COREMEDIA CONTENT CLOUD

Personalization Hub Manual

COREMEDIA

CoreMedia GmbH

Altes Klöpperhaus, 5. OG

Rödingsmarkt 9

20459 Hamburg

March 14, 2024 (Release 2310)

List of Figures

List of Tables

List of Examples

1. Preface

CoreMedia Content Cloud supports two different methods of personalization, which provide the basis for creating personalized websites. In both cases, Studio allows you to create personalization content that selects content to be displayed.

- Client-side Personalization
- Adaptive Personalization which runs on server-side

You should start with reading the overview section to understand the basic concepts and scenarios underlying Adaptive Personalization and Client-side Personalization. Then, jump to the section that concerns you the most as they are self-contained and don't need to be read in order.

- In Chapter 2, *Overview* [14] you will get an overview over the aim and features of Adaptive Personalization and Client-side Personalization.
- In Chapter 3, *Adaptive Personalization* [15] you will learn how to configure Adaptive Personalization and how to develop your own extensions.
- In Chapter 4, *Client-side Personalization* [59] you will learn, how to install and configure Client-side Personalization.
- In Chapter 5, *Reference* [95] you will find the supplied context sources, condition types and content types for Adaptive Personalization.

1.1 Audience

This manual is intended for all technical users of *CoreMedia Personalization Hub* that is administrators and developers. Administrators should have read the *CoreMedia Operations Basics Manual* to have basic knowledge of the administration of CoreMedia components. Developers should be familiar with *CAE* development as it is described in the *CoreMedia Content Application Developer Manual* and with the customization of *CoreMedia Studio*. The use of *CoreMedia Personalization Hub* is described in the Chapter 7, *Working with Personalized Content* in *Studio User Manual*.

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 CoreMedia Services

This section describes the CoreMedia services that support you in running a CoreMedia system successfully. You will find all the URLs that guide you to the right places. For most of the services you need a CoreMedia account. See Section 1.3.1, “Registration” [5] for details on how to register.

NOTE

CoreMedia User Orientation for CoreMedia Developers and Partners

Find the latest overview of all CoreMedia services and further references at:

http://documentation.coremedia.com/new-user-orientation

- Section 1.3.1, “Registration” [5] describes how to register for the usage of the services.
- Section 1.3.2, “CoreMedia Releases” [6] describes where to find the download of the software.
- Section 1.3.3, “Documentation” [7] describes the CoreMedia documentation. This includes an overview of the manuals and the URL where to find the documentation.
- Section 1.3.4, “CoreMedia Training” [10] describes CoreMedia training. This includes the training calendar,the curriculum and certification information.
- Section 1.3.5, “CoreMedia Support” [10] describes the CoreMedia support.

1.3.1 Registration

In order to use CoreMedia services you need to register. Please, start your initial registration via the CoreMedia website. Afterwards, contact the CoreMedia Support (see Section 1.3.5, “CoreMedia Support” [10]) by email to request further access depending on your customer, partner or freelancer status so that you can use the CoreMedia services.

1.3.2 CoreMedia Releases

Downloading and Upgrading the Blueprint Workspace

CoreMedia provides its software as a Maven based workspace. You can download the current workspace or older releases via the following URL:

https://releases.coremedia.com/cmcc-11

Refer to our Blueprint Github mirror repository for recommendations to upgrade the workspace either via Git or patch files.

> NOTE
>
> If you encounter a 404 error then you are probably not logged in at GitHub or do not have sufficient permissions yet. See Section 1.3.1, "Registration" [5] for details about the registration process. If the problems persist, try clearing your browser cache and cookies.

Maven artifacts

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

https://repository.coremedia.com

You have to add your CoreMedia credentials to your Maven settings file as described in section Section 3.1, "Prerequisites" in *Blueprint Developer Manual* .

npm packages

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

https://npm.coremedia.io

Your pnpm client first needs to be logged in to be able to utilize the registry (see Section 3.1, "Prerequisites" in *Blueprint Developer Manual*).

License files

You need license files to run the CoreMedia system. Contact the support (see Section 1.3.5, "CoreMedia Support" [10]) to get your licences.

1.3.3 Documentation

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

https://documentation.coremedia.com

The manuals have the following content and use cases:

Manual	Audience	Content
Adaptive Personalization Manual	Developers, architects, administrators	This manual describes the configuration of and development with *Adaptive Personalization*, the CoreMedia module for personalized websites. You will learn how to configure the GUI used in *CoreMedia Studio*, how to use predefined contexts and how to develop your own extensions.
Analytics Connectors Manual	Developers, architects, administrators	This manual describes how you can connect your CoreMedia website with external analytic services, such as Google Analytics.
Blueprint Developer Manual	Developers, architects, administrators	This manual gives an overview over the structure and features of *CoreMedia Content Cloud*. It describes the content type model, the *Studio* extensions, folder and user rights concept and many more details. It also describes administrative tasks for the features. It also describes the concepts and usage of the project workspace in which you develop your CoreMedia extensions. You will find a description of the Maven structure, the virtualization concept, learn how to perform a release and many more.
Connector Manuals	Developers, administrators	This manuals gives an overview over the use cases of the eCommerce integration. It describes the deployment of the Commerce Connector and how to connect it with the CoreMedia and eCommerce system.
Content Application Developer Manual	Developers, architects	This manual describes concepts and development of the *Content Application Engine (CAE)*. You will learn how to write JSP or Freemarker templates that access the other CoreMedia modules and use the sophisticated caching mechanisms of the CAE.

Manual	Audience	Content
Content Server Manual	Developers, architects, administrators	This manual describes the concepts and administration of the main CoreMedia component, the *Content Server*. You will learn about the content type model which lies at the heart of a CoreMedia system, about user and rights management, database configuration, and more.
Deployment Manual	Developers, architects, administrators	This manual describes the concepts and usage of the CoreMedia deployment artifacts. That is the deployment archive and the Docker setup. You will also find an overview of the properties required to configure the deployed system.
Elastic Social Manual	Developers, architects, administrators	This manual describes the concepts and administration of the *Elastic Social* module and how you can integrate it into your websites.
Frontend Developer Manual	Frontend Developers	This manual describes the concepts and usage of the Frontend Workspace. You will learn about the structure of this workspace, the CoreMedia themes and bricks concept, the CoreMedia Freemarker facade API, how to develop your own themes and how to upload your themes to the CoreMedia system.
Headless Server Developer Manual	Frontend Developers, administrators	This manual describes the concepts and usage of the *Headless Server*. You will learn how to deploy the Headless Server and how to use its endpoints for your sites.
Importer Manual	Developers, architects	This manual describes the structure of the internal CoreMedia XML format used for storing data, how you set up an *Importer* application and how you define the transformations that convert your content into CoreMedia content.
Multi-Site Manual	Developers, Multi-Site Administrators, Editors	This manual describes different otions to desgin your site hierarchy with several languages. It also gives guidance to avoid common pitfalls during your work with the multi-site feature.

Manual	Audience	Content
Operations Basics Manual	Developers, administrators	This manual describes some overall concepts such as the communication between the components, how to set up secure connections, how to start application.
Search Manual	Developers, architects, administrators	This manual describes the configuration and customization of the *CoreMedia Search Engine* and the two feeder applications: the *Content Feeder* and the *CAE Feeder*.
Site Manager Developer Manual	Developers, architects, administrators	This manual describes the configuration and customization of *Site Manager*, the Java based stand-alone application for administrative tasks. You will learn how to configure the *Site Manager* with property files and XML files and how to develop your own extensions using the *Site Manager API*. The Site Manager is deprecated for editorial work.
Studio Developer Manual	Developers, architects	This manual describes the concepts and extension of *CoreMedia Studio*. You will learn about the underlying concepts, how to use the development environment and how to customize *Studio* to your needs.
Studio User Manual	Editors	This manual describes the usage of *CoreMedia Studio* for editorial and administrative work. It also describes the usage of the *Adaptive Personalization* and *Elastic Social* GUI that are integrated into *Studio*.
Studio Benutzerhandbuch	Editors	The Studio User Manual but in German.
Supported Environments	Developers, architects, administrators	This document lists the third-party environments with which you can use the CoreMedia system, Java versions or operation systems for example.
Unified API Developer Manual	Developers, architects	This manual describes the concepts and usage of the *CoreMedia Unified API*, which is the recommended API for most applications. This includes access to the content repository, the workflow repository and the user repository.

Manual	Audience	Content
Utilized Open Source Software & 3rd Party Licenses	Developers, architects, administrators	This manual lists the third-party software used by CoreMedia and lists, when required, the licence texts.
Workflow Manual	Developers, architects, administrators	This manual describes the *Workflow Server*. This includes the administration of the server, the development of workflows using the XML language and the development of extensions.

Table 1.3. CoreMedia manuals

If you have comments or questions about CoreMedia's manuals, contact the Documentation department:

Email: documentation@coremedia.com

1.3.4 CoreMedia Training

CoreMedia's training department provides you with the training for your CoreMedia projects either live online, in the CoreMedia training center or at your own location.

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

http://www.coremedia.com/training

Contact the training department at the following email address:

Email: training@coremedia.com

1.3.5 CoreMedia Support

CoreMedia's support is located in Hamburg and accepts your support requests between 9 am and 6 pm MET. If you have subscribed to 24/7 support, you can always reach the support using the phone number provided to you.

To submit a support ticket, track your submitted tickets or receive access to our forums visit the CoreMedia Online Support at:

http://support.coremedia.com/

Do not forget to request further access via email after your initial registration as described in Section 1.3.1, "Registration" [5]. The support email address is:

Email: support@coremedia.com

Create a support request

Support request

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

- Which CoreMedia component(s) did the problem occur with (include the release number)?
- Which database is in use (version, drivers)?
- Which operating system(s) is/are in use?
- Which Java environment is in use?
- Which customizations have been implemented?
- A full description of the problem (as detailed as possible)
- Can the error be reproduced? If yes, give a description please.
- How are the security settings (firewall)?

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

Support checklist

To put it in a nutshell, CoreMedia needs:

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

Log files

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

Which Log File?

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

Where do I Find the Log Files?

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

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

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

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

```
kubectl logs --timestamps <pod>
```

1.4 Changelog

In this chapter you will find a table with all major changes made in this manual.

Section	Version	Description

Table 1.4. Changes

2. Overview

CoreMedia Content Cloud supports two different methods of personalization, which provide the basis for creating personalized websites. In both cases, Studio allows you to create personalization content that selects content to be displayed.

- Adaptive Personalization
- Client-side personalization

Adaptive Personalization uses the CAE to deliver personalized content and uses rules defined in the CoreMedia system.

Client-side personalization decides on the user client software (mostly the browser) which content to show. The rules are defined in a third-party system, such as Monetate. Figure 2.1, " Personalization architecture " [14] gives a high-level architectural overview.

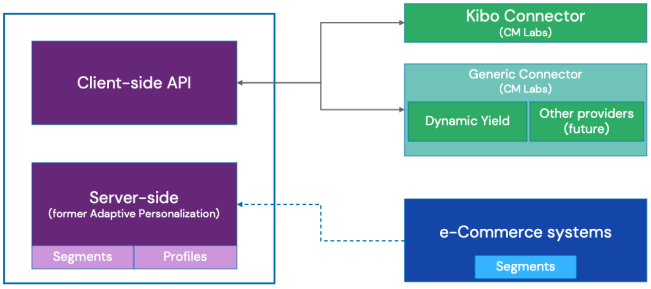

Figure 2.1. Personalization architecture

In this manual, you will find installation, configuration and development information. For the usage of Personalization in Studio refer to Chapter 7, *Working with Personalized Content* in *Studio User Manual*,

3. Adaptive Personalization

*CoreMedia Adaptive Personalization*p rovides the basis for creating a personalized web experience on top of the *CoreMedia Content Application Engine*. It offers user interest profile management as well as dynamic content selection - the building blocks for your personalized site, whether you want to implement explicit (manual) personalization, implicit (automatic) personalization, or both.

Personalization is used by leading web companies to increase user engagement by providing a better user experience. Typical examples of website personalization are:

- Showing more relevant ads by taking a user's browsing behavior into account.
- Recommending products on an eCommerce site based on the user's purchase history.
- Automatically listing the most common answers on a support site for the operating system and browser used by the current user.
- Selecting news stories for a user by analyzing the user's reading history.
- Ranking search results for an individual user based on his personal search history.

The underlying idea of all these examples is to be more relevant to the individual user. With personalized content, you can increase the satisfaction and loyalty of your users, which leads to higher user retention and number of visits.

3.1 Adaptive Personalization Overview

This section gives you an overview of Adaptive Personalization.

3.1.1 Example Scenario

CoreMedia Adaptive Personalization provides everything you need to implement rule-based personalization for your website out of the box. But what does this mean, exactly? Here's a simple example:

Assume you're the editor of a news site and there's a single main teaser region on your entry page. You know that placing a relevant teaser in this region is critical as it drives a high percentage of clicks, and more clicks mean more revenue. By inspecting the reports of your analytics system, you've noticed that in the morning, most visitors read *World News*, while during lunch break and in the evenings, interests are more diverse. In particular, you see some visitors focusing on *Lifestyle*, others on *Economy*, and still others on *Sports*. You decide to optimize your entry page by placing a *Personalized Content* in the main teaser region. This content item is configured to show the most important (as defined by the editorial team) article in *World News* each morning until 10am. At 10am, it switches to another *Personalized Content* that selects the most important teaser from *Lifestyle*, *Economy*, or *Sports* depending on the interests of the current user. If *CoreMedia Adaptive Personalization* is installed in your site, you can do all of this without the need to do any programming or to redeploy the system.

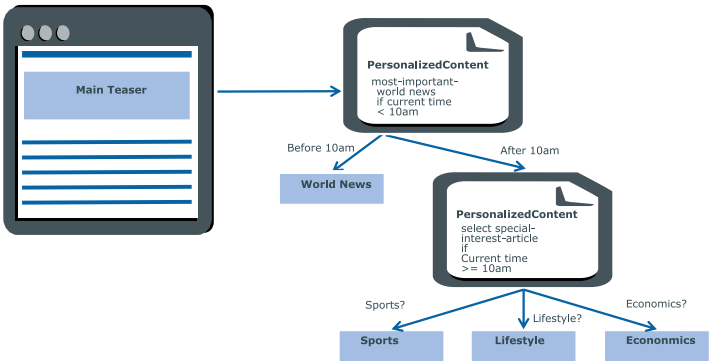

Figure 3.1. Example Page with Main Teaser

Have a quick look at the components of *CoreMedia Adaptive Personalization* that would be used to implement a system that supports this scenario. Detailed descriptions of the components can be found in the corresponding chapters within this manual.

All context data for a request, containing the current time of day, is stored within the *CAE* in a `ContextCollection`. If you tag your pages with keywords, as is typically the case if you use an ad server, you can use a `ScoringContext` in combination with the `KeywordInterceptor` to track the most often seen keywords for each user (if he read a lot of articles tagged with 'Sports', 'Sports' will have a large score in the context).

Personalized Content contains a list of rules of the form *select content X if the contexts satisfy conditions Y*. Given a `ContextCollection`, it renders the first content for which the conditions are satisfied. So in the scenario above, the main teaser would contain the rules select most-important-world-news if current time < 10am and select special-interest-article if current time >= 10am. The content most-important-world-news could use a search to determine the most current, highly rated editorial article from *World News*, while special-interest-article would be another *Personalized Content* selecting articles based on the users' keyword scores, for example select most-important-sports-news if score of 'Sports' > 0.8. These *Selection Rules* are defined from within *CoreMedia Studio* using a specialized editor component and are deployed to the *CAE*s via content item publication, so there's no need for any code changes.

3.1.2 Architectural Overview

CoreMedia Adaptive Personalization is a collection of building blocks intended to assist you in leveraging the versatility of the *CAE* to implement dynamic and personalized content delivery. The basic idea is that each request to the site by a visitor is associated with context data and that this data is used to determine what is to be delivered to the visitor.

Contexts represent arbitrary things about the user and his environment, such as the user's current interests, the location from which the user accesses the site, and the device used. A context can also contain general information such as the current date and time or the day of the week.

To determine the content to be delivered, *CoreMedia Adaptive Personalization* provides the implementation of a rule-based approach ("select some specific content if the context data fulfills some requirements") as well as an extension to the search engine integration that allows using context data within search queries.

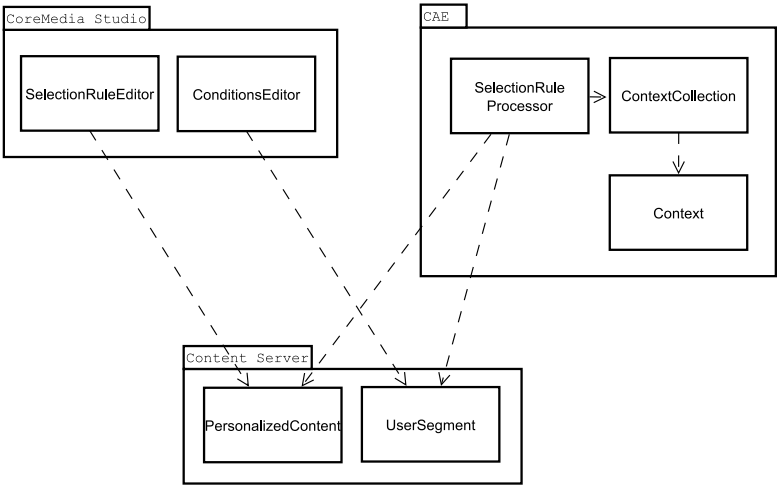

Figure 3.2. Architectural overview

To configure rule-based personalization and define customer segments, *CoreMedia Adaptive Personalization* includes *CoreMedia Studio* components that provide corresponding editing functionality for editors. All configurations are stored within content item properties which are freely configurable - you are not required to use the predefined content types.

Within the CAE, you evaluate rules in a Content Bean implementation using the `SelectionRuleProcessor`. The processor expects to be supplied with a collection of user contexts, which may include all customer segments for which the defined conditions are satisfied by the user.

Not shown in the diagram above is the search engine extension. It provides a query preprocessor that allows you to add macro calls within query strings and evaluate these macros at time of search. For example, if you define a macro `userSegments` that looks up and returns the set of segments the user is a member of and tag your content with segment names which are indexed in field `segments`, you can search for all content items tagged with the segments of the user via the query `segments:userSegments()`.

3.1.3 Building Blocks

CoreMedia Adaptive Personalization provides the basis for creating a personalized web experience on top of the *CoreMedia Content Application Engine*. This section lists all building blocks with a short description and is intended as an overview for programmers

and technical consultants. See Section 3.3, "Developing With Adaptive Personalization" [31] for a more detailed discussion.

