COREMEDIA CONTENT CLOUD

Commercetools Connector Manual

List of Figures

List of Tables

1. Preface

This manual describes how the CoreMedia system integrates with *commercetools*.

- Chapter 2, *Overview* [6] gives a short overview of the integration.
- Chapter 3, *Connecting to a commercetools System* [11] describes how you connect a CoreMedia web application with a *commercetools* system.
- Chapter 4, *Studio Integration of Commerce Content* [18] shows the eCommerce features integrated into *CoreMedia Studio*.
- Chapter 5, *Commerce Caching* [31] describes the CoreMedia cache for eCommerce entities.
- Chapter 6, *The eCommerce API* [38] describes the basics of the eCommerce API.
- Chapter 7, *Commerce Adapter Properties* [40] describes the configuration properties for the commerce adapter.

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*, , *commercetools, Spring*, *Maven* and *Docker*.

1.2 Typographic Conventions

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

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

Table 1.1. Typographic conventions

In addition, these symbols can mark single paragraphs:

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

Table 1.2. Pictographs

1.3 Change Record

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

Section	Version	Description

Table 1.3. Changes

2. Overview

This manual describes how the CoreMedia system integrates with *commercetools*. You will learn how to access the *commercetools* catalog from the CoreMedia system and how to develop with the *eCommerce API*.

2.1 Commerce Hub Architecture

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

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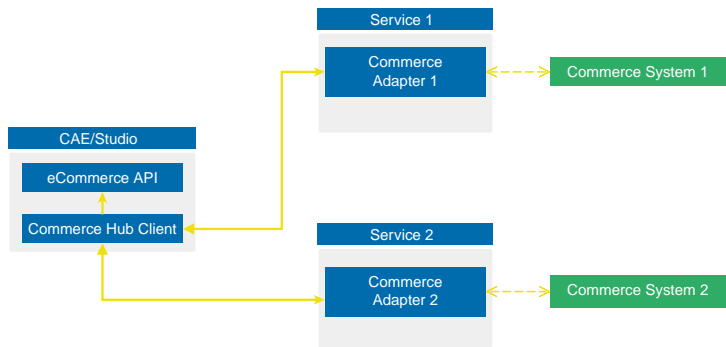
Figure 2.1. Architectural overview of the Commerce Hub

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

The commerce system specific part exists in a service with the commerce system specific connector. The connector uses the API of the commerce system (often REST) to get the commerce data. In contrast, the generic Commerce Hub client and the Commerce Connector use gRPC for communication (see https://grpc.io/) for details.
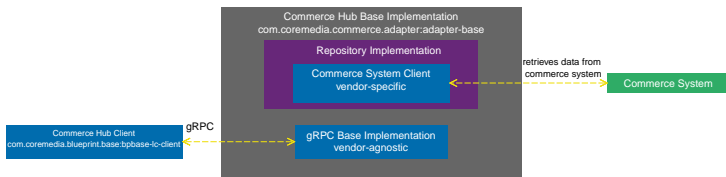
Figure 2.2. More detailed architecture view

Figure 2.2, " More detailed architecture view " [7] shows the architecture in more detail. At the Commerce Hub Client, you only have to configure the URL of the service and some other options, while at the Commerce System Client, you have to configure the commerce system endpoints, cache sizes and some more features.

2.2 Commerce Hub API

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

The gRPC API

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

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

The Java API

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

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

Request flow

The request flow, using the above described APIs, starting from the generic client is as follows. Please have a look at Figure 2.2, " More detailed architecture view " [7] first.

1. The generic client sends a gRPC request to the vendor agnostic *base adapter*. The Entities API is used to convert the Java entity to the corresponding gRPC message.

2. The gRPC service implementation in the *base adapter* receives the gRPC request and invokes the corresponding repository methods.

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

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

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

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

3. Connecting to a commercetools System

The connection of your *Blueprint* web applications (*Studio* or *CAE*) to a *commercetools* 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 *commercetools* system (see Section 3.1, "Configuring the Commerce Adapter" [12]).

- Settings configuration in *Studio*. It references the Commerce Adapter endpoint, which *Studio* and *CAE* use to indirectly communicate via the Commerce Adapter with *commercetools* (see Section 3.2, "Configuring the Shop in Content Settings" [13]).

