Connector for Salesforce Commerce Cloud Manual

CoreMedia Content Cloud – v12

List of Figures

List of Tables

List of Examples

1. Preface

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

- Chapter 2, *Overview* [6] gives a short overview of the integration.
- Chapter 3, *Customizing Salesforce Commerce Cloud* [11] describes how you have to configure the commerce system to work with *CoreMedia Content Cloud.*
- Chapter 5, *Commerce-led Integration Scenario* [22] describes the commerce-led scenario and shows how you extend commerce pages with CMS fragments.
- Chapter 4, *Connecting to a Salesforce Commerce Cloud System* [12] describes how you connect a CoreMedia web application with a *Salesforce Commerce* system.
- Section 6.2, "Enabling Preview in Shop Context" [66] describes how you activate the preview of *Salesforce Commerce* pages in *Studio.*
- Chapter 6, *Studio Integration of Commerce Content* [62] shows the eCommerce features integrated into *CoreMedia Studio.*
- Chapter 7, *Commerce Caching* [80] describes the CoreMedia cache for eCommerce entities.
- Chapter 8, *The eCommerce API* [88] describes the basics of the eCommerce API.

1.1 Audience

This manual is intended for architects and developers who want to connect *CoreMedia Content Cloud* with an eCommerce system and who want to learn about the concepts of the product. The reader should be familiar with *CoreMedia CMS*, , *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
(i)	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 *Salesforce Commerce Cloud*. You will learn how to add fragments from the CoreMedia system into a *Salesforce* generated site, how to access the *Salesforce* catalog from the CoreMedia system and how to develop with the *eCommerce API*. The configuration of your *Salesforce* system is described in Chapter 3, *Customizing Salesforce Commerce Cloud* [11]

Integration scenarios

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.
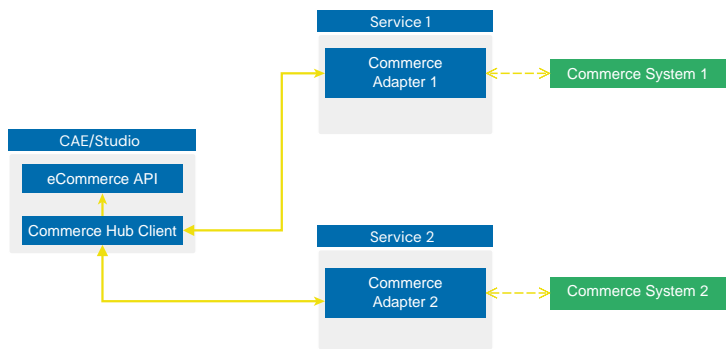
Figure 2.1. Architectural overview of the Commerce Hub

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 independ–ent 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.
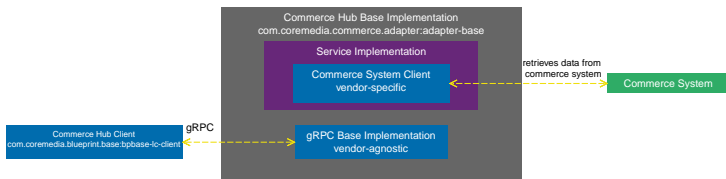
Figure 2.2. More detailed architecture view

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 Salesforce Commerce Cloud

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

The [CoreMedia Connector for Salesforce Commerce Cloud] manual contains documentation which describes how to adapt your Salesforce project workspace in order to integrate with *CoreMedia Content Cloud*. You will find the instruction in the LiveContext Connector for Salesforce workspace Zip file.

Section 4.3, "Check if everything is working" [18] describes how to check if everything is wired up correctly and works as expected.

4. Connecting to a *Salesforce Commerce Cloud* System

The connection of your *Blueprint* web applications (*Studio* or *CAE*) to a *Salesforce Commerce Cloud* 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 *Salesforce Commerce Cloud* system (see Section 4.1, "Configuring the Commerce Adapter" [13]).

• Settings configuration in *Studio*. It references the Commerce Adapter endpoint, which *Studio* and *CAE* use to indirectly communicate via the Commerce Adapter with the *Salesforce Commerce Cloud* (see Section 4.2, "Shop Configuration in Content Settings" [15]).

NOTE

Prerequisite

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

4.1 Configuring the Commerce Adapter

Configuring the Commerce Adapter

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

The Commerce Adapter comes along with a set of configuration properties. For detailed documentation and defaults see Chapter 9, *Commerce Adapter Properties* [90].

Starting the Commerce Adapter

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

Prerequisites to be installed:

- Maven
- Docker
- Docker Compose (optional)

