CoreMedia Content Cloud

Connector for HCL Commerce Manual

CoreMedia

CoreMedia GmbH

Altes Klöpperhaus, 5. OG

Rödingsmarkt 9

20459 Hamburg

July 11, 2024 (Release 2404)

List of Figures

List of Tables

List of Examples

1. Preface

This manual describes how the CoreMedia system integrates with *HCL Commerce*.

- Chapter 2, *Overview* [6] gives a short overview of the integration.
- Chapter 3, *Customizing HCL Commerce 9.0* [11] describes how you have to configure the commerce system to work with *CoreMedia Content Cloud*.
- Chapter 6, *Commerce-led Integration Scenario* [53] describes the commerce-led scenario and shows how you extend commerce pages with CMS fragments.
- Section 5.1, "Configuring the Commerce Adapter" [41] describes how you connect a CoreMedia web application with an *HCL Commerce* store via the Commerce Adapter.
- Section 6.7, "Link Building for Fragments" [84] describes deep links from fragments of the CMS system to pages of the Commerce system.
- Section 8.3, "Enabling Preview in Shop Context" [102] describes how you activate the preview of Commerce pages in *Studio*.
- Chapter 8, *Studio Integration of Commerce Content* [94] shows the eCommerce features integrated into *CoreMedia Studio*.
- Chapter 9, *Commerce Caching* [119] describes the CoreMedia cache for eCommerce entities.
- Chapter 10, *The eCommerce API* [127] describes the basics of the eCommerce API.
- Chapter 11, *HCL Commerce REST Services used by CoreMedia* [129] lists the REST services of *HCL Management Center* used by CoreMedia.

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*, *HCL 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 *HCL Commerce Server*. You will learn how to add fragments from the CoreMedia system into a *HCL Commerce* generated site, how to access the *HCL Commerce* catalog from the CoreMedia system and how to develop with the *eCommerce API*.

In general *CoreMedia Content Cloud* offers two integration scenarios with *HCL Commerce*: Content-led and commerce-led (see Chapter 6, *Commerce-led Integration Scenario* [53]).

Integration scenarios

- In the commerce-led scenario, pages are delivered by the *HCL 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.
- In the content-led scenario, pages are delivered by both systems, transparent for the user. You can manipulate the navigation through the catalog pages and add complete new navigation paths. You can augment product detail pages with content from the CMS. Categories are rendered from the CAE. However, content and settings are inherited along the catalog category structure.

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 2.1, " Architectural overview of the Commerce Hub " [7] gives a rough overview of the architecture.

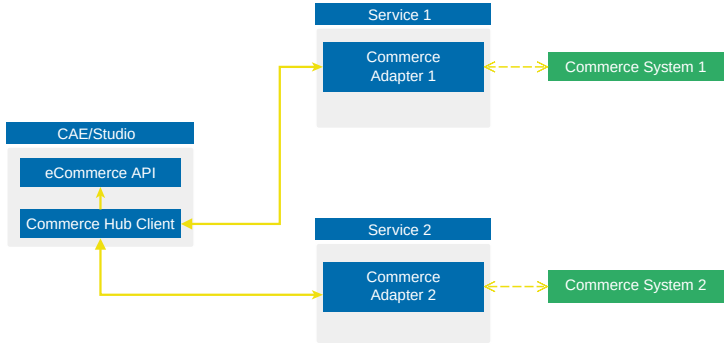

Figure 2.1. Architectural overview of the Commerce Hub

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.

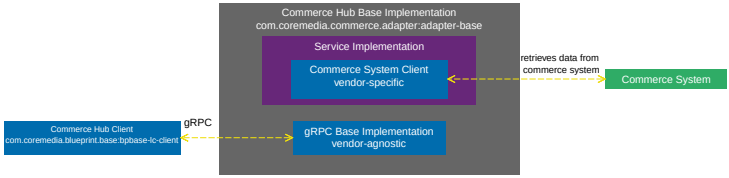

Figure 2.2. More detailed architecture view

Figure 2.2, " More detailed architecture view " [7] 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 2.2, " More detailed architecture view " [7] 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 *HCL Commerce* 9.0

NOTE

Only required when you want to use the eCommerce Blueprint

This section describes how you have to adapt your HCL Rational Application Development (RAD) environment in order to integrate with *CoreMedia Content Cloud*.

In general, certain configuration files need to be adapted in the *HCL Commerce* workspace. Depending on your degree of already applied customization, you might need to merge the provided configuration snippets with your custom code.

This chapter also contains small configurations in the CoreMedia system. These tasks are highlighted in the margin.

NOTE

Deployment to *HCL Commerce* servers, including Staging, Production and Development, is not part of this manual. Please refer to appropriate HCL documentation in the info center at https://help.hcltechsw.com/commerce/9.0.0/install/concepts/v9enhancement.html

The configuration should be performed by an experienced RAD developer.

NOTE

This chapter does not apply to *HCL Commerce* 9.1 either. With *HCL Commerce* 9.1 no customizations are required. Please refer to Chapter 4, *Supporting HCL Commerce 9.1* [38].

Scope of delivery

In order to connect *Content Cloud* with your *HCL Commerce* server you will get the following artifacts from CoreMedia:

- The customization package for the store server (`websphere-commerce-crs` archive). It contains the required customized `crs-web` package to be added to the `CusDeploy` directory of your store server Docker image.
- The customization package for the transaction server (`websphere-commerce-ts` archive). It contains the required customized code to be added to the `CusDeploy` directory of your transaction server Docker image.
- The customization package for the search server (`websphere-commerce-search` archive). It contains the required search configuration and search customization code to be added to the `CusDeploy` directory of your transaction server Docker image.
- The *Sample Data for HCL Commerce* archive (`websphere-commerce-sample-data` archive). The archive contains sample data for the HCL system, which corresponds with the test data for the CoreMedia system in *CoreMedia Blueprint*.

You will find all files on the CoreMedia releases download page at https://releases.coremedia.com/cmcc-12

The customization involves the following aspects:

Installation steps

1. Section 3.1, "Building Custom Docker Image" [14] describes how to deploy the deployable custom packages in your *HCL Commerce*.
2. Section 3.2, "Preparing the RAD Workspace" [15] describes how to apply the required customization to your *HCL Commerce* workspace.
3. Section 3.3, "Copy Libraries" [16] describes how to copy libraries to your *HCL Commerce* workspace.
4. Section 3.4, "Configuring the Search" [17] describes how you have to add the CoreMedia search profile and the Solr index. This enables the CoreMedia system to get additional information necessary for the integration.
5. Section 3.5, "Extending REST Resources to BOD Mapping" [22] describes how you have to configure the mapping of REST resources to the Business Object Document nouns.
6. Section 3.6, "Configuring REST Handlers" [23] describes which REST handlers you have to add and configure.
7. Section 3.7, "Applying Changes to the Management Center" [24] describes the deployment of the Management Center customization.
8. Section 3.8, "Deploying the CoreMedia Fragment Connector" [25] describes the deployment of the fragment connector, which renders content from *Content Cloud* as fragments to *HCL Commerce* pages.
9. Section 3.9, "Customizing HCL Commerce JSPs" [30] describes how to apply customizations to *HCL Commerce* JSPs.
10. Section 3.10, "Deploying the CoreMedia Widgets" [31] describes the deployment of the CoreMedia widgets, which can be used to add content or assets from *Content Cloud* to *HCL Commerce* pages using the fragment connector.

11. Section 3.11, "Setting up SEO URLs for CoreMedia Pages" [35] describes how to set up SEO URLs for CoreMedia Pages.
12. Section 3.12, "Deploying the CoreMedia Catalog Data" [37] describes how to import the CoreMedia catalog content from the Sample archive into the *HCL Commerce*.

NOTE

In the following sections `WCDE_INSTALL` stands for the installation directory of your *HCL Commerce* RAD installation.

3.1 Building Custom Docker Image

Custom Packages

CoreMedia Content Cloud integrates with *HCL Commerce* 9 using the Commerce REST API, therefore you have to deploy the custom packages in the *HCL Commerce*. These custom packages are for the remote store server, the transaction server and the search server.

WARNING

Only follow these instructions when you have no other customizations in your *HCL Commerce Server*. Otherwise, you have to adapt your RAD workspace as described in the other sections of this chapter and create new deployable custom packages.

Deployment Procedure

The following procedure shows how to build the custom Docker images from the customized packages that include the customization code.

1. Create separate `CusDeploy` directories for the remote store server, the transaction server and the search server docker image. For example,
   - `/opt/WebSphere/store/CusDeploy`
   - `/opt/WebSphere/app/CusDeploy`
   - `/opt/WebSphere/search/CusDeploy`
2. Extract every customization packages to the appropriate directory. For example,
   - websphere-commerce-crs archive to `/opt/WebSphere/store/CusDeploy`
   - websphere-commerce-ts archive to `/opt/WebSphere/app/CusDeploy`
   - websphere-commerce-search archive to `/opt/WebSphere/search/CusDeploy`
3. In order to create or update the Dockerfile to build each custom docker image, you need to:
   a. copy`CusDeploy` directory to `/SETUP/Cus` directory.
   b. run applyCustomization.sh script.
4. Stop and remove the running docker containers.
5. Run the docker compose command to build the new custom images. For example, **docker compose -f docker-compose.yml** build

3.2 Preparing the RAD Workspace

REST modules

CoreMedia Content Cloud integrates with *HCL Commerce* using the Commerce REST API, therefore you have to deploy/enable all the REST modules in the *HCL Commerce* workspace for *Content Cloud* to function properly. These modules include: `Rest` and `Search` modules.

Content of the ZIP file

The *HCL Commerce Workspace* archives (download at https://releases.coremedia.com/cmcc-12 contain all new and extended files required to install *Content Cloud* in the *HCL Commerce* RAD workspace. In principle, you can copy the workspaces on top of a fresh Aurora RAD workspace, but only when you do not already have customizations. Make sure you download the Zip archive that matches your WebSphere Commerce version.

WARNING

If you have already customized the Aurora RAD workspace, you cannot copy the CoreMedia Zip content above it, because this would overwrite the former changes. In this case, unzip the files and add and merge the files manually as described in the subsequent sections.

3.3 Copy Libraries

Copy the libraries of the `Code/ts-app/lib` folder of the transaction server archive file into the HCL RAD workspace folder `workspace/WC/lib/`

Make sure that the lc-connector library from the CoreMedia workspace archive are in the corresponding locations of the Stores workspace: `workspace/crs-web/WebContent/WEB-INF/lib/lc-connector-<version>.jar` or `workspace/Stores/WebContent/WEB-INF/lib/lc-connector-<version>.jar`

3.4 Configuring the Search

WebSphere Commerce search provides enhanced search functionality to a store and also influences the search results by using search term association and search-based merchandising rules. In this section you will adapt WebSphere Commerce search to allow *Content Cloud* to leverage these search features. This includes browsing and searching of all catalog assets in *CoreMedia Studio* which is the editorial interface of *Content Cloud*. The configuration consists of two tasks:

1. Add the search profiles
2. Add a new field to the Solr index

3.4.1 Search Customization in *HCL Commerce* 9

Search Customization in *HCL Commerce* 9 take place inside the search server and the transaction server. All the customizations that take place inside the search server (search profiles and search schemas) are provided in the `websphere-commerce-search` archive and all search-related customizations that take place on the transaction server (search index preprocessing) are provided under the `xml/search` folder in the `websphere-commerce-ts` archive.

Search Customization

The project directories and any relevant subdirectories and files are listed in the following table.

Customization	Server (container)	Location
Preprocess configuration files	Transaction server	• `xml\search\dataImport\v3\db2\wc-dataimport-preprocess-custom.xml` • `xml\search\dataImport\v3\db2\wc-dataimport-preprocess-x-final build.xml`
Solr related configuration files	Search server	`search-config-ext\src\index\managed-solr\config\v3\*`

Customization	Server (container)	Location
Search configuration files	Search server	`search-config-ext\src\runtime\config`

Table 3.1. Search customization configuration

3.4.2 Adding Search Profiles

In WebSphere Commerce Search, search profiles (defined in the `wc-search.xml` configuration file) are used to control the storefront search experience at a page level by grouping sets of search runtime parameters. The search runtime parameters set needs to be extended to support the feature set introduced by *Content Cloud*.

The search customization can be found in the `Code/search-app/search-config-ext.jar` of the search server archive file.

Additional information for Commerce Cloud

Content Cloud requires additional information like SEO identifier or pricing which the WebSphere Commerce REST API does not provide by default. Providing this information via REST API is achieved by customizing the `wc-search.xml` configuration file to include that information.

To change/add the value of an existing property in the WebSphere Commerce search configuration file, you have to create a customized version of the search configuration file and add a profile to that file. Follow the steps below to customize the search profiles:

1. Add the search profiles:

   Open the file `WCDE-INSTALL/workspace/search-config-ext/src/runtime/config/com.ibm.commerce.search/wc-search.xml` in the *HCL Commerce Workspace* and copy all the `config:profile` definitions with a name starting with CoreMedia to the corresponding file in your HCL RAD workspace.

2. You have to extend the existing REST API search handlers to provide the additional information now exposed by the search profiles.

   Change the search profile for existing search based REST handlers by creating/updating the file `WCDE-INSTALL/workspace/search-config-ext/src/runtime/config/com.ibm.commerce.rest/wc-rest-resourceconfig.xml` with the corresponding changes from the *HCL Commerce Workspace* archive.

3.4.3 Enabling Dynamic Pricing

Dynamic Pricing supports different prices for different price rules. By default, the feature is disabled.

You activate dynamic pricing by an update of the `STORECONF` table. Set the `wc.search.priceMode` property in the STORECONF table to value "2". See also https://help.hcltechsw.com/commerce/9.0.0/search/concepts/csdsearchstoreconf.html

3.4.4 Customizing the HCL Commerce Solr Index

Content Cloud comes with Solr schema customizations to be applied to the HCL Commerce Solr schema definition.

The schema customization can be found in the search server zip file below `SEARCH-ZIP/Code/search-app/search-config-ext/index/managed-solr/config/v3/CatalogEntry/x-schema.xml` and `SEARCH-ZIP/Code/search-app/search-config-ext/index/managed-solr/config/v3/CatalogGroup/x-schema.xml`.

Adapt the additional fields and field types to the corresponding `x-schema.xml` and `x-schema-field-types.xml` files below `WCDE-INSTALL/workspace/search-config-ext/index/managed-solr/config` to your *HCL Commerce Workspace*.

Read Section 3.4.5, “Adding New PARENT_PARTNUMBER Field to the Solr Index” [19] and Section 3.4.6, “Adding New CM_SEO_TOKEN Field to the Solr Index” [21] to learn more about the specific fields in detail.

3.4.5 Adding New PARENT_PARTNUMBER Field to the Solr Index

Searching HCL Commerce catalog assets in *CoreMedia Studio* is part of the seamless integration experience that *Content Cloud* brings to the table. Almost all the catalog assets are searchable in *Content Cloud* without any need of customization except for the catalog product asset which acts as a template for a group of items (or SKUs) that exhibit the same attributes.

This needs an extra property to explicitly define the hierarchical relationship between the product and its variants in order to make the variants also searchable in *Studio*. This subsection describes all the steps required to introduce the custom *CoreMedia Content Cloud* parent part number field which establishes the relationship between product and variant in WebSphere Commerce.

1. Preprocessing data involves querying WebSphere commerce tables and creating a set of temporary tables to hold the data. The file `Code\ts-app\xml\search\dataImport\v3\db2\CatalogEntry\wc-dataimport-preprocess-parent-partnumber.xml` in the customization package for the transaction server defines a custom preprocessing task for this. The file contains the new temporary table definition, database schema metadata, and a reference to the Java class used in the preprocessing steps for an Oracle database.

   Simply copy the file to the corresponding location in your *HCL Commerce* RAD system. The workspace contains files for other databases which you can use similarly.

2. Extend the HCL Solr configuration files as follows:

   a. Add the following new field to the HCL `x-schema.xml` file `WCDE-INSTALL/workspace/search-config-ext/src/index/managed-solr/config/v3/CatalogEntry/x-schema.xml`

   ```
   <field name="parent_partNumber_ntk"
                type="wc_keywordTextLowerCase" indexed="true"
                stored="true"  multiValued="false"/>
   ```

   Example 3.1. New Solr schema field

   b. Extend the query select and the query from for parent part number using the wc-data-preprocess-x-finalbuild.xml file `WCDE-INSTALL\workspace\WC\xml\search\dataImport\v3\db2\CatalogEntry\wc-data-preprocess-x-finalbuild.xml`.

3. Rebuild the index as described in the HCL documentation at https://help.hcltech-sw.com/commerce/9.0.0/search/tasks/tsdsearchbuildindex.html

WebSphere Commerce search contains a scheduler job (UpdateSearchIndex) to synchronize the catalog changes with the search index. The default update interval is 5 minutes. You can change this default value according to your needs in the WebSphere Commerce Administration Console.