> **WARNING**
>
> In addition to these configurations, CoreMedia requires an external identifier for every commerce item in order to provide stable references for augmented content. In the *commercetools* system these external identifiers are called `keys`.
>
> Setting these keys for every commerce item is a prerequisite for a working *commercetools* integration.

3.1 Configuring the Commerce Adapter

The physical connection to the *commercetools* system is configured in the Commerce Adapter. The Commerce Adapter itself makes use of the JVM SDK, provided by *commercetools*.

The Commerce Adapter comes along with a set of configuration properties. Most of them have defaults and need no further customization.

For basic configuration set the following properties:

- `commercetools.api.project-key`
- `commercetools.api.client-id`
- `commercetools.api.client-secret`
- `commercetools.api.auth-url`
- `commercetools.api.api-url`

Spring Boot offers several ways to set the configuration properties, see Spring Boot Reference Guide - Externalized Configuration.

For more details and the full set of configuration properties see Chapter 7, *Commerce Adapter Properties* [40].

3.2 Configuring the Shop 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, for example, which store ID 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 configure these settings.

▼ commerce		Struct
endpoint	${commerce.hub.data.endpoints.commerce...	String
currency	USD	String
locale	en-US	String
▼ storeConfig		Struct
name	Commercetools Sunrise Shop	String
id	DefaultStore	String
▼ catalogConfig		Struct
id	commercetools	String

Figure 3.1. Example Commerce Settings

The following store specific settings must be configured below the struct property named `commerce` as shown in Figure 3.1, "Example Commerce Settings" [13]

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

Name	Type	Description	Example	Required
locale	String Property	The ISO locale code for the connected Catalog. This over-writes 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 be retrieved from the site locale.
storeConfig	Struct Property	Struct property containing store configuration		true
storeConfig.id	String Property	The ID of the store.	DefaultStore	true
storeConfig.name	String Property	The name of the store as it is set in the commerce system.	Commercetools Sunrise Shop	true
catalogConfig	Struct Property	Struct property containing catalog configuration.		true
catalogConfig.id	String Property	The ID of the catalog.	commercetools	true

Table 3.1. 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.

3.3 Building and Running the Commerce Adapter

You can run the Commerce Adapter in a Docker container provided by CoreMedia.

In order to build and run the container, you need the following tools:

- Maven
- Docker
- Docker Compose (optional)

Proceed as follows:

1. Clone the workspace from https://github.com/coremedia-contributions/commerce-adapter-commercetools. It contains a Docker setup for the *commercetools Connector*.

2. Build the workspace with `mvn clean install` to create a `coremedia/commerce-adapter-commercetools` Docker image

3. When you run the Docker container, you have to provide the required configuration properties for the adapter (see Section 3.1, "Configuring the Commerce Adapter" [12]). The most common options would be either setting environment variables (using the Docker option `--env` or `--env-file`) or mounting a configuration file (using the Docker option `--volume`).

   Start the Docker container with the following command:

   ```
   docker run \
     --detach \
     --rm \
     --name commerce-adapter-commercetools \
     --publish 44165:6565 \
     --publish 44181:8081 \
     [--env ...|--env-file ...|--volume] \
     coremedia/commerce-adapter-commercetools:${ADAPTER_VERSION}
   ```

