMedia CMS* contains a Content Management Environment for content creation and management and a Content Delivery Environment for content delivery. Content has to be published from the Management Environment to the Delivery Environment in order to become visible to customers. Before content can be published, it has to be approved. This way, *CoreMedia CMS* supports the dual control principle.
Blob	Binary Large Object or short blob, a property type for binary objects, such as graphics.
Content Delivery Environment	The *Content Delivery Environment* is the environment in which the content is delivered to the end-user. It may contain any of the following modules: • *CoreMedia Master Live Server* • *CoreMedia Replication Live Server* • *CoreMedia Content Application Engine* • *CoreMedia Search Engine* • *Elastic Social* • *CoreMedia Adaptive Personalization*
Content item	In *CoreMedia CMS*, content is stored as self-defined content items. Content items are specified by their properties or fields. Typical content properties are, for example, title, author, image and text content.
Content Management Environment	The *Content Management Environment* is the environment for editors. The content is not visible to the end user. It may consist of the following modules: • *CoreMedia Content Management Server* • *CoreMedia Workflow Server* • *CoreMedia Importer* • *CoreMedia Site Manager* • *CoreMedia Studio* • *CoreMedia Search Engine* • *CoreMedia Adaptive Personalization* • *CoreMedia Preview CAE*
Content Management Server	Server on which the content is edited. Edited content is published to the Master Live Server.

Content Repository	*CoreMedia CMS* manages content in the Content Repository. Using the Content Server or the UAPI you can access this content. Physically, the content is stored in a relational database.
Content Server	*Content Server* is the umbrella term for all servers that directly access the CoreMedia repository: *Content Servers* are web applications running in a servlet container. • *Content Management Server* • *Master Live Server* • *Replication Live Server*
Content type	A content type describes the properties of a certain type of content. Such properties are for example title, text content, author, ...
Control Room	*Control Room* is a *Studio* plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other *Studio* users.
CoreMedia Studio	*CoreMedia Studio* is the working environment for business specialists. Its functionality covers all the stages in a web-based editing process, from content creation and management to preview, test and publication. As a modern web application, *CoreMedia Studio* is based on the latest standards like Ajax and is therefore as easy to use as a normal desktop application.
Dead Link	A link, whose target does not exist.
Derived Site	A derived site is a site, which receives localizations from its master site. A derived site might itself take the role of a master site for other derived sites.
Elastic Social	*CoreMedia Elastic Social* is a component of *CoreMedia CMS* that lets users engage with your website. It supports features like comments, rating, likings on your website. *Elastic Social* is integrated into *CoreMedia Studio* so editors can moderate user generated content from their common workplace. *Elastic Social* bases on NoSQL technology and offers nearly unlimited scalability.
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.
Folder hierarchy	Tree-like connection of folders, where the root folder forms the origin of the tree.
Home Page	The main entry point for all visitors of a site. Technically it is often referred to as root document and also serves as provider of the default layout for all subpages.
IETF BCP 47	Document series of *Best current practice* (BCP) defined by the Internet Engineering Task Force (IETF). It includes the definition of IETF language tags, which are an abbreviated language code such as en for English, pt-BR for Brazilian Portuguese, or nan-Hant-TW for Min Nan Chinese as spoken in Taiwan using traditional Han characters.

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.
Markup	Marking of parts of a document, structurally (section, paragraph, quote, ...) or with layout (bold, italic, ...).
Master Live Server	The *Master Live Server* is the heart of the *Content Delivery Environment*. It receives the published content from the *Content Management Server* and makes it available to the *CAE*. If you are using the *CoreMedia Multi-Master Management Extension* you may use multiple *Master Live Server* in a CoreMedia system.
Master Site	A master site is a site other localized sites are derived from. A localized site might itself take the role of a master site for other derived sites.
MIME	With Multipurpose Internet Mail Extensions (MIME), the format of multi-part, multimedia emails and of web documents is standardised.
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.
Publication	Creates or updates resources on the Live Server.
Resource	A folder or a content item in the CoreMedia system.
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.
Root folder	The uppermost folder in the CoreMedia folder hierarchy. Under this folder, CoreMedia users can add further folders and content items.
Site	A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In *CoreMedia CMS* a site especially consists of a site folder, a site indicator and a home page for a site. A typical site also has a master site it is derived from.

Site Folder	All contents of a site are bundled in one dedicated folder. The most prominent document in a site folder is the site indicator, which describes details of a site.
Site Indicator	A site indicator is the central configuration object for a site. It is an instance of a special content type, most likely `CMSite`.
Site Manager	Swing component of CoreMedia for editing content items, managing users and workflows. The Site Manager is deprecated for editorial use.
Site Manager Group	Members of a site manager group are typically responsible for one localized site. Responsible means that they take care of the contents of that site and that they accept translation tasks for that site.
Teaser	A short piece of text or graphics which contains a link to the actual editorial content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
Variants	Most of the time used in context of content variants, variants refer to all localized versions within the complete hierarchy of master and their derived sites (including the root master itself).
Version history	A newly created content item receives the version number 1. New versions are created when the content item is checked in; these are numbered in chronological order.
Weak Links	In general *CoreMedia CMS* always guarantees link consistency. But links can be declared with the *weak* attribute, so that they are not checked during publication or withdrawal. Caution! Weak links may cause dead links in the live environment.
Workflow	A workflow is the defined series of tasks within an organization to produce a final outcome. Sophisticated applications allow you to define different workflows for different types of jobs. So, for example, in a publishing setting, a document might be automatically routed from writer to editor to proofreader to production. At each stage in the workflow, one individual or group is responsible for a specific task. Once the task is complete, the workflow software ensures that the individuals responsible for the next task are notified and receive the data they need to execute their stage of the process.
Workflow Server	The *CoreMedia Workflow Server* is part of the Content Management Environment. It comes with predefined workflows for publication and global-search-and-replace but also executes freely definable workflows.
XLIFF	XLIFF is an XML-based format, standardized by OASIS for the exchange of localizable data. An XLIFF file contains not only the text to be translated but also metadata about the text. For example, the source and target language. *CoreMedia Studio* allows you to export content items in the XLIFF format and to import the files again after translation.

Index