3.4.6 Adding New CM_SEO_TOKEN Field to the Solr Index

Per default HCL behavior, you cannot distinguish the SEO keyword overridden by a store. If you have overridden the SEO keyword in the store, then you will get multiple SEO keywords in the response, without knowing which SEO keyword belongs to which store. To be able to distinguish the SEO keyword you need to extend the Solr field by adding the custom CM_SEO_TOKEN field in the Solr index. This custom CM_SEO_TOKEN field concatenates the store ID and the SEO keyword.

1. Add a preprocessing file for CM_SEO_TOKEN field. The file `Code\ts-app\xml\search\dataImport\v3\db2\CatalogEntry\wc-dataimport-preprocess-cm-seo-token.xml` in the CoreMedia HCL Commerce Workspace defines a custom preprocessing task for this. The file contains the new temporary table definition, database schema metadata and a reference to the Java class used in the preprocessing steps for an Oracle database.

   Copy the file to the corresponding location in your *HCL Commerce* RAD system. The workspace contains files for other databases which you can use similarly.

2. Extend the HCL Solr configuration files by including CM_SEO_TOKEN into the SQL statements as follows:

   a. Add the following new field to the HCL `x-schema.xml` file `WCDE-INSTALL/workspace/search-config-ext/src/index/managed-solr/config/v3/CatalogEntry/x-schema.xml`

   ```
   <field name="cm_seo_token_ntk"
                type="wc_cmKeywordTextLowerCase" indexed="true"
                stored="true" multiValued="true"/>
   ```

   Example 3.2. New CM_SEO_TOKEN Solr field

   b. Extend the query select and the query from for parent part number using the wc-data-preprocess-x-finalbuild.xml file `WCDE-INSTALL\workspace\WC\xml\search\dataImport\v3\db2\CatalogEntry\wc-data-preprocess-x-finalbuild.xml`.

3. Rebuild the index as described in the HCL documentation at https://help.hcltech-sw.com/commerce/9.0.0/search/tasks/tsdsearchbuildindex.html

WebSphere Commerce search contains a scheduler job (UpdateSearchIndex) that synchronizes catalog changes with the search index. The default update interval is 5 minutes. You can change the default value in the WebSphere Commerce Administration Console.

3.5 Extending REST Resources to BOD Mapping

> NOTE
>
> The BOD Mapping only needs to be extended if you do not make use of the search based REST handlers. Per default search based REST handlers are active and there is no need to apply the following.

In order to retrieve more detailed information from the REST handlers, the mapping of the REST resources to the Business Object Document (BOD) nouns has to be extended.

1. To retrieve the SEO identifier of a product, create and edit the file `WCDE_INSTALL/workspace/Rest/WebContent/WEB-INF/config/bodMapping-ext/rest-productview-clientobjects.xml` accordingly to the *HCL Commerce Workspace* archive.
2. To retrieve the SEO identifier of a category, create and edit the file `WCDE_INSTALL/workspace/Rest/WebContent/WEB-INF/config/bodMapping-ext/rest-categoryview-clientobjects.xml` accordingly to the *HCL Commerce Workspace* archive.

3.6 Configuring REST Handlers

Content Cloud requires additional REST handlers and some configuration of existing handlers.

Adding New REST Handlers

CoreMedia eCommerce API comes with additional REST handlers in order to make more data accessible and to provide additional data processing capabilities. The handler classes reside in the `WebSphereCommerceServerExtensionsLogic` module.

You have to add the following handlers:

LanguageMapHandler	The `LanguageMapHandler` returns a list of all available languages of the WebSphere Commerce Server with its mapping on the internal language identifier which is used for certain REST calls.
StoreInfoHandler	The `StoreInfoHandler` returns the storeId and the catalog information of all available stores in the WebSphere Commerce Server.

In order to add the handlers proceed as follows:

1. Add the CoreMedia LiveContext library package to the `Rest` module in your commerce development workspace.
2. Add the following fully qualified names of the handlers to the file `WCDE-INSTALL/workspace/Rest/WebContent/WEB-INF/config/resources-ext.properties` accordingly to the *HCL Commerce Workspace* archive.
3. Add a resource element for each handler to the file `WCDE-INSTALL/workspace/Rest/WebContent/WEB-INF/config/com.ibm.commerce.rest-ext/wc-rest-resourceconfig.xml` accordingly to the *HCL Commerce Workspace* archive.
4. For the CacheInvalidationHandler add the file `WCDE-INSTALL/workspace/WC/xml/config/com.ibm.commerce.catalog-ext/wc-query-CoreMedia-LiveContext.tpl` from the *HCL Commerce Workspace* archive. The file contains a database template to access *HCL Commerce* CACHEIVL table.
5. Adapt all `dbtype` properties to your target database.

3.7 Applying Changes to the Management Center

Studio integrates the Management Center into its GUI. For the integration do as follows:

1. Add the file `WCDE-INSTALL/workspace/LOBTools/WebContent/CoreMediaManagementCenterWrapper.html` from the *HCL Commerce Workspace* archive to the `LOBTools` module.

This file is used from *CoreMedia Studio* for displaying products, categories and e-Marketing Spots in the *HCL Commerce* Management Center. The wrapper uses the original HCL Management Center JSP files embedded and delegates deep links to the appropriate HCL functions.

3.8 Deploying the CoreMedia Fragment Connector

The *CoreMedia Fragment Connector* is the component that connects with *CoreMedia CAE* in order to integrate CoreMedia content fragments in store pages. In order to perform a fragment request, the `LiveContextEnvironment` has to be configured in the `WCDE_installdir/workspace/crs-web/WebContent/WEBINF/web.xml` configuration file, as described below.

Changing the web.xml file

There are different approaches to configure the loading mechanism for properties for the fragment connector. The `LiveContextEnvironment` can load its configuration directly from `web.xml`, from a properties file and from the `STORECONF` table. The default implementation is `PropertiesBasedIBMLiveContextEnvironmentFactory`.

The `PropertiesBasedIBMLiveContextEnvironmentFactory` extends the `IBMLiveContextEnvironmentFactory` and in addition loads properties from a resource file on the classpath. If the resource file cannot be found - or the resource cannot be loaded, it will throw RuntimeExceptions. The location of the properties resource must be given in a servlet context parameter named `livecontext.properties.location`. In the first place this factory tries to get a parameter from `STORECONF` table, in the second place from the properties file and if not found as fallback from `web.xml`.

Other approaches are the following:

- The `DefaultLiveContextEnvironmentFactory` reads the connector properties directly as context parameters directly from the `web.xml`.
- The `IBMLiveContextEnvironmentFactory` extends the `DefaultLiveContextEnvironmentFactory` and can be configured via the `STORECONF` table. If properties are not available in the `STORECONF` table the factory reads directly from the `web.xml` configuration.

The fragment connector is the central component in the commerce-led integration scenario (see Chapter 6, *Commerce-led Integration Scenario* [53]). Configure the fragment connector for example as follows:

1. Add the `LiveContextEnvironment` configuration as shown in `WCDE-INSTALL/workspace/crs-web/WebContent/WEB-INF/web.xml` to the corresponding file in the HCL RAD workspace.

2. In the file `WCDE-INSTALL/workspace/crs-web/WebContent/WEB-INF/coremedia-connector.properties` configure at least the parameter *`com.coremedia.fragmentConnector.liveCaeHost`* with the host URL of your Content Application Engine (CAE). If you use a single commerce system 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 commerce system with separate Production System, you only need to configure one CAE host, each. Find the meaning of all parameters in the list below.

`com.coremedia.fragmentConnector.cookieDomain`	The `cookieDomain` is used when a fragment request is created. All accessible cookies are copied and added to this request using the specified cookie domain. This way it is ensured that the CAE session cookie is detected by the CAE and fragments can be rendered depending on the logged on user. The `cookieDomain` can contain multiple cookieDomains separated by comma.
`com.coremedia.fragmentConnector.unconditionalCookieNames`	A fragment request promotes cookies from the commerce request to the CAE. However, this policy is overruled by other features (for example, the `newPreviewSession` URL parameter). In the `unconditionalCookieNames` property you can specify cookies that are always to be passed with the fragment request. The value must be a comma separated list of cookie names.
`com.coremedia.fragmentConnector.environment`	The optional parameter is used to identify the *HCL Commerce* system that is requesting a fragment from a CAE. It may be used to serve different sites for each commerce system that is connected to a single CMS. The strategy for resolving this parameter is implemented in the class `LiveContextSiteResolver`. The method `findSiteFor(@NonNull FragmentParameters fragmentParameters)` checks if the `environment` parameters has been passed as request matrix parameter. If set (for example: `site:Aurora`), a lookup is made if a site with a matching name and locale exists. If no site is found with the given name, the default lookup strategy, implemented in `findSiteFor(@NonNull String storeId, @NonNull Locale locale)` is used.

com.coremedia.fragmentConnector.liveCaeHost	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`.
com.coremedia.fragmentConnector.previewCaeHost	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 *HCL Commerce* instance being able to access the preview CAE in case of *HCL Commerce* system preview 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 instance for staging and separate production commerce system, you do not need to set this property. If this parameter is not set, the parameter `liveCaeHost` will be used instead.
com.coremedia.fragmentConnector.urlPrefix	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`.
com.coremedia.widget.templates	Configures the template lookup path that is used when rendering CoreMedia Widget includes. Default is `/Widgets-CoreMedia/com.coremedia.commerce.store.widgets.CoreMediaContentWidget/impl/templates/`
com.coremedia.fragmentConnector.defaultLocale	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.
com.coremedia.fragmentConnector.contextProvidersCSV	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 *HCL Commerce* and can be provided to the backend

	CAE via a `ContextProvider` implementation. See Section 6.3, "Extending the Shop Context" [69] for details.
com.coremedia.fragmentConnector.isDevelopment	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.
com.coremedia.fragmentConnector.disabled	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.
com.coremedia.fragmentConnector.connectionTimeout	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 is "10000".
com.coremedia.fragmentConnector.socketTimeout	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 is "30000".
com.coremedia.fragmentConnector.connectionPoolSize	Maximum number of connections used by the fragment connector. Default is 200.
com.coremedia.fragmentConnector.previewCaeAccessTokenHeader	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.
com.coremedia.fragmentConnector.liveCaeAccessTokenHeader	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.
com.coremedia.fragmentConnector.isPrefetchEnabled	If set to true the connector tries to prefetch fragments for the current commerce page.
com.coremedia.fragmentConnector.parameterIncludeList	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`.
`com.coremedia.fragmentConnector.parameterExcludeList`	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.

3.9 Customizing HCL Commerce JSPs

When the `CoreMedia Fragment Connector` has been installed, the `lc:include` tag can be used in any JSPs of the Commerce Workspace to include content from the CoreMedia CMS. See Section 6.2.2, "The CoreMedia Include Tag" [61] for more details.

The *HCL Commerce Workspace* contains web content like JSP and JavaScript files in the `crs-web/<STORE_NAME>` folder. These files are mostly adapted versions of the JSP files of an original HCL RAD workspace. The CoreMedia customizations are highlighted with the following comment lines:

```
<!-- Begin CoreMedia XXX -->
CoreMedia snippet data
<!-- END CoreMedia XXX  -->
```

The corresponding files in the HCL RAD workspace are in the `workspace/crs-web/WebContent/<STORE_NAME>` folder.

How to adapt the files

If you have an Aurora RAD workspace without any customizations, you can copy the *HCL Commerce Workspace* archive content above it. Otherwise, you have to unzip the file and check for each file if you can copy the CoreMedia change into the corresponding file of your HCL RAD workspace.

Example

The CoreMedia archive contains custom `Header.jsp` and `Footer.jsp` files. These JSPs contain some `include` tags, highlighted with comments, to replace the default Aurora store header and footer with CoreMedia page grid placements. The placements contain the navigation and footer elements of the CAE. The original files are located in the folder `workspace/crs-web/WebContent/<STORE_NAME>/Widgets` of the RAD workspace.

In addition, CoreMedia JavaScript and CSS that is used by the CAE must be included in the store front. To do so adapt the CoreMedia specific changes in `WebContent/<STORE_NAME>/Common/CommonJSToInclude.jspf`.

3.10 Deploying the CoreMedia Widgets

The CoreMedia widgets are *HCL Commerce* Composer Widgets. You can use the *CoreMedia Content Widget* to add CoreMedia content fragments to your *HCL Commerce* pages and the *CoreMedia Asset Widget* to add product images to product detail pages.

Prerequisites

In order to use the CoreMedia widgets to embed CoreMedia fragments, the Fragment Connector needs to be deployed before executing these steps.

Register the Widget definition and subscribe your Store to it

See the HCL documentation at https://help.hcltechsw.com/commerce/9.0.0/data/concepts/cmlbatchoverview.html: for more details about data load.

1. Stop the *HCL Commerce* server in the HCL RAD environment.
2. Adapt the database settings in the *Data Load* environment configuration file (`SAMPLEDATA-ZIP\workspace\DataLoad\dataload\common\wc-dataload-env.xml`) from the *CoreMedia Sample Data for HCL Commerce* Zip file to the settings of your WebSphere database.

   You can retrieve your database settings from the HCL RAD environment configuration file `WC/xml/config/wc-server.xml`, at the following XML element:

   ```
   <InstanceProperties>
      <Database>
        <DB>
   ```

   For a DB2 database, the attribute `schema` in `wc-dataload-env.xml` corresponds to the attribute `DBNode` in `wc-server-xml`.

   Find your store identifier in the HCL Management Center in *Store Management*. If you use the default HCL shop, the value is "Aurora".
3. Use the *Data Load* business object configuration files from the *Sample Data for HCL Commerce* ZIP file for registering the widget definition (`workspace\DataLoad\dataload\common\[store_name]\Widget\wc-loader-registerWidgetdef.xml`) and for subscribing the widget definition (`workspace\DataLoad\dataload\common\[store_name]\Widget\wc-loader-subscribeWidgetdef.xml`) where *`store_name`* is the store identifier of your store ("AuroraESite", for instance).

4. Use the CSV input files from the *CoreMedia Sample Data for HCL Commerce* ZIP file for registering the widget definition (`workspace\DataLoad\dataload\common\[store_name]\Widget\registerWidgetdef.csv`) and for subscribing the widget definition (`workspace\DataLoad\dataload\common\[store_name]\Widget\subscribeWidgetdef.csv`).
5. Configure the *Data Load* order configuration file (`wc-dataload.xml`). The *Data Load* file has pointers to the environment settings file, the business object configuration file and the input file.

```
<?xml version="1.0" encoding="UTF-8"?>

<_config:DataLoadConfiguration
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
      "http://www.ibm.com/xmlns/prod/commerce/foundation/config
        ../../../../xml/config/xsd/wc-dataload.xsd"
    xmlns:_config=
      "http://www.ibm.com/xmlns/prod/commerce/foundation/config">

  <_config:DataLoadEnvironment configFile="wc-dataload-env.xml"/>

  <_config:LoadOrder commitCount="100"
                     batchSize="1"
                     dataLoadMode="Replace">
    <_config:property name="firstTwoLinesAreHeader" value="true"/>
    <_config:property name="loadSEO" value="true"/>

    <!-- Configuration for the file to register a widget -->
    <_config:LoadItem
        name="RegisterWidgetDef"
        businessObjectConfigFile=
          "wc-loader-registerWidgetdef.xml">
      <_config:DataSourceLocation
          location="registerWidgetdef.csv"/>
    </_config:LoadItem>

    <!-- Configuration for the file to subscribe a store to a widget -->
    <_config:LoadItem
        name="SubscribeWidgetDef"
        businessObjectConfigFile=
          "wc-loader-subscribeWidgetdef.xml">
      <_config:DataSourceLocation
          location="subscribeWidgetdef.csv"/>
    </_config:LoadItem>
  </_config:LoadOrder>

</_config:DataLoadConfiguration>
```

Example 3.3. wc-dataload.xml

6. Run the *Data Load* utility command syntax with the *dataload.bat* tool which is located in `workspace\bin` of the RAD environment. Give the absolute path to the `wc-dataload.xml` file. The call might look as follows:

```
..\bin\dataload.bat [path_to_your_dataload]\wc-dataload.xml
```

Load the custom access control policies for the CoreMedia Widget

1. Stop the *HCL Commerce* server in the HCL RAD environment.
2. Copy the custom access control policies file `workspace/DataLoad/acp/common/CoreMediaContentDisplay.xml` to the access control policies directory which is located in `xml\policies\xml` of the RAD environment.
3. Run the *ACP Load* utility with the *acpload.bat* tool which is located in `workspace\bin` of the RAD environment. Give the absolute path to the `acp-filename.xml` file. The call might look as follows:

   ```
   ..\bin\acpload.bat [path_to_your_acp_dir]\acp-filename.xml
   ```

   The *ACP Load* documentation can be found here: https://help.hcltechsw.com/commerce/9.0.0/admin/refs/raxacpload.html.

> **NOTE**
>
> The *acpload* tool itself does not report any problems. So, check if the tool created two new XML files with the suffixes `_xmltrans.xml` and `_idres.xml` in `..\xml\policies\xml` for each policy file. Also, look into `..\logs\acpload.log` and `..\logs\messages.txt` for errors.