Integrating the adapter container into Blueprint Docker environment

To run the `commerce-adapter-commercetools` Docker container with the *CoreMedia Content Cloud* Docker environment, add the `commerce-adapter-commercetools.yml` compose file, which 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-commercetools.yml
COMMERCETOOLS_API_AUTH_URL=...
...
```

The `commerce-adapter-commercetools` container is started with the *Core-Media Content Cloud* Docker environment when running

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

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

3.4 Checking the Functionality

Prerequisites

- All commerce entities in you *commercetools* project are equipped with an external identifier, the `key`.
- The *CoreMedia Content Cloud* infrastructure has been deployed and is running.

Check the Studio - *commercetools* Connection

1. Open *Studio*, select the "Commercetools Sunrise - English (United States)" site, open the Library. If necessary, switch the Library to browse mode.
2. In the repository tree view, locate a node named *Commercetools Sunrise Shop*. This is the entry point to browse the connected *commercetools* product catalog.
3. Browse the catalog in Studio and check if everything works as expected. Section 4.1, "Catalog View in CoreMedia Studio Library" [19] 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 3.2, "Configuring the Shop in Content Settings" [13].
- Check if the *Connector for commercetools* is configured correctly (see Section 3.1, "Configuring the Commerce Adapter" [12]).

4. Studio Integration of Commerce Content

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

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

- Section 4.1, "Catalog View in CoreMedia Studio Library" [19] gives a short overview over the Catalog Integration in the *Studio* Library.