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

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

To run the `commerce-adapter-sfcc` 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-sfcc \
--publish 44165:6565 \
--publish 44181:8081 \
[--env ...|--env-file ...|--volume] \
coremedia/commerce-adapter-sfcc:${ADAPTER_VERSION}
```

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

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

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

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

4.2 Shop Configuration in Content Settings

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

Refer to the Javadoc of the class `com.coremedia.blueprint.base.live-context.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.

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

Name	Type	Description	Example	Required
`endpoint`	String Property	Host and Port of the Commerce Adapter.	sfcc-commerce-adapter:6565	true (if endpointName is not set)
`endpoint Name`	String Property	The endpoint name to lookup the Spring gRPC service configuration .	sfcc	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.	GBP	false. If not set, the currency will

Name	Type	Description	Example	Required
				be re-trieved from the site locale.
storeConfig	Struct Property	Struct property containing store configuration		true
storeConfig.id	String Property	The ID of the store.	SiteGenesis-Global	true
storeConfig.name	String Property	The name of the store as it is set in the commerce sys-tem.	SiteGenesis Global Shop	true
catalogConfig	Struct Property	Struct property containing catalog configuration.		true
catalogConfig.id	String Property	The ID of the catalog.	storefront-catalog-non-en	true
catalogConfig.name	String Property	The name of the catalog.	storefront-catalog-non-en	true
catalogConfig.alias	String Property	The alias of the catalog.	catalog	false. If not set, 'cata-log' will be used as de-fault alias.
customEntityParams	Struct Property	Site specific custom entity parameters, which are at-tached to the communica-tion with the commerce		false. If not set, no site specific custom en-

Name	Type	Description	Example	Required
		adapter. See Section 4.4, "Configuring Custom Entity Parameters" [20] for more information.		tities will be used.

Table 4.1. Livecontext settings

NOTE

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

4.3 Check if everything is working

Prerequisites

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

Check the Studio – *Salesforce Commerce* REST Connection

1. Open *Studio*, select the "SFRA – English (United Kingdom)" site, open the Library. If necessary, switch the Library to browse Mode.
2. In the repository tree view, locate a node named *SFRA Global Shop*. This is the entry point to browse the connected *Salesforce* product catalog.
3. Browse the catalog in studio and check if everything works as expected. Section 6.1, "Catalog View in CoreMedia Studio Library" [63] describes what it looks like.

If errors occur:

- Check the Studio log and the Commerce Adapter log for errors.
- Check in *CoreMedia Studio* if the "LiveContextSettings" are configured correctly, see Section 4.2, "Shop Configuration in Content Settings" [15].
- Check if the REST connector is configured correctly (see Section 4.1, "Configuring the Commerce Adapter" [13]). Check for example, if the deployment property `sfcc.ocapi.host` is configured correctly.

Check Studio – *Salesforce Commerce* Preview Integration

1. Open the Homepage of the "SFRA – English (United Kingdom)" site in Studio

   The *Salesforce* 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 `sfcc.link.storefront-url` is configured correctly for Commerce Adapter.
- Check if your customer specific Open Commerce API client ID is set in the `sfcc.oauth.client-id` and `sfcc.oauth.client-password` properties in Commerce Adapter.
- Check if, `CM-RedirectUrl` controller is accessible. Call `https://sandboxhost/on/demandware.store/Sites-RefArchGlobal-Site/en_GB/CM-RedirectUrl?link=Home-Show,preview,true`. The call should be redirected to the SFRA homepage.

Check Fragment Connector

1. Open the SFRA - English (United Kingdom) homepage and check if CoreMedia Demo content is displayed.

If errors occurred or no CoreMedia Content is displayed

- Check for errors in the *Salesforce Commerce* log and the Preview *CAE* log and the Commerce Adapter log.
- Check in *Salesforce Commerce Business Manager* and the Developer Tools if the homepage has content slots containing *CoreMedia Content Widgets* or if render templates contain an `islcinclude` tag.

4.4 Configuring Custom Entity Parameters

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

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

Example:

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

Adapter Configuration

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

Client Configuration

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

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

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

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

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

5. Commerce-led Integration Scenario

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

This chapter describes how you include the content from the CMS into shop pages. Have also a look into Section 6.4, "Augmenting Commerce Content" [70] and Chapter 6, *Working with Product Catalogs* in *Studio User Manual* for more details about the *Studio* usage for eCommerce.

- Section 5.1, "Commerce-led Scenario Overview" [23] gives an overview over the request flow in the commerce-led integration scenario.
- Section 5.2, "Adding CMS Fragments to Shop Pages" [25] describes how you can add fragments to the commerce system via the CoreMedia widgets and the `islcinclude` tag and how you can augment shop pages in *Studio*.
- Section 5.3, "Extending the Shop Context" [42] describes how you extend the shop context that is delivered to the CMS.
- Section 5.4, "Caching In Commerce-Led Scenario" [45] describes the caching in the commerce-led scenario.
- Section 5.6, "Prefetch Fragments to Minimize CMS Requests" [55] describes how to prefetch fragments in the commerce-led scenario.
- Section 5.7, "Configure Logging" [60] describes how to configure logging for the *CoreMedia Cartridge for Salesforce*.

5.1 Commerce-led Scenario Overview
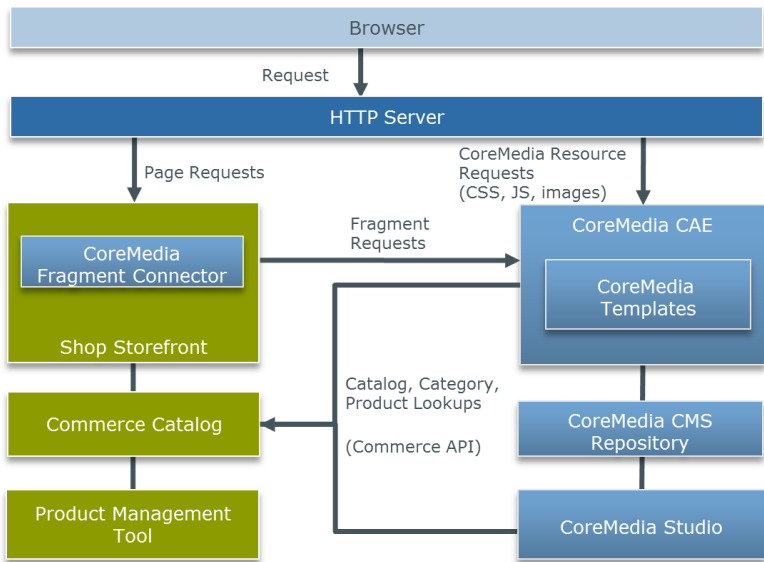
Figure 5.1. Commerce-led Architecture Overview

Figure 5.1, "Commerce-led Architecture Overview" [23] shows the commerce-led integration scenario where the CoreMedia *CAE* operates behind the commerce server for all page request. Moreover, you can see two kinds of requests. While the left side shows HTTP page requests to the commerce server, that include fragments delivered by the *CAE*, the right side shows resource or Ajax requests directly redirected by the one virtual host in front of both servers to the *CAE*.

A typical flow of requests through a commerce-led system is as follows:
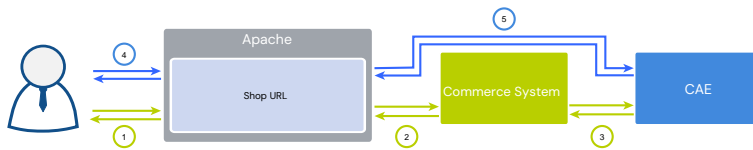
Figure 5.2. Commerce-led Request Flow

1. A user requests a product detail page that is received by the virtual host.

2. The virtual host identifies the request as a commerce request and forwards it to the commerce server.

3. Part of the requested Product Detail Page (PDP) is a CMS content fragment. Hence, the commerce system requests the fragment from the *CAE*.

4. The resulting HTML page flows back to the user's browsers. Because the page contains dynamic *CAE* fragments which have to be fetched via Ajax, the browser triggers the corresponding request against the virtual host.

5. As this is a *CAE* request, the virtual host forwards it directly to the *CAE*.

5.2 Adding CMS Fragments to Shop Pages

A pure eCommerce system is focused on the more transactional aspects of the buying process. To create a more engaging user experience you can augment the catalog pages with editorial content from the CMS. This includes, articles, images or videos.
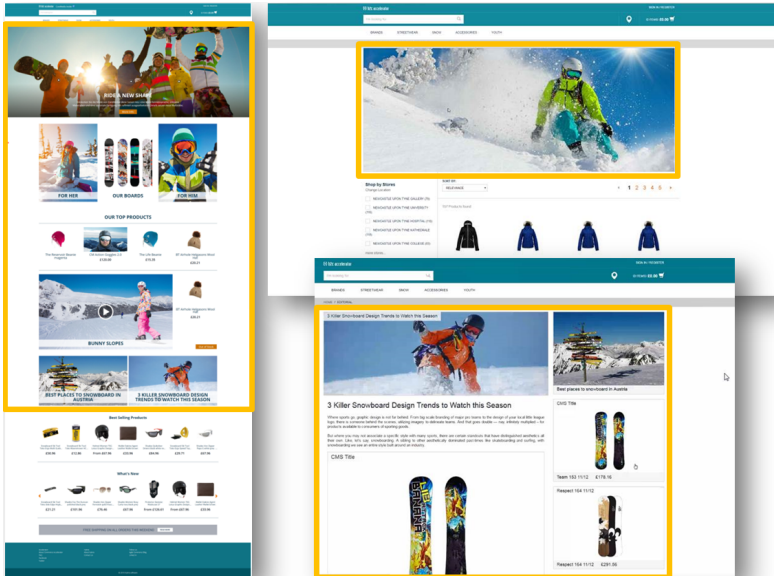
Figure 5.3. Various Shop Pages with CMS Fragments

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

Types of augmentable pages

- **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 basis for augmentation is the use of the *CoreMedia Content Widget* in content slots or the `islcinclude` tag in `ISML` templates.

The augmentation process

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

Section 6.4, "Augmenting Commerce Content" [70] describes the procedure.

5.2.1 CoreMedia Content Widget

The CoreMedia Content Widget is used to display content from the CoreMedia system on pages delivered by the eCommerce system. It is implemented as an extension of the *Salesforce* content slot mechanism. The slot configuration is extended with three custom attributes that can be filled when a content uses the CoreMedia Content Widget.

Technical Background of the Core-Media Content Widget

Furthermore, there is an `ISML` template that must be executed when a content slot should be used for CoreMedia content (see Figure 5.5, "Content Slot Configuration Example" [29]).

The configuration file that extends the content slot edit form, `system-object type-extensions.xml`, and the `ISML` template `coremedia-content-widget.isml` are both part of the *CoreMedia Cartridge for Salesforce* and come with the *Salesforce Commerce Cloud workspace archive*. Upload the *CoreMedia Cartridge for Salesforce* to the *Salesforce Commerce Cloud* system to activate the *CoreMedia Content Widget*. This is described in the instructions inside the CoreMedia Workspace for Salesforce Commerce Cloud Zip file.

Using the CoreMedia Content Widget

You can have one or more slots using a *CoreMedia Content Widget* per page. You might have, for example, a page with a main slot with content from the CMS or another page with a header and a footer coming from the CMS. Figure 5.4, "Using the *CoreMedia Content Widget* – A Homepage Fragment" [28] shows a site from Salesforce SiteGenesis, that uses the *CoreMedia Content Widget*. It fills the main area of the page (everything within the blue frame) and, in addition, shows a sales banner at the top (in the orange frame).

You can have one or more slots using a *CoreMedia Content Widget* per page. You might have, for example, a page with a main slot with content from the CMS or another page with a header and a footer coming from the CMS. The figure below shows a site from Salesforce SiteGenesis, that uses the *CoreMedia Content Widget*.
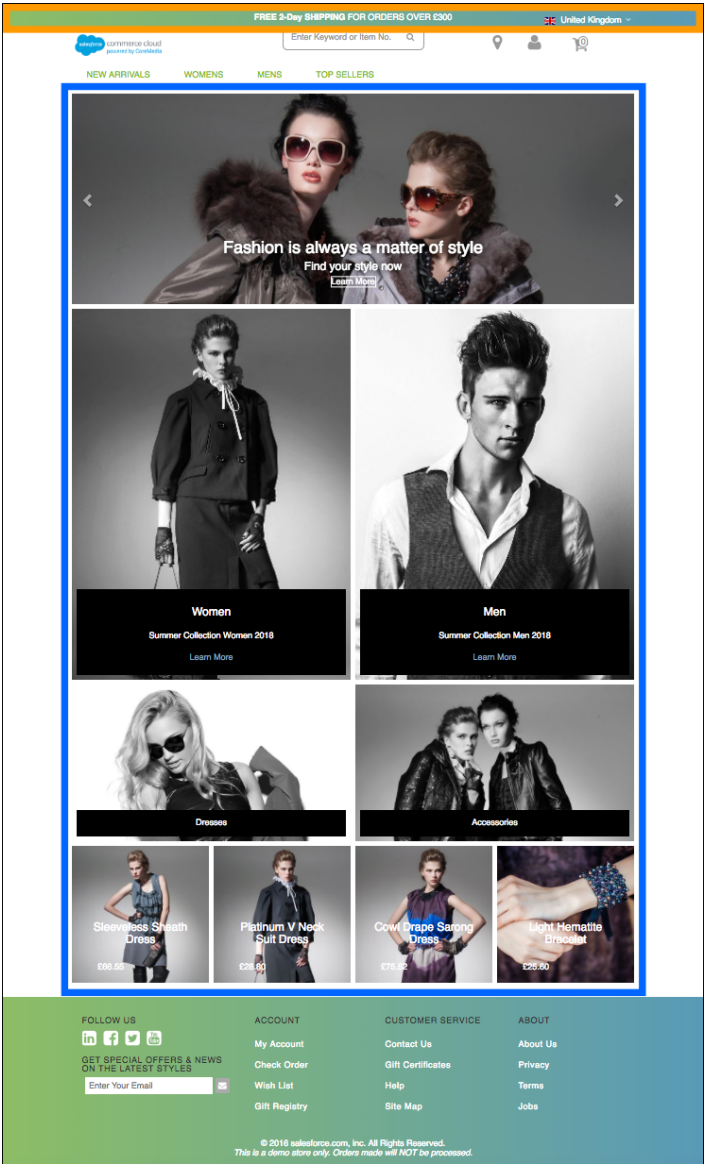
Figure 5.4. Using the CoreMedia Content Widget – A Homepage Fragment

Configuring a Content Slot for Content Widget

To show CoreMedia content on the pages, you need to create a content slot and use it on the page. You can use the *Salesforce Commerce Business Manager* for this task. Figure 5.5, "Content Slot Configuration Example" [29] shows the editing form of such a content slot. To use the *CoreMedia Content Widget* set the *Content Type* field to `Content Asset` and type `slots/content/core media-content-widget.isml` into the *Template* field. This is the path where the template is stored in the *CoreMedia Cartridge for Salesforce*.

Figure 5.5. Content Slot Configuration Example

In the *CoreMedia* section of the form, three additional values can be set to identify the content and the view that should be used on the CMS side.

The CoreMedia Content Widget gets its content from pages in the CoreMedia system. Therefore, the parameter $pageId$ is sent to the CMS. By default, the value of the parameter is taken from the commerce content in which the slot is used. However, when you want to access a different page, you can set the ID in the *"CoreMedia Content ID"* field. The value must correspond to the *"External Page ID"* field that is set on the proxy page in *CoreMedia Studio* on the CMS side. Figure 5.6, "External Page ID set via CoreMedia Studio" [30] shows the corresponding *CoreMedia Studio* form, but for another example, an `about-us` page.

CoreMedia Content ID Parameter
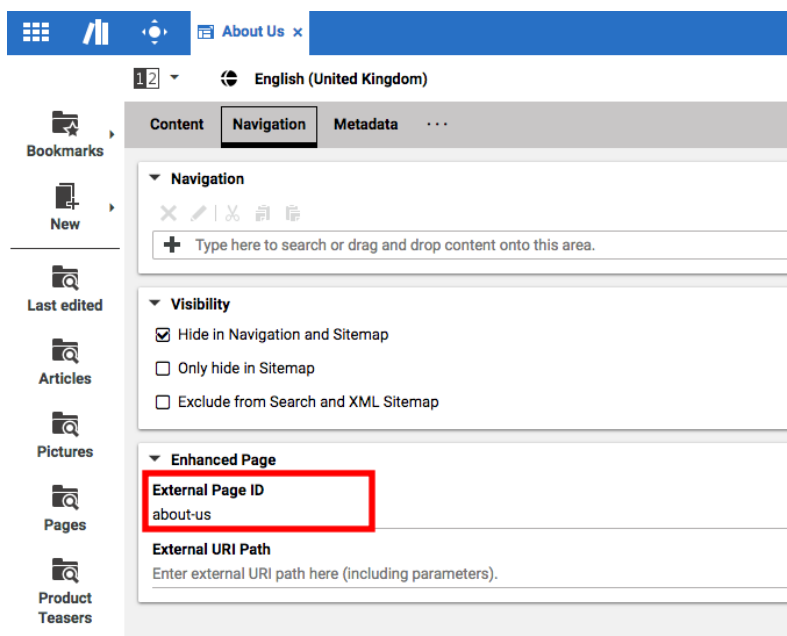
Figure 5.6. External Page ID set via CoreMedia Studio

The content of a page in the CMS is located in so–called placements, a specific, named position in the page grid of a page layout. Here, a *Studio* editor enters the content. In the *"Name of the placement to render"* field, you enter the name of the placement from which you want to get the content for the commerce page. If the field is left empty, the full page grid is taken. However, the placement setting can be overridden by the *Name of view to render* field.

Placement and View Parameter

> ## NOTE
>
> The name of the placement shown in *Studio* is the localized label. The value of the placement field in the *CoreMedia Content Widget* must match the technical name in the page grid definition. You can find the definitions in the `Option/Settings/Pagegrid/Layout` folder in *Studio*. The name is the value of the *Section* entry in the Struct property. Usually this is written in small letters.

The *Name of view to render* field defines a view, which will be used to display the content of the page. Such views have to be prepared on the CMS side, be–cause they must exist at runtime. A view overrides the placement parameter.

That is, it might use it, but it can also take content from other placements and arrange them in the way the developer of the view intended. With such a view it is possible to recompose the content completely. If no view is set, the default view is taken on the CMS side. The CoreMedia default view shows the placement set in the *"Name of the placement to render"* field.

Pitfalls: When to use the CoreMedia Content Widget and when to use the islcinclude Tag

Technically, the *CoreMedia Content Widget* can be used easily on content slots with a `global` context (such as the Homepage), but also in the catalog area with the context `category`, so that you have the current category available as a render parameter.

The "category" Context Problem

However, it is not possible to express an "and all subcategories" semantic in the category based slot configuration. That means, a slot defined in a Category is not automatically inherited in its subcategories. Therefore, the slot configuration must be done for each category where the *CoreMedia Content Widget* should be displayed. This might make sense on category landing pages or on other special featured categories but certainly not on all other lower categories. This is even more important, when the categories change frequently, since the slots are cached.

So, when it is not sensible to use the Content Widget, consider to change the template and add the `islcinclude` tag directly instead of using an `isslot` tag. See the `categoryproducthits.isml` as an example.

Providing the product as the current context is not supported by the *CoreMedia Content Widget*. Therefore, when you want the current product being available you cannot use the Content Widget on Product Detail Pages (PDPs). In addition, as the slot mechanism is also used for independent caching of fragments, it would be questionable to do so on product basis. For CMS fragments on PDPs use the `islcinclude` tag directly in templates and pass the `productId` as a parameter.

Don't use the Content Widget on Product Detail Pages!

There are still other conceivable constellations in which a *CoreMedia Content Widget* does not fit well or it would be rather too expensive to change an existing template structure completely. Generally spoken, as soon as the flexibility the Content Widget offers you is not necessary, for example, when there will be no change of a page structure between two releases, then always use the `islcinclude` tag instead of the *CoreMedia Content Widget*. The `islcinclude` tag is easier to control that all required parameters are reaching the fragment context (see Section 5.2.2, "The CoreMedia Include Tags" [33] for the description of the tag).

When in doubt, use the islcinclude tag directly in templates!

CoreMedia Content Widget on Other Pages

"Other Pages" ("about-us", for instance) are not part of the catalog hierarchy and for such pages the *CoreMedia Content Widget* can also be used. The same additional attributes as for slots are placed on the *Salesforce* editing form for Content Assets. See the following screenshot of the "about-us"page as an example.

Figure 5.7. Content Asset Configuration Example

The additional attributes *"CoreMedia Content ID"*, *"Name of placement to render"* and *"Name of view to render"* have the same meaning as in the slots described above. However, you do not have to set the rendering template in the form. The CoreMedia supplied SiteGenesis template `contentpage.isml` renders the content fragment above the original content defined in the *Body* field of the Content Asset. To replace the whole content with the content delivered by the

CMS, remove the text from the *Body* field. However, you can also change the behavior in the template, instead.

The *"CoreMedia Content ID"* is used again to set the transmitted `pageId` parameter explicitly to identify the page within the CMS. The parameter is optional and if not given, the page identifier is automatically taken from the commerce system. Set this field when the same CMS page is reused on multiple shop pages.

5.2.2 The CoreMedia Include Tags

islcinclude

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