Add the Widget UI to the Management Center app

1. Copy and merge the `LOBTools` folder content into the `LOBTools` folder of the HCL RAD workspace.

Copy the crs-web Folder and Apply JSP Customizations

Copy and merge the content of the `crs-web/` folder of the *HCL Commerce Workspace* archive into the HCL RAD workspace folder `crs-web/` as described in Section 3.9, “Customizing HCL Commerce JSPs” [30]

Using Placeholder Resolution for Asset URLs

If you have licensed *CoreMedia Advanced Asset Management* you can use placeholders for the CMS host and the store ID in your image URLs. Section 6.6.4.2, “Placeholder

Resolution for Asset URLs" in *Blueprint Developer Manual* describes further details and how you enable placeholder resolution.

Refresh and Rebuild the workspace in Eclipse (RAD)

Now you have to refresh and rebuild the HCL workspace in the HCL RAD environment.

1. Refresh the projects in the HCL RAD system so that the new files are recognized:
   a. Select the `crs-web` project and press **F5**
   b. Select the `WebSphereCommerceServerExtensionsLogic` project and press **F5**
   c. Select the `LOBTools` project and press **F5**
2. Rebuild the `LOBTools`:
   a. Rebuild the `LOBTools` in order to apply the changes to the management Center application.

   This steps might take some time.
3. Republish the *HCL Commerce* server workspace in order to apply the changes to the shop web application. In the server view (bottom left corner) right click on the server instance and select **Publish** from the context menu.

You have updated the Management Center tools and the development workspace and the *HCL Commerce* server has been restarted.

3.11 Setting up SEO URLs for CoreMedia Pages

HCL Commerce contains a default SEO-URL configuration for its shopping pages, such as product detail pages or category landing page. For a seamless integration of CoreMedia content pages like CoreMedia article pages the SEO-URL configuration needs to be extended. The *HCL Commerce Workspace* archive comes with a SEO-URL configuration, which you can apply to your project *HCL Commerce* workspace.

The CoreMedia SEO-URL configuration is required for the usage of CoreMedia Content Display in your *HCL Commerce* environment.

As a prerequisite, SEO URLs require the custom access control policies, installed in Section 3.10, "Deploying the CoreMedia Widgets" [31].

In order to enable the CoreMedia SEO URLs do the following steps:

1. Define the SEO pattern and its mapping for a given StoreName (Aurora or AuroraEsite, for instance). See the HCL documentation at https://help.hcltechsw.com/commerce/9.0.0/seositemap/concepts/csdSEOpatternfiles.html for more details about SEO configuration.

   To do so, copy the SEO pattern file `workspace/crs-web/WebContent/WEB-INF/xml/seo/stores/{StoreName}/SEOURLPatterns-CoreMedia.xml` to your project workspace.

   > NOTE
   >
   > For development, create a file `.reload` (text file) in the same directory and add this line: `reloadinterval = 30`. This will reload the SEO patterns file every 30 seconds.

2. Configure the handling of SEO Requests as follows:

   Extend the existing Spring MVC `views.xml` within the custom stores web archive. The location of the file is `crs-web/WEB-INF/spring/views.xml`

   ```
   <import resource="classpath:/WEB-INF/spring/widgets-views-ext.xml"/>
   ```

   Example 3.4. Import the customized widgets views

3. Check if the copied JSP files already contain the parameter *`externalSeoSegment`*:

The SEO pattern specifies that the path segment after `/cm/` will be mapped to a JSP parameter *`externalSeoSegment`*. Make sure the parameter is actually recognized and prepared to be passed to the lc-include tag as *`lc_externalRef`* parameter.

```
<c:if test="${not empty param.externalSeoSegment}">
        <c:set var="lc_externalRef"
value="cm-seosegment:${param.externalSeoSegment}"/>
        </c:if>
```

Otherwise, check the JSP files in the CoreMedia archive file and copy the settings to the JSPs in the HCL workspace.

4. Check SEO links

   As defined in `SEOURLPatterns-CoreMedia.xml`, the URL pattern *`Core-MediaContentURL`* can be used from within the HCL `wcf:url` tag. You can find the implementation of URL generation for CoreMedia content with this tag in the JSP file `WCDE-ZIP/workspace/crs-web/WebContent/Widgets-CoreMedia/com.coremedia.commerce.store.widgets.CoreMediaContentWidget/impl/templates/Content.url.jsp`. Check that this file is already included in your HCL workspace. Otherwise, copy it.

NOTE

In oder to adapt the predefined URL prefix "/cm" for SEO URLs for CoreMedia Content Pages to your needs, you need to customize

- the *HCL Commerce* SEO URL pattern for CoreMedia Content Pages
- the property `wcs.link.cm-path-identifier` in your Commerce Adapter deployment

3.12 Deploying the CoreMedia Catalog Data

The Sample archive file contains CoreMedia store data that can be used together with the CoreMedia CMS Blueprint demo data. The data can be imported via data load.

Importing Data via Data Load

See the *HCL Commerce* documentation https://help.hcltechsw.com/commerce/9.0.0/data/concepts/cmlbatchoverview.html for more details about data load.

1. Stop the *HCL Commerce* server in the HCL RAD environment.
2. Adapt the database settings in the *Data Load* environment configuration files [`SAMPLEDATA-ZIP\workspace\DataLoad\dataload\common\wc-dataload-env[-<siteName>].xml`] from the *Sample archive* Zip file to the settings of your WebSphere database.

   You can retrieve your database settings from the HCL RAD environment configuration file `WC/xml/config/wc-server.xml`, at the following XML element:

   ```
   <InstanceProperties>
      <Database>
        <DB>
   ```

3. Use the *Data Load* utility to load the data for all sites. Give the absolute path to the `wc-dataload.xml` file, for example `c:\lc-demo-data\workspace\DataLoad\dataload\common\AuroraESite\wc-dataload.xml`.

4. Supporting *HCL Commerce* 9.1

There are two types of systems come with *HCL Commerce* 9.1, the new React-based store with elasticsearch-based search and a legacy JSP-based store with solr-based search.

NOTE

The customization for a legacy JSP-based store with solr-based search in *HCL Commerce* 9.1 is still compatible with the customization for *HCL Commerce* 9.0 (see also Chapter 3, *Customizing HCL Commerce 9.0* [11]). The workspace archive for version 9.0 can also be used for this legacy variant of version 9.1. Please note that this integration scenario requires a project-approval and is not supported by default.

The current support for *HCL Commerce* version 9.1 only applies to the headless scenario. The React-based storefront of the *HCL Commerce* 9.1 requires the CMS content to be provided via the *Headless Server*. There is consequently no need for a *CAE* application that delivers fragments to a storefront. Only in *CoreMedia Studio* the catalog tree from the *HCL Commerce* catalog is visible and *Studio* can be used to define additional content or navigation for your commerce system and to augment category and product pages.

Since the delivering of CMS content is done via the *Headless Server Studio* can show a JSON Preview for each page in the Emerald example site. That JSON can be taken as an example to build a frontend accordingly. (see also Figure 8.5, “Product in tab with JSON preview (HCL Commerce 9.1)” [98] and Figure 8.8, “Category in tab preview (HCL Commerce 9.1)” [99]).

The connection of *Headless Server* and *CoreMedia Studio* to a *HCL Commerce* 9.1 system uses the Commerce Hub and thus the Commerce Adapter. The configuration consists of two parts:

- Configuration of the Commerce Adapter to connect to a *HCL Commerce* 9.1 system is using the same commerce adapter basic configuration with an additional property `wcs.search-profile-prefix` (see also Section 5.1, “Configuring the Commerce Adapter” [41]) but note, there are some properties that are not used for a headless integration. All properties that are needed to build links will be ignored because links are built in the frontend in the headless scenario.
- Settings configuration in *Studio* references the Commerce Adapter endpoint, which *Studio* and *Headless Server* use to communicate via the Commerce Adapter with the

HCL Commerce 9.1 (see also Section 5.2, "Shop Configuration in Content Settings" [43]).

NOTE

For the headless scenario either with elasticsearch-based search or with solr-based search, there is no customization needed on the *HCL Commerce* side. Only standard REST handlers will be called and no search profile adjustment is required. There is no active code deployed to the *HCL Commerce* to fetch content. This needs to be done in the headless storefront client. For more information please refer to the *Headless Server* manual.

NOTE

The `commerce-adapter-wcs` supports *HCL Commerce* 9.1 since version `1.4.0`.

5. Connecting with an HCL Commerce Shop via Commerce Adapter

The connection of your *Blueprint* web applications (*Studio* or *CAE*) to a *HCL 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 *HCL Commerce* system
- Settings configuration in *Studio*. It references the Commerce Adapter endpoint, which *Studio* and *CAE* use to indirectly communicate via the Commerce Adapter with the *HCL Commerce*.

NOTE

Prerequisite

Before connecting the CoreMedia system to the *HCL Commerce* system deploy first the CoreMedia extensions into your *HCL Commerce Workspace* as described in Chapter 3, *Customizing HCL Commerce 9.0* [11].

5.1 Configuring the Commerce Adapter

Configuring the Commerce Adapter

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

The Commerce Adapter comes along with a set of configuration properties. For detailed documentation and defaults see Chapter 12, *Commerce Adapter Properties* [132].

The `commerce-adapter-wcs` provides Spring profiles for the different *HCL Commerce Server* versions that are supported. These profiles configure the suitable URLs that are required to connect to the *HCL Commerce Server*. To use these profiles, set the `wcs.host` property and activate the Spring profile `wcs-[VERSION]` when starting the adapter application.

Starting the Commerce Adapter

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

Prerequisites to be installed:

- Maven
- Docker
- Docker Compose (optional)

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

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

To run the `commerce-adapter-wcs` 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 - 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-wcs \
  --publish 44365:6565 \
  --publish 44381:8081 \
  [--env ...|--env-file ...|--volume] \
  coremedia/commerce-adapter-wcs:${ADAPTER_VERSION}
```

To run the `commerce-adapter-wcs` Docker container with the CoreMedia CMCC Docker environment, add the `commerce-adapter-wcs.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-wcs.yml
WCS_HOST=...
...
```

The `commerce-adapter-wcs` 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*.

NOTE

For *HCL Commerce* 9.1, it is recommended to use the following search profile prefix:

- `HCL` for the headless integration with elasticsearch-based search
- `IBM` for the headless integration with solr-based search

For earlier *HCL Commerce* versions, it is recommended to use the default search profile prefix `CoreMedia`.

5.2 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 item linked with the root channel. This is the perfect place to make these settings.g

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

Name	Type	Description	Example	Required
`endpoint`	String Property	Host and Port of the Commerce Adapter.	wcs-commerce-adapter:6565	true (if endpointName is not set)
`endpointName`	String Property	The endpoint name to lookup the Spring gRPC service configuration .	wcs	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-US	false
`currency`	String Property	The displayed currency for all product prices.	USD	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	Store id that is used to access the store. If the `StoreInfoHandler` is deployed on the *HCL Commerce Server* side, it can be retrieved automatically by mapping an existing store name.	700012345678	false
`storeConfig.name`	String Property	Store name that is used to access the store. If the `StoreInfoHandler` is deployed on the *HCL Commerce Server* side, the name is used to retrieve the store id.	AuroraESite	true
`catalogConfig`	Struct Property	Struct property containing catalog configuration. In a multi-catalog scenario additional catalog configurations can be added via the `additionalCatalogConfigs` configuration. The catalog behind the `catalogConfig` entry is treated as default catalog then.		true
`catalogConfig.id`	String Property	Catalog id that is used to access the catalog. If not set, the ID of the default catalog is used.	300012345678	false
`catalogConfig.name`	String Property	Catalog name that is used to display a catalog name (e.g in the *Studio* library). If not set, the ID of the default catalog is used. setting.	AuroraESite-SalesCatalog	false

Name	Type	Description	Example	Required
`catalogConfig.alias`	String Property	Catalog alias that is used in content to store links to catalog items. The alias *`catalog`* is reserved and used for the default catalog. If not set, the string *`catalog`* is used.	master	false
`additionalCatalogConfigs`	Struct List	List of additional catalog configurations used for multi-catalog scenario. Each entry should provide the properties described earlier for the `catalogConfig` entry. The property `alias` and at least one of `id` or `name` must be defined.		
`customEntityParams`	Struct Property	Site specific custom entity parameters, which are attached to the communication with the commerce adapter. See Section 5.4, "Configuring Custom Entity Parameters" [51] for more information.		false. If not set, no site specific custom entities will be used.

Table 5.1. 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.

Configuring Multiple Catalogs

By default, *CoreMedia Studio* only shows the default catalog of the *HCL Commerce* system. However, you can configure multiple catalogs which can be defined in *Studio* via a struct list property `additionalCatalogConfigs` below the `commerce` struct. Proceed as follows:

1. Open the `LiveContext Settings` content in `Sites/<Site Name>/<Locale Country>/<Locale Language>/Options/Settings` (for example `Sites/Aurora Augmentation/United States/English/Options/Settings`).
2. If it does not exist, add a Struct List property named *additionalCatalogConfigs* below the `commerce` Struct to the *Settings* field.
3. For each catalog add a Struct item to the Struct List property `additionalCatalogConfigs`. Each entry should at least define an `alias` and an `id` or `name` property. The property `alias` is used to link to catalog items internally and shouldn't be changed anymore. The property `id` corresponds to the id of the catalog in the commerce system. The property `name` corresponds to the name of the catalog in the commerce system.

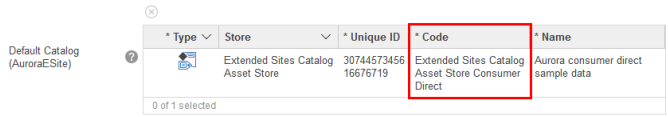

Figure 5.1. Catalog code in commerce system

For backward compatibility, the default catalog needs to have the alias "catalog".

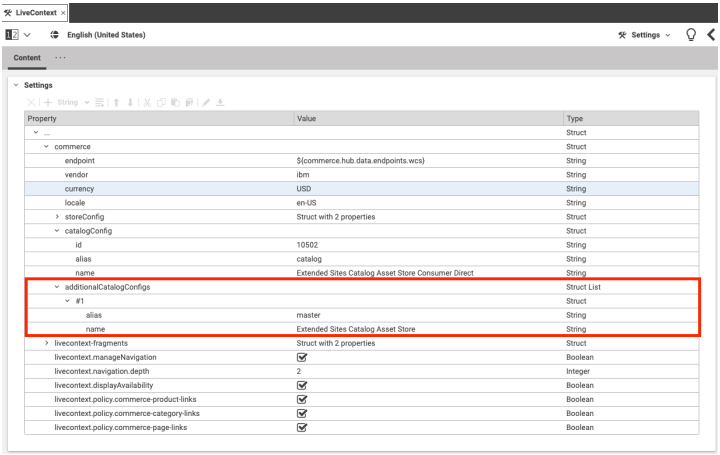

Figure 5.2. Catalog settings

Name	Type	Value
The alias for the catalog. You can freely define a name which must be alphanumeric including '_' and '-'. Only the default catalog requires the alias "catalog".	String	The *HCL Commerce* code of the catalog

Table 5.2. Catalog aliases

CAUTION

The defined aliases are then used as part of internal IDs which are persisted in the system.

Therefore, choose the alias wisely before the multi-catalog feature is used. Changing the alias afterward would require some cumbersome data migration.

Enabling Dynamic Pricing

Dynamic price rendering is disabled by default. If this feature is not used on *HCL Commerce* side, then it is not necessary to turn it on on CMS side. It avoids an additional call to *HCL Commerce* that is not needed in such a scenario.

But if you use personalized price rules in *HCL Commerce* then it is necessary to switch this feature on.

Name	Type	Description	Example	Required
`config.id`	String Property	The configuration ID defined in Spring configuration	myStore	true
`dynamicPricing.enabled`	Boolean Property	Personalized product prices enabled	true	false

Table 5.3. Currency configuration

Please see Section 5.1, “Configuring the Commerce Adapter” [41] to get the information how the dynamic prices can be switched on on *HCL Commerce* side.

5.3 Check if everything is working

Prerequisites

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

Check the Studio - *HCL Commerce* REST Connection

1. Open *Studio*, select the "Aurora Augmentation - English (United States)" site, open the Library. If necessary, switch the Library to browse Mode.
2. In the repository tree view, locate a node named *AuroraESite*. This is the entry point to browse the connected *HCL Commerce* product catalog.
3. Browse the catalog in studio and check if everything works as expected. Section 8.1, “Catalog View in CoreMedia Studio Library” [95] 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 5.2, “Shop Configuration in Content Settings” [43].
- Check if the REST connector is configured correctly (see Section 5.1, “Configuring the Commerce Adapter” [41]). Check for example, if the deployment property `wcs.host` is configured correctly.

Check Studio - *HCL Commerce* Preview Integration

1. Open the Homepage of the "Aurora Augmentation - English (United States)" site in Studio

   The *HCL Commerce* 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 `wcs.link.storefront-url` is configured correctly for Commerce Adapter.

Check Fragment Connector

1. Open the Aurora Augmentation - English (United States) homepage and check if CoreMedia Demo content is displayed.

If errors occurred or no CoreMedia Content is displayed

- Check for errors in the *HCL Commerce* log and the Preview *CAE* log and the Commerce Adapter log.
- Check in *Management Center* if the homepage has content slots containing *CoreMedia Content Widgets* or if render templates contain a `lcinclude` tag.