- Section 4.2, "Augmenting Commerce Content" [23] describes how you augment commerce content in the commerce-led scenario in *CoreMedia Studio*.

4.1 Catalog View in CoreMedia Studio Library

When the connection to a *commercetools* system and a concrete shop for a content site are configured as described in Chapter 3, *Connecting to a commercetools System* [11] 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:
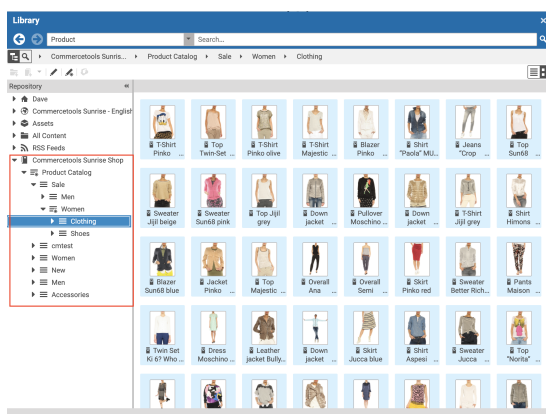
Figure 4.1. Library with catalog in the tree view

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.
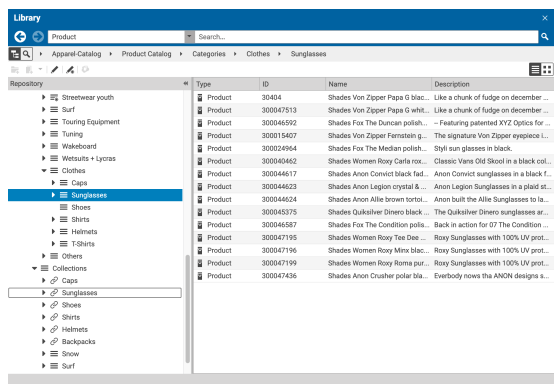
Figure 4.2. Library tree with multiple occurrences of the same category

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.
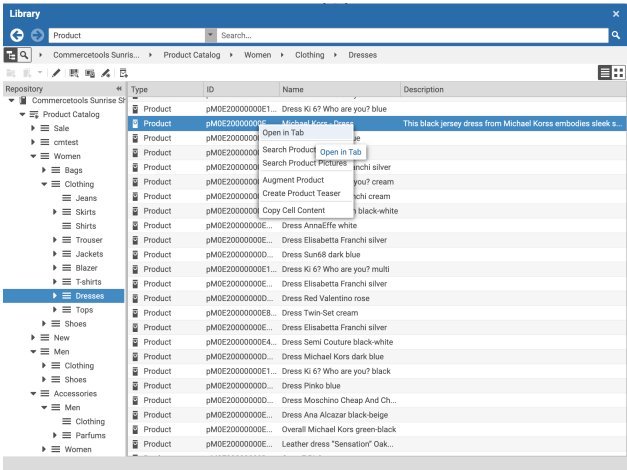
Figure 4.3. Open Product in tab
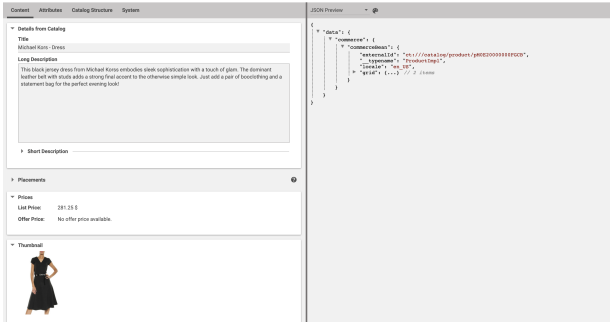
Figure 4.4. Product in tab with JSON preview

> **NOTE**
>
> For Information on how to enable the JSON preview have a look at Section 9.32, "Multiple Previews Configuration" in *Studio Developer Manual*.
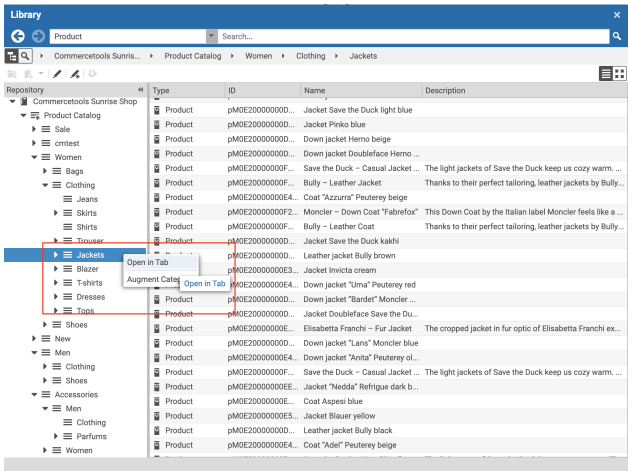
Figure 4.5. Open Category in tab

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.

4.2 Augmenting Commerce Content

CoreMedia Content Cloud enables the user to 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 4.2.1, "Augmenting the Root Nodes" [23].

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.

5. Drop the augmenting content into the right placements of the augmented content item.

4.2.1 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, *Commercetools Sunrise Shop,*). 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 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.

Augmented catalog roots
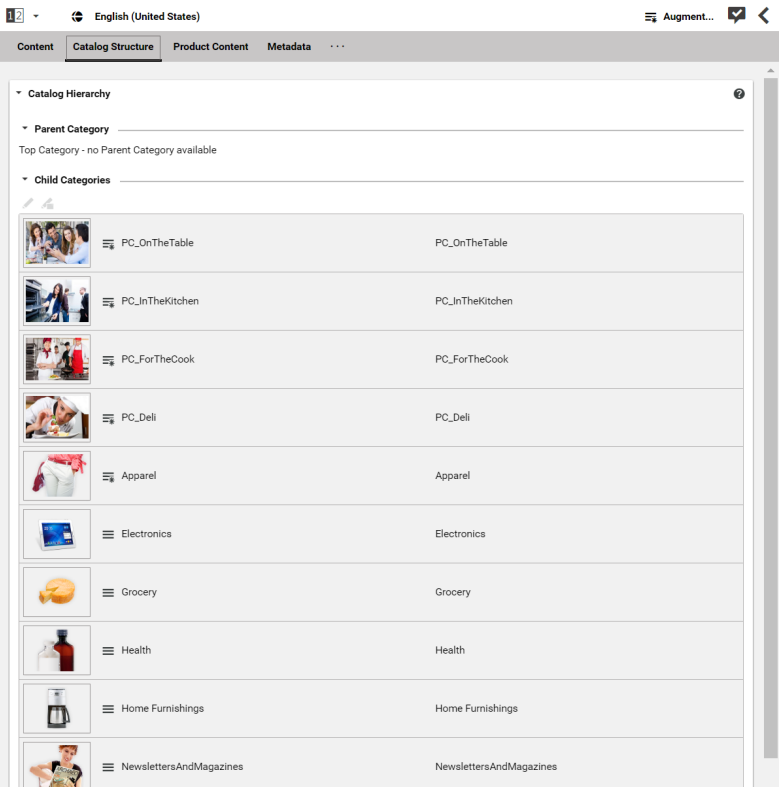
Figure 4.6. Catalog structure in the catalog root content item