CoreMedia Adaptive Personalization comes with an API that offers five main building blocks:

- Personalized Content
- Customer Segments
- User Contexts
- Test User Contexts
- Behavior tracking

Personalized Content

Personalized Content is content that uses a list of rules to determine what to show to a visitor. This is similar to content that stores a search query and displays the results of executing the search, but covers different use cases as you've got finer control over the selection process. For example, you can define rules that show different content items.

- ... to users above a certain age AND at specific times of the day.
- ... to users who previously bought a specific service or product.
- ... to users visiting the website with a specific device.
- ... to users who previously showed interest in content tagged by repeated keywords (such as soccer, baseball, travel, politics etc.) through keyword tracking.

Selection rules are stored in a Markup document property using a specific XML grammar. Rules are parsed and evaluated within the *CAE* and all content selected by rules whose conditions are satisfied is returned.

Adaptive Personalization provides a *CoreMedia Studio* plug-in with an easy to use interface for users to define personalization criteria. *Adaptive Personalization* comes with a number of predefined condition types that can be bound to arbitrary context parameters, thus allowing you to adapt the UI to the semantics of your application domain. These condition types are described in detail regarding usage and configuration in Section 5.1, "Condition Types" [96].

Customer Segments

With personalization you can group the visitors of your website into segments according to a set of logical conditions. For example, if a user context (explained below) is provided, you could create visitor segments such as

- Male users aged between 30 and 40 AND with a yearly available income of US$45,000.
- Users with an interest in the fashion topic AND with at least five social connections within the site's user base.

- Users which have bought a certain number of products through the website correlating with an interest in a specific content topic.

Customer segments are evaluated by a specialized ContextSource (a component that adds a context to the ContextCollection associated with a request to the CAE) and added to the current user's context data. Thus, they can be used within conditions in selection rules.

The *CoreMedia Studio* plug-in provides an interface to define segments with conditional expressions. As with Personalized Content, the UI can be adapted to your application's needs.

User Contexts

CoreMedia Adaptive Personalization allows you to use arbitrary user contexts as sources of information accessible in conditional criteria of Personalized Content and Customer Segments. A context can be an arbitrary Java object, but usually is a map-like entity that stores key-value pairs. A user's request is associated with an arbitrary number of contexts collected in a `ContextCollection`, which typically is injected in all CAE beans that require access to the context data.

User contexts are populated in the *CAE*, in the `preHandle` phase of request processing. Thus, context data is available to handlers as well as content beans. The context API also allows you to persist information into user contexts at the end of request processing.

Test User Contexts

Test User Contexts are CMS content items containing lists of context properties. A specialized `ContextSource` reads these content items and adds corresponding context objects to the context collection of each request to the CAE. Using *Test User Contexts*, you can simulate a user having specific context properties and thus test the behavior of your personalized site.

Behavior Tracking

CoreMedia Adaptive Personalization provides a specialized Context class that is intended for tracking and scoring the behavior of individual users on your site. This ScoringContext can be informed about (weighted) events, such as visits to keyword-tagged pages or initiated downloads. The collected weights for an event are combined and the event name as well as its weight are made available as context properties to be used in customer segments or selection rules.

3.1.4 Data Privacy Considerations

CoreMedia delivers building blocks as part of the *CoreMedia Adaptive Personalization* add-on module and the respective Blueprint Extensions that enable you to build personalized experiences. CoreMedia provides tooling to facilitate compliance with legal privacy regulations including requests for information, change and deletion of personal data - however establishing compliance remains the responsibility of the customer implementing and operating the product. Depending on whether or where technically you choose to persist personal data of your end users, you may need to seek and document consent from your users and/or establish other legal grounds for use of personal data based on your applicable legal regulations. Any recommendations provided by CoreMedia are not to be established as legal advice or consultation, please contact your legal counsel.

3.2 Adaptive Personalization Configuration and Operation

This section describes how you configure personalization in CoreMedia Adaptive Personalization.

3.2.1 Defining Property Editors

This chapter describes how you configure *CoreMedia Adaptive Personalization* features in the underlying platforms.

- Section 3.2.1, "Defining Property Editors" [22] describes how you can integrate the delivered property editors into *CoreMedia Studio* content forms.
- Section 3.2.2, "Configuring Caching For Rules and Condition Evaluation" [25] describes how to cache rules and conditions.

CoreMedia Adaptive Personalization includes two property editors for editing personalization specific content properties in *CoreMedia Studio*:

- `SelectionRulesField` is an editor to be used to define content selection rules
- `ConditionsField` is an editor to be used to define customer segment conditions

`SelectionRulesField` and `ConditionsField` can be used for a content property of type XML using schema `coremedia-selectionrules-1.0`. This schema is defined in `cap-personalization-schema-bundle.jar` and can be imported into a content type declaration file by adding the following code near the top of the file:

```
<XmlSchema Name="coremedia-selectionrules-1.0"
   SchemaLocation="classpath:xml/coremedia-selectionrules-1.0.xsd"
   Language="http://www.w3.org/2001/XMLSchema"/>
```

You configure a property editor for a specific content property as explained in the *CoreMedia Studio Manual*.

The *CoreMedia Blueprint* development workspace provides a Studio form using these condition fields to edit personalized content items.

Setting up the Property Editors

CoreMedia Adaptive Personalization offers different types of conditions that are listed in Section 5.1, "Condition Types" [96]. Therefore, you can adapt the property editors for selection rules and segment conditions to the types of properties your application is using. For example, if your context contains a property `dateOfBirth` that holds the current visitor's date of birth, the property editors should use a `DateCondition` instead of a `StringCondition` for conditions using the property.

You configure the editors in the `ext-xml` files defining the property editors for your content types.

SelectionRulesField

`SelectionRulesField` supports the attributes `propertyName` and `allowedContentType`.

- `propertyName` is required and denotes the name of the content property to be associated with the field. This attribute is common to all property editors in *CoreMedia Studio*.
- `allowedContentType` is optional and denotes the name of the type of content that can be selected via rules defined using this property editor.

For example, if `allowedContentType="CMTeasable"` is used, only content items of type `CMTeasable` or of any subtype can be added to the rules created via this editor. Thus, you won't be able to create a rule that selects a `CMChannel`.

The child element `conditionItems` defines the condition types the `SelectionRulesField` will support. The following table lists the allowed attributes in conditions.

Property Name	Description
`conditionName`	The text the user sees in the combo box used to select the type of a condition. It is not further processed by the rule editor and thus can be an arbitrary string. *Required*
`propertyPrefix`	The prefix denotes the context name of the property and does not include the separating '.'. For example, to denote all properties in the 'foo' context, such as 'foo.bar' and 'foo.zork', supply 'foo' as the `propertyPrefix` value. `ConditionTypes` support either `propertyPrefix` or `propertyName`, but not both.
`propertyName`	The name of the property the condition is associated with. The rule editor compares the name of the property used in a condition with this string to identify the UI element it should use for rendering the condition. `ConditionTypes` support either `propertyPrefix` or `propertyName`, but not both.

Property Name	Description
`isDefault`	If set to "true", the condition type is used as the selected condition type if a new condition is added to a rule via the UI. Make sure that there's only a single default item because otherwise you cannot be sure which one will be selected. Default is 'false'.

Table 3.1. All properties

Example with *`propertyName`* attribute:

```
<perso:dateCondition conditionName="Date of Birth"
  propertyName="personal.dateofbirth"/>
```

This element makes the `SelectionRulesField` use a `DateCondition` if a condition is defined on the `personal.dateofbirth` property.

Example with `propertyPrefix` attribute:

```
<perso:keywordCondition conditionName='Explicit Interest'
  propertyPrefix='explicit' isDefault='true'/>
```

This element makes the `SelectionRulesField` use a `KeywordCondition` for all properties starting with the prefix "explicit" followed by ".", for example, "explicit.science".

The order of elements in `conditionItems` is relevant for item selection. The `SelectionRulesField` searches the list top to bottom to find the Condition for a given property name. It uses the first item whose `propertyName` or `propertyPrefix` matches.

ConditionsField

The `ConditionsField` property editor is similar to the `SelectionRulesEditor` in that it allows you to define a list of customer segment conditions using the same components and configuration, except for the `SegmentCondition`.

Using the AddConditionItemsPlugin to add conditions to the property editors

The `SelectionRuleField` as well as the `ConditionsField` support the `AddConditionItemsPlugin` to allow the configuration of condition items via plugin rules. Plugin rules are a mechanism provided by *CoreMedia Studio* to allow *Studio* plugins to modify common UI components.

For example, you might want to keep the configuration of condition items specific to your CRM system in the same project as your CAE/CRM integration. To this end, create a *CoreMedia Studio* plugin containing plugin rules that configure the condition items

using the `AddConditionItemsPlugin` and introduce it as a Maven dependency to your *CoreMedia Studio* web application (for details, see the *CoreMedia Studio Developer Manual*).

Module `p13n-studio` of the *CoreMedia Blueprint* development workspace shows how to configure selection rules based on Elastic Social contexts.

3.2.2 Configuring Caching For Rules and Condition Evaluation

Selection rules as well as segment conditions are stored in textual form in content properties. To be evaluated in the *CAE*, they have to be parsed and transformed into an executable form. This transformation is expensive and thus should only be performed if necessary, that is, if the corresponding content properties were modified. Therefore, you should use CoreMedia data views and the CoreMedia cache for caching.

`SelectionRulesProcessor` as well as `ConditionsProcessor` can be cached. In your content beans, use a property getter that returns the appropriate processor for your content item and create a data view with association type 'static' for this getter. In the methods that use the processor, access it via the getter. This guarantees that parsing is only done if necessary.

If you use the `SegmentSource`, you do not need to care about caching segment conditions, as this is done by the source itself. You'll find an example data view declaration for the type `CMSelectionRules` in the *CoreMedia Content Cloud* p13n extension. For further information on how data views work, refer to the *Content Applications Developer Manual*.

3.2.3 Configuring The Customer Persona Form

You can change the used context properties and/or the appearance of the context property editors of the *Customer Persona Form* by reconfiguring the `CMUserProfile` content type.

If you add context properties to the content you do not need to adapt the content type definition for the *Content Server* because all context properties are stored in one, already defined plain text blob property.

Underneath a `PersonaGroupContainer` there are special property fields which are responsible for handling the forwarded property. You can write your own property fields for custom properties.

There are already the most common property fields available:

- `PersonaNumberPropertyField` - accepts just digits, '-' and '.'
- `PersonaStringPropertyField` - accepts all kind of characters
- `PersonaTimePropertyField` - accepts time in the specified time format; you can choose time from the combo box as well
- `PersonaDatePropertyField` - accepts a date in the specified date format; you can pick the date from the date picker as well
- `PersonaDateTimeProperty` - combined time and date property fields. You need to fill both values.

To write your own property fields have a look at Section 3.3.2.4, "Working With Test Contexts" [39].

3.2.3.1 Configure the displayed User Segments

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. For this reason there is the possibility to provide a predefined list with user segments to display. This list can be configured in the site-specific *LiveContext* settings content item. To define the list, open the *LiveContext* settings content item and create a new *String List* named `configuredSegmentIds` below the *commerce* node. Fill this list with the IDs of the desired user segments. The IDs have the following structure *`[vendor]`*:///catalog/segment/*`[externalId]`*

3.2.4 Configuring The PersonaSelector

The `PersonaSelector` is a component of *CoreMedia Adaptive Personalization* that is shown in the Preview Toolbar of *CoreMedia Studio*. As depicted in Figure 3.3, "The PersonaSelector in CoreMedia Studio" [27], you can unfold it by pressing the corresponding button in the Preview's Toolbar (1.). It contains *Customer Personas* that represent typical visitors of your website. When selecting a Customer Persona its artificial context properties are read from the CMS and the Preview is rendered accordingly. For example, a *Customer Persona* could explicitly simulate a specific date to test a Personalized Content displaying special offers on Christmas Eve.

In addition to simply selecting a *Customer Persona*, the `PersonaSelector` allows you the following:

- navigate to the location of the *Customer Personas*' backing content item in the Content Management Server (2.) and

- open the *Customer Persona Info Window* with detailed information about the context properties of a specific *Customer Persona* (3.).

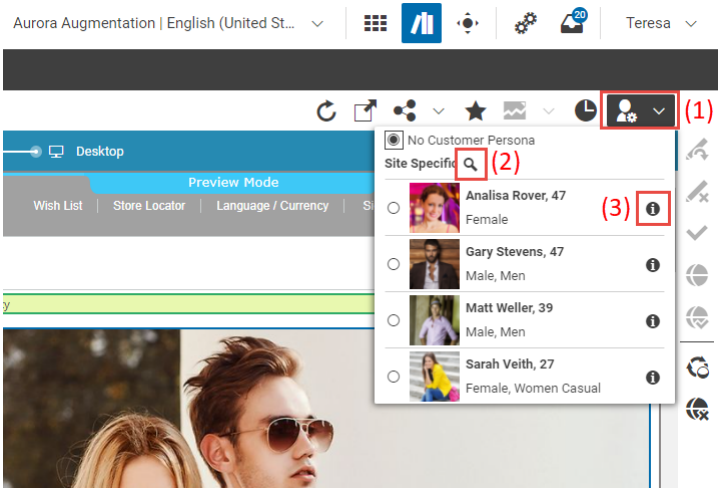

Figure 3.3. The PersonaSelector in CoreMedia Studio

The initial view of the *Customer Persona Info Window* displays the basic context properties as shown in Figure 3.4, “The Customer Persona Info Window in CoreMedia Studio” [28]. You can display a grouped list of all contained properties by switching to the "Details" tab (1.). To permanently modify a context property press the "Edit" button (2.), which opens the Customer Persona's backing CMS content item in a new content tab. You can also activate a *Customer Persona* from the *Customer Persona Info Window* by clicking the "Activate Customer Persona" button (3.). If you want to know how to customize localized context properties of the *Customer Persona Info Window*, have a look at Section 3.2.5, “Localizing the Customer Persona Info Window” [29].

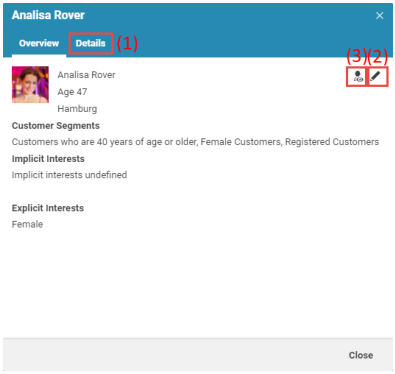

Figure 3.4. The Customer Persona Info Window in CoreMedia Studio

By default, the `PersonaSelector` offers a list of all *Customer Personas* - which are contents of type `CMUserProfile` - that are located in the `/System/personalization/profiles` folder (which is different in *CoreMedia Blueprint*, see further below). Furthermore, it offers a method that can be used to adapt the paths from which *Customer Personas* are retrieved:

- `public function addPath(repositoryPath:String, groupHeaderLabel:String)`
- `public function clearPaths()`

The `groupHeaderLabel` argument of the `addPath` method defines a label that is used to group the *Customer Personas* within the `PersonaSelector` that are retrieved from the same path.

Example

If you do not want to retrieve *Customer Personas* from the default path, but from the paths `/context` and `/experimental` where all *Customer Personas* from the latter location should be suffixed with "experimental" you would do the following in a plugin:

```
...
public function init(component:Component):void {
  const selector:PersonaSelector = component as PersonaSelector;
  if (!selector) {
    throw Error("plugin is only applicable to components of
          type PersonaSelector");
  }
  selector.clearPaths();
  selector.addPath('/contexts');
  selector.addPath('/experimental', 'experimental');
```

```
}
...
```

CoreMedia Adaptive Personalization contains ready-made plugins for use with the `PersonaSelector`:

ptype	Description
`disablefortypes`	Disables the selector if one among a set of preconfigured content types is being previewed.
`addpath`	Adds a path to the list of path used by the selector.
`addsitespecificpath`	Adds a site specific path containing a placeholder to the selector.

Table 3.2. Plugins for PersonaSelector

You add plugins to a component via the plugin rules of your project module (see the "Understanding Studio Plugins" section in the *CoreMedia Studio Developer Manual* for details). *CoreMedia Blueprint* provides a ready to use example of the `PersonaSelector` with the side independent default path `/Settings/Options/Personalization/Profiles` and the site specific default path `Options/Personalization/Profiles`.

3.2.5 Localizing the Customer Persona Info Window

The data shown in the *Customer Persona Info Window* can be localized, so that the right language version is shown in *CoreMedia Studio*. You can localize the following items:

- context names
- property keys
- property values

The *Customer Persona Info Window* searches for the localized form of an element by looking for global resource bundle properties of the form (where `name` is the name of a context, `key` is a property key and `value` is a property value):

- `p13n_context_<name>` for the name of a context
- `p13n_context_<name>_<key>` for the name of a property key within a context

- `p13n_context_<name>_<key>_<value>` for a property value within a context

Any non-word characters (everything except alphanumeric characters and '_') are removed before the look-up key is constructed, that is, the localization property for the context "a sample context" would be `p13n_context_asamplecontext`.

Property values representing time stamps are not looked up in a localization file, but automatically transformed into a date representations matching the selected locale.

If the *Customer Persona Info Window* cannot find a matching localization property, the original value is used. Refer to the CoreMedia Studio Developer manual on how to set up resource bundles in CoreMedia Studio.

3.2.6 Monitoring Components With JMX

Key components of *CoreMedia Adaptive Personalization* expose management functionality via the following JMX MBeans:

- `ContextCollectorManager`
- `SelectionRuleProcessorManager`

You can find a detailed list of all available JMX properties in the corresponding API documentation of the classes.

ContextCollectorManager

This class provides statistics about the performance of the `ContextCollector` and each registered `ContextSource`. By default, only performance tracking of the `ContextCollector` is enabled. If you want to enable tracking of the sources, use the `perSourcePerformanceEnabled` flag in your JMX console.

You can use the `ContextCollectorManager` bean to activate and deactivate the `ContextCollector`. This might be useful if you have an unexpected spike in high traffic and you want to disable *Adaptive Personalization*. Use the `ContextCollectorEnable` flag for this task.

SelectionRuleProcessorManager

This class provides statistics about the performance of all `SelectionRuleProcessor` instances used in a *CAE*.

3.3 Developing With Adaptive Personalization

CoreMedia Adaptive Personalization is a set of building blocks by nature. As such, there is a lot of room for customizations and custom implementation. Each of the following sections explains how to use and combine the available building blocks and features.