5.4 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.

6. 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 8.5, "Augmenting Commerce Content" [107] and Chapter 6, *Working with Product Catalogs* in *Studio User Manual* for more details about the *Studio* usage for eCommerce.

- Section 6.1, "Commerce-led Scenario Overview" [54] gives an overview over the request flow in the commerce-led integration scenario.
- Section 6.2, "Adding CMS Fragments to Shop Pages" [56] 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 6.3, "Extending the Shop Context" [69] describes how you extend the shop context that is delivered to the CMS.
- Section 6.4, "Solutions for the Same-Origin Policy Problem" [71] describes how the same-origin policy problem has been solved for the CoreMedia solution.
- Section 6.5, "Caching In Commerce-Led Scenario" [74] describes the caching in the commerce-led scenario.
- Section 6.6, "Prefetch Fragments to Minimize CMS Requests" [79] describes how to prefetch fragments in the commerce-led scenario.

NOTE

This chapter does not apply to *HCL Commerce* 9.1. More information on the Headless Integration Scenario can be found in Chapter 4, *Supporting HCL Commerce 9.1* [38].

6.1 Commerce-led Scenario Overview

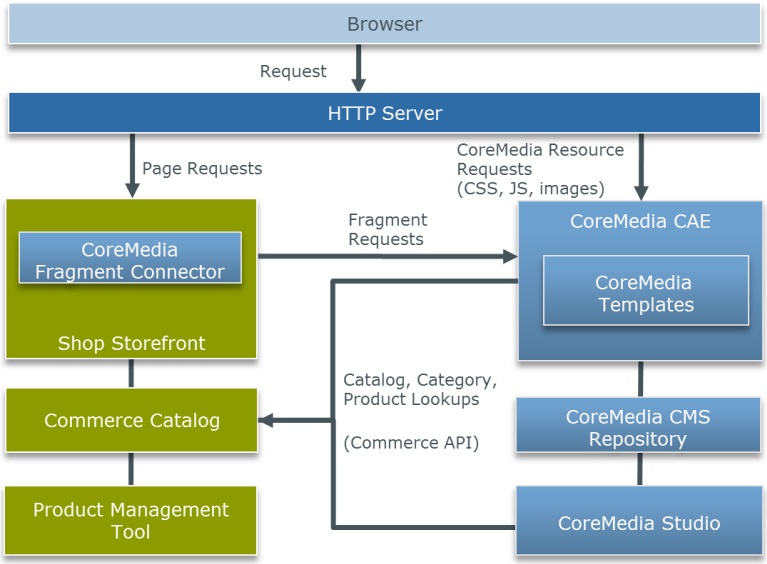

Figure 6.1. Commerce-led Architecture Overview

Figure 6.1, "Commerce-led Architecture Overview" [54] 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:

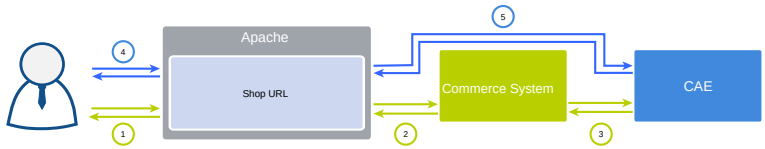

Figure 6.2. Commerce-led Request Flow

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 6.4, “Solutions for the Same-Origin Policy Problem” [71].

6.2 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.

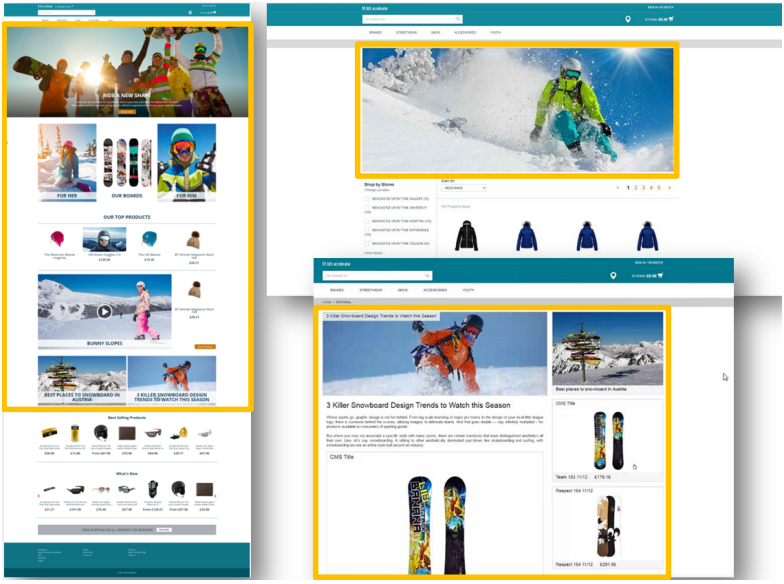

Figure 6.3. Various Shop Pages with CMS Fragments

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 6.2.1, "CoreMedia Widgets" [57] and Section 6.2.2, "The CoreMedia Include Tag" [61] 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 8.5, "Augmenting Commerce Content" [107] describes the procedure.

6.2.1 CoreMedia Widgets

Adding the CoreMedia Content Widget

On the *HCL Commerce* side it is necessary to define slots where the CMS content can be displayed. This is normally done by adding the *CoreMedia Content Widgets* to an *HCL Commerce* page layout.

Using the lc:include tag

In other cases, where a widget cannot be used, it can also be achieved by directly adding an `lc:include` tag into a JSP within the *HCL Commerce* workspace. This is typically done in advance during the project phase. Later, editors will only deal with `Augmented Categories` and `Augmented Pages` that they can edit and preview via *CoreMedia Studio*.

The content that is shown in the *CoreMedia Content Widget* is taken from a placement in the augmented content item, whose name corresponds with the name set in the widget. See Figure 6.4, "Connection via placement name" [58] for an example. Note, that the name of the placement shown in the Studio form is only a localized label. The name in the *Content Widget* must match with the technical name in the page grid definition. If the widget defines no placement, the full page grid is taken.

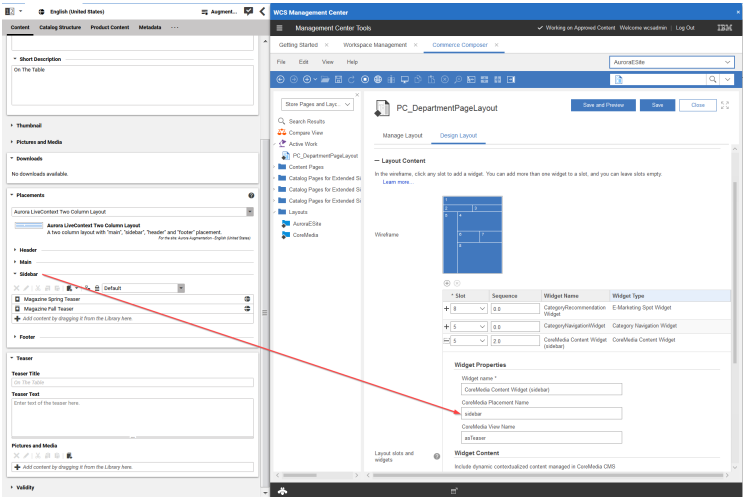

Figure 6.4. Connection via placement name

The CoreMedia widgets are *HCL Commerce Composer Widgets* that display content or assets from the CMS on any page managed through the *HCL Commerce Composer*. After the CoreMedia widgets have been deployed on the commerce side (see Section 3.10, "Deploying the CoreMedia Widgets" [31]), two CoreMedia widgets are available in the *HCL Commerce Composer*:

- CoreMedia Content Widget
- CoreMedia Asset Widget

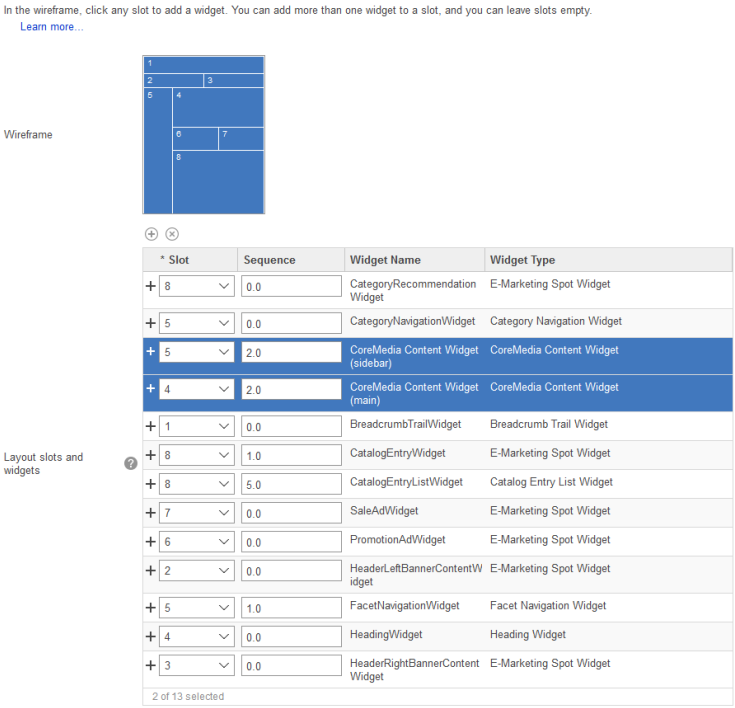

Figure 6.5. CoreMedia Widgets in Commerce Composer

Technically, the CoreMedia Widgets use the `lc:include`. See Section 6.2.2, "The CoreMedia Include Tag" [61] for a description.

The CoreMedia Content Widget

You can use the *Content Widget* like any other *Commerce Composer Widget*. It has the following configuration options:

Option	Description
Widget name	The widget name.
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

Option	Description
	will be shown at the associated position of the page grid and subsequently in the layout of this CoreMedia Content Widget.
CoreMedia View Name	The view of the placement as defined in CoreMedia CMS. Each placement can be rendered with a specific view which needs to be predefined to handle the content in a placement.

Table 6.1. CoreMedia Content Widget configuration options

The CoreMedia Product Asset Widget

> NOTE
>
> The Product Asset Widget is part of the *CoreMedia Advanced Asset Management* module described in Section 6.6, "Advanced Asset Management" in *Blueprint Developer Manual* . This module requires a separate license.

You can use the *CoreMedia Product Asset Widget* like any other *Commerce Composer Widget*. It has the following configuration option:

Option	Description
Display Pictures and Videos	If checked, a picture gallery is rendered from CMS pictures and videos that are associated with the product.
Orientation	The orientation of the pictures (only relevant if pictures are included). The possible values are *Square* and *Portrait*
Include Downloads	If checked, an *Additional Downloads* list is rendered from CMS *Download* content item that are associated with the product.

Table 6.2. CoreMedia Product Asset Widget configuration options

6.2.2 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" %>
<lc:include
        storeId="${WCParam.storeId}"
        locale="${WCParam.locale}"
        catalogId="${WCParam.catalogId}"
        productId="${WCParam.productId}"
        categoryId="${WCParam.categoryId}"
        placement="${param.placement}"
        view="${param.view}"
        externalRef="${WCParam.externalRef}"
        exposeErrors="${not empty WCParam.externalRef
                       && empty WCParam.categoryId
                       && empty WCParam.categoryId}"
        httpStatusVar="fragmentHttpStatus"/>
```

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 Blueprint side an *Augmented Page* will be retrieved to serve the fragment HTML. The transmitted page ID parameter must match the *External Page ID*

Parameter	Description
	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 6.3. 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 6.4. 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 6.1, “Default fragment handler order” [66] 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 6.1. 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 6.5. 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.

For customers using *HCL Commerce* the property `metadata.additional-metadata.allow-tech-ids=true` is set by default.

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 6.3, “Extending the Shop Context” [69].

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 3.8, “Deploying the CoreMedia Fragment Connector” [25]) 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 *HCL Commerce* reference workspace the usage of the *`exposeErrors`* parameter is demonstrated in the `CommonJSToInclude.jspf` template. 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 WCParam.externalRef && empty WCParam.productId &&
 empty WCParam.categoryId}"
```

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>
```

6.3 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 6.2. ContextProvider interface method

Such implementations of the `ContextProvider` interface must be provided with the *HCL Commerce* workspace. This is typically done below the `WebSphereCommerceServerExtensionsLogic` directory of the your *HCL Commerce* project workspace. Such context provider implementations will use the *HCL Commerce* 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 creates and stores a `Context` object in the request. You can access the `Context` object via `LiveContextContextHelper.fetchContext(HttpServletRequest request)`.

Example 6.3. Access the Shop Context in CAE via Context API

6.4 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:

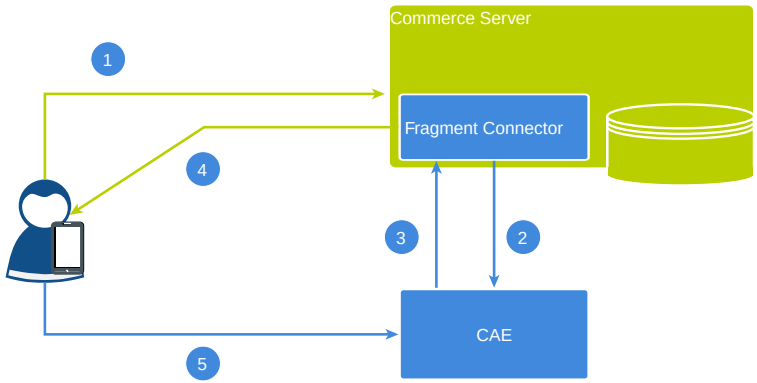

Figure 6.6. Cross Domain Scripting with Fragments

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.

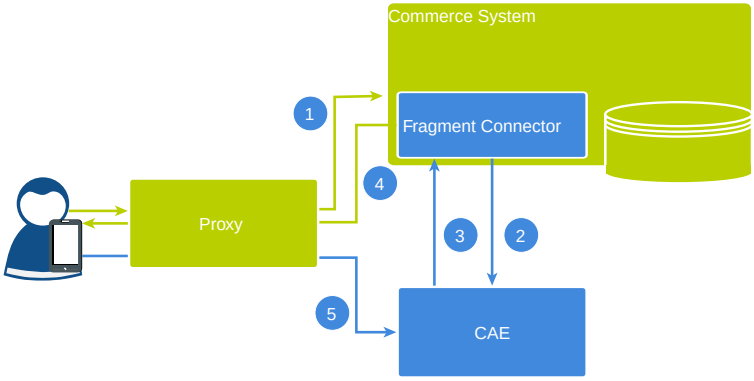

Figure 6.7. Cross Site Scripting with fragments

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;
    }
}
```

6.5 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 (CMSelectionRules 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

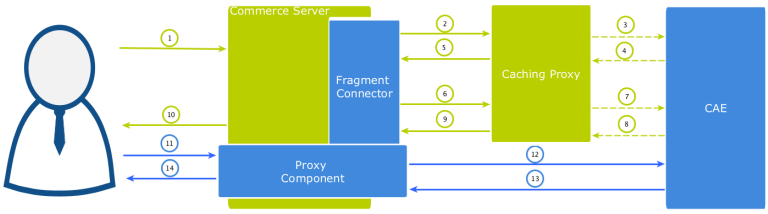

Figure 6.8. Example request flow

Figure 6.8, "Example request flow" [75] 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 *HCL Commerce Workspace* and will be installed with all other CoreMedia customizations. Technically it is a Struts Action that uses the request mapping `/CmDynamic` with a `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 6.7.2, “How fragment links are build” [85] for more information about link building on the commerce site.

```
<div class="cm-fragment"
data-cm-fragment="/webapp/wcs/stores/servlet/CmDynamic?catalogId=3074457345616676719&langId=-1
&storeId=715838084&urlLangId=&url=%2Fblueprint%2Fservlet%2Fdynamic%2Fplacement%2Fp13n%2Faurora%2F136%2Fplacement%2F
hero%3FtargetView%3D%255Blandscape%255D%26fragmentContext%3D%2F715838084%2Fen-US%2F
params%3BcatalogId%253D3074457345616676719%3Bplacement%253Dhero%3BpageId%253Dauroraesite"></div>
```

Example 6.4. AJAX Stub

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

```
/blueprint/servlet/dynamic/placement/p13n/aurora/136/placement/hero?targetView=%5Blandscape%5D
&fragmentContext=/715838084/en-US/params;catalogId%3D3074457345616676719;placement%3Dhero;pageId%3Dauroraesite
```

Example 6.5. 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 6.3, “Extending the Shop Context” [69]. for details.

Double Click Handler

HCL by default enables a so called DoubleClickHandler that avoids the same requests being processed in parallel. The purpose of double-click handling in WebSphere Commerce is to prevent processing the same request twice to ensure data integrity within the system. This feature prevents multiple personalized fragments on a page with dynamic Ajax loading. To use dynamic Ajax loading for multiple personalized fragments on one page set `EnableDoubleClickHandler` property for the Instance in *HCL Commerce* Configuration File to `false` or exclude the CoreMedia `CmDynamic` command in the `DoubleClickMonitoredCommands` section.