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

4.2.2 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.
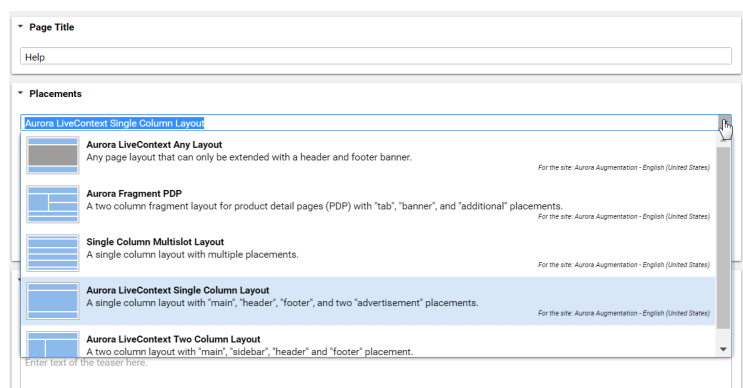
Figure 4.7. Choosing a page layout for a shop page

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.

4.2.3 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

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

Locating the content in the CoreMedia system

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 .

   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 client query .

Figure 4.8, "Decision diagram" [26] 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).
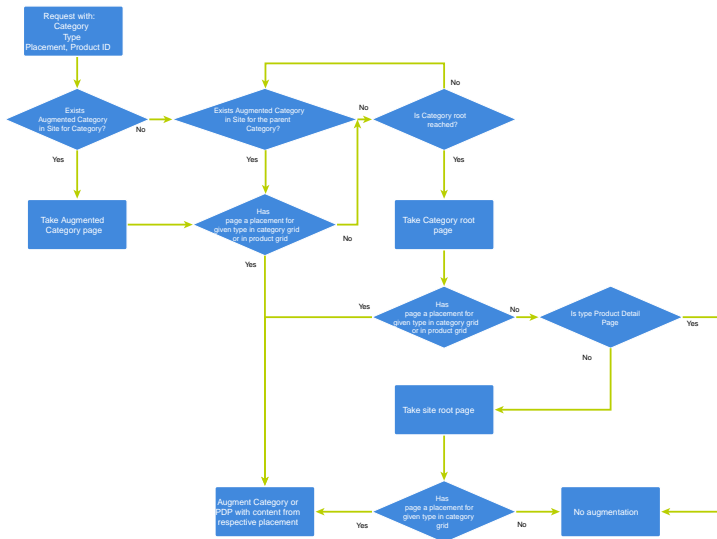
Figure 4.8. Decision diagram

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

- You do not have to create an *Augmented Category* for each category. It's enough to create such a page for a parent category. It is also quite common to create pages only for the top level categories especially when all pages have the same structure.

- You can even use the site root's `Augmented Page` to define a placement that is inherited by all categories of the site.

- If you want to use a completely different layout on a distinct page (a landing page's layout, for example, differs typically from other page's layouts), you should use different placement names for the "Landing Page Layout", for example with a `landing-page` prefix (as part of the technical identifier in the struct of the layout content item). This way, pages below the intermediate landing page, which use the default layout again, can still inherit the elements from pages above the intermediate page

[from the root category, for instance], because the elements are not concealed by the intermediate page.