3.3.1 Architectural Overview

CoreMedia Adaptive Personalization is a collection of building blocks intended to assist you in leveraging the versatility of the `CAE` to implement dynamic and personalized content delivery. The basic idea is that each request to the site by a visitor is associated with context data and that this data is used to determine what is to be delivered to the visitor. Contexts might represent arbitrary things about the user and his environment, such as the user's current interests, the location from which the user accesses the site, and the device used.

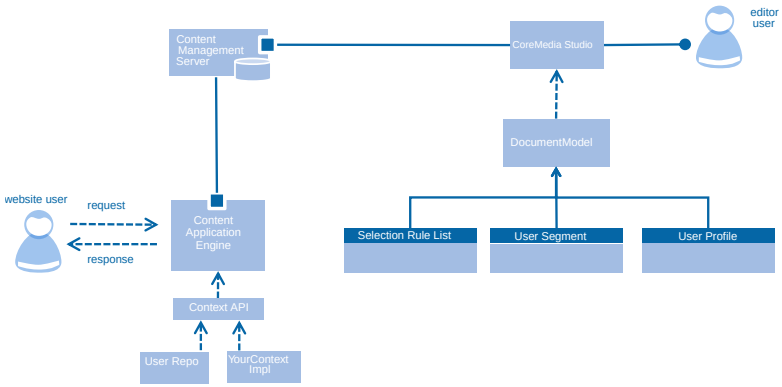

Figure 3.5. Adaptive Personalization overview

CoreMedia Adaptive Personalization runs partly within the CAE delivery component to evaluate the selection and choice of content based on your settings. *CoreMedia Adaptive Personalization* also depends on content types in the CoreMedia content repository to persist certain settings and personalization rules, representing the personalized content you want to place on your site. These content types can be edited conveniently through CoreMedia's web based editor by using the *Adaptive Personalization Editor Plugin*. Using the *CoreMedia Adaptive Personalization* content types in your publication workflow, you

can place personalized content just like you would place any other content, using the same editing metaphors and workflows as with any other CoreMedia content.

Both components are integrated into *CoreMedia Blueprint* by default. *CoreMedia Blueprint* already has suitable content types in place. When using a custom content type model, it will be necessary to model suitable content types for *Adaptive Personalization* and configure their usage according to documentation.

Dedicated personalization content items in the content repository are used to manage personalization of a site editorially. The type *Personalized Content* represents personalized content by storing a Markup property with a set of selection rules used to decide what content to render when a request is processed in the *CAE*. The type Customer Segment allows you to define segments of website users based on conditional rules. Using the same selection rule logic as the type *Personalized Content*, this type stores the rules as a String property. *Customer Segments* can then in term be used within a matching condition type in *Personalized Content* content items. The type *Test User Context* can be used by editors within *CoreMedia Studio* to switch user contexts within the preview pane to test and preview the effects of personalization settings before publishing any content items to a live website. These content items are edited, placed and published from within *CoreMedia Studio* like any other content item - except the test user contexts which have no effect or use when published. During delivery of those content items, *CoreMedia Adaptive Personalization* components running within the *CAE* will interpret and evaluate the contents of those content items in order to render matching, personalized content based on the user's request and the user context.

The *CAE* has access to a pool of context sources addressed through the Context API, which is also described in detail in this manual. Out of the box, *CoreMedia Adaptive Personalization* supports storing user context information in cookies. For each request, the *CAE* can determine the specific context using the contexts available through the Context API implement context sources. The information stored in those contexts can be used to define selection rules in *Personalized Content* and *User Segment* content items.

The evaluation of dynamic, request specific selection rules per request is costly in terms of computation. Because of this, *CoreMedia Adaptive Personalization* facilitates the caching features already in place in the *CAE* and computes a cacheable, precomputed representation of a set of selection rules, using both *CAE* data views and cache keys where appropriate. This minimizes the impact of personalization on *CAE* performance.

Adaptive Personalization in the CAE

Within the CAE a high level point of view request processing looks like in figure.

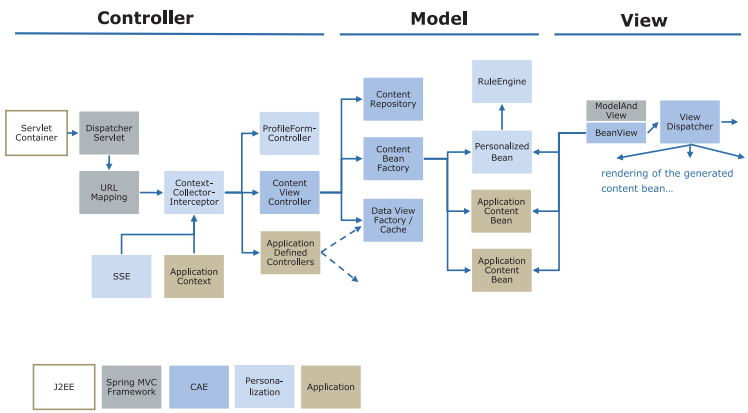

Figure 3.6. Request processing in the CAE

CoreMedia Adaptive Personalization integrates into the *CAE* using the standard Spring facilities and API. Within the *CoreMedia Blueprint* development workspace, *Adaptive Personalization* is already be integrated in the *CAE* setup. Refer to the installation documentation for details about how to manipulate the Spring configuration of *CoreMedia Adaptive Personalization*.

Within the *CAE*, *Adaptive Personalization* performs two basic functions:

- collecting information from all available contexts for the current request
- evaluating content selection rules as they are used within Personalized Content and Customer Segments

Context information must be collected before processing a request and can be persisted after having processed the request. This can be achieved through Spring Web MVC interceptors or servlet filters. Evaluation of content selection rules may be performed while processing a request, for example, using content bean logic.

How contexts, properties, conditions and personas work together

In *CoreMedia Adaptive Personalization* the information about a website users context is stored in a so called `ContextCollection` that can be best thought of as a request scope map holding the request's context objects. All context sources that are

configured via the Spring application context are called to retrieve and store their context information for the given Request into the request - and therefore usually user-specific - *ContextCollection*. A common scenario is to instantiate a `ContextCollection` when a request hits the *CoreMedia Content Application Engine* (CAE) with enabled *CoreMedia Adaptive Personalization*. Alternatively, a `ContextCollection` can be implemented using thread local storage, so that it is effectively a singleton bean (as the `DefaultContextCollection`).

A context is identified by a name ("keywords", "personal" or "system", for example) and can store arbitrary data. Usually (at least the default contexts that are shipped with the product) the context sources implement the `PropertyProvider` interface which requires that a context stores Map-like information in key/value pairs. Therefore, the properties of a given context are identified by the context name and property names with corresponding values, for example a numeric value, a string value, a date value.

Example

```
<contextname>.<propertyname>=<value>
```

The `<contextname>.<propertyname>` pattern is also used in personalization selection rules to identify the context information that will be used in a rule.

Examples

```
select <content> if <contextname>.<propertyname> \
  <operator> <value>
select content:1234 if keyword.sports > 0.5
```

In the *Selection Rules* editor, which is part of the *CoreMedia Studio* plug-in, you can use different UI components to define different conditions in personalization rules. Which UI component is used, can be configured by a manually mapping from context property names to component types. This is, for example, done in `CMSelectionRulesForm.mxml` and `CMSegmentForm.mxml` of the CoreMedia Blueprint development workspace.

When the *CAE* evaluates a personalization rule for a given request, the `SelectionRulesProcessor` uses the already known `<contextname>.<propertyname>` pattern to check whether the values in the current `ContextCollection` match the rules or not. For more details on the selection rule execution please refer to Section 3.3.3, "Working With Selection Rule Lists" [43].

Due to the map-style nature of the context data, it is very easy to create test data for editorial usage. That is exactly how the persona contexts work in the personalization UI (the `PersonaSelector`).

Instead of actually instantiating a `ContextSource` with an identifier "keyword" and the property "sports" and value 70% you can simply write "keyword.sports=0.7" into the persona context. This information is then used in the *CAE* as context information and the real "keyword" `ContextSource` is ignored.

When the *CAE* evaluates a personalization rule, an executable representation of the rule string is created or retrieved from the cache and supplied with the active user's `ContextCollection`. This representation uses the `<contextname>.<propertyname>` pattern encoded in the individual conditions to retrieve the corresponding property values from the `ContextCollection` and applies the specified comparison operator from the personalization rule.

3.3.2 Working With the User's Context

Personalizing the user's experience relies on data about the user. Within the system, this data is represented as so called context objects (simple POJOs) stored in a `ContextCollection`. The `ContextCollection` is made available to all components requiring access to context objects.

In a personalized web application, the `ContextCollection` is filled with all objects relevant for processing the request prior to actually processing the request. Relevant context data may be located in disparate sources (for example, internal CRM systems and external social community sites), thus a simple way to collect and combine this data is required. This is the responsibility of the `ContextCollector`. The `ContextCollector` can be invoked by either a Spring Web MVC handler interceptor that is installed in all handler chains requiring context data, or a servlet filter. For this purpose, the implementations `PersonalizationHandlerInterceptor` and `PersonalizationServletFilter` are provided. In the following, it is assumed that the `ContextCollector` is set up as a handler interceptor, if not stated otherwise.

The request flow

The sequence diagram below shows an example of how context objects are retrieved and provided for further manipulation and decision making. In general, for every request all context objects for the active user are loaded. These objects can be used, for example, to select content to be rendered or keep track of the pages the user visits. After request processing is finished, changed context objects are written back to their source. The `ContextCollection` is then cleared.

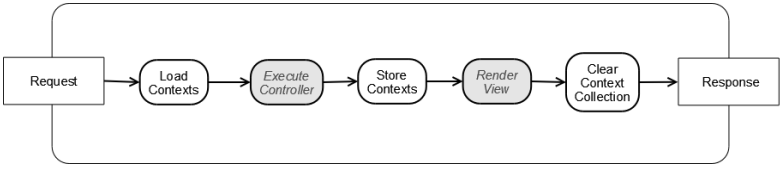

Figure 3.7. ContextObject usage

Loading Contexts

For each request, the `ContextCollector` asks each of its `ContextSources` to load its context objects and place them into the `ContextCollection` of the active user.

Each `ContextSource` retrieves part of the user's context objects. For example, the `CookieSource` checks if a specific cookie is available in the current request. If it is, the value of the cookie is decoded into a context object and put into the `ContextCollection` collection. If not, a new and empty context is created.

Using Contexts

Contexts objects can be read and modified throughout request processing. For example, the contexts can be used to determine which content to show to the user or to capture user behavior (see the Section 3.3.5, "Working With Scoring" [49]).

Storing Contexts

After request processing, each `ContextSource` gets the chance to persist the contexts objects it is responsible for.

Supplied ContextSources

CoreMedia Adaptive Personalization comes with a set of `ContextSources` ready to be used in your project. See Section 5.3, "Supplied Context Sources" [99] for a table of all delivered sources.

A `ContextSource` typically requires a context name and a `ContextFactory` or `ContextCoDec` instance to be appropriately configured. The name is used as the key under which the context object is stored in the `ContextCollection`. Make sure these names are unique to prevent replacing context objects added by other sources. The `ContextFactory` or `ContextCoDec` is used by the source to create new context instances and serialize as well as deserialize a context.

3.3.2.1 Configuring the Context Collector

The `ContextCollector` is responsible for collecting context data from `ContextSources`. It can be invoked through a Spring MVC interceptor or a servlet filter both of which must be installed in all handler chains that require user context data.

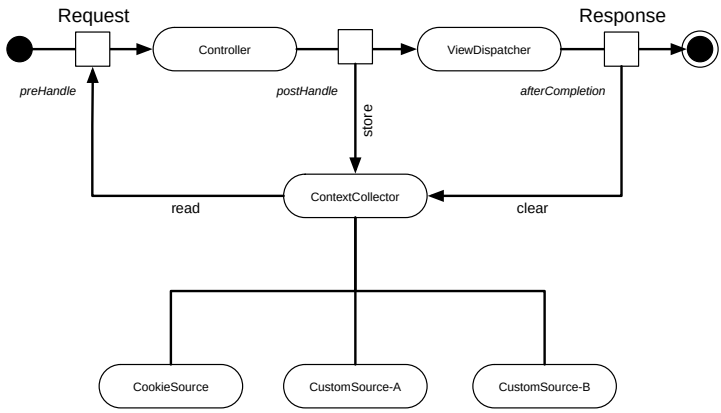

Figure 3.8. ContextCollector position

The `ContextCollector` manages a list of `ContextSources` to fill before processing the request. Sources are processed in the order implied by the respective list and the request and session lifecycle are mapped as follows to the `ContextSource` methods: `preHandle` and `postHandle` can be invoked by a servlet filter or by the corresponding lifecycle methods of a Spring `HandlerInterceptor`, `preSession` and `postSession` relate to `sessionCreated` and `sessionDestroyed` of an `HttpSessionListener`.

In addition to the lists of context sources, you have got to provide a `LicenseHelper` bean, configured with a connection to the content server, as well as the `ContextCollection` bean to be filled by the collector.

3.3.2.2 Implementing ContextSources

Implementing your own `ContextSource` is straightforward. It is quite similar to the implementation of a Spring `HandlerInterceptor` in that the interface declares several methods called in a request's lifecycle. What you do within those methods is entirely up to you, but keep in mind that they are executed for each request, so

- make them fast and
- make them robust.

You are free to throw any kind of exception within a `ContextSource` implementation - the `ContextCollector` represents an exception firewall that will log the exception and continue with the next source.

You will notice that almost all methods in the `ContextSource` interface expect a `ContextCollection` argument. This argument represents the collection used for the current request in the state at the time of the call. Hence, if source A's `preHandle` method is executed before source B's, A will not see any objects added later by B. Keep this in mind if you think about the order of your sources.

There are a couple of conventions you should follow to create a proper `ContextSource`:

- If you want your `ContextSource` to be independent of the type of context object it manages (if your source is only concerned with storing and not modifying contexts in any way, for example), support the `ContextFactory` or `ContextCoDec` interfaces. Most context objects implement these interfaces and thus can readily be used by any source that supports them.
- If your source serializes and persists context objects, check for the `DirtyFlagMaintainer` interface on a context object before storing it. If the interface is implemented and the dirty flag is not set, you do not need to store the context because it has not changed since it was last read. Make sure that you reset the dirty flag if you save the context.

Finally, if you do not need to execute logic in all request phases, you might want to derive your source from `AbstractContextSource`, which provides empty implementations of all `ContextSource` methods.

3.3.2.3 Implementing Context

Context objects are arbitrary POJOs, so you can define and implement them in the way most suitable for your application.

If you want to reuse some of the functionality provided by *CoreMedia Adaptive Personalization*, a specific `ContextSource` for example, you need to implement the required interfaces. In particular, most `ContextSource` implementations require a `ContextFactory` or a `ContextCoDec` implementation for your context, which provide the knowledge of how to create, serialize, and deserialize an instance of your context. Most of them also use the `DirtyFlagMaintainer` interface, writing a context object back into their respective stores only if the context's dirty flag is set.

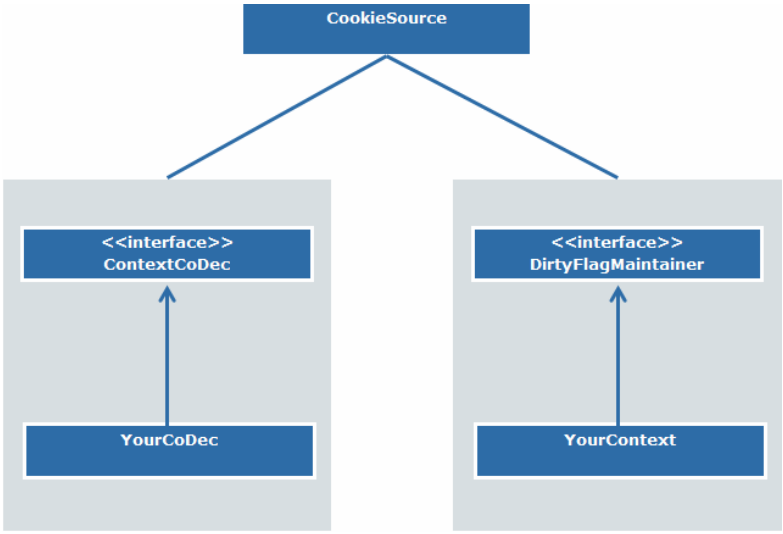

Figure 3.9. A ContextSource implementing typical interfaces

If your context objects contains properties that should be available in selection rules, simply implement the `PropertyProvider` interface.

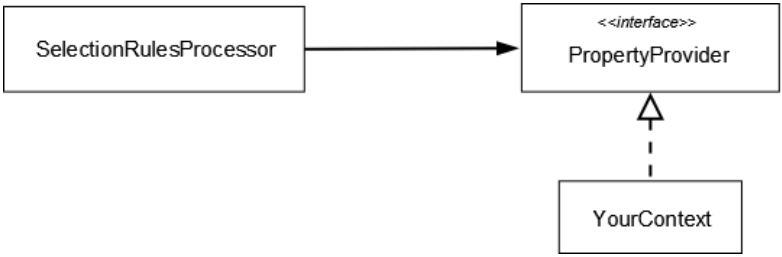

Figure 3.10. PropertyProvider Interface

3.3.2.4 Working With Test Contexts

Test contexts allow you to test your personalized web pages by viewing them with different user-context data in a preview *CAE*. You create a test context as a content of type `CMUserProfile` in *CoreMedia Studio*. Within the CAE, test contexts are created by an instance of `TestContextSource`.

By convention, test contexts are located in the `/System/personalization/profiles` folder of the CoreMedia repository. A content item with name 'DEFAULT' in this folder will be used as the preselected test context for each newly created tab in *CoreMedia Studio*.

The default settings of a `TestContextSource` assume that they are of content type `CMUserProfile` and contain a blob property with MIME type `text/plain` containing the context-property definitions using the syntax of a Java property file. These properties are parsed into one or more context objects that implement the `PropertyProvider` interface.

Setting Up a TestContextSource Instance

The `TestContextSource` requires an instance of `CapConnection` to be able to retrieve the test contexts from *CoreMedia CMS*. In addition, the name of the expected content type can be set. By default, it is assumed that test contexts are defined in content items of type `CMUserProfile`.

Typically, you may want to set up a separate `ContextCollector` instance based on test contexts. To this end, add the `TestContextSource` instance to that `ContextCollector` bean and switch the collector instances before processing a request. The `PreviewPersonalizationHandlerInterceptor` switches context collectors dependent on a request parameter indicating that test context sources are to be used. See Section 3.3.2, "Working With the User's Context" [35] for details on how the `ContextCollector` works.