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.

6.6 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 6.9, “Multiple Fragment Requests without Prefetching” [79]). 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.

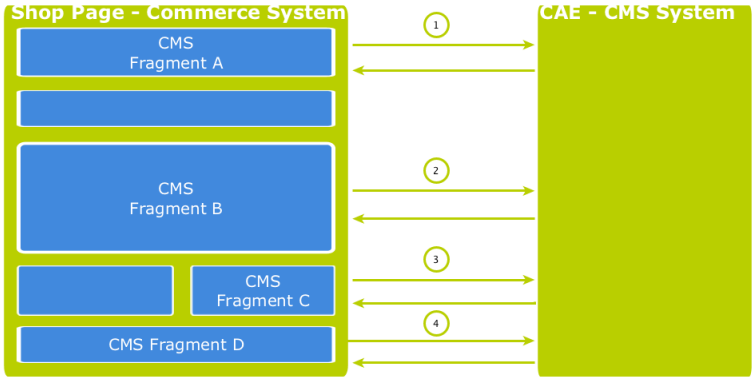

Figure 6.9. Multiple Fragment Requests without Prefetching

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 6.10, "LiveContext Settings: Prefetch Views per Placement" [81] 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 6.11, "LiveContext Settings: Prefetching Additional Views" [82] 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 6.10. 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 6.11. 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 6.2, "Adding CMS Fragments to Shop Pages" [56] 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 *HCL Commerce*

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

6.7 Link Building for Fragments

Overview

If you include CoreMedia fragments into *HCL Commerce* pages, these fragments might contain links to commerce pages; a link to an Augmented Category, for example. Depending on the scenario that you use, this link should led to a page rendered by the CAE (content-led scenario) or to a page rendered by the *HCL Commerce* (commerce-led scenario). The latter is named "deep link".

6.7.1 Configuring Deep Links

A use case for deep links might be the following: You have an existing eCommerce solution with carefully styled category and product pages. While you want to switch to *Content Cloud* in order to enhance your site with editorial content, there is no need to port the commerce pages to *Content Cloud*. Instead, you want to reuse the existing pages (possibly enhanced with *Content Cloud* fragments).

Properties for deep link activation

Content Cloud supports two settings to switch to deep links for categories and products:

- `livecontext.policy.commerce-product-links`
- `livecontext.policy.commerce-category-links`

Default setting "true"

The settings are at the root channel of each site. The default setting is `true`, which means that the CAE creates deep links to the product or category pages of the *HCL Commerce*. However, for links to other content types, such as `HTML`, `CSS` or `JavaScript`, links to the CAE will be generated. Also, URLs to dynamic resources (`UriConstants.Prefixes.PREFIX_DYNAMIC`) won't be converted to JSON. See Section 8.3, "Enabling Preview in Shop Context" [102] to learn how to enable the preview for *HCL Commerce* pages in *Studio*.

The settings are evaluated by the `LiveContextPageHandlerBase` and its subclasses.

Link building and request handling

If a setting is `true`, the corresponding `@Link` method creates links to *HCL Commerce*, so there is no need for a matching `@RequestMapping` method. If it is `false`, the `@Link` method creates CAE links. So you must keep the according `@RequestMapping` method in sync with changes to the URL pattern and provide (or customize) the `ProductPageHandler` or `ExternalNavigationHandler` classes. See also the Section 4.3, "The CAE Web Application" in *Content Application Developer Manual* for request handling and link building.

6.7.2 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.

```
http://localhost/webapp/wcs/stores/servlet/CoreMediaContentURL?
      storeId=10202&externalSeoSegment=spring-salads-1888&
      urlRequestType=Base&langId=-1&catalogId=10051
```

Example 6.6. Commerce URL

NOTE

The SEO feature has not been configured for this example, otherwise the `externalSeoSegment` value would be used to render a SEO friendly URL.

Other templates are located in the folder `workspace\Stores\WebContent\Widgets-CoreMedia\com.coremedia.commerce.store.wid`

`gets.CoreMediaContentWidget\impl\templates` by default. The path is configurable via property `com.coremedia.widget.templates` in `core media-connector.properties`. New templates can be added by extending the `CommerceLinkResolver` in the *Blueprint* workspace. Custom object types can be added, depending on the content type of the content or its property values. Also, additional rendering types can be defined for an object type. Using this templating mechanism, it is possible to support different layouts for content depending on its context.

7. Content-led Integration

In the content-led scenario, *HCL Commerce* system and CMS system are equal partners. It is possible, that the CoreMedia CAE delivers all content to the customer, while augmenting the pages with content, such as prices, from the commerce system.

- Section 7.1, "Content-led Integration Overview" [88] gives an overview over the request flow in the content-led scenario.
- Section 7.2, "Status Synchronization in the Content-led Integration Scenario" [90] describes how the user state is synchronized between the commerce system and CMS systems.

NOTE

This chapter does not apply to *HCL Commerce* 9.1. More information on the Headless Integration Scenario can be found in Chapter 4, *Supporting HCL Commerce 9.1* [38].

7.1 Content-led Integration Overview

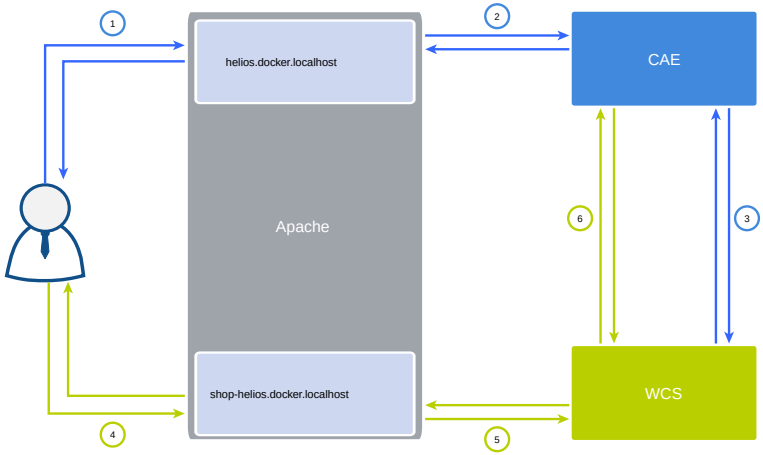

Figure 7.1. Content-led integration scenario

The most obvious difference to the commerce-led scenario in the content-led scenario is the presence of a second virtual host, that separates both systems, the *CAE* and the commerce system, clearly from one another. Here the *CAE* is the fully equal partner of the commerce system with the potential to become the driving force for rendering the whole front end.

The description of a typical request flow through the system, as shown in Figure 7.1, "Content-led integration scenario" [88], clarifies the different roles of the *CAE* and the commerce system in this scenario.

1. The user requests a marketing driven landing page of a shop system.
2. The virtual host for the *CAE* forwards the request to the *CAE*.
3. Part of the requested page are various product teasers, with dynamic prices. Hence, the *CAE* needs to fetch corresponding information from the commerce system.
4. After receiving the page from the *CAE*, the user decides to click on a product teaser to see the corresponding product details. The link, rendered by the *CAE* as part of the landing page, directs the user to the virtual host of the commerce system.
5. The virtual host forwards the request to the commerce server.

6. As the requested Product Detail Page (PDP) contains a CoreMedia fragment, the commerce system requests it from the *CAE* and sends the whole PDP back to the user.

From the example follows, that the commerce-led integration scenario described in Chapter 6, *Commerce-led Integration Scenario* [53] is a subset of the content-led scenario. The request flow 4->-5->-6 uses the exact same technique to handle included CoreMedia fragments into *HCL Commerce* pages as described in the commerce-led scenario. The only difference is that resources or dynamic fragments fetched via Ajax requests are not handled by the virtual host of the commerce system. Instead, they are sent to the CAEs virtual host.

7.2 Status Synchronization in the Content-led Integration Scenario

Motivation

Take a look at figure Figure 7.1, "Content-led integration scenario" [88]. As you can see, the CAE and the commerce system stand side by side as equal partners from a users point of view. A user is allowed to request pages from both systems at any given time.

This architecture forces the CAE to synchronize any user sessions on the commerce system with its own. A user that browses the CAE and afterwards visits the *HCL Commerce* must keep his session and vice versa a user browsing the *HCL Commerce* going to the CAE afterwards must keep his state as well.

This section describes how the synchronization of this state is implemented by the CoreMedia CAE.

7.2.1 What Is The Users State?

HCL Commerce represents the state of a user session using cookies. To understand the synchronization of a users state across both systems you need to understand how those cookies may flow through the system. Take a closer look at Figure 7.2, "Content-led integration scenario with cookies" [91]. In addition to the request flow, the dashed green and blue arrows represent the flow of cookies.

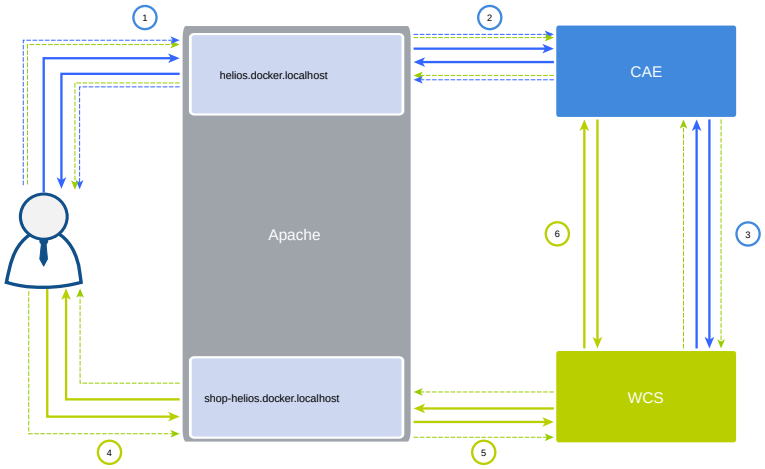

Figure 7.2. Content-led integration scenario with cookies

You can see that cookies may flow nearly everywhere. No matter where a request starts and where it ends, either between the browser and the *CAE* or between the *CAE* and the *HCL Commerce* system, every node may be the source as well as the receiver of cookies.

Two things that need explanation. First, two kinds of cookies flow from the browser to the *CAE*, cookies which were originally created in the commerce system and cookies that are created by the *CAE*. This is necessary because the *CAE* must send the commerce cookies to the commerce system as part of its backend calls. Second, for fragment requests (labeled with 6), no CoreMedia cookies are needed, hence, the browser does not need to send the *CAE* cookies to the commerce server.

Therefore, CoreMedia had to answer the following questions:

7.2.1.1 How does the CAE render fragments without its own cookies?

Cookies are used for dynamic HTML snippets, which are snippets that cannot be cached because they contain user specific content. Fragments that the *CAE* delivers to the commerce server should never include such dynamic HTML snippets because this would prevent a CDN or other caching infrastructure from caching complete *HCL Commerce* pages.

7.2.1.2 How Does the Browser Deliver Commerce System Cookies to the CAE?

The browser sends cookies to a server that runs in the same domain, that is saved with the cookie. In general the cookie domain of a cookie is left empty, so that the browser stores the exact host name of the server that responded to a request. But because the *CAE* and the commerce system must have different host names (via their virtual host), the *CAE* would never receive commerce system cookies.

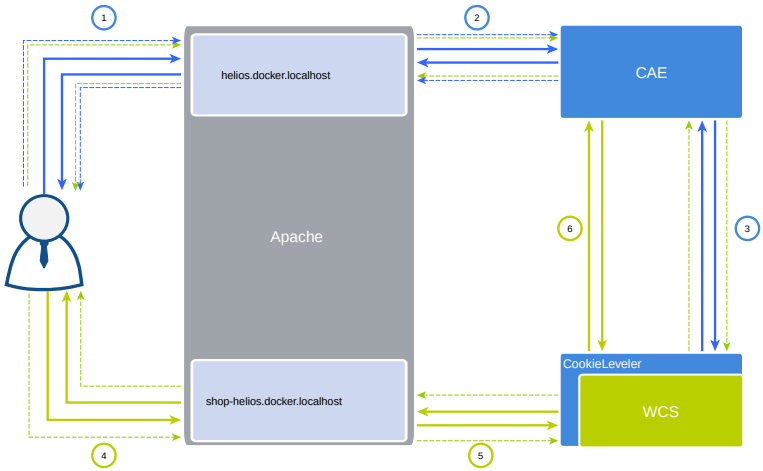

Figure 7.3. Content-led integration scenario

The solution to this problem is fairly simple. A servlet filter, the so called cookie leveler, runs in front of any *HCL Commerce* storefront call. It wraps the `HttpServletResponse` into a custom one, that intercepts `addCookie()` method calls in order to set the cookie domain to a configurable value.

The cookie leveler should be executed prior to any other filter that may add cookies to the response. In general CoreMedia recommends you to put its filter mapping definition in front of any other filter mapping.

There is one cookie that cannot be customized that way, the JSESSION cookie, which is set by the WebSphere servlet container. You have to configure it via the usual mechanisms provided by HCL, for example via the HCL console.

Now the *CAE* and the commerce system only need to be put into the same domain, for example helios.docker.localhost for the *CAE* and shop-helios.docker.localhost for the

HCL Commerce system. The cookie domain must then be configured to be .docker.localhost

NOTE

The cookie domain must not be a top level domain, for example .com, because that would mean, every website in the .com domain will receive the cookies. Because that does not make any sense, cookies with only a top level domain are generally not sent at all.

8. Studio Integration of Commerce Content

CoreMedia Content Cloud integrates with *HCL Commerce Server*. 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 8.1, "Catalog View in CoreMedia Studio Library" [95] gives a short overview over the Catalog Integration in the *Studio* Library.
- Section 8.2, "HCL Management Center Integration in CoreMedia Studio" [100] gives a short overview over the *HCL Commerce* Management Center integration in *CoreMedia Studio*.
- Section 8.4, "Commerce related Preview Support Features" [103] gives a short overview over the commerce related preview functions that are supported in *CoreMedia Studio*.
- Section 8.5, "Augmenting Commerce Content" [107] describes how you augment commerce content in the commerce-led scenario in *CoreMedia Studio*.

8.1 Catalog View in CoreMedia Studio Library

When the connection to a *HCL Commerce* system and a concrete shop for a content site are configured as described in Section 5.1, "Configuring the Commerce Adapter" [41] the *Studio* Library shows the default commerce catalog. You can also configure multiple catalogs as described in section "Configuring Multiple Catalogs" [45]. Then you will see all configured catalogs in the library. You can browse product categories, products and marketing spots in the commerce catalog and search for products, product variants and marketing spots. After the editor has selected a preferred site with a valid store configuration the catalog view will be enabled and the catalog[s] will be shown in the Library:

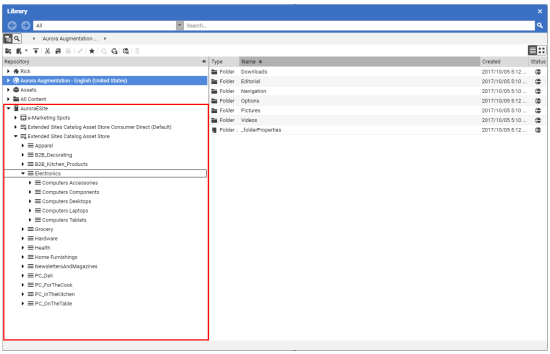

Figure 8.1. Library with catalog in the tree view

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.

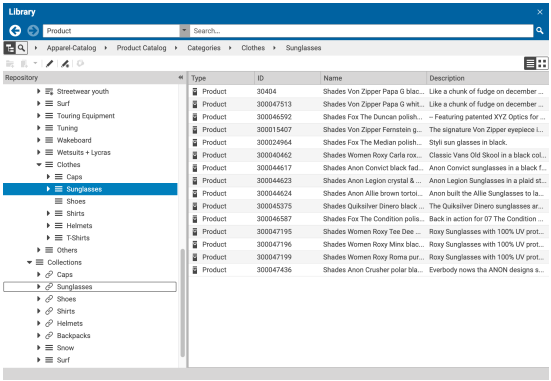

Figure 8.2. Library tree with multiple occurrences of the same category

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.