4.2.4 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 similar to the category overview page.

Product Detail Pages

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

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.`
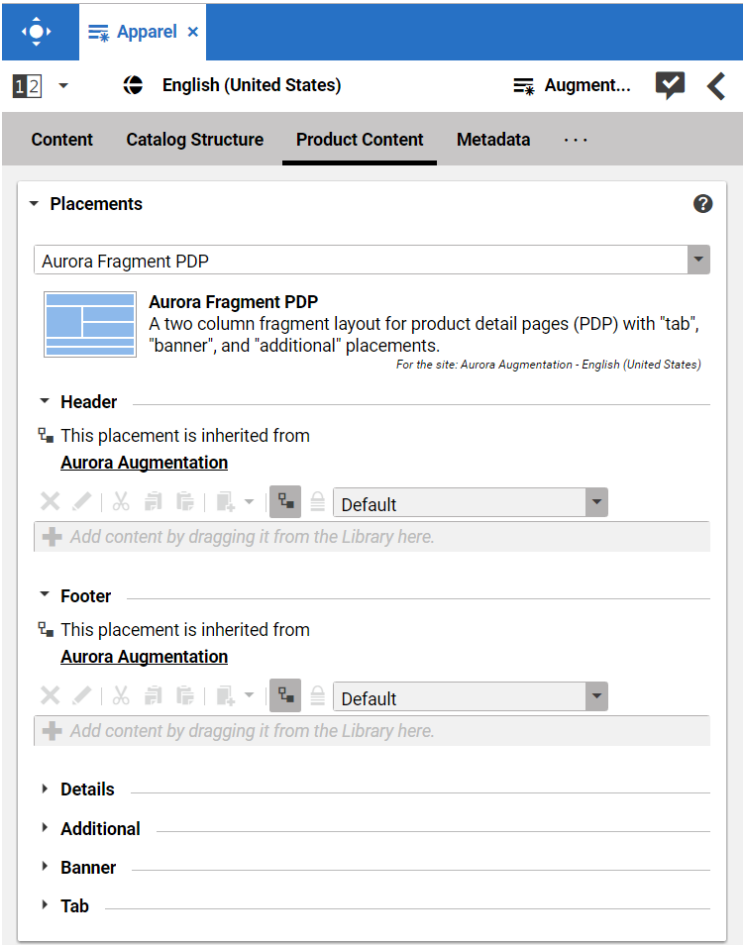
Figure 4.9. Page grid for PDPs in augmented category

Adding CMS Assets to Product Detail Pages

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

Locating the assets in the CoreMedia system

4.2.5 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 by following steps using the common content creation dialog:

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

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

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 4.10. 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.

Difference between the augmentation of catalog and other pages

> **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.

5. 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 5.1. Multiple levels of caching

- Caching is implemented in the Commerce Adapter to accelerate access to commerce entities and to avoid heavy traffic on the *commercetools* 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 `Re moteBeans` 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 per-sistent 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 capa-cities for commerce entities.

Cache Invalidation by Actuator

Commerce entities are cached for a configurable time span. Changes made to commerce items on the *commercetools* 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 *commercetools*. 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 *commercetools*. If the update cycles of the *commercetools* 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 5.2. 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",
  "id": "dresses"
}
```