Adapting a TestContextSource to Project-Specific Requirements

A `TestContextSource` retrieves a test-context content item from the CMS and applies `TestContextExtractors` to the content. The responsibility of a `TestContextExtractor` is to create test contexts from the values of content properties and add them to the supplied `ContextCollection` instance. By default, `TestContextSource` applies the `PropertiesTestContextExtractor`, which creates test contexts given a plaintext blob containing Java-style property declarations.

```
public interface TestContextExtractor {
  void extractTestContextsFromContent(final Content content,
                          final ContextCollection contextCollection);
}
```

You can set the extractors to be applied using the `ContextExtractors` property of the source. This allows you to use new properties or properties with differently structured values to define your test contexts without reimplementing the functionality of `TestContextSource`. For example, to use another property in your test-context content item, follow the following steps:

1. Add the property to the content type definition of `CMUserProfile`.
2. Implement a new `TestContextExtractor` that knows how to create test contexts from the value of your new property.
3. Set the list of extractors to be used by the `TestContextSource` in your CAE to contain the default `PropertiesTestContextExtractor` as well as your own extractor.

You can also change the name of the test-context content type by setting the `TestContextDocType` property of `TestContextSource`.

Customizing the Customer Persona Form

In order to customize the rendering of a `CMUserProfile` via the Customer Persona Form component, you need to understand the underlying basic architecture: The UI component consists of property containers (`PersonaGroupContainer`, for example) that hold one or more property fields (`PersonaStringProperty`, for example). You can configure each of the existing implementations or add your own. To change the appearance of property fields and containers have a look at Section 3.2.3, “Configuring The Customer Persona Form” [25].

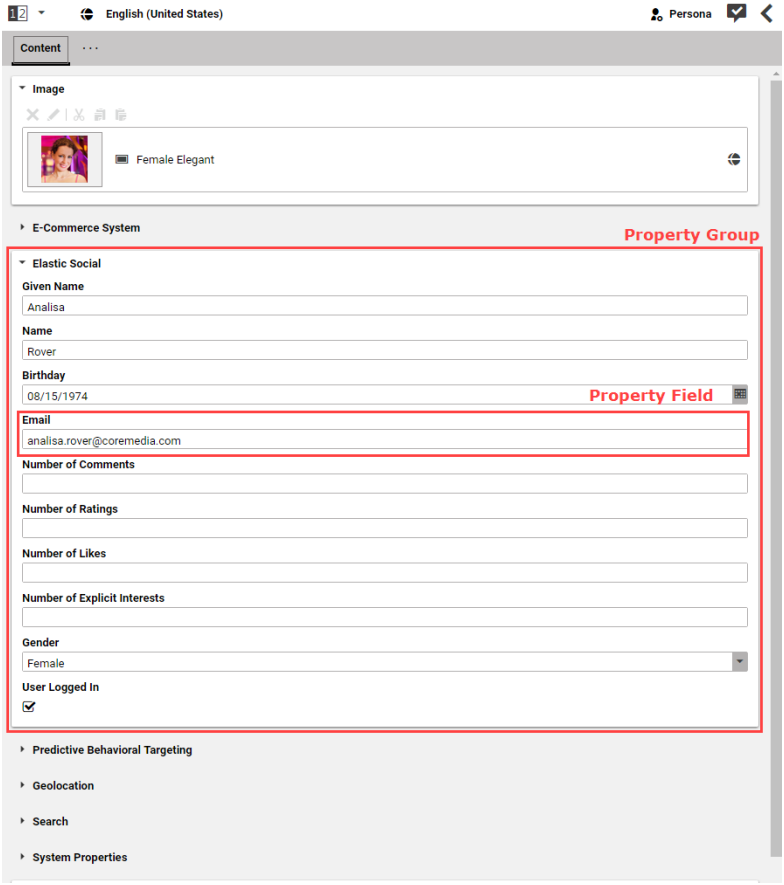

Figure 3.11. Property container and field

Adding your own property field

You can define your own property field in addition to the already existing ones, such as the `PersonaNumberPropertyField`. Your new field needs to contain three major parts:

- It needs to get the context data
- It needs to access the `propertyContext` and `propertyName`
- It needs to bind the entered data to the context property

Get the context data

The user context data is actually a text blob which is interpreted as a properties object. The blob information is stored in a `ValueExpression` accessed via the `bindTo` property of the content item's backing `config` object (see the *Studio Developer Manual* for details). This `ValueExpression` is "forwarded" to the child components of the `CMUserProfile` content. Each child component can access and listen to changes of its given (sub)property. Furthermore, each child component needs to implement the forwarding mechanism as well. You do this by adding a default attribute to your component which is responsible for telling every item to get the corresponding `ValueExpression`.

Access propertyContext and propertyName

If you write your own property field, you need to specify the name and the context of the property you want to add. Therefore, you need to configure two attributes to accept the forwarded `propertyContext` and `propertyName`. This could be done by adding the following snippet underneath your imports:

```
<fx:Declarations>
  <!---
    The context of the Bean-property to bind in this field.
  -->
  <fx:String id="propertyContext"/>

  <!---
    The property of the Bean to bind in this field.
  -->
  <fx:String id="propertyName"/>
</fx:Declarations>
```

Bind your field to the property

By configuring these attributes, you are able to access your property by setting these values to your `propertyBinding` inside your property field. Examples are given in the `p13n-studio` module of the CoreMedia Blueprint development workspace.

3.3.3 Working With Selection Rule Lists

Content Selection Rules allow an editor to define a set of rules that determine which content items to show based on the active user's context. For example, the entry page of a site could take the user's local time into account when selecting a welcome message. To this end, rules that determine what to show under certain conditions are stored in a content property which is evaluated in the *CAE* at time of delivery.

A selection rule is of the general form:

```
select <some content> if <some conditions>
```

Here `<some content>` specifies the content to be selected if `<some conditions>` evaluate to true. The content is specified by its unique id using the syntax

`content:<id>`, while conditions are specified using `<context property name> <operator> <value>`.

<context property name>

The `<context property name>` can have two different forms:

- It can consist of the name of the context object, followed by a dot ['.'] and followed by the name of the context property you want to test in the condition.

Example:

```
select content:23 if count.foo > 12
```

- It can consist of the name of the context object, followed by some more information in brackets ['[]']. Using this notation, the information can simply consist of the context property name, or of a content ID using the syntax `content:<id>`, or an arbitrary string in double quotes. The property name is handled as in the form above.

Example:

```
select content:23 if count[foo] > 12
select content:23 if count[content:12] > 12
select content:23 if count["some complex key"] > 12
```

<operator>

`<operator>` is one of the supported comparison operators. These are:

Operator	Description
=	Equals
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
!=	Not equal

Operator	Description
#	Contains as substring. Only used for string literals

Table 3.3. Supported operators

<value>

`<value>` is the literal value to compare the property value to. Supported types are:

Type	Description
`Boolean`	true or false
`Float`	Examples: 2.34, 0.543e-12
`Integer`	Examples: 42, 1093
`Date`	A date in ISO8601 format (yyyy-mm-ddThh:mm:ss) 2010-12-15T17:08:52, for instance
`Time`	Time of day in the format hh:mm:ss, 23:01:00, for example
`String`	A string literal enclosed in double quotes. Java escape sequences are supported. Examples: "foo", "frob\\\bnitz"
`ContentId`	A representation of a content ID, following the syntax `content:<id>`. For example, `content:4712`. Only equal and not equal operators are supported.

Table 3.4. Supported values

The Evaluation of a condition is performed as follows:

1. Determine the type of the value used in the condition.
2. Retrieve the value of the context property.
3. If the type of the context property value can be compared to the type of the condition value, perform the comparison.
4. Otherwise, evaluate to false.

If the context does not contain the property specified in the condition, the behavior depends on the type of the comparison value:

Type	Behavior
`Boolean`	Assume property is `false`
`Float`	Assume property is 0
`Integer`	Assume property is 0
`Date`	Evaluate to `false`
`Time`	Evaluate to `false`
`String`	Evaluate to `false`
`ContentId`	Evaluate to `false`

Table 3.5. Behavior when the context does not contain the specified property

Conditions can be combined using "and" and "or" in their familiar semantics. Furthermore, negation (not) and parentheses are supported. Thus, the following is a valid condition:

```
behavior.good = true and not
  (datetime.date > 2010-12-25T00:00:00 or vcard.name = "Santa")
```

Rules are separated via a newline character or a semicolon, for example

```
select content:23 if count.foo > 12; select content:42 if count.foo < 5
```

The SelectionRuleProcessor

Rules are evaluated by an instance of `SelectionRuleProcessor`. Its constructor expects a string containing the rules which are transformed into a representation that can be evaluated very efficiently. The `process*` method apply the rules to the supplied `ContextCollection` and return a list of all content items selected in order of their corresponding rules.

The `SelectionRuleProcessor` can only access context objects of type `PropertyProvider`, so make sure that all properties you are using in your rules are accessible via such an object. All context classes supplied with *CoreMedia Adaptive Personalization* implement the `PropertyProvider` interface.

`SelectionRuleProcessor` instances can and should be cached, because the process of transforming a string of rules into an internal representation is expensive and not user or context dependent. The recommended pattern is to add a property getter to your content beans that returns a `SelectionRuleProcessor` instance representing the rules stored in the associated content item, then define a data view on the getter with association type `static`. See the *Content Applications Developer Manual* for a detailed description of data views.

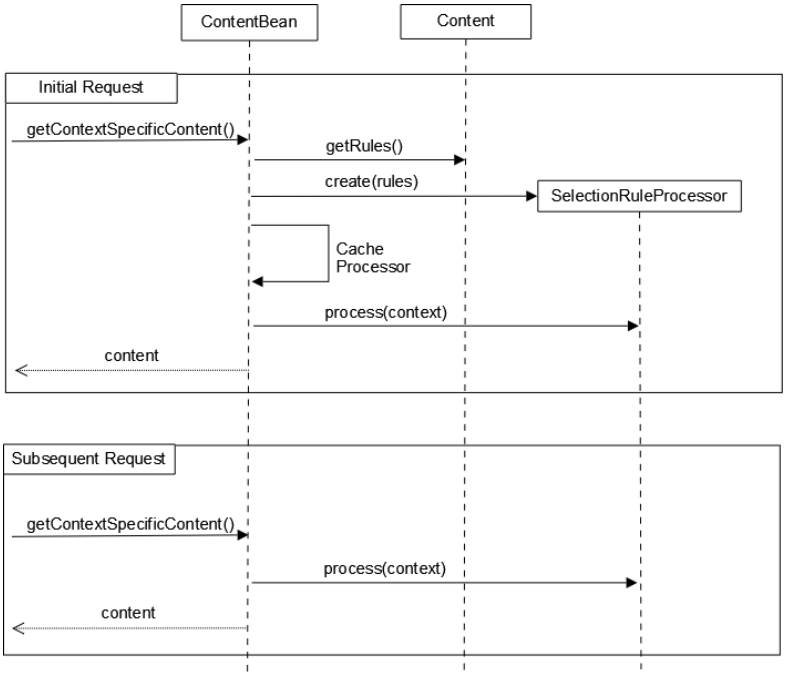

Figure 3.12. Caching `SelectionRuleProcessor` instances

Saving Rules as an XML Property

Selection Rules created via *CoreMedia Studio* are saved in XML format using the grammar `coremedia-selectionrules-1.0`. In this representation, references to content objects (including customer segment definitions) are encoded as `xlink` attributes allowing the *CoreMedia Content Server* to check whether the referenced content is available on the live servers before publishing the rules.

To convert rules in XML format into the plain text format expected by the `SelectionRuleProcessor`, use the helper class `XMLCoDec`.

3.3.4 Working With Customer Segments

Customer segments represent groups of website visitors. Users belong to a segment if they satisfy the conditions associated with the respective segment, for example if a user is a premium user and at most 35 years old.

Segment conditions are stored in a property of a content type that represents segments in your application. In *CoreMedia Blueprint*, this content type is called `CMSegment`. These conditions are used by the `SegmentSource` to determine membership in a segment.

CoreMedia Adaptive Personalization offers a *CoreMedia Studio* field editor for segment conditions called `ConditionsField`.

Configuring the SegmentSource

`SegmentSource` is a `ContextSource` implementation that evaluates segment conditions to determine the current user's membership in a segment. The source evaluates the conditions in its `preHandle` method for each request. The conditions are applied to the contents of the `ContextCollection` at the time of invocation of `preHandle`, thus the `SegmentSource` must be placed behind all other sources that provide context information used in the segment conditions.

Membership in a segment is indicated by a property of the segment's simplified content ID (content:<id>) of the content item representing the segment. So a segment represented by content 42 will be mapped to the property 'content:42'. This property is set to the Boolean value 'true' if the user is a member of the segment; segments a user does not belong to are either not represented in the context or are assigned a value of 'false'.

The `SegmentSource` requires a reference to the `Cache` used for storing preprocessed segment conditions and to the `ContentRepository` to retrieve segment content items. Further, as with all sources, you've got to provide the name of the context to be used to store the segment properties.

Optionally, you may configure in which folder of the repository the source looks for segment content items, the content type used to represent segments, and the name of the property of the content type that contains the segment conditions.

Property Name	Required	Default	Description
`cache`	Yes		Reference to the CoreMedia Cache to be used to store preprocessed segment conditions.

Property Name	Required	Default	Description
`contentRepository`	Yes		Reference to the content repository containing the segment content items.
`contextName`	Yes		Name to be used for the context containing the segment properties.
`pathToSegments`	No	/System/personalization/segments	Repository folder in which to look for segment content items.
`segmentDocType`	No	CMSegment	Name of the content type used to represent customer segments.
`conditionPropertyName`	No	condition	Property of the segment content type that contains the segment conditions.

Table 3.6. Properties of SegmentSource

Configuring the property editor used for segment conditions

`ConditionsField` is a property editor for conditions. This editor is configured similar to the `SelectionRuleField` by supplying the list of supported condition types and their mapping to user profile properties.

Configuring the SelectionRulesField to offer conditions on customer segments

To enable conditions on customer segments in the `SelectionRulesField` property editor, configure the `SegmentCondition` component. Make sure its `propertyPrefix` attribute matches the name of the context object used for storing segments in the *CAE*.

3.3.5 Working With Scoring

Scoring is a simple means to abstract an individual user's behavior on a website. In general, the idea is to assign scores to certain observable events and to combine these scores with the user's current scores whenever the events are observed.

NOTE

Please note that profiling user behavior as well as taking automated decisions based on this profiling may be restricted by applicable privacy law and may require specific user consent to be obtained and may depend on how you implement the Adaptive Personalization module for your project.

Example

Assume the pages on a website are tagged with keywords and you want to keep track of how often the user visits pages tagged with a specific keyword. In this scenario, a visit on a page is an observable event, and the scores are the counters associated with each keyword. Whenever the user visits a page, the scores of all associated keywords are incremented by 1.

CoreMedia Adaptive Personalization supports scoring via the `ScoringContext`. It manages a set of scores and uses a `ScoringStrategy` to update scores if events are observed.

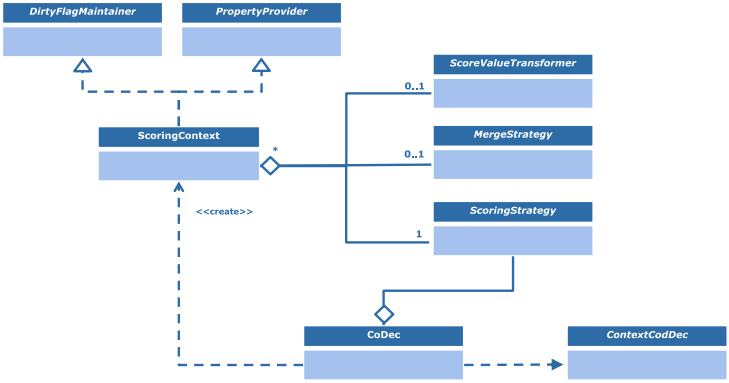

Figure 3.13. Scoring classes

CoreMedia Adaptive Personalization comes with a set of predefined scoring strategies:

- `CountScoring` This strategy simply counts the occurrence of events. That is, for each supplied event, the corresponding score is incremented. This strategy can be used to implement the keyword scenario described above.

- `PercentageFromMaxScoring` This strategy weights each score by its percentage of the maximum score value. For each event, a score of the corresponding key is maintained and incremented by 1 whenever the event is observed.
- `PercentageFromTotalScoring` This strategy weights each score by its percentage of the sum of all scores. For each event, a score of the corresponding key is maintained and incremented by 1 whenever the event is observed.

Going back to the keyword example, assume that page A is tagged with keywords 'foo' and 'bar', and page B is tagged with 'bar'. Further, assume that a new user visits page A once and page B twice. Here is the table of scores that result from applying the different strategies:

Strategy	foo	bar
CountScoring	1	3
PercentageFromMaxScoring	1/3	3/3
PercentageFromTotalScoring	1/4	3/4

Table 3.7. Example results

In addition to the application specific scores, two general scores are maintained by all three strategies:

- __max__ contains the maximum score of all scores maintained by the context
- __total__ contains the current total of all scores maintained by the context

The scoring strategies are interchangeable, that is, if you start with one you can reconfigure your system later to use a different one without loosing any data.

Configuring a ContextSource to use the ScoringContext

`ScoringContext` provides its own `ContextCoDec` implementation in the static inner class `ScoringContext$CoDec`. The codec can be used in any source that accepts a `ContextCoDec` or a `ContextFactory`. Because the `ScoringContext` requires a `ScoringStrategy`, you must inject the strategy you want to use for all decoded and created contexts into the codec.

Here is an example of how to configure a `CookieSource` to use a `ScoringContext` with the `PercentageFromTotalScoring` strategy:

```
<bean id="scoringCookie"
 class="com.coremedia.personalization.context.collector.CookieSource"
 type="singleton">
  <property name="contextCoDec">
    <bean class="com.coremedia.personalization.scoring.
```

```
                ScoringContext$CoDec">
      <property name="strategy">
        <bean class="com.coremedia.personalization.scoring.
                     PercentageFromTotalScoring"/>
      </property>
    </bean>
  </property>
  <property name="contextName" value="scoringContext"/>
</bean>
```

Writing your own ScoringStrategy

Writing your own scoring strategy is as simple as implementing the `ScoringStrategy` interface. Keep in mind that your implementation must be thread-safe because it is typically shared by several `ScoringContext` instances. Ideally, you simply do not use any modifiable state that is shared among threads.

In typical scenarios, processing events is far more frequent than reading scores. Thus, it's sensible to perform costly updates lazily only when scores are requested. To this end, your strategy may implement the `ScoreValueTransformer` interface. If a strategy implements this interface, the `ScoreValueTransformer#transform` method is called by the `ScoreContext#getScore` method and its result returned as the score. The supplied strategies `PercentageFromMaxScoring` and `PercentageFromTotalScoring` use this to perform the normalization of values only at the time of access.