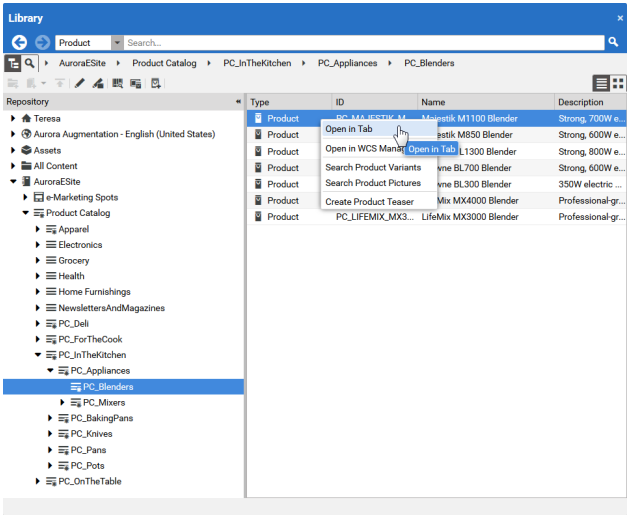

Figure 8.3. Open Product in tab

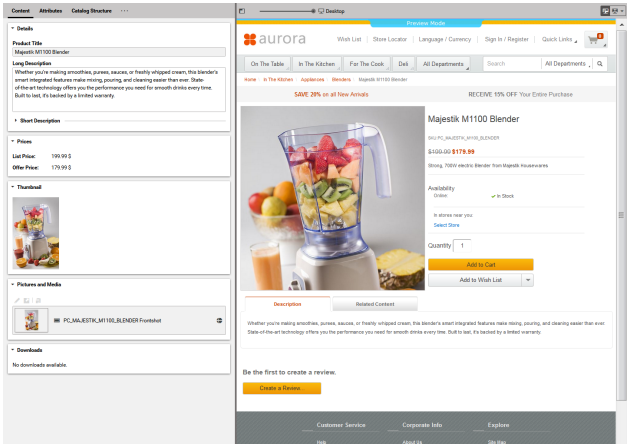

Figure 8.4. Product in tab preview

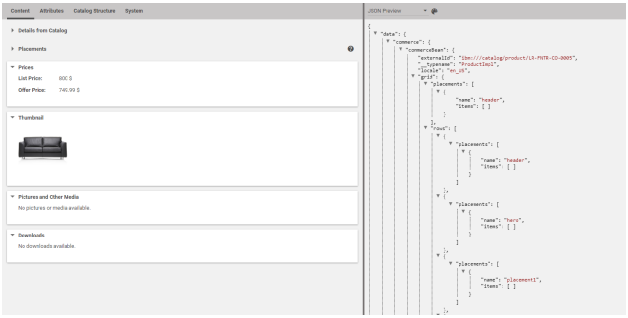

Figure 8.5. Product in tab with JSON preview (HCL Commerce 9.1)

NOTE

For Information on how to enable the JSON preview have a look at Section 9.32, "Multiple Previews Configuration" in *Studio Developer Manual*.

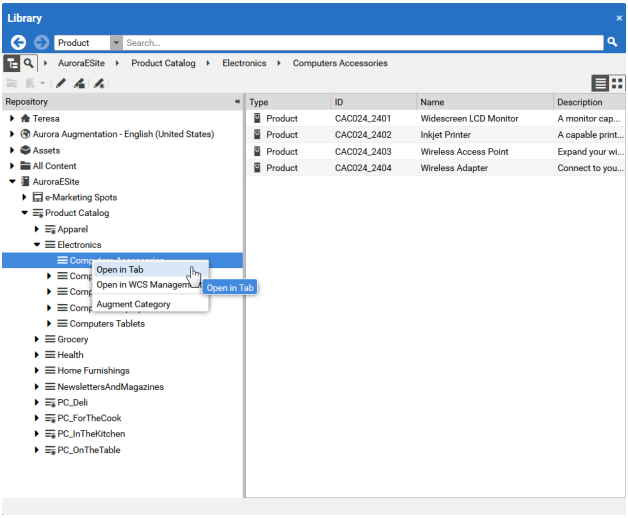

Figure 8.6. Open Category in tab

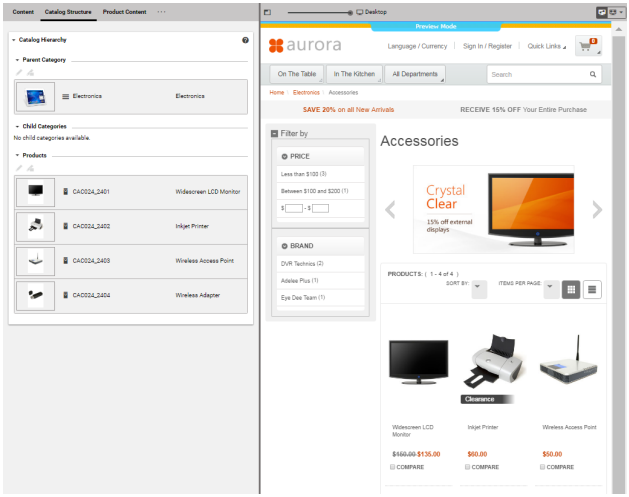

Figure 8.7. Category in tab preview

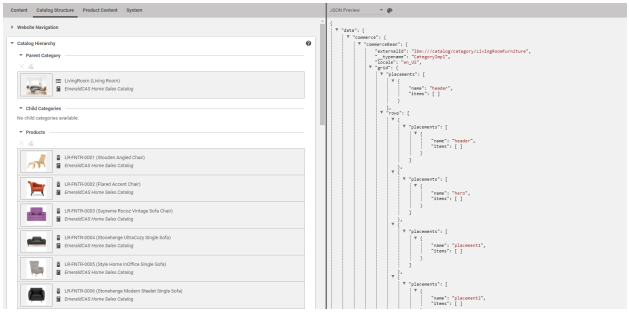

Figure 8.8. Category in tab preview (HCL Commerce 9.1)

In addition to the ability to browse through the commerce catalog in an explorer-like view it is also possible to search for products, variants and marketing spots from catalog. Similar to 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 will be displayed.

8.2 HCL Management Center Integration in CoreMedia Studio

> NOTE
>
> The HCL Management Center Integration is only available if the HCL Commerce Extension is used.

In addition to the eCommerce catalog library integration you can directly access the *HCL Management Center* from *CoreMedia Studio*. A context menu action on a product, product variant, category or e-marketing spot opens the item in a window within *CoreMedia Studio* where catalog item properties can be edited directly. This applies to all components in *CoreMedia Studio* which represent a product, product variant, category or `e-marketing spot`. Categories in the library do not open in Management Center by double click as this is the default behavior for navigation in the library tree.

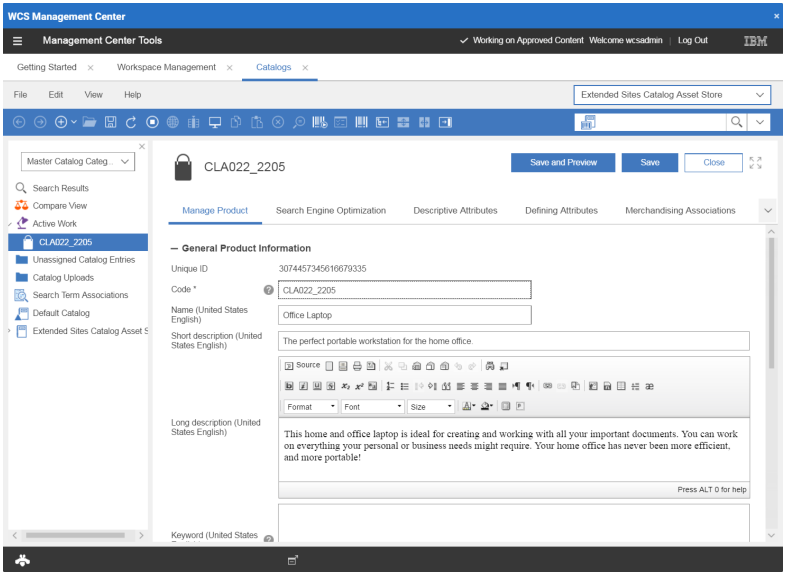

Figure 8.9. Management Center in Studio

NOTE

Known restriction:

- Up to FEP 7, the only supported web browsers are Internet Explorer and Firefox as these are supported web browsers for *HCL Commerce Server* Tools. Since FEP 8, Chrome is also supported.
- Currently there is no Single Sign On implemented between *CoreMedia Studio* and *Management Center*. You have to login to the *Management Center* with your *HCL Commerce* login credentials.

8.3 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 category pages in Studio, proceed as follows:

1. Open the `CommonJSToInclude.jspf` file and ensure that `${jsAssetsDir}javascript/CoreMedia/coremedia-pbe.js` is included if `_cm_page_pbe_pageData` is not empty.
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`). Be aware that this property overwrites the `studio.previewUrlPrefix` property, so you have to add the default CAE preview URL to the `studio.previewUrlWhitelist` property too.

Configure in the CoreMedia system

NOTE

If your *HCL Commerce* shop storefront uses any clickjacking prevention features (for example, X-Frame-Options (see https://help.hcltechsw.com/commerce/8.0.0/admin/tasks/tseiframerestrictxframe.html for details), please make sure to allow the shop preview (*HCL Commerce* Staging-/Authoringserver) being embedded as an iframe within *CoreMedia Studio*.

8.4 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 the commerce system, it is desirable that also these content reflects the given preview time. That could be a marketing spot containing activities with different validity time ranges. A specific activity could be valid only after a certain time or a marketing teaser that announces a happy hour could be another example.

  If such preview is requested from *HCL Commerce* the preview date is also sent to *HCL Commerce* as a genuine *HCL Commerce* preview token. The *HCL Commerce* recognizes the transmitted preview date and renders a control on top of the page that lets you inspect the currently active settings. Figure 8.10, “Time based preview affects also the *HCL Commerce* preview” [104] gives an example.

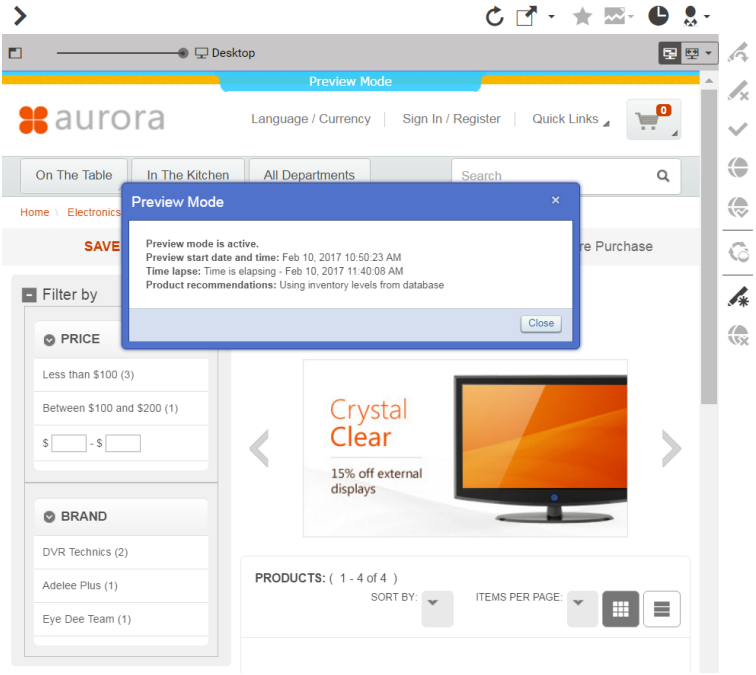

Figure 8.10. Time based preview affects also the HCL Commerce preview

- Customer segment based preview

> The commerce segment personalization is not available in HCL Commerce [FEP6].

FEP7+

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 8.11, “Test Customer Persona with Commerce Customer Segments” [105] shows an example where the test persona is female and has already been registered.

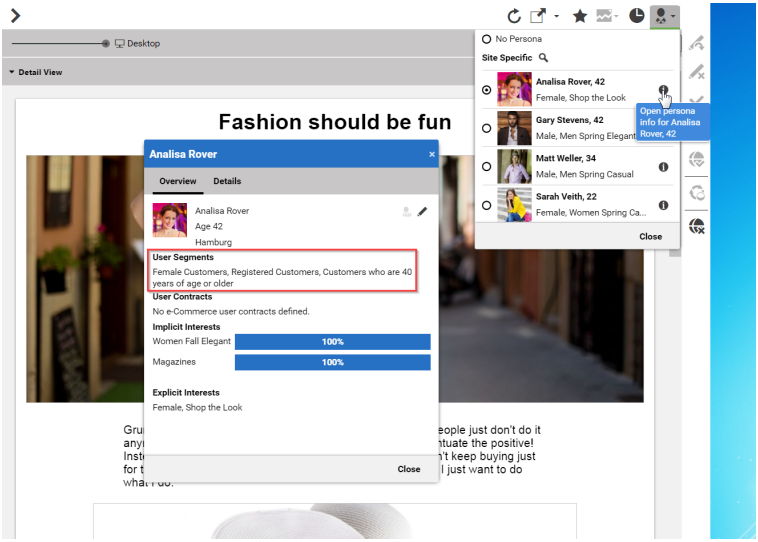

Figure 8.11. Test Customer Persona with Commerce Customer Segments

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*.

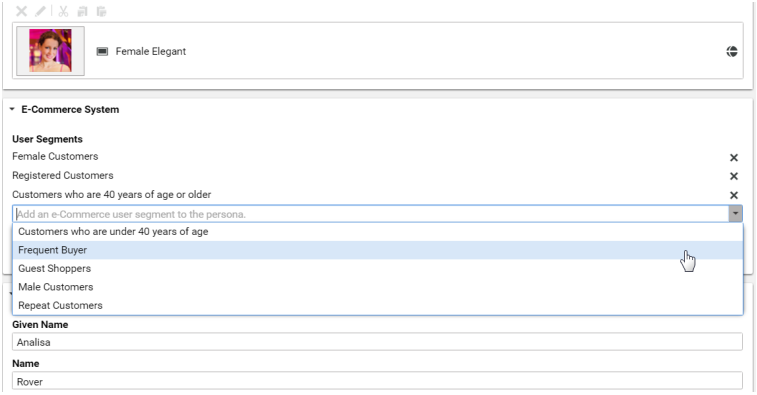

Figure 8.12. Edit Commerce Segments in Test Customer Persona

For personalized content based on commerce customer segmentation, it depends on the content type, if rules can be applied in the different rendering scenarios. In the case of catalog items, like products and categories, the commerce-led and the content-led scenarios are supported. In the content-led scenario the *CoreMedia CAE* is responsible for rendering, but the given user ID is also sent to the *HCL Commerce*. So all content that is received from the *HCL Commerce* is delivered within the context of the current *HCL Commerce* user. For marketing spots, the commerce system is responsible for rendering and therefore only the commerce-led scenario is supported.

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.

8.5 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 8.5.1, "Augmenting the Root Nodes" [107].
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*.

8.5.1 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, *AuroraESite,*). 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.

When multiple catalogs are configured, you will see multiple nodes under the store node. They represent catalogs' root categories. Each catalog has the *HCL Commerce* code of the catalog as its name.

Augmented catalog roots

To have a common ancestor for all augmented catalog pages, every root node of all configured catalogs 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.

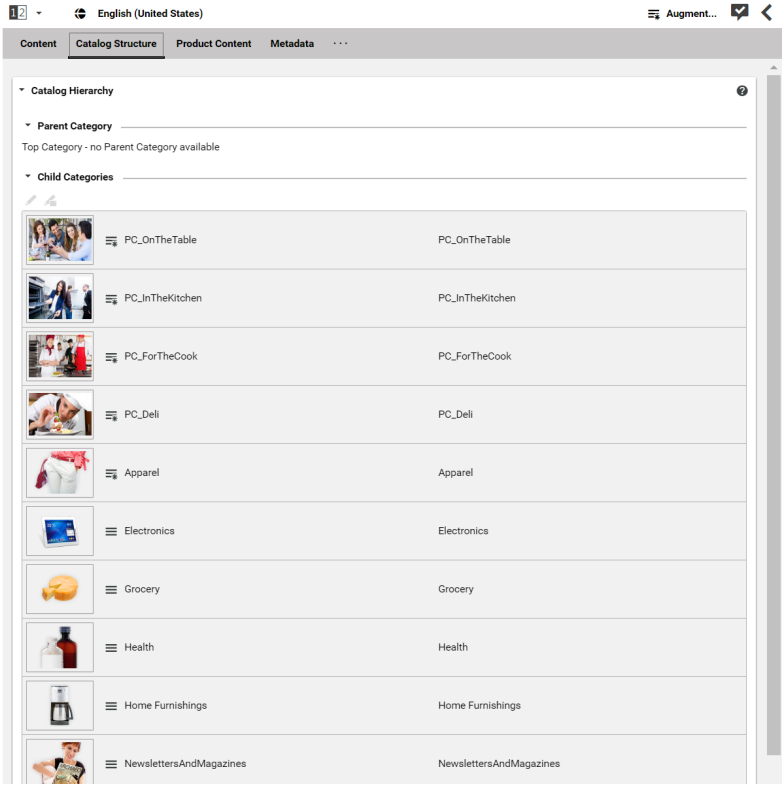

Figure 8.13. Catalog structure in the catalog root content item

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

8.5.2 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.

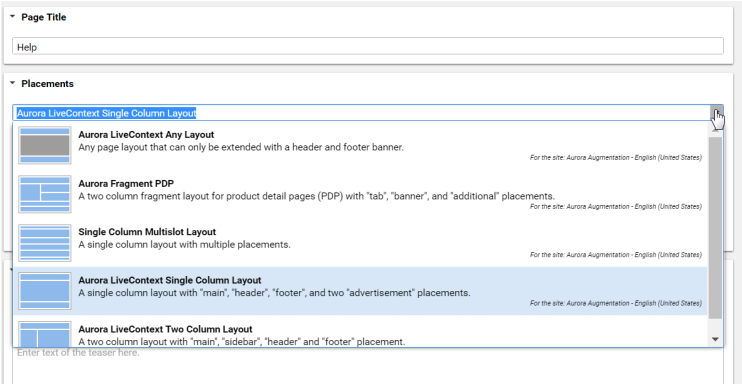

Figure 8.14. Choosing a page layout for a shop page

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.