```
<iscontent type="text/html" charset="UTF-8" compact="true"/>
<isinclude template="coremedia/modules.isml"/>

<!-- COREMEDIA HEADER -->
<isset name="pageId" value="${cmUtil.pageId(pdict)}" scope="page"/>
<isset name="categoryId" value="${cmUtil.categoryId(pdict)}" scope="page"/>
<isset name="productId" value="${cmUtil.productId(pdict)}" scope="page"/>
<islcinclude pageId="${pageId}" categoryId="${categoryId}"
productId="${productId}" placement="header"/>
```

All parameters are described in the Include Tag Reference section.

The `islcinclude` tag from CoreMedia renders the CMS fragments in the same context of the caller. That means all the following code would have access to the results of this call. This technique is, for example, especially useful for the `metadata` call. This is different to the `islcincludeRemote` tag that will be described describe later.

islcincludeVar

In some cases you might want to decide what to do next, depending on the result of a fragment call. For such a case you can use the `islcincludeVar` tag. It stores the result in a `fragmentPayload` page variable and the HTTP status in a separate `fragmentHttpStatus` variable. You could now, depending on the status, either print the fragment payload to the output stream or do an alternative rendering.

As an example you can use this technique to decide whether the navigation should be rendered by the CMS or the shop. In the template you can ask the

CMS if it is able to render the navigation. If there is a result status of "200", then the fragment payload can be printed to the response. Otherwise, the original shop template should do the work.

```
<isinclude template="coremedia/modules.isml"/>
<iscomment>Render CoreMedia Navigation if available</iscomment>
<isset name="pageId" value="${cmUtil.pageId(pdict)}" scope="page"/>
<islcincludeVar pageId="${pageId}" view="asNavigation" />
<isif condition="${fragmentHttpStatus == '200'}">
  <iscomment>
    Render the output of the navigation fragment call into the page.
    The fragment response is already encoded and shouldn't be encoded twice!

  </iscomment>
  <isprint value="${fragmentPayload}" encoding="off"/>
  <iselse/>
  <iscomment>
    The original SFRA template was copied. Please verify if the original is
    changed and should be renewed.
  </iscomment>
  <isinclude template="components/header/menu-original" />
</isif>
```

islcincludeRemote

As a specialty of the *Salesforce Commerce* platform fragments can be rendered in a remote call for the reason of cacheability and reusability. In ISML templates an iscomponent can be used to achieve this. With the islcincludeRemote tag it is possible to enforce a remote call to gather a CMS fragment. The CMS fragment will then be rendered in the remote context with its own pipeline dictionary. But the parameters of this tag are mostly the same as for the islcinclude tag except of the prefetch and ajax parameters. Both parameters make no sense in the remote case, because the fragment is requested in a completely new context (by a new HTTP call). This new context serves only this single fragment and a further prefetch of all fragments would result in an unnecessary rendering effort on the CAE side. Same applies to the ajax parameter. The actual fragment call is made by the browser. The required AJAX stub code is so small that it does not have to be cached separately.

```
<div class="header-banner">
  <iscomment>CoreMedia include of header</iscomment>
  <isset name="pageId" value="${cmUtil.pageId(pdict)}" scope="page"/>
  <islcincludeRemote pageId="${pageId}" placement="header"
```

```
view="asDefaultFragment"/>
</div>
```

> **NOTE**
>
> The *CoreMedia Content Widget* is using the `islcinclude` tag. The reason for this is that it makes it easier to transfer computed values into the caller context and thus influence the subsequent rendering. For example, the processing of the HTML metadata makes use of it (to set the HTML title and meta tags).

ⓘ

Include Tag Reference

The tag attributes have the following meaning:

Parameter	Description
productId, *category-Id*	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 over-written. 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* 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 CMS page you may omit it. 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.

Parameter	Description
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 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. The attribute can be used in combination with the *view* and/or *parameter* attribute.
prefetch	This attribute is used to signal the *CoreMedia Fragment Connector* a prefetch of all fragments should be made before requesting the fragment. At best, this should lead to a single call that gets all wanted fragments that will follow in the same request context. A following fragment call can then be served from the local cache, or if not found, will be made in the traditional way. This attribute is optional and the default value is *false*. That means, you have to actively find out which fragments are the first to be rendered on a page and set the parameter to true.
ajax	This attribute is used to signal the *CoreMedia Fragment Connector* that a AJAX stub code should be written into the output instead of calling the CAE. The link that is set into stub code points to the CM-Dynamic controller with the fragment URL as parameter. The usage of this parameter has two advantages. The stub code can be written in no time and therefore does not delay the processing of the whole page. In addition, these fragments are not counted into the quota for external requests. This parameter is optional and the default value is *false*. Please note that setting this parameter to true cannot be combined with prefetch=true because such an include will not trigger a CAE request that can do a prefetch.
parameter	This attribute is optional and can 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

Parameter	Description
	`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 built when the commerce system requests the fragment information from the CAE (see next section).

Table 5.1. Attributes of the Include tag

Finding Handlers

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

The CoreMedia `islcinclude` tag requests data from the CAE via HTTP. Each attribute value of the include tag is passed as path or matrix parameter to the `FragmentPageHandler`. In order to find the matching handler, the `FragmentPageHandler` class calls the `include` method of all fragment handler classes defined in the file `livecontext-fragment.xml`. The first handler that returns "true" generates the HTML. Example 5.1, "Default fragment handler order" [37] shows the default order:

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

Example 5.1. Default fragment handler order

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

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

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

Table 5.2. Fragment handler usage

Using Commerce-side Includes

Up to this point you have already seen CMS fragments that are embedded in the store-side HTML output. But one twist further it is also possible the other way around: to define placeholders in CMS templates that will be replaced later during the shop rendering (as server-side includes). This is already used by default for creating URLs in CMS fragments.

A "Velocity rendering technique" is used to achieve this. The *Salesforce* system has already the possibility to write Velocity expressions in templates as an alternative scripting mechanism. For example such Velocity expressions can be used to include other components or even to call each publicly exported script function.

It is possible to write Velocity script directly into CMS-side FreeMarker templates. Such a Velocity script section must be included into an HTML comment section to have an unbroken output of fragments even without the Velocity script engine (for example, if you call the fragments directly in a browser).

A Velocity sections must start with a `<!--VTL` text and end with `VTL-->`. The following examples will illustrate this.

- VTL scripts cannot be nested. Be careful with includes of further templates within such a Velocity section that may contain more Velocity scripts. Be aware that rendering a link within a Velocity script (using `cm.getLink()`) would lead to such a situation. Rather don't use any includes in VTL scripts.

- Velocity expressions start with a "$" char. Additionally, the "#" char is also reserved. If you want to use these chars around a Velocity expression but within a VTL section you have to mask these characters manually. Use "$D" instead of "$" and $H instead of "#".

This mechanism is currently prepared for four use cases. To support these cases, there is a file `cartridge/scripts/cmInclude.js` which contains publicly exported script functions that can be used directly.

Rendering Commerce Links

It is already used by default. The `SfccLinkResolver` class is part of the *eCommerce Blueprint* and generates Velocity expression instead of HTML links into the fragment output. It ensures that commerce links are built exclusively on the Commerce side. The CMS does not need to know anything about the resulting format (for example, SEO mapping on/off is transparent for the CMS).

In general the CMS-side format complies to the canonical URL format in the *Salesforce Commerce Cloud* platform. Parameters can be passed directly to certain controllers. There are various types of possible target pages/controllers. In the following example a category page link is shown. Each target page type has its own allowed set of parameters. Please see the class `SfccLinkResolver` to get further information.

```
<!--VTL
$include.url('Search-Show','cgid','womens-clothing-dresses','preview','true')
 VTL-->
```

Overwriting HTML Metadata

The CMS can overwrite the finally used HTML metadata for the HTML title, keywords and description tag by using the `metadata` function in CMS templates. This is typically the case for content driven pages. The following code shows an example from the `Page.metadata.ftl` template.

```
<!--VTL $include.metadata('${content.htmlTitle}','${content.htmlDescription}',
'${content.keywords}') VTL-->
```

Including any Salesforce Controller

Salesforce Commerce Cloud reusable components are typically implemented as a controller with its own URL to call. Any *Salesforce* controller can be included by giving its name and all required parameters. The same ones you need to call the controller on the *Salesforce* side. There is a `controller()` function prepared in the `cmInclude.js` file that can be used. The following code example shows how the include a product teaser that is rendered by the *Salesforce* platform. All parameters can also be calculated dynamically.