The third interface that is relevant to scoring is `MergeStrategy`: A `ScoringStrategy` that allows merging of two sets of scores should implement this interface. Merging of scores is useful if you want to combine data from different context. A typical scenario is as follows: A user logs into your site and his scoring context is persisted in a database. Later, the user returns to the site and browses without logging in, thus new scores are collected. Then, with the user logging in, the formerly persisted data becomes available and can now be merged with the scores collected while the user was anonymous.

If your `ScoringStrategy` implements the `MergeStrategy` interface, a `ScoringContext` using your strategy will be able to perform the `mergeWith` operation.

Using a ScoringContext to track Keyword Clicks

CoreMedia Adaptive Personalization provides the `KeywordInterceptor` for the common use case in which you want to count the keywords associated with the pages a user clicks on. The `KeywordInterceptor` intercepts a *CAE* request after the controller but before the view is rendered and attempts to extract keywords from the 'self' bean in the model that is to be supplied to the view dispatcher. These keywords

are sent as events to the configured `ScoringContext`. See the respective Javadoc for details.

3.3.6 Working With Search Queries

You can use queries to the *CoreMedia Search Engine* to dynamically compile parts of your website's pages. Nevertheless, using this method, you do not have context information for your queries. To solve this problem, search functions provided by *CoreMedia Adaptive Personalization* come in handy. They enable you to include context-specific data into your queries, thus providing you with another means to adapt your site to the visitor.

Example

You use folders in the CMS repository, that represent a specific customer segment. That is, each folder contains content that will be shown to a user who is member of the respective segment. Now, you compile a page about sports products and want to show content depending on the user's segment. Let's say, Skateboard products for the young urban segment and Golf products for the successful prime-age manager segment. Now, you can use a search query similar to `sports userSegment()`. Where `userSegment()` is a search function that is evaluated at query time and presumably adds the required folder constraint to the query. That is, if the user is in the segment mapped to the folder of id 23, the string actually sent to the search engine would be `sports folderid:23` (assuming `folderid` is the field, IDs of folders get fed to).

CoreMedia Adaptive Personalization comes with some generally useful functions in the `com.coremedia.personalization.search` package. Nevertheless, since search functions are very project specific, you will use these delivered functions as a starting point for your own functions.

- In Section 3.3.6.1, “Evaluation Of Search Functions” [53] you will learn how search functions are evaluated.
- In Section 3.3.6.2, “Implementing Search Functions” [54] you will learn how to write your own search functions.

3.3.6.1 Evaluation Of Search Functions

Typically, you access your *Search Engine* from within a content bean implementation. Within the bean, you will do the following things:

1. Read the query string from a property of the associated Content object

2. Use a *Search Engine* connection to send the query to the *Search Engine*
3. Retrieve the result object
4. Iterate over the results to map them to content beans which can then be provided to the template for rendering.

If your query string contains calls to search functions, you can't just provide the string to the *Search Engine* because the *Search Engine* doesn't know what to do with the functions. So, you first got to evaluate the functions and replace their calls by their respective results, thus creating a syntactically correct query string that can be send to the engine. Evaluation and replacement of search function calls is performed by the `SearchFunctionPreprocessor`.

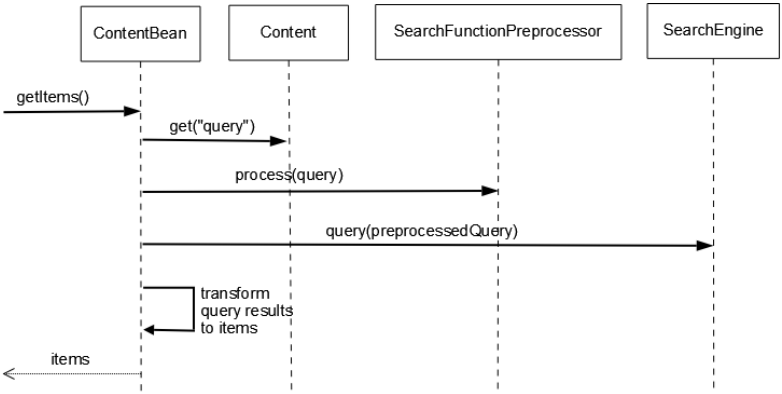

Figure 3.14. Evaluating a Search Function

3.3.6.2 Implementing Search Functions

The `SearchFunctionPreprocessor` maintains a map of search function names and implementations. The registered name of a function is used to call it from within the query string and, if a call is encountered in the query, it's replaced by the result of the executed implementation.

A search function implementation is an instance of a Java class that implements the `SearchFunction` interface. This interface contains a single method only; `evaluate`. The preprocessor supplies the `ContextCollection` associated with the current request and all function arguments supplied in the function call to this method.

What's happening inside of the evaluate method is entirely up to you. The only constraint is that the resulting string should by a syntactically valid [sub]query to your *Search Engine*.

Search function arguments are in the form `<parameter name>:<value>` and are supplied to a function in an instance of class `SearchFunctionArguments`. The latter provides a number of convenience methods to access arguments and convert their values to appropriate types.

If you implement your own search functions, make sure they are thread safe because the `SearchFunctionPreprocessor` is usually declared as a singleton Spring bean. This means that several request threads may access the preprocessor and the registered search functions in parallel.

Example

The search function `SolrGeneralProperty`, which is provided as part of *CoreMedia Adaptive Personalization*, provides access to a general context property from within a query in Solr syntax. If it is registered with the `SearchFunctionPreprocessor` under the name "contextProperty", preprocessing the query `recommendations contextProperty(property:personal.name, field:user)` calls the `evaluate` method of the registered instance of `SolrGeneralProperty` supplying the current `ContextCollection` and function arguments `property:personal.name` and `field:user`.

`SolrGeneralProperty` looks up the context object named "personal" in the `ContextCollection` and retrieves the value of its property name, which is assumed to be "bob". Then, it concatenates the field argument with the retrieved name to the valid Solr search query "user:bob" and returns this string.

The preprocessor replaces the function call by the returned string, resulting in the query "recommendations user:bob".

Exception Handling

The `SearchFunctionPreprocessor` wraps any exception that is thrown while evaluating a search function's evaluate method in a runtime exception of type `SearchFunctionEvaluationException`. In addition to the exception cause, the `SearchFunctionEvaluationException` is supplied with the name under which the executing search function is registered.

Implementations of `SearchFunction` are encouraged to use one of the `Argument*Exception` classes if there is any problem with the arguments supplied in `SearchFunctionArguments`. These exception classes are known to the `CoreMedia Studio` integration provided as part of *CoreMedia Blueprint* and are used to provide improved feedback to *CoreMedia Studio* users in case they make any mistakes using search functions.

Spring Configuration

The `SearchFunctionPreprocessor` is intended to be configured as a Spring bean. It is thread safe so using the default Spring singleton scope is fine.

Here is an example configuration that registers three search functions with the processor:

```
<bean class="com.coremedia.personalization.search. \
      SearchFunctionPreprocessor">
  <property name="functions">
    <map>
      <entry key="userKeywords">
        <bean class="com.coremedia.personalization. \
              search.solr.SolrScoredKeys">
          <property name="defaultLimit" value="5"/>
          <property name="defaultThreshold" value="0"/>
          <property name="defaultContextName" value="keyword"/>
          <property name="defaultField" value="keywords"/>
        </bean>
      </entry>
      <entry key="userSegments">
        <bean class="com.coremedia.personalization. \
              search.solr.SolrSegments"/>
      </entry>
      <entry key="contextProperty">
        <bean class="com.coremedia.personalization.search. \
              solr.SolrGeneralProperty"/>
      </entry>
    </map>
  </property>
</bean>
```

3.3.6.3 Adding Help Texts

In order to support the users of your search functions, you can add a help text to *CoreMedia Studio*. This text might describe, for example, how to call the function, what the function does and what arguments are required.

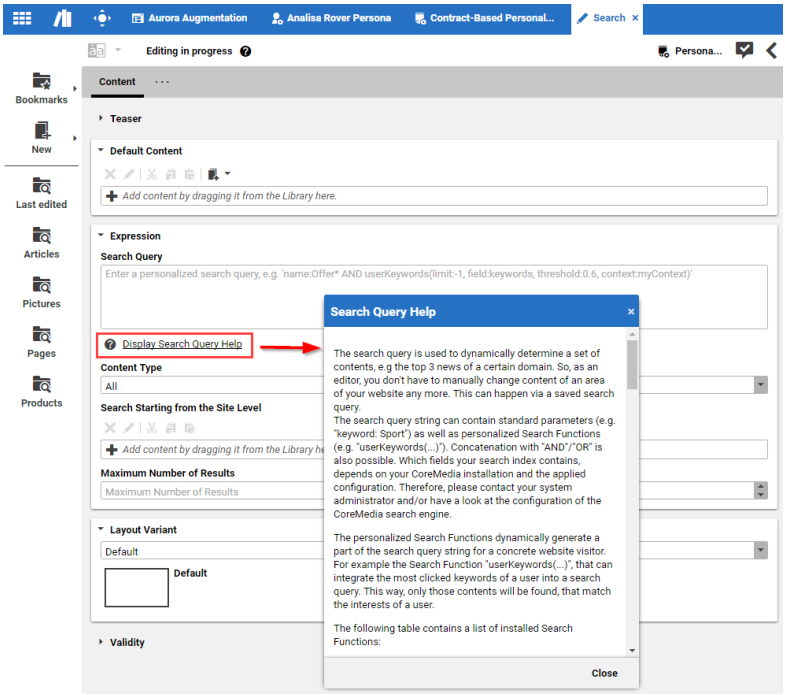

Figure 3.15. Example of a help text

To add a help text, *CoreMedia Adaptive Personalization* provides the `SearchQueryHelper` component in `cap-personalization-ui`. The help text is written as an HTML file. Proceed as follows:

Write your help text and store the file as `SearchFunctionsHelp.html` in the directory `sencha/resources/p13n-search-query-help/` of your web application.

Add the `SearchQueryHelper` with the tag `<perso:SearchQueryHelper>` to the content form where it should be shown.

3.3.7 Localizing the Studio Plugin

The Studio plugin of *CoreMedia Adaptive Personalization* enhances *CoreMedia Studio* with several UI components. You can adapt any labels shown by these components. To do so, override the respective properties in the global *CoreMedia Studio* `Resource Bundle` either programmatically or by using a property file.

All changes that are done programmatically have to be applied in the `init` method of the class `PersonalizationEditorPlugin` that is located in the `p13n-studio` module of the CoreMedia Blueprint development workspace.

4. Client-side Personalization

Client-side Personalization decides on the user client software (mostly the browser) which content to show. The rules are defined in a third-party system, such as Monetate.

Client-side Personalization consists of three extensions, which can be integrated into the CoreMedia system:

- The p13n-core extension which offers the basis functionality used by the adapter extensions. This extension must always be installed.
- The p13n-adapter-monetate extension which connects with Monetate.
- The p13n-adapter-generic extension which connects with Evergage and Dynamic Yield.

You must always install p13n-core as the base extension required by the other ones.

Based on the capability of the provider the following use cases are supported:

- Optimization and Testing: Run experiences with split traffic allocation (A/B/n tests) and targeting
- Personalization: Run experiences with machine learning and targeting
- Segmentation: Use targeting capabilities of the provider to serve custom content experiences to different user segments

For all these use cases, content editors in CoreMedia Studio can easily create and manage content and connect them to the provider for optimized and personalized delivery.

p13n-core Extension

The CoreMedia p13n-core extension provides the base functionality for integrating third party web personalization, optimization, and testing providers with CoreMedia.

p13n-adapter-monetate Extension

Monetate is a leading provider of personalization, targeting and optimization solutions for websites.

The p13n-adapter-monetate extension for CoreMedia Blueprint enables you to use Monetate's advanced technology to personalize, target and test content in an easy way to build a whole new customer experience.

p13n-adapter-generic Extension

The p13n-adapter-generic adapter is a generic means to connect third party web personalization, optimization and targeting solutions that don't provide dedicated client APIs to query metadata state.

The current implementation has been tested with the following systems:

- Dynamic Yield
- Evergage

Architecture

For all supported personalization providers, a client-side integration approach is used:

The provider's JavaScript tag is rendered into the head of the generated HTML output. The script tag loads and calls additional JavaScript code which evaluates the current request and determines the actions to run on this page, for example which variant to show from what experience. When actions related to CoreMedia experiences are run, custom CoreMedia JavaScript callbacks are triggered, loading personalized content items from the CAE via AJAX, replacing default content items on the page. Using this approach offers two main benefits:

In addition to CMS content actions, all capabilities of the provider can be used for use cases that are currently not supported by CoreMedia. The generated CoreMedia pages are not dependent on the personalization actions and can be cached.

Content Placement and Rendering Restrictions

Client-side Personalization bases on a few assumptions that affect the way you can place content and render content:

- Experiences and segments target single teasable items only - not collections. This means it is not possible to replace an item with a collection of items and vice versa.
- A baseline item must always exist, therefore, it is not possible to add content to the variants/segments while the baseline remains empty. Moreover, for all variants/segments a content item must be defined, which means it is not possible to add a baseline content item and leave some or all the variant/segment content empty.
- The design of segments follows the idea that a user is part of only one segment. There is no ruleset to combine different segments to target content.
- Different experiences/segmentation documents cannot target the same baseline slot.

Personalization based on React and Spark

This manual shows in addition how you can implement the CoreMedia Client-side Personalization by extending Spark, an example application based on React, TypeScript and the Headless Server of CoreMedia Content Cloud.

See Section 4.3, “Client-Side Personalization Using Headless Server” [89].

4.1 Installing Client-Side Personalization

Using CoreMedia Client-side Personalization requires the installation of at least two extensions (see Section 4.1.5, "Project Extensions" in *Blueprint Developer Manual* for more details about extensions):

- The p13n-core extension at https://github.com/coremedia-contributions/p13n-core.
- At least, one of the adapters p13n-adapter-generic (https://github.com/coremedia-contributions/p13n-adapter-generic or p13n-adapter-monetate (https://github.com/coremedia-contributions/p13n-adapter-monetate) to connect with a third-party system.

You will find the extensions in the following repositories:

- https://github.com/coremedia-contributions/p13n-core

  The core extensions that must be installed.
- https://github.com/coremedia-contributions/p13n-adapter-monetate

  This adapter connects to Monetate and provides a direct integration using the Kibo metadata API.
- https://github.com/coremedia-contributions/p13n-adapter-generic

  This adapter provides a generic integration base for providers without a client API. Experiences must be mirrored in the CoreMedia system with a special content type. Built-in support is included for Dynamic Yield and Evergage

You can either add the extension repositories as Git submodules, the recommended approach, or copy them into your Blueprint workspace in the `modules/extensions` folder. Copying the extensions requires more work when upgrading the Blueprint workspace to newer releases. In the following section you will learn how to add them as submodules.

Adding Extension as Submodule

If you plan to customize the extensions, first create a fork of the repositories and add your forks as submodules. Otherwise, you can simply add the CoreMedia repositories as submodules as shown in the following example.

Add the core and adapter extensions to the Blueprint workspace as Git submodules as follows (this example uses the Monetate adapter). To do this, open a terminal window and run the following commands:

```
cd /<blueprint-root-dir>

mkdir -p modules/extensions

git submodule add https://github.com/coremedia-contributions/p13n-core.git
modules/extensions/p13n-core

git submodule add
https://github.com/coremedia-contributions/p13n-adapter-monetate.git
modules/extensions/p13n-adapter-monetate

git submodule init
```

Example 4.1. Adding submodules

After the submodules are added, go to each submodule directory and check-out the branch matching your Blueprint version.

```
cd /<blueprint-root-dir>/modules/extensions/p13n-core
git checkout -b <branch-name>
cd ../p13n-adapter-monetate
git checkout <branch-name>
```

Example 4.2. Checkout branch in submodule

Then commit the changes to the submodules:

```
cd /<blueprint-root-dir>/modules/extensions
git add p13n-core
git add p13n-adapter-monetate
git commit -m 'Add personalization submodules'
```

Example 4.3. Commit changes to submodules

Now, you have to activate the extensions.

Activating the Extensions

Run the extensions tool (see Section 4.1.5, "Project Extensions" in *Blueprint Developer Manual* for more details about the extensions tool) in `workspace-configuration/extensions` to activate the extensions like this (here, the core and the monetate extension are activated):

```
mvn extensions:sync
mvn extensions:sync -Denable=p13-core,p13n-adapter-monetate
```

Example 4.4. Activate extensions

Now you are done with the installation and activation of the extensions. In order to work with the extensions, you have to configure them, as described in the next sections.

Updating the Extensions

If you are upgrading your Blueprint workspace to another major release, an update of the personalization extension may be required. Check the extension's repositories for the available branches and select the branch matching your Blueprint release version or the highest release version lower than your Blueprint version.

Update the extension's submodules by opening a terminal window and running the following commands (the example shows the steps for the Monetate adapter):

```
cd /<blueprint-root-dir>/modules/extensions
cd p13n-core
git fetch
git checkout <branch-name>
cd ..
git add p13n-core

cd p13n-adapter-monetate
git fetch
git checkout <branch-name>
cd ..
git add p13n-adapter-monetate/

git commit -m 'Update personalization submodules to release xxxx.x'
```

Example 4.5. Updating an extension

4.2 Client-Side Personalization Configuration and Operation

This section describes, how you configure Client-side Personalization. Getting Client-side Personalization working contains the following steps:

1. Install the p13n-core extension and at least one adapter extension. See Section 4.1, "Installing Client-Side Personalization" [62] for details.
2. Configure the p13n-core extension which is the basis for the adapter extensions. See Section 4.2.1, "Configuring the p13n-core Extension" [65] for details.
3. If you want to use segmentation, install P13NSegment content items in *CoreMedia Studio*. See Section 4.2.2, "Creating Segments in Studio" [67] for details.
4. Configure the installed adapter extensions. See Section 4.2.3, "Configuring the p13n-monetate-adapter Extension" [68] and Section 4.2.4, "Configuring the p13n-adapter-generic Extension" [77] for details.

4.2.1 Configuring the p13n-core Extension

The `p13n-core` extension adds, for example, the following features:

- Content types
- API definition
- Studio forms and preview
- Studio backend implementation
- CAE core logic

In order to use the p13n-core extension you have to extend your frontend theme with a brick

4.2.1.1 Frontend Integration

The frontend theme can be extended with a brick that acts on the callbacks received from the personalization provider. The brick is responsible for loading variant fragments and displaying the baseline content if no variant is triggered. The provider's script tag is independently included in the adapter CAE extensions.