8.5.3 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, mar-

keting 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.

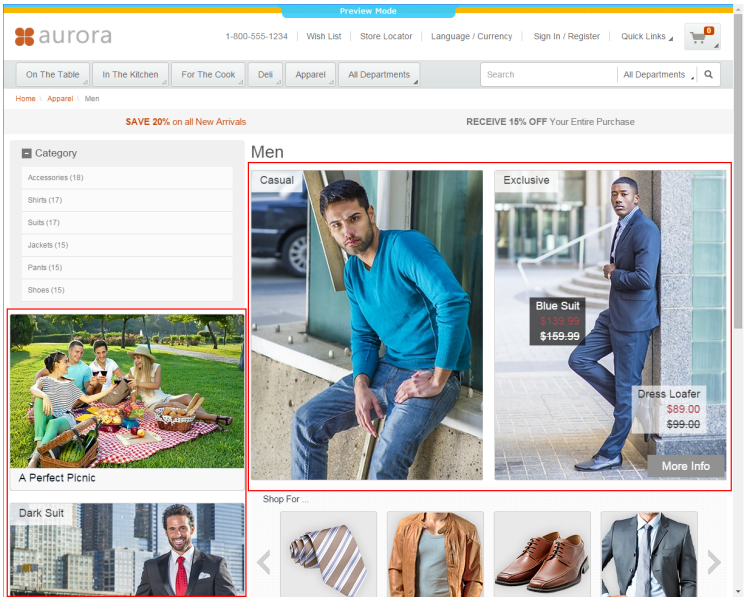

Figure 8.15. Category Overview Page with CMS Content

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

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:

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 8.16, "Decision diagram" [111] 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).

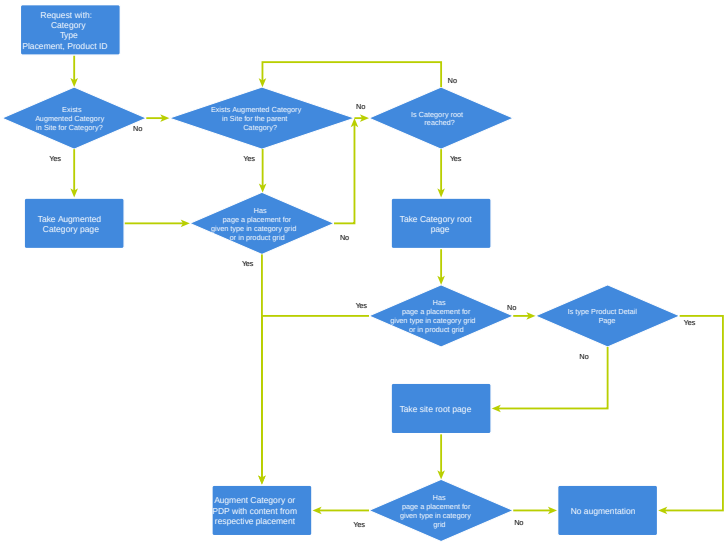

Figure 8.16. Decision diagram

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.

8.5.4 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.

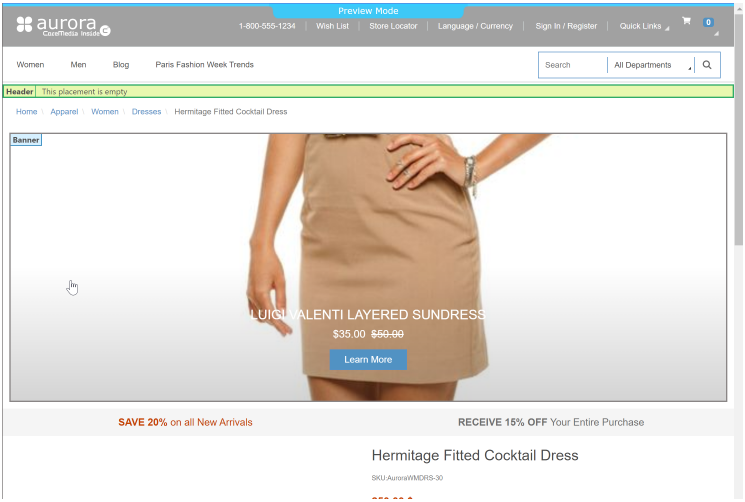

Figure 8.17. Product detail page with CMS content in the Banner section and empty Header placement

Information passed to the CoreMedia system

Similar to the category overview pages, the Category ID and placement name are passed to *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, *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`.

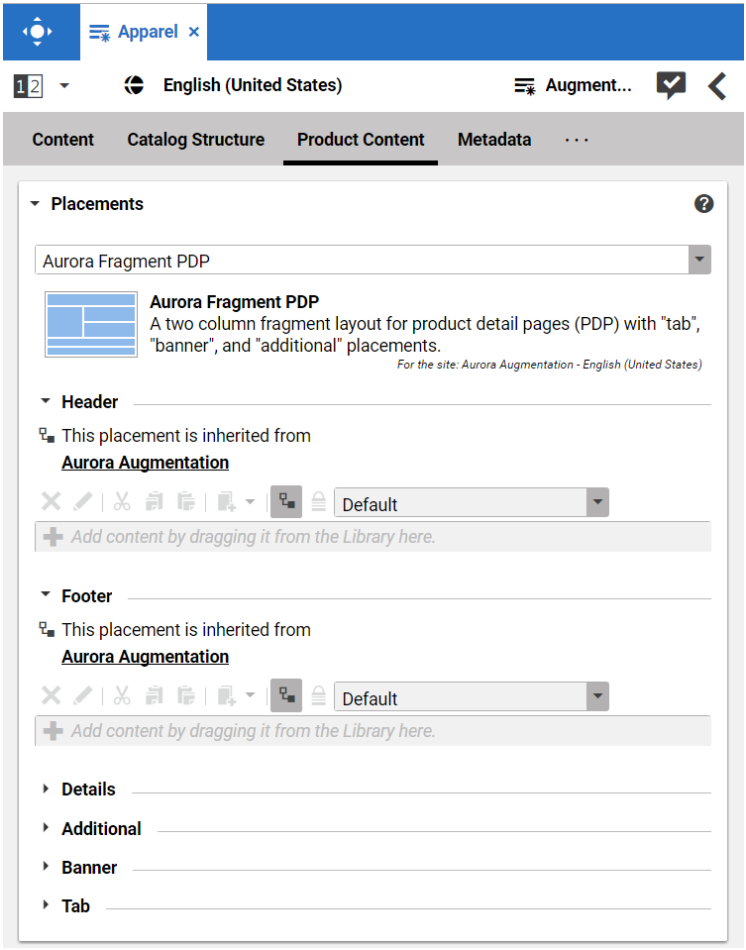

Figure 8.18. Page grid for PDPs in augmented category

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*.

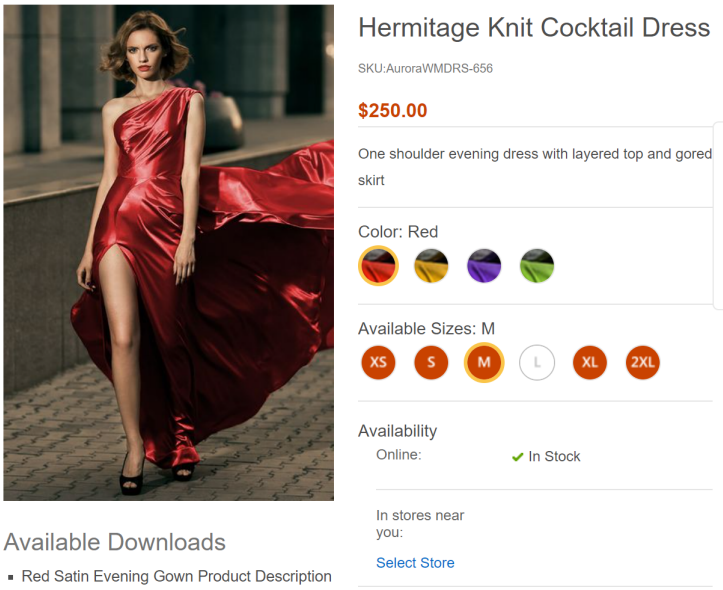

Figure 8.19. Product detail page with CMS assets

Information passed to the CoreMedia system.

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

Locating the assets in the CoreMedia system

To find assets for product detail pages, *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.

8.5.5 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.

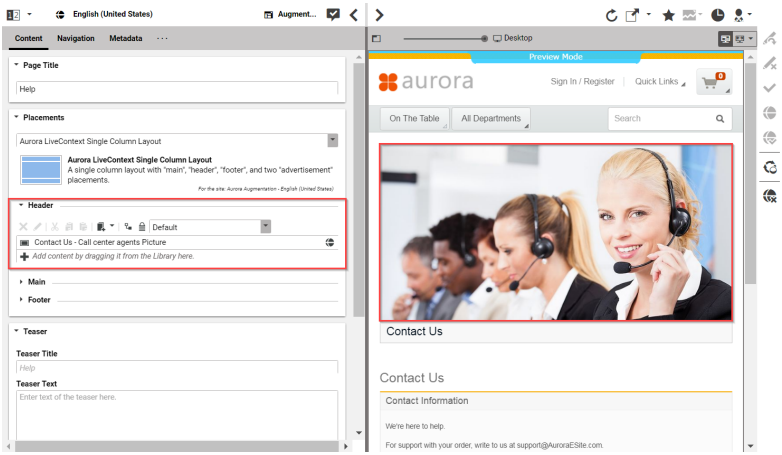

Figure 8.20. Example: Contact Us Pagegrid

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 8.21. Example: Navigation Settings for a simple SEO Page

URLs of non SEO pages

When the URL to a shop page is not a standard SEO URL but contains, for example, additional parameters, you can add this additional information via the *External URI Path* field (see Figure 8.22, “Example: Navigation Settings for a custom non SEO Form” [117]). This is necessary in order to get the *Studio* preview for the augmented page or for links rendered from the CMS. Therefore, if you have entered the correct URL, you will see the page in the preview.

In the *External URI Path* field, you redefine the URL path starting from `/en/aurora/...` and add required parameters. For example the advanced search page does not use the standard SEO path and in turn it has additional parameters:

```
.../AdvancedSearchDisplay?catalogId=10152&langId=-1&storeId=10301
```

Some of the standard parameters are well known and can be replaced by tokens, because they are very typical for all such URLs. In order to flexibly copy these URLs to other sites with different shop configurations the following tokens can be used:

Token	Description
`storeId`	The current store ID.
`catalogId`	The current catalog ID.

Token	Description
`langId`	The current language ID.

Table 8.1. config.id

Tokens have to be enclosed with curly braces. In case of the Advanced Search Page it would be possible to enter to following String into the *External URI Path*:

```
/AdvancedSearchDisplay?catalogId={catalogId}&langId={langId}&storeId={storeId}
```

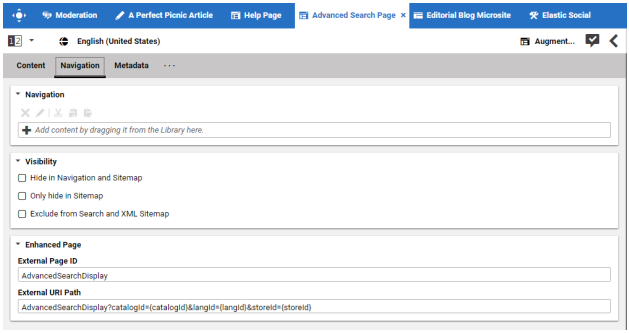

Figure 8.22. Example: Navigation Settings for a custom non SEO Form

> **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.

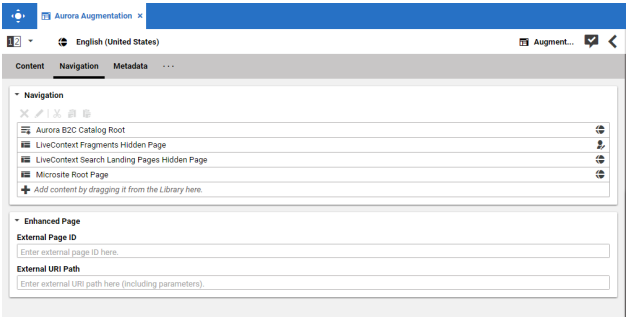

Figure 8.23. Special Case: Navigation Settings for the Homepage

9. 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:

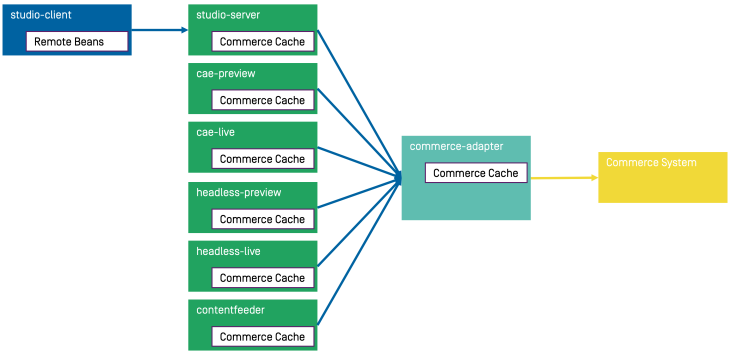

Figure 9.1. Multiple levels of caching

- Caching is implemented in the Commerce Adapter to accelerate access to commerce entities and to avoid heavy traffic on the *HCL Commerce* 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 *HCL Commerce* 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 *HCL Commerce*. 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 *HCL Commerce*. If the update cycles of the *HCL Commerce* 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.

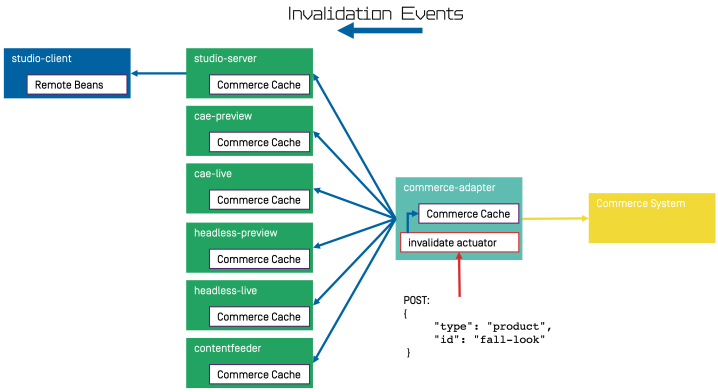

Figure 9.2. Commerce Cache Invalidation

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 *HCL Commerce*. 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 *HCL Commerce*.

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.
marketingspots	The list of all marketing spots referenced by ID.
marketingspot	A marketing spot with its properties. The memory consumption of each cache entry is very small.

Vendor specific config keys:

previewtoken	Preview tokens to support the Studio preview of commerce pages. A new token is requested for each new combination of preview parameters (e.g. customer segments, preview date). The cache time should be less than the default expiration time in the commerce system.
storeinfo	The global store info with all available catalogs referenced by name and ID.
categoryid	Used to map tech IDs to external IDs of categories. The memory consumption of each cache entry is very small.
categorydata	Used to build storefront URLs and in services that are not already cached in the base adapter (e.g. `PriceService`, `LinkService`, `CartService`). Each entry consumes ~10kB heap memory.
productid	Used to map tech IDs to external IDs of products and variants/SKUs. The memory consumption of each cache entry is very small.
productdata	Used to build storefront URLs and in services that are not already cached in the base adapter (e.g. `PriceService`, `LinkService`, `CartService`). 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.
dynamicprice	To retrieve personalized prices for products and SKUs. 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 is small.
staticprice	To retrieve static list prices for products and SKUs. 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 is small.

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 12, *Commerce Adapter Properties* [132] 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.*`

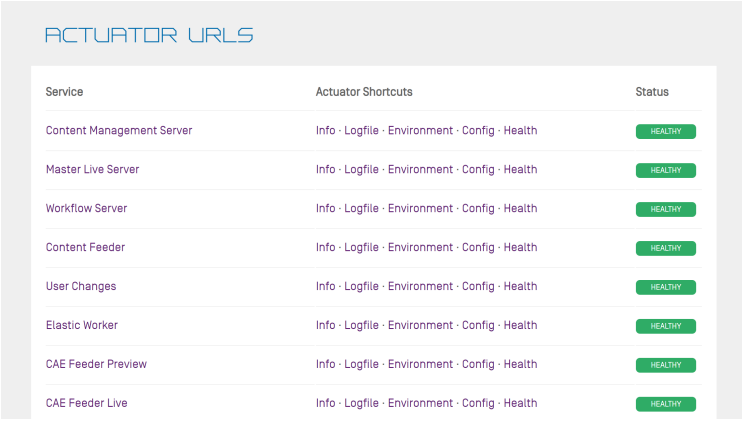

Figure 9.3. Actuator URLs in overview page

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 9.4. Actuator results for cache.timeout-seconds.ecommerce properties

10. 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*.