```
<!--VTL
$include.controller('Product-HitTile','pid','25448070','showswatches','true',
'showpricing','true','showpromotion','true','showrating','true') VTL-->
```

Including any ISML Template

Less often it will be necessary to embed an `ISML` template. A `template()` function is prepared in the `cmInclude.js` file for such cases. This example shows the include of a template `example.isml` which renders a product teaser again. The called `ISML` template must be located in the `cart ridge/templates/default/coremedia/cms` directory of the *CoreMedia Cartridge for Salesforce*.

```
<!--VTL $include.template('example','pid','682875090845','showswatches','true',
'showpricing','true','showpromotion','true','showrating','true') VTL-->
```

Including the Availability of a Product

The availability of a product in stock can be tested on the shop side by calling the `availability()` function. The prepared function in the `cmInclude.js` file expects at least one argument: the product ID. The result of this method is a string that indicates if the product is available in stock or not. If not given as separate parameters the method returns `true` or `false`. Alternative strings can be passed as second and third argument.

```
<!--VTL $include.availability('${self.product.externalId}') VTL-->
or
<!--VTL
$include.availability('${self.product.externalId}','available','not-available')
 VTL-->
```

Calling custom Script Functions

Only exported functions from the `cartridge/scripts/cmInclude.js` file can be called by default. As you can see in the examples above, they are all exposed below the context `include`. To call your own functions you can add these functions to the file. To call other function from other files or even other cartridges, more `requires` directives would have to be added to the `renderVelocity()` function in `cartridge/scripts/cmVtlPro cessor.js`. An alternative would be to overwrite the whole `cmInclude.js` module in our own custom cartridge and copy and extend the code.

Fragment Request Context

In addition to the passed request parameters, a personalization context is built via the `cmContextProvider.js` script as part of the *CoreMedia Cartridge for Salesforce*. The default implementation can be extended with custom values. The context information is then passed as header attributes to the CAE. For more details see Section 5.3, "Extending the Shop Context" [42].

5.3 Extending the Shop Context

To render personalized or contextualized info in content areas it is important to have relevant shop context info available during CAE rendering. It will be most likely user session related info, that is available in the Commerce system only and must now be provided to the backend CAE. Examples are the user id of a logged in user, gender, the date the user was logged in the last time or the names of the customer 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 place 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 script where the context info is gathered. To add custom information to the context please extend the prepared `scripts/context/cmContextProvider.js` script in the *CoreMedia Cartridge for Salesforce*. The exported functions in this script are called by the `cmFragmentService` when the context is built to pass it to the backend CAE. The packing, transmitting and unpacking of the values happen automatically.

Extending the ContextProvider

To extend the shop context you have to edit the `cmContextProvider.js`. There are three prepared exported function that are called by the `cmFragment Service` to build up the context information. By default, a base set of context information is already gathered and can be extended with custom values. Alternatively you can implement your own `cmContextProvider` by overwriting this module in your own customization cartridge and prepending it in the cartridge path.

Function	Description
getPreviewContext	Gathers all preview related context information that should be sent as request headers. By default, an existing preview date is provided in the format *2018-04-17 18:30:00.000* and the associated timezone. The used request headers are: *wc.preview.timestamp* and *wc.preview.timezone*. This information is used by the Studio to render a preview assuming a certain date in the future.

Function	Description
getPreviewParams	Gathers all preview related context information that should be sent as request parameters. By default, an existing test persona (*p13n_test* and *p13n_testcontext*) and a preview date (*timestamp*) with its timezone (*timezone*). This information is used by the Studio to render a preview assuming a certain test persona (like Sarah or Matt) and a different time.
getUserContext	Gathers all user session related context information that should be sent as request headers. By default, current customer groups of a logged in user are provided. The used request header is: *wc.user.membergroupids*. This information is used by the CAE to personalize the rendering accordingly.
getUserParams	Gathers all user session related URL parameters. By default, this list is empty.

Table 5.3. Functions of the cmContextProvider.js script

NOTE

The prefixes *wc.preview* and *wc.user* are automatically added by the connector and must not be provided as prefixes.

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.

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 `FragmentCommerceContex tInterceptor` in the *CAE*. This interceptor creates and stores a `Context` object in the request. You can access the `Context` object via `LiveCon textContextHelper.fetchContext(HttpServletRequest  request)`.

Example 5.2. Access the Shop Context in CAE via Context API

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

Example Request Flow
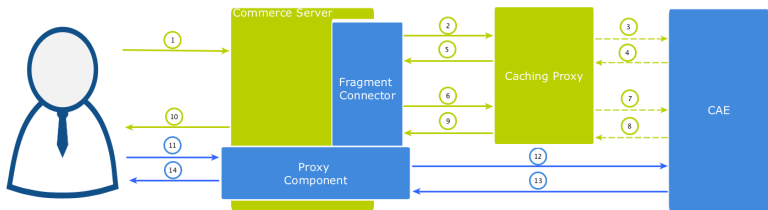
Figure 5.8. Example request flow

Figure 5.8, "Example request flow" [46] 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 post-processing of the received fragment payload is an important step carried out by both the Proxy Component and the *CoreMedia Fragment Connector*. At this point, their processing is similar. Links to other shop pages which may be contained in a fragment coming from the CAE must be post-processed in the Commerce system. This is because the knowledge about the final link format is in the Commerce system. In addition, other server side includes can also be done, for example, the rendering of a price info.

See the section Section 5.2.2, "The CoreMedia Include Tags" [33] for more information concerning the topic "Using Commerce-side Includes".

The CoreMedia Proxy Component is part of *CoreMedia Cartridge for Salesforce* and will be installed with all other CoreMedia customizations. Technically it is a *Salesforce* controller with the name `CM-Dynamic` and a single *url* parameter. This parameter contains an encoded *CAE* URL that is then be called by the controller, post-processed (all containing links will be generated) and the result is finally sent to the browser.

```
<div class="cm-fragment"
data-cm-fragment="/on/demandware.store/Sites-SiteGenesisGlobal-Site/en_GB/CM-Dynamic?
url=%2fblueprint%2fservlet%2fdynamic%2fplacement%2fp13n%2fsitegenesis-en-gb%2f132%2fplacement%2fmain%3f
targetView%3d%255Bcarousel%255D%26amp%3bp13n_test%3dtrue%26amp%3bp13n_testcontext%3d0%26amp%3b
fragmentContext%3d%2fSiteGenesisGlobal%2fen-GB%2fparams%3b...%3bview%253DmergedPlacements...&preview=true">
</div>
```

Example 5.3. AJAX Stub

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

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

Adding Context Information to Dynamic Calls

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

personalized content be used, along with caching between Commerce server and *CAE* please make sure all relevant context data are provided in the *CoreMedia Fragment Connector*. Please see the Section 5.3, "Extending the Shop Context" [42]. for details.

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

5.5 Using Salesforce Page Cache for CMS Fragments

This section discusses the ability of using the Salesforce Page Caching for CMS fragments. In general, the CMS fragments are added to the *Salesforce Page Cache* just like the parts that render the shop itself. Since this cache operates on the granularity of Salesforce controllers, usually several CMS fragments are cached together if they weren't included with a `islcincludeRemote` tag for themselves.

After a fragment is retrieved from the CMS the *Connector for Salesforce Commerce Cloud* can set cache directives to control the Salesforce Page Caching. This is essentially a *setExpires* call on the response. *Salesforce Commerce* automatically evaluates all cache times for a page (or a certain controller output) and will choose the minimum time to cache the page.

With the *Salesforce Storefront Developer Tools* you can see the current effective cache times per controller output. In this example, several homepage fragments are put together to one cacheable page. The responsible controller is `Home-Show`.
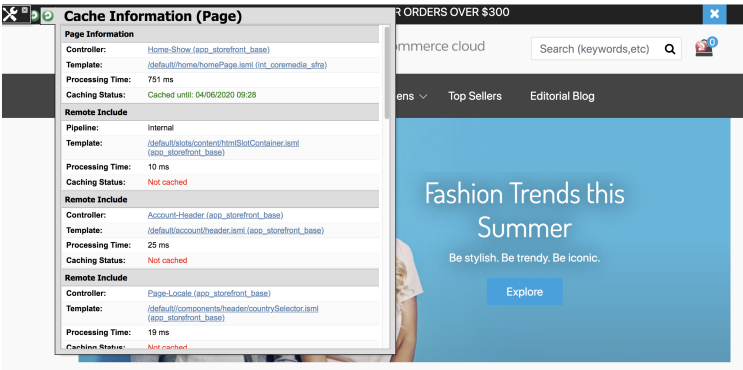
Figure 5.9. Storefront Cache Information

Every CMS fragment within this cacheable unit can also influence the cache time by setting the minimal value. There are two possible situations that can be handled differently, either if the CMS fragment was loaded successfully or if an error has occurred. For both cases, there is a configuration setting (see Table 5.4, " Cache settings " [51]) in the *CoreMedia Custom Site Preferences* that controls

the CMS fragment caching. You can add them in the *Salesforce Commerce Business Manager*.

cmPageCacheOnErrorTTL	
Default	Disabled
Description	If an error occurs, the fragment should probably not be cached for a long time. By default, the expiration time is not set. CoreMedia recommends entering a moderate value here, for example, 60 seconds, to avoid flooding the server that is in trouble with too many requests.

cmPageCacheDefaultTTL	
Default	Disabled
Description	If a fragment could be loaded successfully, you can define the expiration time. By default, no expiration time is set. This value should be aligned with the expected frequency of page changes and the requirement for the topicality of the site. CoreMedia recommends a higher value, for example, 3600 seconds.

Table 5.4. Cache settings

CAUTION

Please note that using the cache TTL for CMS fragments affects the enclosing page. And please also note that page caching is switched off, by default, in *Salesforce Commerce*. That means if the surrounding template doesn't already use page caching for itself, a `setExpires()` call on the response would enable the caching of the whole page/fragment. If such a page must not be cached (for example, to display the most current information), caching can be disabled in individual cases as described below.

Disabling caching on demand

Only for CoreMedia Cartridge for Salesforce version 3.4.x and higher

NOTE

The following functionality is only available with *CoreMedia Cartridge for Salesforce* version 3.4.x and higher.

The standard routine looks for a custom request attribute that prevents fragment caching (`setExpires()` will not be called). If this is desired, then set the request attribute `request.custom.shouldBeCached` to `false`.

If this simple logic is not sufficient for your demands, you can also overwrite it in your own cartridge that is placed in front of the `int_coremedia` cartridge in the path. To do this, create a script `cmCacheControl.js` in the directory `scripts` and implement your own `shouldBeCached` function.