Together with a small piece of JavaScript proxy code injected into the HTML head by the CAE extension, the brick provides the callback functions called from the 3rd party service if a variant or segment is to be displayed and replaces the default content fragment with the variant loaded via AJAX.

The frontend code adds the object `cm_p13n` to the `window` object providing the following methods:

- `pushVariant(variantId:String)`

  Activates the variant with the given ID.

- `pushSegment(segmentName:String)`

  Activates the segment with the given name.

- `exchangeVariant(variantId:String, contentId:String)`

  Replaces a variant when using the preview icon of a variant in the content item tab.

- `completed(providerId)`

  Called when the 3rd party service is finished and displays the baseline for all fragments where no variant or segment is active.

In addition, an event listener is installed for integration with the Studio preview.

Adding the brick to the theme

1. Add the brick `p13n-dynamic-include` to your frontend bricks and add a dependency for your theme into the `package.json` file:

```
"@coremedia/brick-p13n-dynamic-include": "^1.0.0",
```

2. Modify the `dynamic-include` brick to ignore the fragments from the p13n integration in the `init.js` file:

```
addNodeDecoratorByData(undefined, FRAGMENT_IDENTIFIER, function (
  $fragment,
  url
) {
  if($fragment.data("cm-experience-links")) {
    return;
```

```
  }
  utils.pushTaskQueue();
```

4.2.1.2 Fragment Caching

Contrary to CoreMedia's Adaptive Personalization which resolves variant selection on the server side, the client-side personalization resolves everything in the client's browser. Thus, the fragments included via AJAX do not require any user specific server side processing and are cacheable for a limited time.

CAE Caching

To enable the required HTTP caching headers add the following line to the live CAE's `application.properties` file and adjust the cache time to your need:

```
cae.cache-control.for-url-pattern[/dynamic/fragment/experience/item/**].max-age=5me
```

Caching in a commerce-led Scenario

If you are running the client-side personalization in a commerce-led scenario, the cache headers from the CAE might not be passed through the eCommerce delivery system. To counter this, add static rules to your CDN, allowing caching and storing of fragments with the path `<prefix>/dynamic/fragment/experience/item/**` on the client's side.

4.2.2 Creating Segments in Studio

For the segmentation use case you have to create a `P13NSegment` content item for each segment you want to use in the `<Site Root>/Options/Personalization/Segments/` folder of the site. This is necessary for all adapters. For the content name use lower- and upper-case letters, numbers, minus and underscore, no special characters are allowed.

4.2.3 Configuring the p13n-monetate-adapter Extension

The `p13n-monetate-adapter` extension connects the CoreMedia system with the Monetate personalization platform.

Configuring the p13n-monetate-adapter extension requires the following steps:

1. Enabling the Monetate connection in Studio (see Section 4.2.3.1, "Connecting Monetate with CoreMedia Content Cloud" [68] for details).
2. Creating experiences for different use cases (see Section 4.2.3.2, "Creating Experiences in Monetate" [70] for details).

4.2.3.1 Connecting Monetate with CoreMedia Content Cloud

To connect *CoreMedia Content Cloud* with the Monetate system, you have to connect the CAE and Studio backend with Monetate and configure each site that should use the Monetate connection.

Configuring the Connection

To connect CoreMedia Content Cloud with the Monetate system, proceed as follows:

1. Create an RSA key pair, for example, with `openssl`.
2. Open the Monetate portal and navigate to **Account Settings** → Sites → **APIKeys**. and create a new API user.
3. Add the newly generated public key to the API user.
4. Configure the connection in the `monetate-connector.properties` file in `p23n-adapter-monetate-lib/src/main/resources/META-INF`. There, you will find a description of the properties. An example configuration would look as follows:

```
p13n.adapters.monetate.connectors[0].id=coremedia
p13n.adapters.monetate.connectors[0].retailer=coremedia
p13n.adapters.monetate.connectors[0].key-string=MIIBrzBJBgkqhkiG...
p13n.adapters.monetate.connectors[0].api-user=api-0815-connector
p13n.adapters.monetate.connectors[0].tag-uri-template=//se.monetate.net/js/2/{channel}/entry.js
p13n.adapters.monetate.connectors[0].analytics-uri-template=https://marketer.monetate.net/
            control/{channel}/analytics/{experienceId}/realtime
p13n.adapters.monetate.connectors[0].edit-uri-template=https://marketer.monetate.net/control/
            {channel}/experience/{experienceId}
```

```
p13n.adapters.monetate.connectors[0].filterExpressions[0]=(.*)\\$
p13n.adapters.monetate.connectors[0].filterExpressions[1]=(.*)#
p13n.adapters.monetate.connectors[0].filterMode=exclude
```

Example 4.6. Example Monetate configuration

Excluding Experiences from Studio

In Section 4.2.3.2, "Creating Experiences in Monetate" [70] you will learn how to create experiences in Monetate. However, not all experiences in the Monetate system are applicable for use with a CoreMedia Monetate Experience content item, for example, experiences used for segmentation, the 'Final' experience (see Section "Creating Experience for Content Masking" [76]), which only serves a technical function, and experiences that don't target content (changing layout, for example). These experiences should be hidden from the Studio user and be not selectable in the experience drop-down list.

For this use case, you can apply the properties `monetate.connectors[n].filterMode` and `monetate.connectors[n].filterExpressions[n]` as shown in section "Configuring the Connection" [68]

Configuring the Site

To connect a site with the Monetate system, you have to create a `Settings` content item and link to it from the root page of the site. In the `Settings` content item set the following properties:

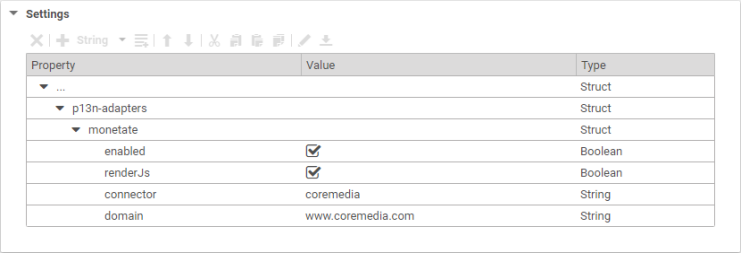

Figure 4.1. Properties for Monetate connection

The properties have the following meaning:

Property	Required	Default	Description
enabled	x		Enables the Monetate integration for this site.

Property	Required	Default	Description
renderJS		true	Instructs the CAE to include the Monetate tag in its head section. Disable in a commerce-led scenario where the shop frontend already includes the Monetate scripts.
connector	x		The ID of the backend connector. See the property `p13n.adapters.monetate.connectors[0].id` in section "Configuring the Connection" [68].
domain	x		The domain name. Corresponds to a site in your Monetate account.

Table 4.1. Monetate properties for site connection

4.2.3.2 Creating Experiences in Monetate

In order to use Monetate experiences and segments in CoreMedia Studio, you need to define experiences that use the CoreMedia JavaScript action.

You can define three types of experiences:

- Experiences for personalization and testing (see Section "Creating Experiences for Personalization and Testing" [72]).
- Experiences for segmenation (see Section "Creating Experiences for Segmentation" [74])
- A technically required experience to minimize visually disturbing effects in the website (see Section "Creating Experience for Content Masking" [76])

Prerequisites

- Installation of the p13n-core extension and the p13n-adapter-monetate extension as described in Section 4.1, "Installing Client-Side Personalization" [62]
- Configuration of the p13n-core Extension as described in Section 4.2.1, "Configuring the p13n-core Extension" [65].
- If you want to use segmentation, creation of segment content items in Studio as described in Section 4.2.2, "Creating Segments in Studio" [67].

- Set up of the Monetate integration as described in Section 4.2.3.1, “Connecting Monetate with CoreMedia Content Cloud” [68]

Setting Up a JavaScript Action

In order to use experiences from Monetate in CoreMedia Sites you first have to create an action in Monetate.

1. Go into the Monetate Action Builder, select **Components** → **Actions** and click **[Create Action]** . The live website opens up and the Monetate overlay is enabled:

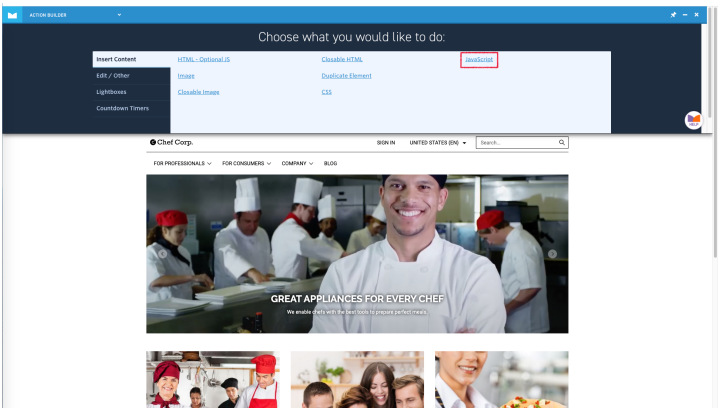

Figure 4.2. Create Action in Monetate

2. Select *JavaScript* and in the following screen set *Only run once?* (1) to *Yes*. Click the arrow (2) to go to the next screen.

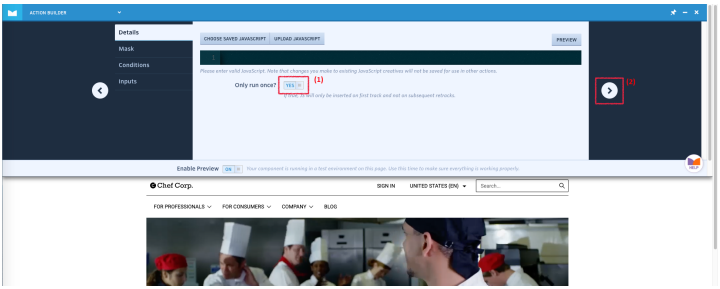

Figure 4.3. Monetate action screen

3. Fill the form fields as shown in the screenshot and click **[Create & Exit]** to create the action and to return to the Monetate portal.

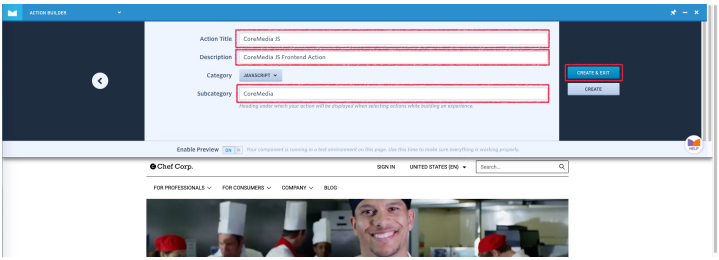

Figure 4.4. Monetate enter data for new action

Creating Experiences for Personalization and Testing

1. Follow the default steps for creating a Monetate experience. You can select any type of goal (WHY). Apply targeting rules as required (WHO). Then create the variants (WHAT). For each variant add a single action — the CoreMedia JavaScript action created in Section “Setting Up a JavaScript Action” [71].

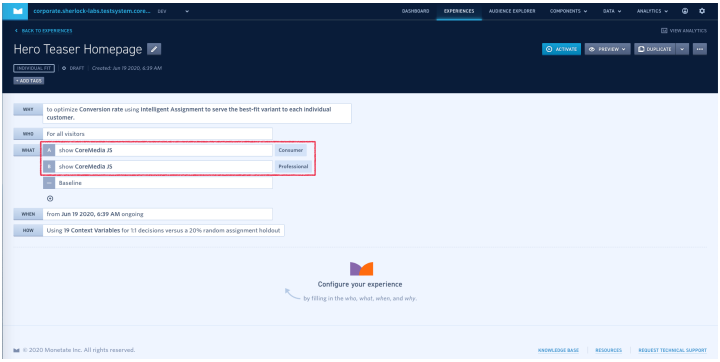

Figure 4.5. Create experience in Monetate

2. Configure each action with the required JavaScript code using one of the following two methods:
   - Look at the URL in your browser. It should look similar to `https://marketer.monetate.net/control/<ACCOUNT>/p/<DOMAIN>/experience/1275940#c1517122:what,a3489594`. Note the number

after #c, in this example 1517122. Copy `cm_p13n.pushVariant("<NUMBER>")` to the clipboard where <NUMBER> is replaced by the number above.

- Change to CoreMedia Studio, select your preferred site and open the dashboard. If the dashboard does not contain the experience widget, add it to the dashboard. Select the newly created experience from the widget's list and open it in a new tab (if the experience is missing, use the reload button to update the list and check the widget's filter settings). In the variants panel each variant has a link button to copy the required JavaScript code the clipboard:

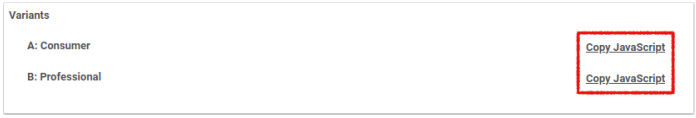

Figure 4.6. Copy JavaScript code from Studio

Click the **[Copy JavaScript]** button.

Go back to the Monetate portal, select the variant action and paste the copied code into the JavaScript code field.

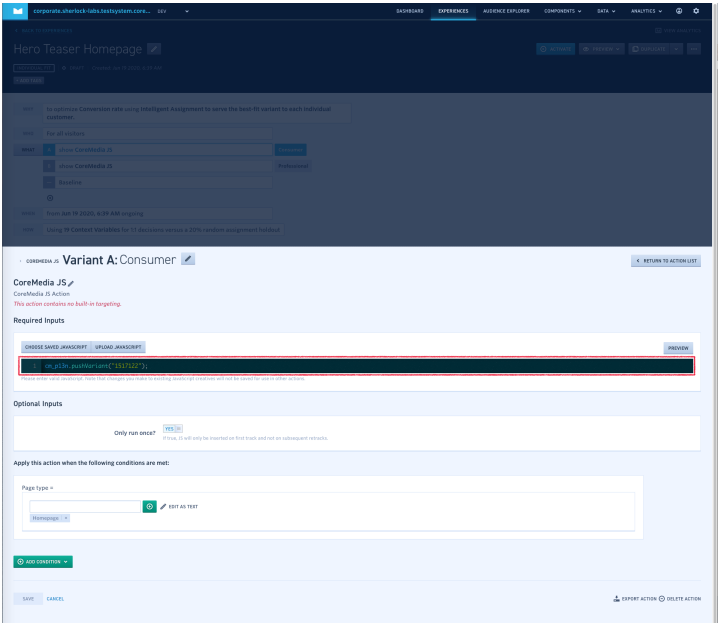

Figure 4.7. Insert JavaScript code into Monetate action

3. If the experience should not be placed on every page, add conditions restricting the action to a single page or set of pages using the **[Add Condition]** button.

Creating Experiences for Segmentation

The supported segmentation use case divides the customer base into disjoint sets of segments. For each segment, you want to use, you have to create a `P13NSegment` in CoreMedia Studio. You have two ways of applying an experience to a segment:

- Each segment can be used as a direct target rule of an experience.
- The Monetate AI decides about the segment with a single individual fit experience.

For the first case you create an experience for each segment, while for the second case you only create one experience with all segments.

Prerequisites

You have already defined P13NSegment content items in Studio as described in Section 4.2.2, “Creating Segments in Studio” [67].

Direct Target Rule

1. Define an experience with the goal *to serve everyone the same experience*. Apply the required targeting rules and add a single variant with the CoreMedia JavaScript action.

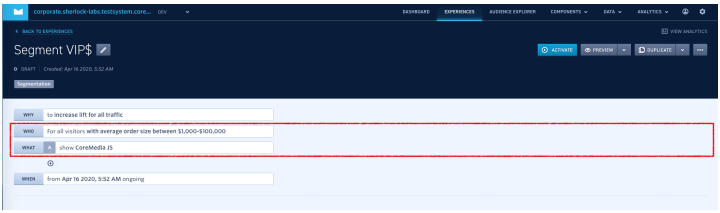

Figure 4.8. Define Monetate experience

2. Select the variant action and paste the following code into the JavaScript code field:

```
cm_p13n.pushSegment("<segment_name>");
```

Replace **`<segment_name>`** with the name of the corresponding Studio content item, which you have created in Section 4.2.2, “Creating Segments in Studio” [67].

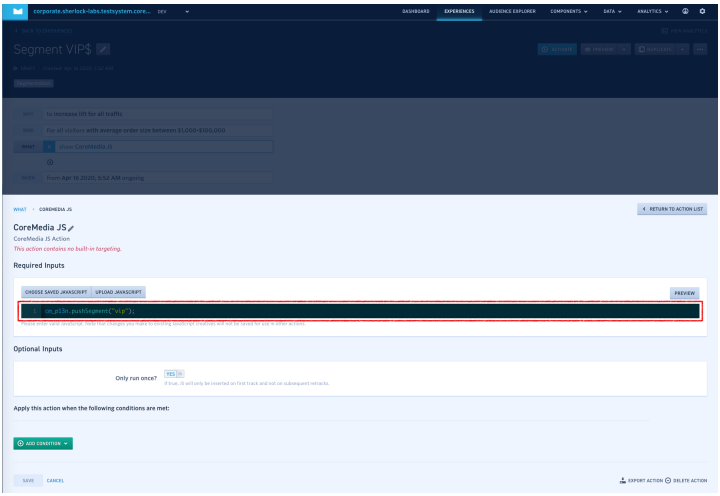

Figure 4.9. Add code to variants

3. Continue with experiences for other segments.

AI Optimized Content Selection

1. Create a single experience with goal *To optimize my goal with Machine Learning. Individual Fit*. For each segment add a variant.

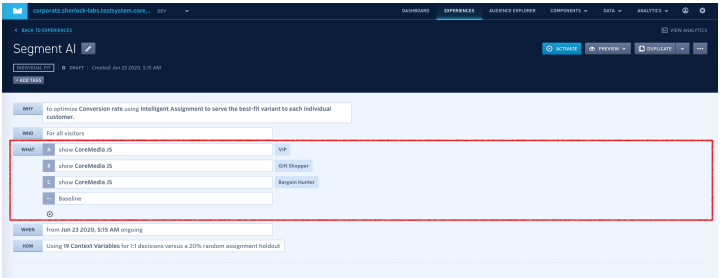

Figure 4.10. Add an experience for AI powered segmentation

For each variant add a single CoreMedia action and paste the following code into the JavaScript code field:

```
cm_p13n.pushSegment("<segment_name>");
```

Replace **<segment_name>** with the name of the corresponding Studio content item, which you have created in Section 4.2.2, "Creating Segments in Studio" [67].