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

```
{
  "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 *commercetools*. 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 *commercetools*.

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:

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 7, *Commerce Adapter Properties* [40] if you want to adjust the cache configuration for your Commerce Adapter

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

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

Figure 5.3. Actuator URLs in overview page

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

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

Figure 5.4. Actuator results for cache.timeout-seconds.ecommerce properties

6. 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*.

7. Commerce Adapter Properties

commercetools.api.api-url	
Type	java.lang.String
Default	
Description	The base URL, with protocol and port (if needed), to access the commercetools API.

commercetools.api.auth-url	
Type	java.lang.String
Default	
Description	The absolute URL, with protocol and port (if needed), used for authorization at the commercetools system.

commercetools.api.client-id	
Type	java.lang.String
Default	
Description	The unique identifier of the API client.

commercetools.api.client-secret	
Type	java.lang.String
Default	
Description	The confidential client secret.

commercetools.api.fail-on-api-deprecation	

Type	java.lang.Boolean
Default	false
Description	Decorate the SphereClient (responsible for all call to the commercetools system) with a DeprecationExceptionSphereClientDecorator in order to throw an exception on usage of deprecated API calls.

`commercetools.api.project-key`

Type	java.lang.String
Default	
Description	The unique key of the commercetools project.

`commercetools.api.scopes`

Type	java.lang.String
Default	
Description	A comma separated list of scopes, the API client should have access to.

`commercetools.default-locale`

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

`commercetools.product-data-version`

Type	com.coremedia.commerce.adapter.commercetools.config.ProductDataVersion
Default	
Description	The version of the product data. Can be current for published data or staged for preview.

`commercetools.search-enable-language-fallback`

Type	java.lang.Boolean
Default	true
Description	True if language of locale shall be used in search requests.

`commercetools.search-max-result-size`

Type	java.lang.Integer
Default	500
Description	Maximum search result size.

`commercetools.single-value-search-facets`

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

`cache.capacities`

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

`cache.timeout-seconds`

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

`entities.circuit-breaker-names`

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

Default	
Description	Mapping of data lookup keys (cache classes) to circuit breaker names. Mapping to 'none' disables circuit breakers for the mapped data lookup keys.
	Example: Mapping 'product' to 'products' will use a separate circuit breaker named 'products' for product calls. The new circuit breaker can have its own configuration via 'resilience4j.circuitbreaker.configs.products'. Mapping 'product' to 'none' will disable the circuit breaker for product requests.

`entities.default-circuit-breaker-name`

Type	java.lang.String
Default	base
Description	The default breaker name.

`entities.disable-circuit-breakers`

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

`entities.exponential-backoff.factor`

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

`entities.exponential-backoff.initial-delay`

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

`entities.message-store.files`

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

`entities.message-store.root`

Type	org.springframework.core.io.Resource
Default	
Description	Root resource to persistently store messages. If this property is not set, no messages will be persisted. Configure a value to enable persistent caching of messages.

`entities.products.register-parent-dependency`

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

`entities.recompute-on-invalidation`

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

`entities.send-invalidations`

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

`metadata.additional-metadata`

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

`metadata.custom-attributes-format`

Type	com.coremedia.commerce.adapter.base.entities.CustomAttributesFormat
Default	
Description	Format of the custom attribute values.
	The keys are always plain strings.
	Used to identify the deserialization format on the CMS side.

`metadata.custom-entity-param-names`

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

`metadata.replacement-tokens`

Type	java.util.Map<java.lang.String,java.lang.String>
Default	
Description	Map of key value pairs.
	Used as replacement map for example for link building in the generic client on the CMS side.

`metadata.vendor`

Type	java.lang.String
Default	

Description	Name of the vendor.
	Used to identify the connected vendor on the CMS side.

Table 7.1. Commercetools Commerce Adapter related Properties

Glossary

Approve	*CoreMedia CMS* contains a Content Management Environment for content creation and management and a Content Delivery Environment for content delivery. Content has to be published from the Management Environment to the Delivery Environment in order to become visible to customers. Before content can be published, it has to be approved. This way, *CoreMedia CMS* supports the dual control principle.
Blob	Binary Large Object or short blob, a property type for binary objects, such as graphics.
Content Delivery Environment	The *Content Delivery Environment* is the environment in which the content is delivered to the end-user.

It may contain any of the following modules:

- *CoreMedia Master Live Server*
- *CoreMedia Replication Live Server*
- *CoreMedia Content Application Engine*
- *CoreMedia Search Engine*
- *Elastic Social*
- *CoreMedia Adaptive Personalization*

Content item	In *CoreMedia CMS*, content is stored as self-defined content items. Content items are specified by their properties or fields. Typical content properties are, for example, title, author, image and text content.
Content Management Environment	The *Content Management Environment* is the environment for editors. The content is not visible to the end user. It may consist of the following modules:

- *CoreMedia Content Management Server*
- *CoreMedia Workflow Server*
- *CoreMedia Importer*
- *CoreMedia Studio*
- *CoreMedia Search Engine*
- *CoreMedia Adaptive Personalization*
- *CoreMedia Preview CAE*

Content Management Server	Server on which the content is edited. Edited content is published to the Master Live Server.

Content Repository	*CoreMedia CMS* manages content in the Content Repository. Using the Content Server or the UAPI you can access this content. Physically, the content is stored in a relational database.
Content Server	*Content Server* is the umbrella term for all servers that directly access the Core-Media 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 function-ality 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, multi-media emails and of web documents is standardised.
Personalisation	On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.
Projects	With projects you can group content and manage and edit it collaboratively, setting due dates and defining to-dos. Projects are created in the Control Room and managed in project tabs.
Property	In relation to CoreMedia, properties have two different meanings:
	In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content item depends on the content type.
	In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.
Publication	Creates or updates resources on the Live Server.
Resource	A folder or a content item in the CoreMedia system.
Responsive Design	Responsive design is an approach to design a website that provides an optimal viewing experience on different devices, such as PC, tablet, mobile phone.
Root folder	The uppermost folder in the CoreMedia folder hierarchy. Under this folder, CoreMedia users can add further folders and content items.
Site	A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In *CoreMedia CMS* a site especially consists of a site folder, a site indicator and a home page for a site.
	A typical site also has a master site it is derived from.

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

Index

C
catalog, 19
commerce adapter
    configuration, 12
    starting, 15
commercetools shop configuration, 11

E
eCommerce API, 38

L
Library
    catalog view, 19