11. *HCL Commerce* REST Services used by CoreMedia

CoreMedia Content Cloud uses REST services of the *HCL Commerce Server* to access content. Here you find a list of URLs used by *Studio* and *CAE*.

REST Services used by CoreMedia Studio

- `http://<search_server>/search/resources/store/<storeId>/categoryview/@top`
- `http://<search_server>/search/resources/store/<storeId>/categoryview/%20?categoryIdentifier=<categoryIdentifier>`

  This search-based REST call allows slash character in the category identifier.
- `http://<search_server>/search/resources/store/<storeId>/categoryview/byId/<uniqueId>`
- `http://<search_server>/search/resources/store/<storeId>/categoryview/byParentCategory/<uniqueId>`
- `http://<search_server>/search/resources/store/<storeId>/productview/byCategory/<categoryId>`
- `http://<search_server>/search/resources/store/<storeId>/productview/bySearchTerm/<term>`
- `http://<wc_server>/wcs/resources/store/<storeId>/spot?q=byTypeAndName&qType=MARKETING&qName=<term>`
- `http://<wc_server>/wcs/resources/store/<storeId>/spot?q=byType&qType=MARKETING`
- `http://<wc_server>/wcs/resources/store/<storeId>/segment/<uniqueId>`
- `http://<wc_server>/wcs/resources/store/<storeId>/segment`
- `http://<wc_server>/wcs/resources/coremedia/languagemap`

  Used to map langId to numeric value
- `http://<wc_server>/wcs/resources/coremedia/storeinfo`

Used to get the storeId and the catalog information from all available stores in *HCL Commerce*

- `http://<wc_server>/wcs/resources/store/<storeId>/catalog`
- `http://<wc_server>/wcs/resources/store/<storeId>`
- `http://<wc_server>/wcs/resources/rest/admin/v2/stores`

REST Services used by the CAE

- `http://<wc_server>/wcs/resources/store/<storeId>/price?q=byPartNumbers&partNumber=<partNumber>`
- `http://<search_server>/search/resources/store/<storeId>/categoryview/%20?categoryIdentifier=<categoryIdentifier>`

  This search-based REST call allows slash character in the category identifier.
- `http://<search_server>/search/resources/store/<storeId>/categoryview/<SeoSegment>`
- `http://<search_server>/search/resources/store/<storeId>/categoryview/byId/<uniqueId>`
- `http://<search_server>/search/resources/store/<storeId>/productview/%20?partNumber=<productIdentifier>`

  This search-based REST call allows slash character in the product identifier.
- `http://<search_server>/search/resources/store/<storeId>/productview/byId/<uniqueId>`
- `http://<search_server>/search/resources/store/<storeId>/productview/bySearchTerm/<term>`
- `https://<wc_server>/wcs/resources/store/<storeId>/loginidentity`
- `https://<wc_server>/wcs/resources/store/<storeId>/previewToken`
- `http://<wc_server>:<searchport>/search/resources/store/<storeId>/productview/%20?partNumber=<productIdentifier>`

  This search-based REST call allows slash character in the product identifier.
- `http://<wc_server>/wcs/resources/store/<storeId>/usercontext/@self/contextdata`

  Used by *Elastic Social*
- `https://<wc_server>/wcs/resources/store/<storeId>/person/@self`

Used by *Elastic Social*

- `https://<wc_server>/wcs/resources/store/<storeId>/segment`

Used by *Adaptive Personalization*

- `http://<wc_server>/wcs/resources/store/<storeId>/cart/@self`
- `http://<wc_server>/wcs/resources/coremedia/languagemap`

Used to map langId to numeric value

- `http://<wc_server>/wcs/resources/coremedia/storeinfo`

Used to get the storeId and the catalog information from all available stores in *HCL Commerce*

- `http://<wc_server>/wcs/resources/store/<storeId>/catalog`

12. Commerce Adapter Properties

`wcs.always-use-master-category`	
Type	java.lang.Boolean
Default	false
Description	Determines that the master category is set on a product. A "master" category must exist in the master catalog and the sales catalog as well. If it is combined with categoryValidationEnabled = true and if the master category cannot be loaded then the next valid category is returned. If set to "true" the master category is set on products.

`wcs.auth-header-name`	
Type	java.lang.String
Default	
Description	The name of an authentication header the REST connector uses the access the WCS REST services. Default is empty, no Authentication header is used.

`wcs.auth-header-value`	
Type	java.lang.String
Default	
Description	The value of an authentication header the REST connector uses the access the WCS REST services.

`wcs.category-validation-enabled`	
Type	java.lang.Boolean

Default	false
Description	Determines that only a loadable category is set on a product. All eligible categories are loaded one after the other. The first one that is successful is used. If set to "true" only a loadable category is set on products.

`wcs.default-locale`

Type	java.util.Locale
Default	
Description	The default locale the REST connector is using if no locale is given.

`wcs.dynamic-pricing-enabled`

Type	java.lang.Boolean
Default	false
Description	Determines if dynamic pricing is enabled. If set to "true" the PriceRepository tries to get personalized prices from the WCS, otherwise an empty price list is returned.

`wcs.password`

Type	java.lang.String
Default	
Description	The service user password the REST connector uses to log in into WCS. This is mandatory and must be set.

`wcs.search-engine`

Type	com.coremedia.commerce.adapter.wcs.client.common.SearchEngineType
Default	
Description	Configures the search engine type of the HCL Commerce System. It is only used since HCL Commerce 9.1 and the search engine ES is used as the default.

`wcs.search-profile-prefix`	
Type	java.lang.String
Default	CoreMedia
Description	Configures the prefix of the HCL Commerce Search profile. For HCL Commerce 9.0 and older the prefix CoreMedia is used as the default search profile prefix. With HCL Commerce 9.1 the prefix should be set to HCL.

`wcs.search-url`	
Type	java.lang.String
Default	
Description	The general WCS URL to access the search-based WCS REST services via http. If a REST service does not need secure access this url prefix is used.

`wcs.secure-search-url`	
Type	java.lang.String
Default	
Description	The secure WCS URL to access the search-based WCS REST services via https. If a REST service needs secure access this url prefix is used.

`wcs.secure-url`	
Type	java.lang.String
Default	
Description	The secure WCS URL to access the WCS REST services via https. If a REST service needs secure access this url prefix is used.

`wcs.single-value-search-facets`	
Type	java.util.List<java.lang.String>

Default	
Description	Configures the keys of the facets that that can only be added with a single value to product search requests. Should e.g. be configured with parentCatgroup_id_search when connecting to WCS 8.0, because it doesn't allow searching with multiple category facets.

`wcs.url`

Type	java.lang.String
Default	
Description	The general WCS URL to access the WCS REST services via http. If a REST service does not need secure access this url prefix is used.

`wcs.username`

Type	java.lang.String
Default	
Description	The service user the REST connector uses to log in into WCS. This is mandatory and must be set.

`wcs.version`

Type	java.lang.String
Default	
Description	The WCS version. Some WCS REST services are version specific.

`wcs.cookie.user.filter-pattern`

Type	java.lang.String
Default	WCP?_.+
Description	The regular expression pattern for which the client should filter the relevant cookies. This should narrow down the cookies on the client side to a subset of cacheable cookies.

`wcs.cookie.user.filter-pattern-for`

Type	java.util.Map<java.lang.String,java.lang.String>
Default	
Description	Cookie filter pattern for specific environment. The structure of the Map should be: key=environment, value=cookie pattern. The environment is the hardcoded name of the entity param which must be configured on the CM App client side e.g. `commerce.hub.data.customEntityParams.environment=PREVIEW|LIVE``

Examples:

```
wcs.link.filter-pattern-for.preview=WCP?_.+
```

```
wcs.link.filter-pattern-for.live=WC?_.+
```

`wcs.cookie.user.user-session-pattern`

Type	java.lang.String
Default	WCP?_USERACTIVITY_[-1002|\d+]
Description	The regular expression pattern for the WCS user session cookie. See description for *WC_USERACTIVITY_ID* in: • HCL Commerce Version 9 User Guide - Session management - WebSphere Commerce session cookies • WebSphere Commerce Version 8 User Guide - Session management - WebSphere Commerce session cookies

`wcs.link.asset-url`

Type	java.lang.String
Default	
Description	Asset URL prefix that is used to build asset links to shop images in the live system. Typically, a proxy url is set including protocol and possibly a context path prefix. Should only be set if the adapter does not need to distinguish environments. In this case no environment metadata.custom-entity-param-names parameter is required.

Examles:

```
https://shop-hcl.coremedia.vm
```

```
https://shop-preview-hcl.coremedia.vm
```

This and the further wcs.link properties are not needed when only connecting to HCL Commerce 9.1+ React stores.

wcs.link.asset-url-for

Type	java.util.Map<java.lang.String,java.lang.String>
Default	

Description	Asset URL prefixes which are used to build asset links to shop images for different environments.
	
	Typically, a proxy url is set including protocol and possibly a context path prefix. The structure of the Map should be: {key=environment, value=url}. The environment is the hardcoded name of the entity param which must be configured in the CMS app, e.g. commerce.hub.data.custom-entity-params.environment=preview|live. **IMPORTANT**: The keys used here must match those used in the CMS app via commerce.hub.data.custom-EntityParams.environment={environment}.
	
	Examples:
	
	`wcs.link.asset-url-for.preview=https://shop-preview-hcl.coremedia.vm`
	
	`wcs.link.asset-url-for.live=https://shop-hcl.coremedia.vm`
	
	For configuration options see also documentation of wcs.link.storefront-url-for.
	
	This and the further wcs.link properties are not needed when only connecting to HCL Commerce 9.1+ React stores.

wcs.link.link-templates

Type	java.util.Map<java.lang.String,java.lang.String>
Default	
Description	Map of StorefrontRef. Used to build shop urls for the Studio Preview and Content-Led integration scenarios.

Known default lookup keys are defined in StorefrontRefKeysCommerceLed and StorefrontRefKeysContentLed. Only lookup keys in lowercase and without "_" are valid.

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
- {langId} ... the current language as WCS specific id, e.g. "-1" as default language
- {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)

This and the further wcs.link properties are not needed when only connecting to HCL Commerce 9.1+ React stores.

`wcs.link.link-templates.categorylinkfragment`

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

`wcs.link.link-templates.categorynonseourl`

Type	java.lang.String
Default	{storefrontUrl}/CategoryDisplay?categoryId={categoryTechId}&storeId={storeId}&langId={langId}&catalogId={catalogId}
Description	Non-seo-friendly shop URLs to category pages.

`wcs.link.link-templates.categorypreviewurl`

Type	java.lang.String
Default	{storefrontUrl}/CategoryDisplay?categoryId={categoryTechId}&storeId={storeId}&langId={langId}&catalogId={catalogId}&newPreviewSession=true&previewToken={previewToken}
Description	Used to build the preview URL to a category page.

wcs.link.link-templates.categoryseourl	
Type	java.lang.String
Default	{storefrontUrl}/{language}/{storeName}/{pageId}
Description	Used to build seo-friendly URLs to category pages.

wcs.link.link-templates.checkoutredirecturl	
Type	java.lang.String
Default	{storefrontUrl}/OrderCalculate?calculationUsageId=-1&storeId={storeId}&update-Prices=1&catalogId={catalogId}&orderId=.&langId={langId}&URL=AjaxOrderItemDis-playView
Description	Used to build the redirect URL to the checkout page.

wcs.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.

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

wcs.link.link-templates.cmcontentpreviewurl	
Type	java.lang.String
Default	{storefrontUrl}/{language}/{storeName}/cm/{seoSegment}?newPreviewSession=true&pre-viewToken={previewToken}
Description	Used to build the preview URL to a shop page which displays a CoreMedia content.

wcs.link.link-templates.cmcontenturl	
Type	java.lang.String
Default	{storefrontUrl}/{language}/{storeName}/cm/{seoSegment}
Description	Used to build seo-friendly URLs to shop pages displaying CoreMedia Articles and Channels.

wcs.link.link-templates.contractpreviewurl	
Type	java.lang.String
Default	true
Description	Used to build a preview url with a contract parameter.

wcs.link.link-templates.externalpagenonseopreviewurl	
Type	java.lang.String
Default	{storefrontUrl}/{externalUriPath}&newPreviewSession=true&previewToken={previewToken}
Description	Used to build the preview URL to a shop page which has no seo support.

wcs.link.link-templates.externalpagenonseourl	
Type	java.lang.String
Default	{storefrontUrl}/{externalUriPath}
Description	Used to build non-seo-friendly URLs to shop pages.

wcs.link.link-templates.externalpagepreviewurl	
Type	java.lang.String
Default	{storefrontUrl}/{language}/{storeName}/{pageId}?newPreviewSession=true&previewToken={previewToken}
Description	Used to build the preview URL to a shop page.

wcs.link.link-templates.externalpageseourl

Type	java.lang.String
Default	{storefrontUrl}/{language}/{storeName}/{pageId}
Description	Used to build seo-friendly URLs to shop pages.

`wcs.link.link-templates.homepagelinkfragment`

Type	java.lang.String
Default	<!--CM {"externalSeoSegment":"","renderType":"url","objectType":"page"} CM-->
Description	Used to the link to the home page.

`wcs.link.link-templates.homepagepreviewurl`

Type	java.lang.String
Default	{storefrontUrl}/{language}/{storeName}/{pageId}?newPreviewSession=true&previewToken={previewToken}
Description	Used to build the preview URL to the shop home page.

`wcs.link.link-templates.loginurl`

Type	java.lang.String
Default	{storefrontUrl}/UserRegistrationForm?catalogId={catalogId}&langId={langId}&storeId={storeId}
Description	Used to build the URL to the Login page.

`wcs.link.link-templates.logouturl`

Type	java.lang.String
Default	{storefrontUrl}/Logoff?storeId={storeId}
Description	Used to build the URL which logs off the current user.

`wcs.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.

`wcs.link.link-templates.productnonseourl`

Type	java.lang.String
Default	{storefrontUrl}/ProductDisplay?productId={productTechId}&storeId={storeId}&langId={langId}&catalogId={catalogId}
Description	Url pattern that is used to build non-seo-friendly shop URLs to product detail pages.

`wcs.link.link-templates.productpreviewurl`

Type	java.lang.String
Default	{storefrontUrl}/ProductDisplay?productId={productTechId}&storeId={storeId}&langId={langId}&catalogId={catalogId}&newPreviewSession=true&previewToken={previewToken}
Description	Used to build the preview URL to a product detail page.

`wcs.link.link-templates.productseourl`

Type	java.lang.String
Default	{storefrontUrl}/{language}/{storeName}/{pageId}
Description	Url pattern that is used to build shop URLs for product detail pages.

`wcs.link.link-templates.searchredirecturl`

Type	java.lang.String
Default	{storefrontUrl}/SearchDisplay?storeId={storeId}&catalogId={catalogId}&langId={langId}&pageSize=12&searchTerm={searchTerm}
Description	Used to build the parameterized search url to be redirected to the shop search result page.

`wcs.link.link-templates.shoppagelinkfragment`

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

`wcs.link.product-max-url-segments`

Type	java.lang.Integer
Default	3
Description	Max url segments of an seo url for products This and the further wcs.link properties are not needed when only connecting to HCL Commerce 9.1+ React stores.

`wcs.link.storefront-url`

Type	java.lang.String
Default	

Description	Storefront URL prefix that is used to build storefront links to shop pages and resources in the live system.
	Typically, a proxy url is set, including protocol and possibly a context path prefix.
	Should only be set if the adapter does not need to distinguish environments In this case no environment metadata.custom-entity-param-names parameter is required.
	Examples:
	`https://shop-hcl.coremedia.vm/webapp/wcs/shop`
	`https://shop-preview-hcl.coremedia.vm/webapp/remote/preview/servlet`
	This and the further wcs.link properties are not needed when only connecting to HCL Commerce 9.1+ React stores.

`wcs.link.storefront-url-for`

Type	java.util.Map<java.lang.String,java.lang.String>

Default	

Description	Storefront URLs which are used to build storefront links to shop pages and resources for different environments. The structure of the Map should be: {key=environment, value=url}.
	
	The multi-environment support needs to be activated via metadata.custom-entity-param-names=environment.
	
	Examples:
	
	`wcs.link.storefront-url-for.preview=https://shop-preview-hcl.coremedia.vm/webapp/remote/preview/servlet`
	
	`wcs.link.storefront-url-for.live=https://shop-hcl.coremedia.vm/webapp/wcs/shop`
	
	The environment name for the custom entity param must be configured on the client side (CAE, Studio, etc.). Global configuration example: commerce.hub.data.customEntityParams.environment=preview|live
	
	You may also configure multiple storefront URLs for different sites/environments via the commerce settings struct: commerce (Struct) customEntityParams (Struct) environment=siteus (String) Keep the lookup keys simple. Use lowercase with no special characters.
	
	Be aware that you need to configure the environment values on the client site first, otherwise lookups can't work and will fail. There is no default fallback as this could lead to even more confusion.
	
	This and the further wcs.link properties are not needed when only connecting to HCL Commerce 9.1+ React stores.

`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 12.1. HCL 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 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 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