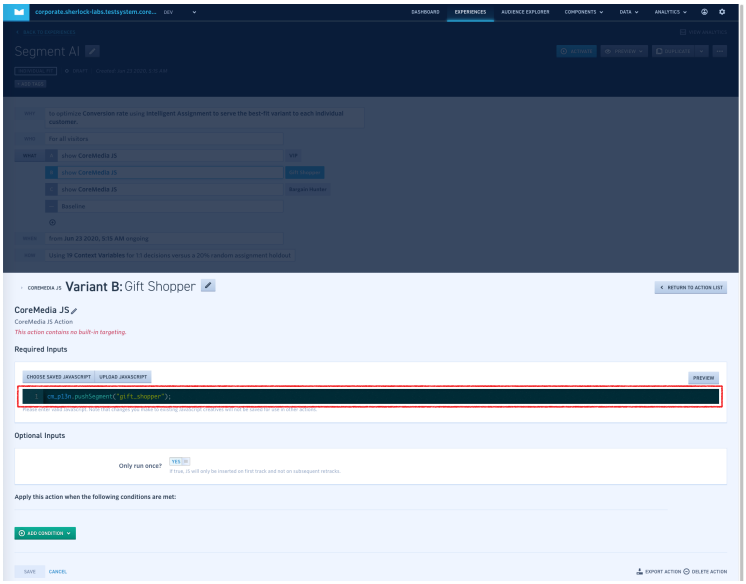

Figure 4.11. Experience for AI powered content selection

Creating Experience for Content Masking

The CoreMedia JavaScript frontend integration code - triggered by the Monetate Tag - dynamically loads content from the CAE backend via AJAX and injects it into the already displayed page. To minimize visually disturbing effects, such as elements flickering, popping up or moving around, during these page updates it is essential for the CoreMedia code to get notified when Monetate has finished processing all decisions. Hence, a special experience needs to be added, which must always be executed last by placing it at the end of the list of experiences:

1. Create a new experience, for example named *Final*.
2. Select *to serve everyone the same experience* as the goal. Don't add any targeting, let it apply to all visitors.

3. Add the CoreMedia JavaScript action to its single variant and insert the following code: `cm_p13n.completed('monetate');`.
4. Save and return to the experience overview.
5. Place the experience at the bottom of the list by giving it the highest priority value.

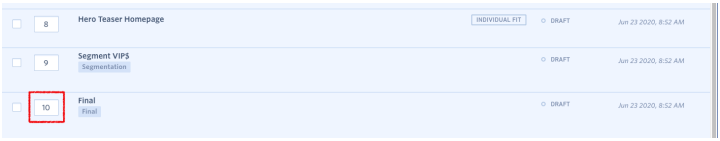

Figure 4.12. Place Final experience

4.2.4 Configuring the p13n-adapter-generic Extension

The generic adapter allows you to connect with Evergage and Dynamic Yield.

Configuring the p13n-adapter-generic requires the following steps:

- Enabling Dynamic Yield or Evergage Connection in Studio (see Section 4.2.4.1, “Connecting Evergage and Dynamic Yield with Studio” [77].
- Creating CMExperienceDefinition content items in Studio to mirror the experiences defined in Evergage and Dynamic Yield. See Section 4.2.4.2, “Creating Experience Definitions in Studio” [79].
- Creating experiences in Evergage and Dynamic Yield. See Section “Creating Experiences for Testing or Personalization” [81] and Section 4.2.4.4, “Creating Experiences for Dynamic Yield” [85].

4.2.4.1 Connecting Evergage and Dynamic Yield with Studio

For both integrations, you have to create a `Settings` content item in your site and link it from the root page of the site. The `Settings` items need to have the following properties:

Evergage Settings Content Item

The properties need to have the following values:

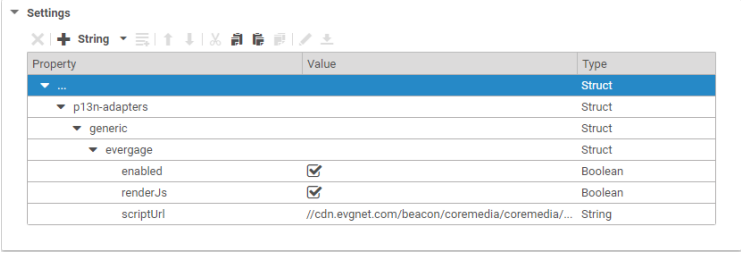

Figure 4.13. Evergage settings item

enabled Required. Enables the Evergage integration for this site.

renderJS Optional. Instructs the CAE to include the Evergage script link in its head section. Disable in a commerce-led scenario where the shop frontend already includes the script. Defaults to true if missing.

scriptUrl Required. The URL of the Evergage script. Can be obtained from the JavaScript integration page in the Evergage portal (**Web** → JavaScript Integration → **Synchronous**).

Dynamic Yield Settings Content Item

The properties need to have the following values:

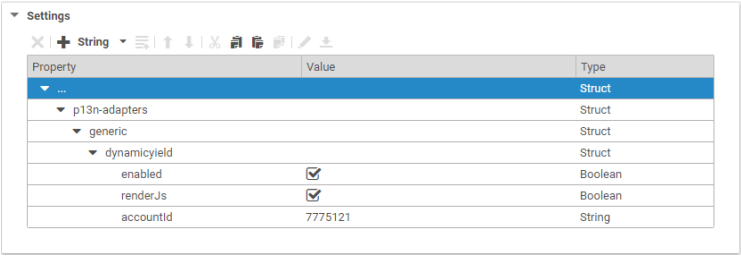

Figure 4.14. Dynamic Yield settings item

enabled Required. Enables the Dynamic Yield integration for this site.

renderJS Optional. Instructs the CAE to include the Dynamic Yield script in its head section. Disable in a commerce-led scenario where the shop frontend already includes the scripts. Defaults to true if missing.

scriptUrl Required. The account ID as displayed on the general settings page in the Dynamic Yield portal.

4.2.4.2 Creating Experience Definitions in Studio

In order to integrate personalization providers into Studio you need to mirror the experiences in Studio with special configuration content item of type `CMExperienceDefinition`.

1. For each site and experience, create `CMExperienceDefinition` content items in the folder `<Site Root>/Options/Personalization/Experiences/<ProviderID>`. Replace *<ProviderId>* with the key of the provider configuration in the `Settings` content item (see Section 4.2.4.1, “Connecting Evergage and Dynamic Yield with Studio” [77]). By default, this is "evergage" and "dynamicyield", respectively.

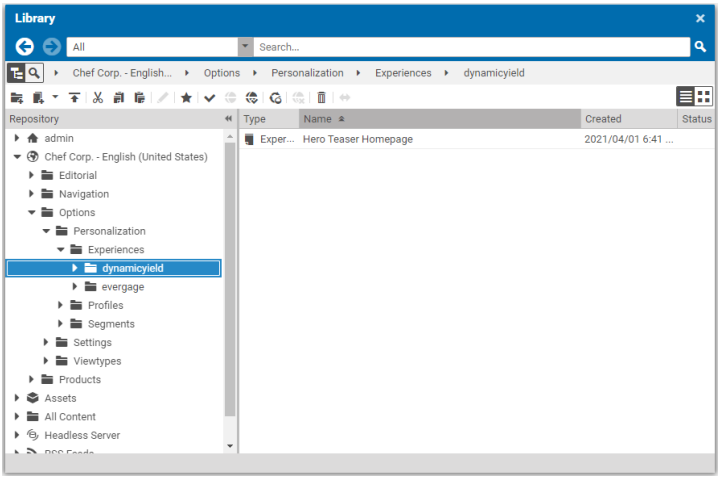

Figure 4.15. Create CMExperienceDefinitions in Studio

The name of the content item is not important, but it is recommended to use the name of the experience, defined in the personalization provider software.

2. In the content item, set a unique name and description and add all variants defined in the third-party system. The concrete name is not important, but it is recommended to use the name of the experience, defined in the personalization provider software.

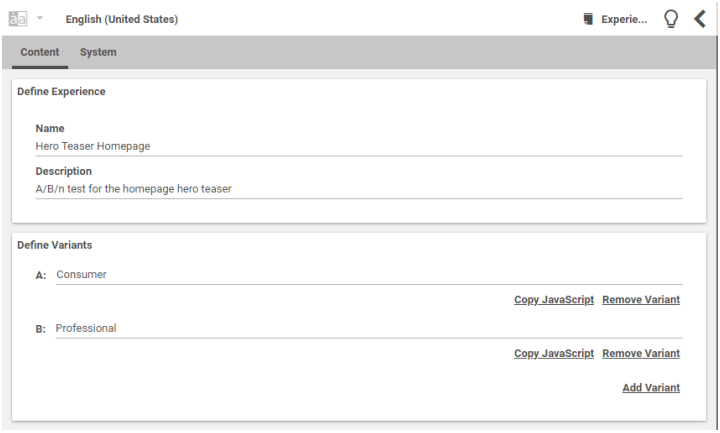

Figure 4.16. Configure experience definition in Studio

Once the definition has been created, it can be selected in experience content items.

4.2.4.3 Creating Experiences for Evergage

Evergage uses different naming conventions than the CoreMedia personalization extension and the other supported personalization providers. The following table shows the mapping of Evergage terms to CoreMedia terms:

Evergage	CoreMedia
Campaign	Experience
Experience	Variant

Table 4.2. Evergage naming

Prerequisites

- Installation of the p13n-core extension and the p13n-adapter-generic extension as described in Section 4.1, “Installing Client-Side Personalization” [62]
- Configuration of the p13n-core Extension as described in Section 4.2.1, “Configuring the p13n-core Extension” [65].

- If you want to use segmentation, creation of segment content items in Studio as described in Section 4.2.2, “Creating Segments in Studio” [67].
- Set up of the generic adapter as described in Section 4.2.4.1, “Connecting Evergage and Dynamic Yield with Studio” [77]

Creating Experiences for Testing or Personalization

1. Create a new web campaign in the Evergage portal. Add the required number of experiences (one for each variation) and switch to the setup menu.
2. Set the global campaign settings like user targeting, goal and metric.
3. Switch to the experiences panel. Set the test mode and traffic allocation, name the experiences, and set each experience to type *Personalize*.

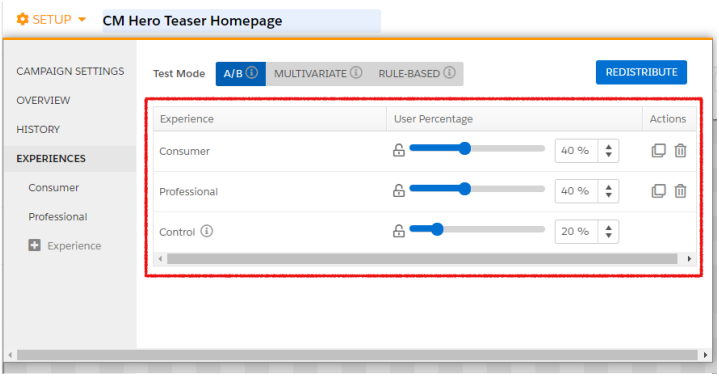

Figure 4.17. Evergage Experiences panel

4. Add the JavaScript snippets to connect the variations to CoreMedia. For this open the mirrored experience definition (see Section 4.2.4.2, “Creating Experience Definitions in Studio” [79]) in CoreMedia Studio. For each variation copy the Javascript snippet to the clipboard, change back to the Experience JavaScript tab and paste the JavaScript.

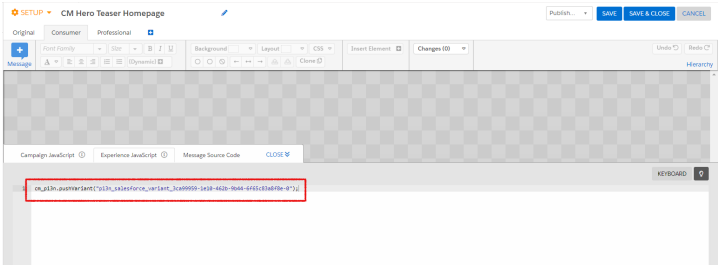

Figure 4.18. Adding JavaScript code to the variants

Creating Experiences for Segmentation

The segmentation use case allocates the customer base to a set of segments.

Prerequisites

You have already defined P13NSegment files in Studio as described in Section 4.2.2, “Creating Segments in Studio” [67].

1. Create a new web campaign and add an experience for each segment.
2. Open the setup menu and switch to the experiences panel.
3. Set the test mode to *Rule-Based* and set traffic allocation for *Control* to 0%. Afterwards edit each experience: Set its name and add a targeting rule with the fitting segment.

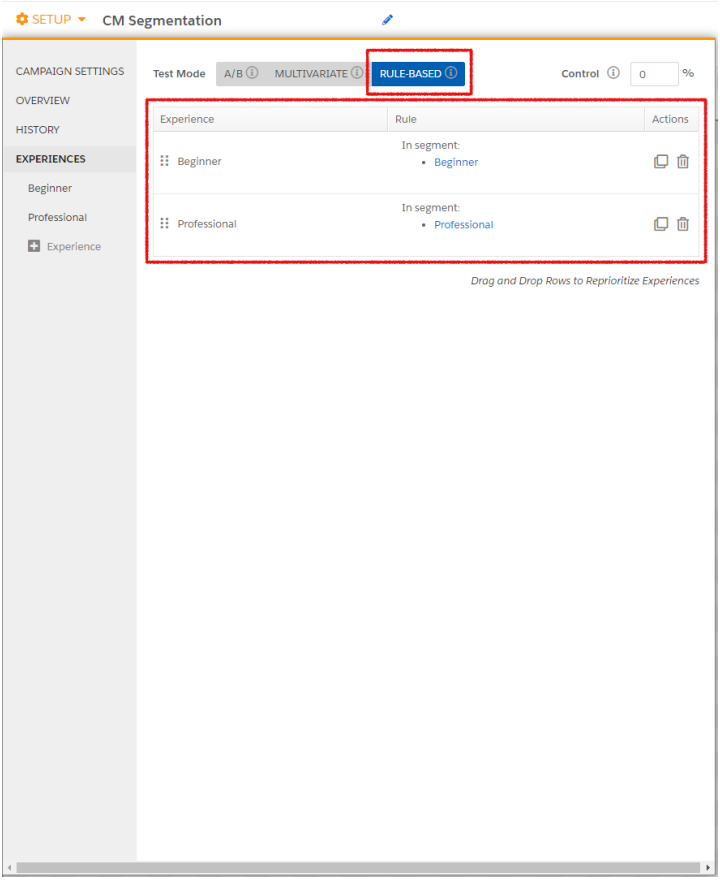

Figure 4.19. Defining Segment experiences in Evergage

4. Add the JavaScript code for connecting the segments to CoreMedia. Add the following code for each experience, substituting the parameter `segment_name` with the name defined in CoreMedia (see Section 4.2.2, “Creating Segments in Studio” [67]).

```
cm_p13n.pushSegment("<segment_name>");
```

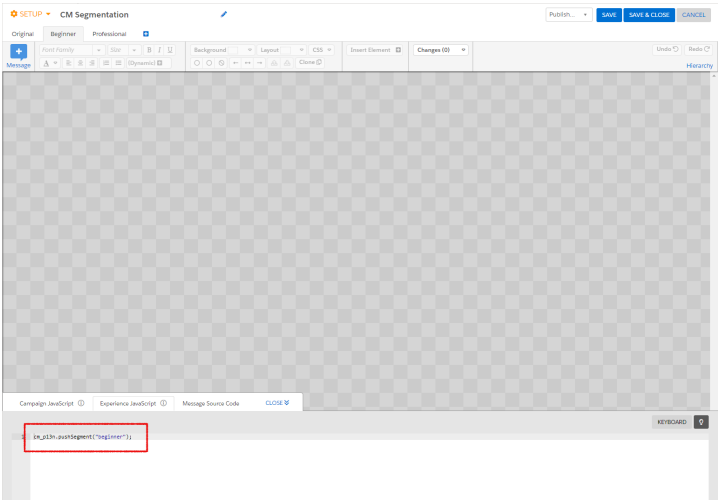

Figure 4.20. Evergage add JavaScript to experience

Creating Experience for Content Masking

The CoreMedia JavaScript frontend integration code - triggered by the Evergage JavaScript - dynamically loads content via AJAX from the CAE backend and injects it into the already displayed page. To minimize visually disturbing effects, such as elements flickering, popping up or moving around, during these page updates it is essential for the CoreMedia code to get notified when Evergage has finished processing all decisions. Therefore a special campaign must be added, which must always be executed last by giving it the lowest priority:

1. Create a new campaign, named CM Final.
2. Set its priority to a lower value than all the other CoreMedia campaigns.
3. Set its *Test Mode* to **A/B**.
4. Add a single experience named Final and allocate 100% traffic to it.
5. Set the experience's JavaScript code to: `cm_p13n.completed('evergage');`.

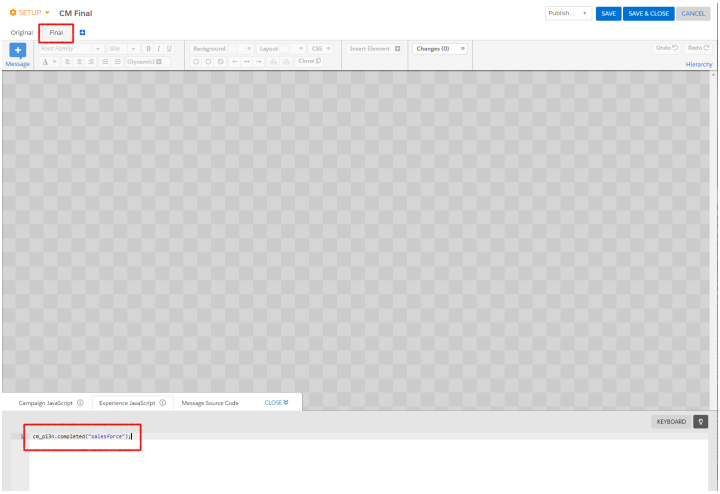

Figure 4.21. Create final experience for Evergage

NOTE

Do not set any targeting or other rules, the experience must run on every page load.

4.2.4.4 Creating Experiences for Dynamic Yield

Creating Experiences for Testing or Personalization

1. In the Dynamic Yield portal create a new Custom Code campaign using the following settings:
   - *Trigger*: Page Load
   - *Frequency*: Once per pageview
2. Add a single experience.