```
/**
 * Function that can be overwritten in customer projects to decide if
 * caching is enabled for a fragment. If page caching is generally
 * switched off and outer, surrounding templates must not be cached,
 * it would be counterproductive if the include of an CMS fragment
 * enables the caching (response.setExpires()).
 * The default implementation evaluates a custom request attribute
 * 'shouldBeCached'. If not found it returns true.
 * Note, this only applies if CoreMedia fragment caching is switched on.
 *
 * @param {string} fragmentUrl - the fragment url
 * @param {Object} request - the current request
 * @returns {boolean} true if caching is enabled
 */
exports.shouldBeCached = function (fragmentUrl, request) {
  var enabled = request.custom.shouldBeCached;
  if (enabled === null) {
    enabled = true;
  }
  if (enabled) {
    Logger.debug('caching is enabled for "' + fragmentUrl + '"');
  } else {
    Logger.debug('caching is disabled for "' + fragmentUrl + '"');
  }
  return enabled;
};
```

Example 5.4. scripts/cmCacheControl.js example

Let the CMS control the fragment caching

> **NOTE**
>
> The following functionality is only available with *CoreMedia Cartridge for Salesforce* version 3.4.x and higher.

Instead of configuring the expiry times in the Salesforce system, you can also use the expiry information sent by the CMS response, either as HTTP response header or within the JSON structure as part of the prefetch (see Section 5.6, "Prefetch Fragments to Minimize CMS Requests" [55]). For example, if the CMS sends the seconds for a day as `max-age` value in the `Cache-Control` header, these seconds are converted into a date and set as expiry date on the response.

> **When CMS expiry information is not used**
>
> • The CMS information will not be used, when the `cmPageCacheDefaultTTL` custom site setting in the *CoreMedia Custom Site Preferences* is set to "-1" or when caching is disabled by the `shouldBeCached` function.
>
> • The CMS information will also not be used when the value configured with the *CoreMedia Custom Site Preferences* property `cmPageCacheDefaultTTL` is smaller than the value send by the CMS.

To control the *Salesforce* page caching CoreMedia provides the script `cmCacheControl.js`. It supports two variants of HTTP headers to extract the expiry information from the CMS response:

• Standard procedure for HTTP 1.1

  The script tries to read the standard headers from the response to determine an expiry date. First of all it looks for a `Cache-Control` header with a `max-age` value in seconds. A given `Age` header is also considered (and subtracted if given).

• Procedure for HTTP 1.0

  The script looks for an `Expiry` header together with a `Date` header (and subtracts it if given).

Depending on the success of the fragment request, the script contains two methods (see Table 5.5, " Cache Control methods " [53]), which decide which expiry to set in the Salesforce response.

`setPageCacheExpiryOnSuccess`

Description	Implements the cache control in case of success (no error has been occurred when getting the fragment from the CMS). Either the expiry date is already determined by the value found in the prefetch response or it is read from existing headers (`Cache-Control`/`Age` headers or `Expires`/`Date` headers). When the configured Salesforce default time (`cmPageCacheDefaultTTL`) is even shorter, then the default is used.
	Note, this method is only called if the `cmPageCacheDefaultTTL` value is set (greater than -1) and the `shouldBeCached` method evaluates to true.

`setPageCacheExpiryOnError`

Description	Implements the cache control in case of an error (when getting the fragment from the CMS). By default the configured `cmPageCacheOnErrorTTL` value is used to set on response.
	Note, this method is only called if the `cmPageCacheOnErrorTTL` value is set (greater than -1) and the `shouldBeCached` method evaluates to true.

Table 5.5. Cache Control methods

This default behavior can easily be overwritten and customized in your own cartridge. It just has to be set in the cartridge path in front of this script.

5.6 Prefetch Fragments to Minimize CMS Requests

A shop page in the commerce-led scenario can contain multiple CMS fragments (placements and views). Normally, each CMS fragment would cause an external HTTP call to the CAE which can lead to performance loss and, depending on the commerce system, reach a limit of outgoing requests on the commerce side (see Figure 5.10, "Multiple Fragment Requests without Prefetching" [55]). Furthermore, each request is processed consecutively. As a result, the response times for each individual CAE request add up to the total pageview time. Therefore, CAE offers a mechanism to lower the amount of CAE requests by prefetching all expected fragments in advance in a single call.
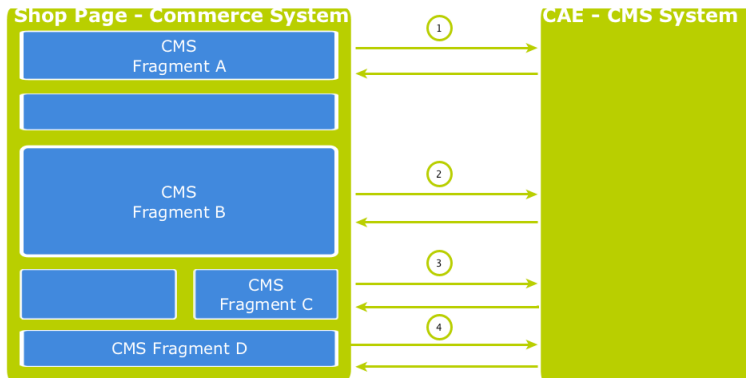
Figure 5.10. Multiple Fragment Requests without Prefetching

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: `externalRef`, `productId`, `categoryId` and

pageId). The CAE therefore uses these parameters to predict the required fragments. Every placement in the assigned page layout can be considered as "potentially to be requested". Therefore, every placement is contained as a separate fragment in the JSON result. To identify the view that should be used to render the placement a configuration is read from the *LiveContext* Settings content. The Figure 5.11, "LiveContext Settings: Prefetch Views per Placement" [57] shows an example configuration. If no setting can be found, it is assumed that the default view should be rendered for a placement.

- Additionally, every shop page requests a few more, mostly technical fragments from the CAE. These fragments are requested as different "views" of the same page. Examples of such views are *metadata*, *externalHead* and *externalFooter* that are likely to be included on every shop page. These "additional views" are also read from the *LiveContext* Settings content and they are also included in the JSON result. The Figure 5.12, "LiveContext Settings: Prefetching Additional Views" [58] shows an example of such a configuration.

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

Prefetch Configuration: View per Placement

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

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

Figure 5.11. 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 over–written by a layout specific configuration in the `layouts` struct element.

layouts   Defines a layout-specific view with which a placement will be prefetched. It overrides the view defined in the `defaults` struct element for this specific placement.

Prefetch Configuration: Additional Views

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

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

Figure 5.12. LiveContext Settings: Prefetching Additional Views

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

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

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

contentTypes   Defines the views that should be prefetched for a specific content type on Content Pages (see Section 5.2, "Adding CMS Fragments to Shop Pages" [25] 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.

Required configuration in the *Salesforce Project Workspace*

The prefetch functionality has to be enabled with the *Custom Site Preference* `cmPrefetch`. Go to the *Merchant/Tools/Site Preferences/CoreMedia* page in the *Business Manager* and set the *Enable Prefetch* flag.

If the feature is turned on for a site, then each occurrence of the `islcinclude` tag also can decide for itself if a prefetch should be performed (in case if it is not already done in this request scope). There is an optional parameter *prefetch* of the `islcinclude` tag. This is, because the *Salesforce Commerce Cloud* system often uses remote includes which trigger sub-calls to the same instance. Every remote include has then a new request context. If another `islcinclude` occurs in such a remote context it would lead to a complete new prefetch call of the page (at least if it was not already done in this new request scope). It turned out to be better to set this parameter to *false* by default and to set all places that should trigger the prefetch explicitly.

The prefetch should only be done within the main request context. All secondary request contexts (triggered by a remote include) should fetch single CMS fragments by a regular fragment call. To do that, all `islcinclude` places that are used in the main request context (or at least the first one) should set the *prefetch* parameter explicitly to *true*. Typically, these are the *metadata* and the *header* calls.

You can find more information about the usage of the `islcinclude` tag in the Section 5.2.2, "The CoreMedia Include Tags" [33].

5.7 Configure Logging

Configure Logging Categories for the CoreMedia Cartridge

The *Custom Log Settings* dialog in the *Business Manager* can be found below *Administration/Operations*. It should be used to control the log output of the *CoreMedia Cartridge for Salesforce*. The following example shows a configuration where all CoreMedia log outputs are set to level `INFO`, apart from the certain log category `coremedia.context`. It is set to `DEBUG`. CoreMedia uses log categories to control the log output and to differentiate between various function blocks.

Figure 5.13. Configure Logging Categories for CoreMedia Cartridge

Existing Logging Categories

The following log categories exist and can be used to control the log output of the *CoreMedia Cartridge for Salesforce*.

coremedia.connector   The main *CoreMedia Fragment Connector* compon-
ents; the central CoreMedia controller functions,
fragment contexts, resources and service classes.

coremedia.service	CoreMedia HTTP service instances that are using the *Salesforce Commerce Cloud* base HTTP service. Here can be logged, which URI is actually used to call the CoreMedia system.
coremedia.context	The context provider that gathers all information that should be passed to the CMS system. These are preview- and/or user-related information. If any info is missing, please look at this category.
coremedia.cache	Fragments can be cached in the request scope when using the prefetch functionality. Use this category to observe the caching behavior (for example, hits and misses).
coremedia.parser	Components which are responsible for fragment parsing to replace placeholders. Such placeholders are used to realize server-side includes. If you have any difficulties that some placeholder are not re-placed as expected, you can use this category.
coremedia.include	This category is used in functions which can be in-cluded as servicer-side includes. If you have problems with one of your function that are meant to be in-cluded, you can log into this category. It is somehow related to the *coremedia.parser* category. If you want to have the complete insight into the parse/in-clude mechanism, you can use both categories at the same time.
coremedia.util	Some basic utility functions which are used by various components.

6. Studio Integration of Commerce Content

In *CoreMedia Content Cloud* each content site can be configured with a specific shop instance to deliver content pages mixed with Commerce catalog items. The term "Commerce catalog items" means all items that live only in the commerce catalog. Nevertheless, these elements are to be interwoven with content on mixed pages.

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

- Section 6.1, "Catalog View in CoreMedia Studio Library" [63] gives a short overview over the Catalog Integration in the *Studio* Library.
- Section 6.3, "Commerce related Preview Support Features" [68] gives a short overview over the commerce related preview functions that are supported in *CoreMedia Studio*.
- Section 6.4, "Augmenting Commerce Content" [70] describes how you augment commerce content in the commerce-led scenario in *CoreMedia Studio*.

6.1 Catalog View in CoreMedia Studio Library

When the connection to a *Salesforce Commerce* system and a concrete shop for a content site are configured as described in Chapter 4, *Connecting to a Salesforce Commerce Cloud System* [12], the *Studio* Library shows the commerce catalog to browse product categories and products in the commerce catalog and to search for products and product variants. After the editor has selected a preferred site with a valid store configuration the catalog view will be enabled and the catalog will be shown in the Library:
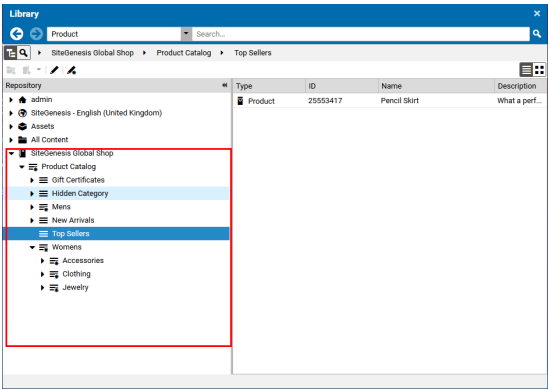
Figure 6.1. Library with catalog in the tree view

In some catalogs it is possible to put a category on multiple places within the catalog tree. But the Commerce Hub ensures that a category can only have one home (a unique parent category). All additional occurrences of a category are shown as a link in the tree. If you click on such a link node you will automatically end up at the place in the tree where the category is actually at home.
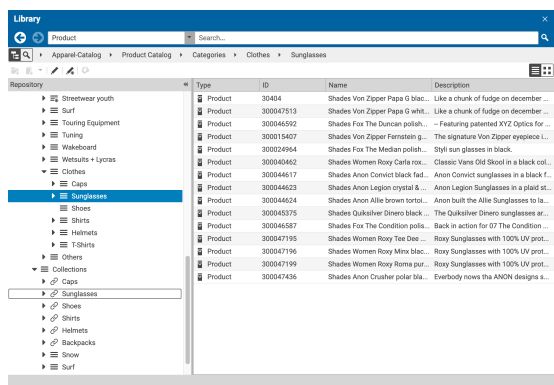
Figure 6.2. Library tree with multiple occurrences of the same category

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

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

While browsing through the catalog tree you can also open a preview of a category or a product from the library. Simply double-click on a product in the product list or use the context menu on a product or a category and choose the entry **Open in Tab** from the context menu as shown in the pictures below.
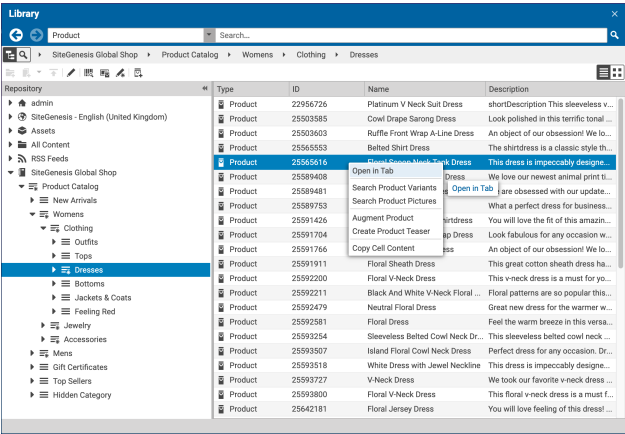
Figure 6.3. Open Product in tab
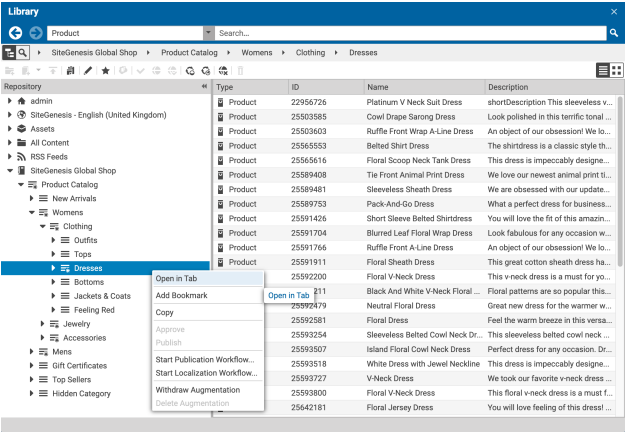
Figure 6.4. Open Category in tab

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

6.2 Enabling Preview in Shop Context

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

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

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

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

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

1. Make sure the customization coming with the *CoreMedia Workspace for Salesforce Commerce Cloud* has been applied to your *Salesforce Commerce Cloud* installation (see Chapter 3, *Customizing Salesforce Commerce Cloud* [11]).

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

Configure in the CoreMedia system

NOTE

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

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

6.3 Commerce related Preview Support Features

CoreMedia Studio supports a variety of commerce preview functions directly:

- Time based preview (time travel)
- Customer segment based preview

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

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

Figure 6.5, "Test Customer Persona with Commerce Customer Segments" [68] shows an example where the test persona is female and has already been registered.
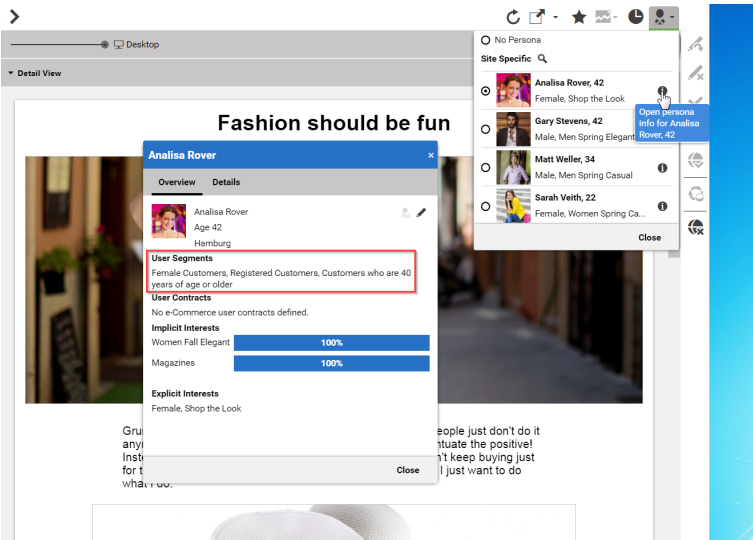
Figure 6.5. Test Customer Persona with Commerce Customer Segments

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

The test persona content can be created and edited in *CoreMedia Studio*. The customer segments available for selection will be automatically read from the commerce system. By default, all user segments available in the eCommerce system are displayed for selection. Under some circumstances it may be desirable to restrict the shown user segments, for instance for studio performance reasons or for better clarity for the editor. See Section 3.2.4, "Configuring The PersonaSelector" in *Personalization Hub Manual*.
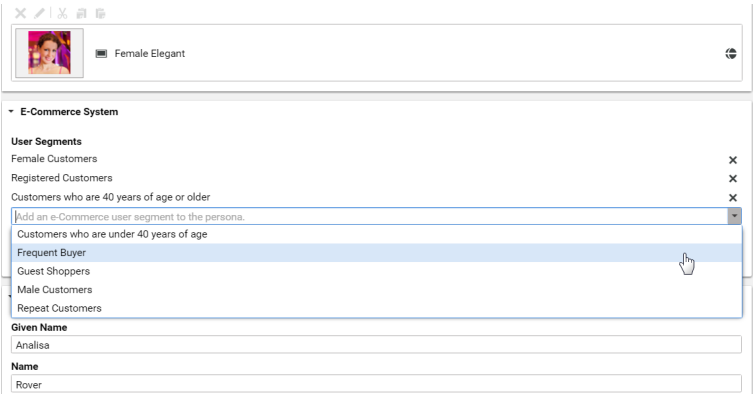
Figure 6.6. Edit Commerce Segments in Test Customer Persona

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

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

6.4 Augmenting Commerce Content

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

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

1. In the CMS create a content item of type `Augmented Category`, `Augmented Product` or `Augmented Page`.

2. Augment the root nodes of the catalogs as described in Section 6.4.1, "Augmenting the Root Nodes" [70].

3. When you augment a category or product, the connection between the category/product and the `Augmented Category`/`Augmented Product` content is automatically created. For the `Augmented Page` you have to create this connection manually via an external page id property

4. In the `Augmented Category`, `Augmented Product` or `Augmented Page` choose a page layout that corresponds to the shop page layout. It should contain all the placements that are referenced in the *CoreMedia Content Widgets* defined on the Commerce side.

5. Drop the augmenting content into the right placements of the augmented content item. That is, into a placement whose name corresponds with the name defined in the *CoreMedia Content Widget*.

6.4.1 Augmenting the Root Nodes

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, *Site Genesis,*). 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.

Catalog view in Studio

To have a common ancestor for all augmented catalog pages, the root node of the configured catalog must be augmented. You can augment the root category by clicking *Augment Category* in the context menu of the root category. An augmented category content opens up, where you can start to define the default elements of your catalog pages, like the page layouts for the Category Overview Pages (CLP) and Product Detail Pages (PDP) and first content elements. All sub

Augmented catalog roots

categories, augmented or not, will inherit these settings. See Section 6.2.3, "Adding CMS Content to Your Shop" in *Studio User Manual* for more information.
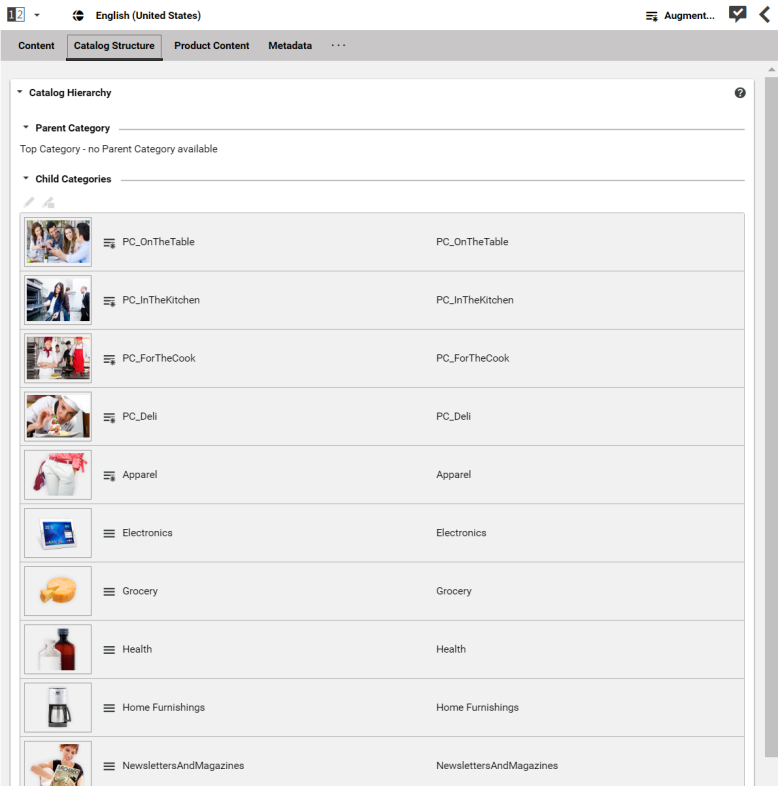
Figure 6.7. Catalog structure in the catalog root content item

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

6.4.2 Selecting a Layout for an Augmented Page

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

to the actual shop page layout. All usable placements should be addressed. The placement names must match the placement names used in the slot definition on the shop side.
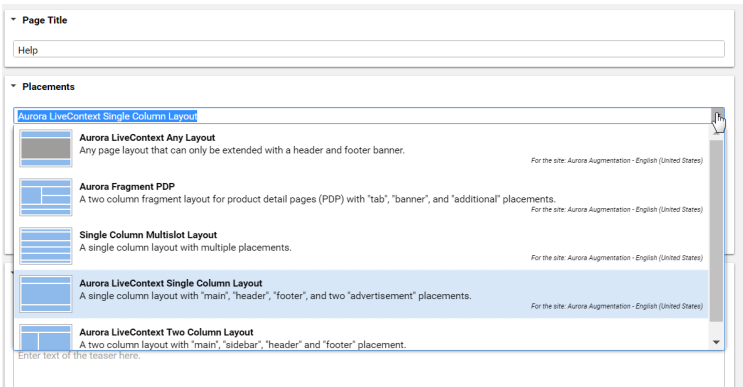
Figure 6.8. Choosing a page layout for a shop page

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

6.4.3 Finding CMS Content for Category Overview Pages

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

Category overview pages

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.

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

Information passed to the CoreMedia system

system uses this information to locate the content in the CoreMedia repository that should be shown on the category overview page.

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

Locating the content in the CoreMedia system

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

1. Select the `Augmented Page` that is connected with the shop.

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

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

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

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

Figure 6.9, "Decision diagram" [74] shows the complete decision tree for the determination of the content for the category overview page or the product detail page (see below for the product detail page).
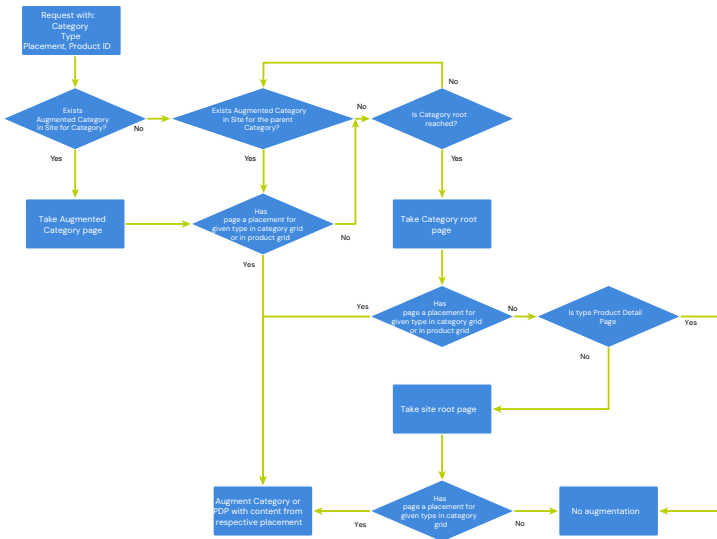
Figure 6.9. Decision diagram

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

6.4.4 Finding CMS Content for Product Detail Pages

Product detail pages 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.

Product Detail Pages

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

Information passed to the CoreMedia system

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

Locating the content in the CoreMedia system

The content to augment is taken from a separate page grid of the `Augmented Category`, called *Product Content* or from the *Content* tab of the `Augmented Product.`
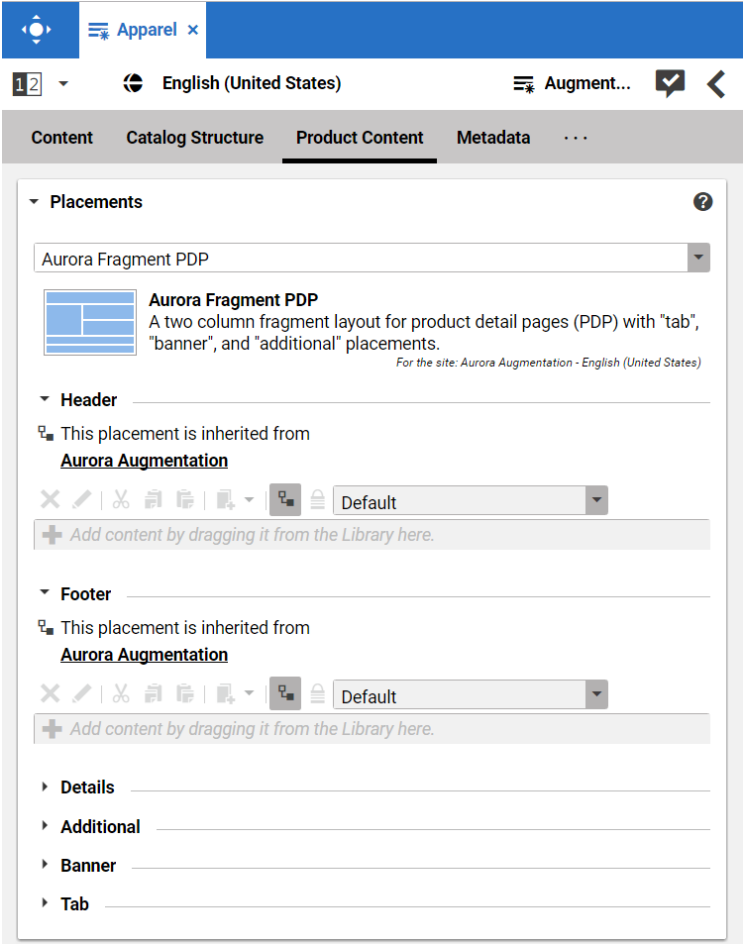
Figure 6.10. Page grid for PDPs in augmented category

Adding CMS Assets to Product Detail Pages

You can enhance product detail pages with assets from the CoreMedia system by adding the *CoreMedia Product Asset Widget*.

Product detail pages

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

Information passed to the CoreMedia system.

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

Locating the assets in the CoreMedia system

6.4.5 Adding CMS Content to 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.

Non Catalog Pages (Other Pages)

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

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

1. Make sure, that the layout of the page in the commerce system contains the *CoreMedia Content Widget*.

2. Create a content item of type *Augmented Page* and add it to the *Navigation Children* property of the site root content.

3. Enter the ID of the other page below the navigation tab into the *External Page ID* field of the `Augmented Page`.

4. Optional: Set the *External URI Path* if special URL building is needed.

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

Figure 6.11. Example: Contact Us Pagegrid

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.

Difference between the augmentation of catalog and other pages

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

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

Figure 6.12. Example: Navigation Settings for a simple SEO Page

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

Special Case: Homepage

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`.

Special Case: Homepage

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

The *External URI Path* field is also likely to remain empty, unless the shop site is to be accessible with an URL, which still has a path component (for example, `../en/aurora/home.html`). But in most cases you wouldn't want that.
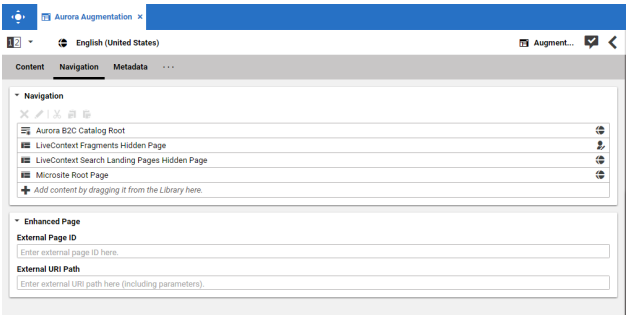
Figure 6.13. Special Case: Navigation Settings for the Homepage

7. Commerce Caching

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

Commerce-Hub Cache Infrastructure

Caching of commerce entities is implemented in different layers of the Commerce Hub infrastructure:
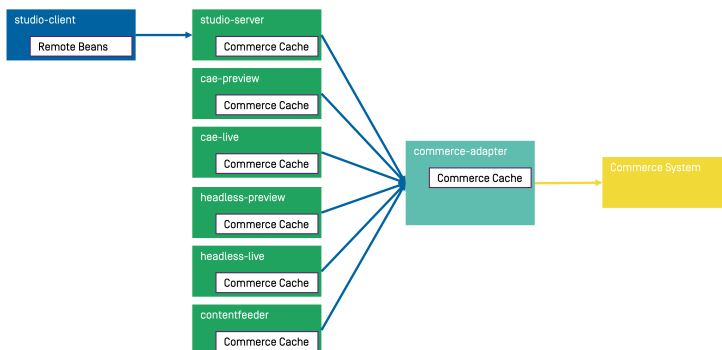
Figure 7.1. Multiple levels of caching

- Caching is implemented in the Commerce Adapter to accelerate access to commerce entities and to avoid heavy traffic on the *Salesforce Commerce Cloud* 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 *Salesforce Commerce Cloud* 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 *Salesforce Commerce Cloud*. 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 *Salesforce Commerce Cloud*. If the update cycles of the *Salesforce Commerce Cloud* are known or if changes can be detected automatically and be used to trigger a script invoking the `invalidate` actuator, then long cache times can be configured to hold commerce entities in the cache as long as possible.

The following figure shows the actuator component in the Commerce Adapter and the direction of events propagating the invalidation.
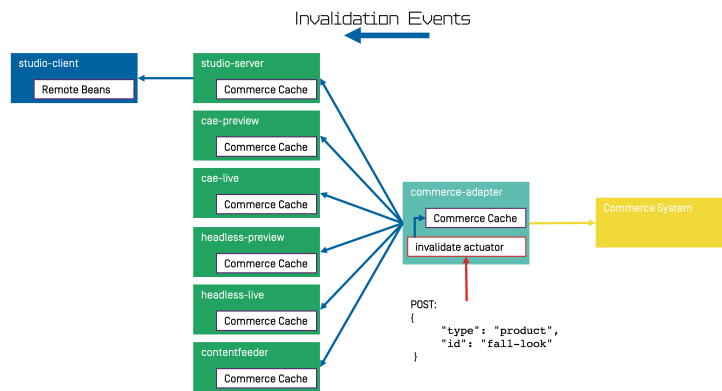
Figure 7.2. Commerce Cache Invalidation

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

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

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

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

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

Examples:

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

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

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

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

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

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

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

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

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

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

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

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

> **NOTE**
>
> Please note, there is no automatic mechanism involved that is able to trigger the invalidation when a commerce item is changed in the *Salesforce Commerce Cloud*. 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:

- Immediately respond to requests with the restored response.
- Replay the restored requests so that the cache fills with up-to-date values served by the *Salesforce Commerce Cloud*.

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

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

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

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

ⓘ

Cache Configuration of the Commerce Adapter

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

ⓘ

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

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

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

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

Common base adapter config keys:

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

of each cache entry should be small, but can increase if custom attributes are used.

segments The list of all customer segments referenced by ID.

segment A customer segment with its properties. The memory consumption of each cache entry is very small.

Vendor specific config keys:

accesstoken API access tokens. There is no effect in setting the cache time. The cache time will be computed according to the expiration time of the requested token.

categoryidbyproduct Used to map products/SKUs to category IDs. The memory consumption of each cache entry is very small.

productshop To retrieve prices for products and SKUs. Prices can only be got from the Shop API. Please note, there is no distinction between base products and variants/SKUs. Keep this in mind when choosing a capacity value! The memory consumption depends on the size of the REST response from the commerce system. Each entry consumes ~20kB heap memory.

productdata Used in services that are not covered by the base adapter caching, like `PriceService`, `LinkService` etc. Please note, there is no distinction between base products and variants/SKUs. Keep this in mind when choosing a capacity value! Each entry consumes ~40kB heap memory.

facetplaceholdermapping The global map of presentation IDs to build product filter facets.

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

Commerce Cache Configuration of Commerce Adapter Clients

NOTE

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

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

Refer to the Chapter 9, *Commerce Adapter Properties* [90] 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.*`

ACTUATOR URLS

Service	Actuator Shortcuts	Status
Content Management Server	Info · Logfile · Environment · Config · Health	HEALTHY
Master Live Server	Info · Logfile · Environment · Config · Health	HEALTHY
Workflow Server	Info · Logfile · Environment · Config · Health	HEALTHY
Content Feeder	Info · Logfile · Environment · Config · Health	HEALTHY
User Changes	Info · Logfile · Environment · Config · Health	HEALTHY
Elastic Worker	Info · Logfile · Environment · Config · Health	HEALTHY
CAE Feeder Preview	Info · Logfile · Environment · Config · Health	HEALTHY
CAE Feeder Live	Info · Logfile · Environment · Config · Health	HEALTHY

Figure 7.3. Actuator URLs in overview page

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

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

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

8. The eCommerce API

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

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

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

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

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

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

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

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

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

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

9. Commerce Adapter Properties

cache.capacities	
Type	`Map<String,Long>`
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	`Map<String,Long>`
Description	TTL in seconds until certain cache entries are invalidated.

entities.circuit-breaker-names	
Type	`Map<String,String>`
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	`String`
Default	base
Description	The default breaker name.

entities.disable-circuit-breakers	
Type	Boolean
Default	false
Description	Disable circuit breakers and cache failed calls in cache class *failed*.

entities.exponential-backoff.factor	
Type	Double
Default	1.5
Description	The factor to be applied to the delay to compute the next delay.

entities.exponential-backoff.initial-delay	
Type	Duration
Default	2s
Description	The initial delay of the backoff.

entities.message-store.files	
Type	Map<String,Long>
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
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	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	Boolean

Default	false

Description	Whether to recompute entities proactively on invalidation.

`entities.send-invalidations`

Type	Boolean

Default	true

Description	Whether or not to propagate invalidations of entities to the clients.

`metadata.additional-metadata`

Type	Map<String,String>

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

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	Collection<String>
Description	List of parameter names, which values need to be transmitted with every entity request from the CMS side.

`metadata.replacement-tokens`

Type	Map<String,String>
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	String
Description	Name of the vendor.
	Used to identify the connected vendor on the CMS side.

`sfcc.default-locale`

Type	Locale
Default	us
Description	The default locale for accessing the commerce system if no locale parameter was passed into request.

`sfcc.image-view-type-large`

Type	String
Default	large
Description	Configure the view type name of image groups used for large product images.

`sfcc.image-view-type-small`

Type	String

Default	medium
Description	Configure the view type name of image groups used for small product images.

`sfcc.link.link-templates`

Type	Map<String,String>
Description	Map of link templates. Only lookup keys lowercase and without "_" are valid.
	Known default lookup keys are defined in StorefrontRefKeysCommerceLed.
	These patterns can include tokens which will be replaced. These tokens must be well known. The following tokens are predefined:

- {storefrontUrl} ... the current store front URL
- {storeId} ... the current store id
- {locale} ... the current locale in java format, eg. en_US
- {language} ... the current language in java format, eg. en
- {catalogId} ... the current catalog id
- {categoryId} ... the current category id
- {productId} ... the current product id
- {seoSegment} ... the current seo segment path (can contain path delimiters)
- {storefrontUrl} ... the current store front URL
- {customerGroup} ... the current user group, if available
- {previewDate} ... the preview date, if available

`sfcc.link.link-templates.categorylinkfragment`

Type	String
Default	<!--VTL $include.url('Search-Show','cgid','{categoryId}') VTL-->
Description	Used to generate category page links into CoreMedia fragments.

`sfcc.link.link-templates.categorypreviewurl`

Type	String
Default	{storefrontUrl}/Sites-{storeId}-Site/{locale}/CM-RedirectUrl?link=Search-Show,cgid,{categoryId},preview,true&__siteDate={previewDate}&__customerGroup={customerGroup}
Description	Used to build the preview URL to a category page.

sfcc.link.link-templates.cmajaxlinkfragment	
Type	String
Default	<!--VTL $include.url('CM-Dynamic','url','{url}') VTL-->
Description	Used to generate ajax urls to CoreMedia contents into CoreMedia fragments.

sfcc.link.link-templates.cmcontentlinkfragment	
Type	String
Default	<!--VTL $include.url('CM-Content','contentId','{seoPath}') VTL-->
Description	Used to build links to shop pages displaying CoreMedia Articles and Channels into CoreMedia fragments.

sfcc.link.link-templates.cmcontentpreviewurl	
Type	String
Default	{storefrontUrl}/Sites-{storeId}-Site/{locale}/CM-RedirectUrl?link=CM-Content,contentId,{seoSegment},preview,true&__siteDate={previewDate}&__customerGroup={customerGroup}
Description	Used to build the preview URL to a shop page which displays a CoreMedia content.

sfcc.link.link-templates.externalpagepreviewurl	
Type	String
Default	{storefrontUrl}/Sites-{storeId}-Site/{locale}/CM-RedirectUrl?link=Page-Show,cid,{pageId},preview,true&__siteDate={previewDate}&__customerGroup={customerGroup}
Description	Used to build the preview URL to a shop page.

sfcc.link.link-templates.homepagelinkfragment	
Type	String
Default	<!--VTL $include.url('Home-Show') VTL-->

Description	Used to the link to the home page.

`sfcc.link.link-templates.homepagepreviewurl`

Type	String
Default	{storefrontUrl}/Sites-{storeId}-Site/{locale}/CM-RedirectUrl?link=Home-Show,preview,true&__siteDate={previewDate}&__customerGroup={customer-Group}
Description	Used to build the preview URL to the shop home page.

`sfcc.link.link-templates.productlinkfragment`

Type	String
Default	<!--VTL $include.url('Product-Show','pid','{productId}') VTL-->
Description	Used to build product detail page links into CoreMedia fragments.

`sfcc.link.link-templates.productpreviewurl`

Type	String
Default	{storefrontUrl}/Sites-{storeId}-Site/{locale}/CM-RedirectUrl?link=Product-Show,pid,{productId},preview,true&__siteDate={previewDate}&__customer-Group={customerGroup}
Description	Used to build the preview URL to a product detail page.

`sfcc.link.link-templates.shoppagelinkfragment`

Type	String
Default	<!--VTL $include.url('Page-Show','cid','{seoPath}') VTL-->
Description	Used to build URLs to shop pages into CoreMedia fragments.

`sfcc.link.storefront-url`

Type	String
Description	Base URL of the commerce storefront

sfcc.link.storefront-url-for	
Type	Map<String,String>
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:

```
sfcc.link.storefront-url-for.us=https://sandbox-us.demandware.net/on/demandware.store/
```

```
sfcc.link.storefront-url-for.de=https://sandbox-de.demandware.net/on/demandware.store/
```

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=us`

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 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.

sfcc.link.timezone-conversion-enabled	
Type	Boolean
Default	true
Description	Enable the conversion of a Studio preview date (perviewDate param) to a sfcc-side preview date (siteDate param). An additional call (shop API /site) is made to determine the "site timezone". Access to the site call must be permitted in the OCAPI Settings of your sandbox for this to work. Set this to "false" to restore the old behavior.

sfcc.oauth.client-id	

Type	String

Description	ClientID used for all Data and Shop REST API Calls to the Salesforce Commerce System. Used to set permissions for the ClientID on Shop and Data API – for example, which resources the ClientID is allowed to access

`sfcc.oauth.client-password`

Type	String

Description	Password used together with the clientId.

`sfcc.oauth.host`

Type	String

Default	account.demandware.com

Description	Host name of central SFCC endpoint for authentication.
	No need to customize.

`sfcc.oauth.path`

Type	String

Default	/dwsso/oauth2/access_token

Description	Path to retrieve access token.

`sfcc.oauth.protocol`

Type	String

Default	https

Description	Protocol used to request access token.

`sfcc.oauth.retry-delay`

Type	Duration

Default	5s

Description	The time after which a retry is attempted to fetch an ocapi access token. Until then, all requests that require an access token will end with an IllegalStateException in log.

sfcc.ocapi.custom-attributes-for

Type	Map<String,List<String>>

Description	Configure attribute names, which are transmitted to the client as customAttributes. The key is the name of the OCAPI Document in lowercase and removed "_" (*com.coremedia.commerce.adapter.sfcc.ocapi.AbstractOCDocument.getOcType()*).

The value is a comma separated list of attributes, which shall be available on the client side via com.coremedia.livecontext.ecommerce.common.CommerceBean#getCustomAttributes.

The value is transmitted as String representation of the JSON Object.

Example:

```
sfcc.ocapi.custom-attributes-for.product=image_groups,c_isSale
```

sfcc.ocapi.custom-expand-parameters-for

Type	Map<String,List<String>>

Description	Configure expand parameter names, which are requested from the commerce system via the "expand" parameter. The keys should be defined in lower case without special characters.

The value is a list of expand parameter values.

Example:

```
sfcc.ocapi.custom-expand-parameters-for.products=images,prices,variations
```

sfcc.ocapi.custom-localized-attributes-for

Type	Map<String,List<String>>

Description	Same as *customAttributesFor* but for localized properties. Only the value for the current locale of the request is transmitted to the client.

sfcc.ocapi.data.customer-groups.count

Type	Integer
Description	Optional count for retrieving only a subset of all customer groups. If not set, the default behaviour of the salesforce Data API will apply.

`sfcc.ocapi.host`

Type	String
Description	Host name (FQDN) of your SFCC Instance.

`sfcc.ocapi.http-client.accept-cookies`

Type	Boolean
Default	false
Description	Setting if cookies should be accepted.
Deprecation	This property has been deprecated and will be removed in a future version. Use `commerce.rest.client.cookie-spec` instead. **Reason:** use base adapter configuration option

`sfcc.ocapi.http-client.connection-pool-size`

Type	Integer
Default	20
Description	Defines the overal connection limit for a connection pool.
Deprecation	This property has been deprecated and will be removed in a future version. Use `commerce.rest.client.connection-pool-size` instead. **Reason:** use base adapter configuration option

`sfcc.ocapi.http-client.connection-request-timeout-ms`

Type	Integer

Default	60000
Description	Http Client Configuration for Rest communication with SFCC OCAPI Services.
Deprecation	This property has been deprecated and will be removed in a future version.
	Use `commerce.rest.client.connect-request-timeout` instead.
	Reason:
	use base adapter configuration option

`sfcc.ocapi.http-client.connection-timeout-ms`

Type	`Integer`
Default	60000
Description	Http Client Configuration for Rest communication with SFCC OCAPI Services.
Deprecation	This property has been deprecated and will be removed in a future version.
	Use `commerce.rest.client.connect-timeout` instead.
	Reason:
	use base adapter configuration option

`sfcc.ocapi.http-client.max-connections-per-route`

Type	`Integer`
Default	2
Description	Defines a connection limit per one HTTP route.
Deprecation	This property has been deprecated and will be removed in a future version.
	Use `commerce.rest.client.max-connections-per-route` instead.
	Reason:
	use base adapter configuration option

`sfcc.ocapi.http-client.network-address-cache-ttl-ms`

Type	`Integer`

Default	30000
Description	Http Client Configuration for Rest communication with SFCC OCAPI Services.
Deprecation	This property has been deprecated and will be removed in a future version. Use `commerce.rest.client.network-address-cache-ttl` instead. **Reason:** use base adapter configuration option

`sfcc.ocapi.http-client.socket-timeout-ms`

Type	Integer
Default	60000
Description	Http Client Configuration for Rest communication with SFCC OCAPI Services.
Deprecation	This property has been deprecated and will be removed in a future version. Use `commerce.rest.client.socket-timeout` instead. **Reason:** use base adapter configuration option

`sfcc.ocapi.http-client.trust-all-ssl-certificates`

Type	Boolean
Default	true
Description	Setting if client should trust all ssl certificates.
Deprecation	This property has been deprecated and will be removed in a future version. Use `commerce.rest.client.trust-all-ssl-certificates` instead. **Reason:** use base adapter configuration option

`sfcc.ocapi.load-all-sku-images`

Type	Boolean

Default	false
Description	Set to true if your base products often have no images assigned. With this flag enabled also SKU image groups are loaded alongside with the product data. If no image is assigned to a product a sku image is used as fallback. Note that this may increase the data footprint between commerce system and adapter. see "all_images" below SFCC Documentation

`sfcc.ocapi.protocol`

Type	String
Default	https
Description	Protocol used for OCAPI REST communication.

`sfcc.ocapi.sandbox`

Type	Boolean
Default	false
Description	Set to true if integrating with a sandbox instance. Adjust base paths for REST API requests
	On sandbox instances the base paths must be prefixed with '/s/{siteId}' see: SFCC Documentation

`sfcc.ocapi.type`

Type	com.coremedia.commerce.adapter.sfcc.common.OcapiType
Description	Configures the OCAPI type to be used to load data from the commerce system. Available types are *shop* and *data*.

`sfcc.ocapi.version`

Type	String
Default	v24_5
Description	Version of OCAPI Rest Service used.

`sfcc.search-limit`

Type	Integer
Default	200
Description	The default search limit as supported by the SFCC backend.

`sfcc.single-value-search-facets`

Type	List<String>
Description	List of facet keys. These facets only support single values to be selected.

Table 9.1. SFCC 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 mod–ules:

- *CoreMedia Content Management Server*
- *CoreMedia Workflow Server*
- *CoreMedia Importer (Deprecated since CMCC v12.2506.0.0)*
- *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	The set of all content items in a multi-site hierarchy related to each other via master references. This includes the top-level master content items themselves.
Version history	A newly created content item receives the version number 1. New versions are created when the content item is checked in; these are numbered in chronological order.
Weak Links	In general *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