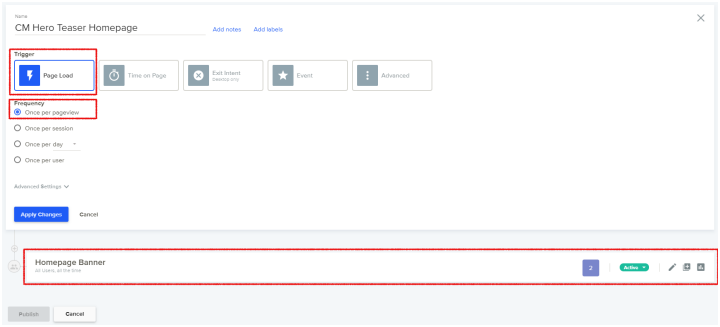

Figure 4.22. Create campaign in Dynamic Yield

3. Open the experience for editing and add the desired targeting. Switch to the *Variations* tab. Choose static or dynamic traffic allocation and select a primary metric. Then add the variations and choose their individual traffic allocation.

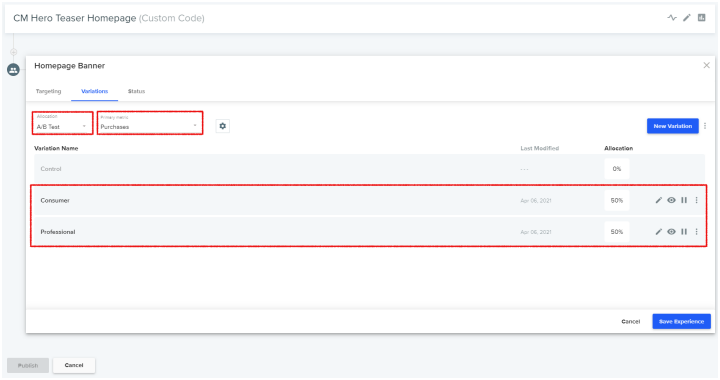

Figure 4.23. Edit Dynamic Yield experience

4. Add the JavaScript code to connect the variations to *CoreMedia Content Cloud*. Open the mirrored experience definitions in CoreMedia Studio (see Section 4.2.4.2, "Creating Experience Definitions in Studio" [79]). For each variation copy the Javascript snippet to the clipboard, change back to the variation's JavaScript tab and paste the code. Save the variation.

Creating Experiences for Segmentation

The segmentation use case allocates the customer base to a set of segments.

1. In the Dynamic Yield portal create a new *Custom Code* campaign using the following settings:
   - *Trigger*: Page Load
   - *Frequency*: Once per pageview
2. For each segment add a new experience. If a user can be a member of multiple segments, be sure to prioritize the segments by ordering them accordingly.

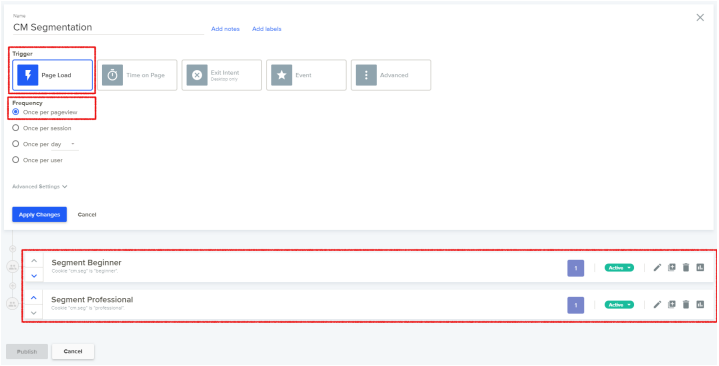

Figure 4.24. Creating campaign for Dynamic Yield segmentation

3. Set up the targeting for each experience and set the traffic *Allocation* to **`A/B Test`**. Add a single variation with 100% traffic *Allocation*.

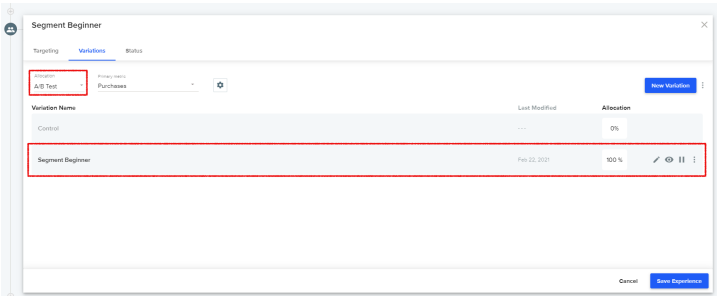

Figure 4.25. Dynamic Yield configure experience for segmentation

4. Add the JavaScript code for connecting the segment to CoreMedia. Add the following code, substituting the parameter `segment_name` with the name defined in CoreMedia (see Section 4.2.2, “Creating Segments in Studio” [67]).

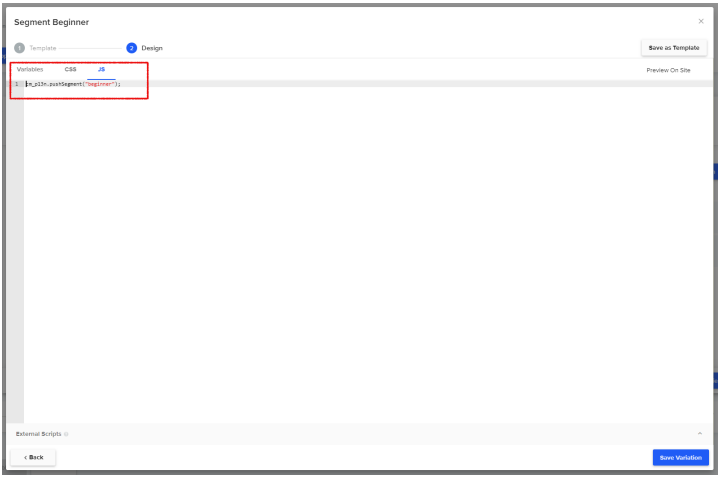

Figure 4.26. Dynamic Yield add JavaScript to segment

4.3 Client-Side Personalization Using Headless Server

This section is a step-by-step guide, which shows you how to extend the Spark application (see https://github.com/CoreMedia/coremedia-headless-client-react) to implement the CoreMedia Client-side Personalization.

4.3.1 Prerequisites

- A CoreMedia Content Cloud system with Client-side Personalization enabled, which is connected to the Monetate system as described in Section 4.1, "Installing Client-Side Personalization" [62] and Section 4.2.3, "Configuring the p13n-monetate-adapter Extension" [68].
- Spark app connected to CoreMedia Content Cloud as described in the quickstart in https://github.com/CoreMedia/coremedia-headless-client-react#readme

4.3.2 Download the Personalization Headless Schema

In the Spark project directory `apps/spark` run the following command:

```
pnpm download
```

Spark downloads all required personalization related resources like schema, metadata and type definitions and saves them to the `packages/graphql-layer/src/__downloaded__` folder.

4.3.3 Extend GraphQL Queries by P13N Fragments

1. Create a fragment file `P13NExperience.fragment.graphql` under `packages/graphql-layer/src/queries/fragments`

```
fragment P13NExperience on P13NExperience {
  baseline {
    ...CMTeasable
```

```
  }
  variants {
    id
    target {
      ...CMTeasable
    }
  }
}
```

2. To enable the fragment preview, modify the file `packages/graphql-layer/src/queries/FragmentPreview.query.graphql`, to include the p13n experience fragment:

```
query FragmentPreview($contentId: String!) {
  content {
    content(id: $contentId) {
      // other fragments here
      ...P13NExperience
    }
  }
}
```

3. Modify in the same way other fragments where the p13n experience can appear. For example, modify `PageGridPlacement.fragment.graphql` so that a p13n experience on a page grid is personalized.

4.3.4 Integrate p13n Experience into the Preview Context

1. Incorporate the p13n experience as part of the preview context. To that end, expand first the interface `PreviewContext` in `PreviewContextProvider.tsx`:

```
interface PreviewContext {
  previewDate?: string;
  previewP13NExperiences?: PreviewP13NExperiences;
  setExperience?: Function;
}

export interface PreviewP13NExperiences {
  variants?: string[];
}
```

2. Expand `PreviewContextProvider` to provide the experience context. Notice that the experience context is used as state. Later you will use the `setExperience` function to change the variant chosen by the experience or segment, which in turn will trigger a new rendering of the experience or segment content items.

```
interface Props {
  previewDate?: string;
  previewP13NExperiences?: PreviewP13NExperiences;
}
```

```
export const PreviewContextProvider: React.FC<Props> = ({ children,
previewDate, previewP13NExperiences }) => {
  const [exp, setExp] = useState(previewP13NExperiences);
  const previewContextValue: PreviewContext = {
    previewDate: previewDate,
    previewP13NExperiences: exp,
    setExperience: setExp,
  };
  return <previewDataContext.Provider
value={previewContextValue}>{children}</previewDataContext.Provider>;
};
```

4.3.5 P13N Variant Rendering of Experiences and Segments

1. Create a new file `P13N.ts` under `models/Banner`. It contains a single function which returns the p13n variant that depends on the experience state.

```
export const getP13NTargets = (self: P13NExperience): Array<CmTeasable>
=> {
  const { previewP13NExperiences } = usePreviewContextState();
  const previewP13Variants = previewP13NExperiences &&
previewP13NExperiences.variants;
  // previewP13Variants can be set to 'baseline' (preview 'eye' usecase)
  const p13nVariants = self && self.variants && self.variants.slice();
  const baseline = self && (self.baseline as CmTeasable);
  if (baseline) {
    p13nVariants?.push({ id: "baseline", target: baseline });
  }
  let p13nTargets = baseline && [baseline];
  if (previewP13Variants && p13nVariants) {
    p13nTargets = p13nVariants
      .filter((p13nVariant) => previewP13Variants.indexOf(p13nVariant.id)
 >= 0)
      .map((p13nVariant) => p13nVariant.target as CmTeasable);
  }
  //again when there are no variants left fallback to baseline
  if (p13nTargets.length === 0) {
    p13nTargets = baseline && [baseline];
  }
  return p13nTargets;
};
```

2. Create a container preview of the p13n experience content item `P13NExperience.asContainerPreview.tsx` under `components/FragmentPreview/views/`:

```
const getContainer = (items: Array<Dispatchable>, rootSegment: string):
Slot => {
  return {
    items: items.map((item) => initializeBannerFor(item,
rootSegment)).filter(notEmpty),
  };
};

const P13NExperienceAsContainerPreview:
React.FC<IncludeProps<P13NExperience>> = ({ self, params }) => {
  const { rootSegment } = useSiteContextState();
```

```
  const Container = slotByName(params?.containerView as string);
  const p13NTargets = self && getP13NTargets(self);
  return <>{p13NTargets && <Container {...getContainer(p13NTargets,
rootSegment)} />}</>;
};
export default P13NExperienceAsContainerPreview;
```

This view uses the function `getP13NTargets` to render the chosen variant.

3. Modify the function `initializeBannerFor` in `models/Banner/Banner.ts`

```
...
    } else if (type && type.indexOf("P13NExperience") >= 0) {
    const p13NTargets = getP13NTargets(self as P13NExperience);
    return initializeBanner(p13NTargets.at(0) as CmTeasableFragment,
rootSegment);
    }
```

4. Add the newly created container preview to the `viewDispatcher` in `components/FragmentPreview/FragmentPreview.tsx`

```
viewDispatcher.addViewComponent(P13NExperienceAsContainerPreview,
"P13NExperience", "asContainerPreview");
```

4.3.6 P13N Studio Integration

You can preview the p13n feature of a p13n experience and segment content items in CoreMedia Studio. This section describes the implementation steps necessary to enable the preview for Spark headless client.

1. Enabling Preview in Preview Toolbar

Client-side Personaliaztion adds a new menu in the preview toolbar (see Section 7.1.3, "Previewing Client Personalization" in *Studio User Manual*). You can use it to enable a segment and a combination of experience variants for preview.

To enable this preview feature in your headless case, proceed as follows:

1. Change `Preview.ts` to use the newly introduced p13n experience context. Implement a new function which extracts the experience *`url`* parameter which is set when the user chooses an experience:

```
export const getPreviewP13NExperiences = (queryParams: string): Object |
undefined => {
  const p13nExperiences = new
URLSearchParams(queryParams).get("p13n_experiences");
  return (isPreview() && p13nExperiences && JSON.parse(p13nExperiences))
```

```
|| undefined;
};
```

2. Modify `App.tsx` so that the preview context provider is set by this function:

```
const previewP13NExperiences = getPreviewP13NExperiences(location.search);

  ...
      <PreviewContextProvider previewDate={previewDate}
previewP13NExperiences={previewP13NExperiences}>
```

Now, when the editor chooses the p13n experience in the preview toolbar in Studio, the React experience state is changed and Spark renders the chosen experience variant in the preview.

2. Enabling Preview in LinkList Toolbar

Each toolbar of a variant/segment and baseline linklist in CoreMedia Studio has a preview icon as mentioned in Section 7.1.3, "Previewing Client Personalization" in *Studio User Manual*. By clicking the icon, the corresponding variant/segment will be loaded in the preview window.

To enable this preview feature in your headless case proceed as follows:

1. Implement a `previewListener` in `PreviewPage.tsx`

```
// catch the function to set the p13n context and state
  const { setExperience } = usePreviewContextState();

  // the p13 hook which is used by studio "eye".
  const previewListener = (event: MessageEvent) => {
    const data = event.data;
    if (data.type === "previewExperience") {
      const variantId = data?.body?.variantId;
      const exp: PreviewP13NExperiences = { variants: [variantId] };
      setExperience && setExperience(exp);
    }
  };
  window.addEventListener("message", previewListener, false);
```

Now, in Studio the preview of the p13n experience shows the chosen variant.

4.3.7 Integration with the P13N Service Provider

As explained in Section 4.2.1.1, "Frontend Integration" [65] the third-party service needs to call the callback functions if a variant should be displayed and replaces the default content fragment with the variant.

Therefore, you have to implement a listener like the `previewListener`, which will then change the experience state of the React app.

1. Implement a `p13nListener` in `Page.tsx`

```
// catch the function to set the p13n context and state
  const { setExperience } = usePreviewContextState();

  // the p13 hook which is then called by p13n provider
  const p13nListener = (event: MessageEvent) => {
    const data = event.data;
    if (data.type === "cm_p13n") {
      const variantId = data?.body?.variantId;
      const exp: PreviewP13NExperiences = { variants: [variantId] };
      setExperience && setExperience(exp);
    }
  };
  window.addEventListener("message", p13nListener);
```

2. Expand the header of `index.html` to implement the callback functions which uses the listener:

```
<script type="text/javascript">
    cm_p13n = {
      pushVariant: function (variantId) {
      window.postMessage({ type: "cm_p13n", body: { variantId: variantId
} });
    }
  </script>
```

5. Reference

This chapter lists condition types, content types and supplied context sources for Adaptive Personalizationn.

5.1 Condition Types

The following condition types exist in *CoreMedia Adaptive Personalization*:

Name	Description
SegmentCondition	Used for defining conditions on **customer segments**. Plugins may use the `addPath`, `removePath` and `clearPath` method to adapt the set of repository paths' that are searched for segment definitions. Supports the `addpath` plugin provided by *CoreMedia Adaptive Personalization*.
DateCondition	Used for defining conditions on **dates**, such as the current date.
StringCondition	Used for defining conditions on **string**-valued properties.
EnumCondition	Used for defining conditions on properties that can take on a **limited set of values**.
FloatCondition	Used for defining conditions on **float**-valued properties.
IntegerCondition	Used for defining conditions on **integer**-valued properties.
TimeCondition	Used for defining conditions on properties that represent **timestamps** consisting of hours, minutes, and seconds.
BooleanCondition	Used for defining conditions on **Boolean**-valued properties.
DateTimeCondition	Used for defining conditions on properties that represent a date and a timestamp, such as March 12, 2011, 15:13:02h
KeywordCondition	Used for defining conditions that test the values of keywords stored as properties. In contrast to the previous conditions, this condition isn't mapped to a property name but a property prefix. The substring following the prefix is assumed to be the keyword.
PercentageKeywordCondition	This corresponds to a `KeywordCondition` but instead of accepting arbitrary floating point values, it only accepts integers between 0 and 100, which are mapped to a floating point value between 0 and 1. This condition isn't mapped to a property name but a property prefix. The substring following the prefix is assumed to be the keyword.

Name	Description
BooleanPropertiesCondition	A condition that tests whether a Boolean property is set to true. You provide the set of available properties to choose from. This condition is not mapped to a property name but a property prefix. The substring following the prefix is assumed to be the name of the Boolean property. For example, if `propertyPrefix="flags"` and `properties="\{\[\['sports', 'Sport News'\]\]\}"`, the UI will show a property `Sport News`. If selected, the condition `flags.sports=true` will be added to the respective selection rule.

Table 5.1. Condition types

5.2 Content Types

CoreMedia Blueprint comes with content types suitable for *CoreMedia Adaptive Personalization*

5.3 Supplied Context Sources

CoreMedia Adaptive Personalization delivers APIs for different context sources (for example, CookieSource, TableSource) that enable persistence of personal data, however CoreMedia only ships example code for the CookieSource implementation in the Personalization Blueprint module. If you choose to persist personal data, you may legally be obliged to disclose this fact to your end users and to seek and document permission which may require additional custom code.

Here is a list of context sources delivered with *CoreMedia Adaptive Personalization*. Find the details about their use in the respective API documentation.

Name	Description
`CookieSource`	This source stores a context object in a cookie. The parameters of the used cookie (such as its max age) can be configured via properties of the source. The source serializes the context into a string and then base-64 encodes this string before writing it to the cookie.
`SystemDateTimeSource`	This source adds a context object containing several properties related to the system's date and time. The added context implements the `PropertyProvider` interface.
`TableStoreSource`	This source stores and retrieves contexts to and from a `TableStore` implementation. A `TableStore` can be anything capable of persisting key-value pairs, such as a relational database or a persistent hash map. `TableStoreSource` also requires a `UserIdProvider` that is expected to return a unique id for the current user. This id is used to construct the key used to store the context object.
`SegmentSource`	This source provides a context that indicates the user segments the current user is a member of. See Section 3.3.4, "Working With Customer Segments" [48] for details.
`TestContextSource`	This source reads test contexts from the CMS repository. See Section 3.3.2.4, "Working With Test Contexts" [39] for details.

Table 5.2. Supplied context sources

Glossary

Blob	Binary Large Object or short blob, a property type for binary objects, such as graphics.
CaaS	Content as a Service or short caas, a synonym for the CoreMedia Headless Server.
CAE Feeder	Content applications often require search functionality not only for single content items but for content beans. The *CAE Feeder* makes content beans searchable by sending their data to the *Search Engine*, which adds it to the index.
Content Application Engine (CAE)	The *Content Application Engine* (*CAE*) is a framework for developing content applications with *CoreMedia CMS*. While it focuses on web applications, the core frameworks remain usable in other environments such as standalone clients, portal containers or web service implementations. The CAE uses the Spring Framework for application setup and web request processing.
Content Bean	A content bean defines a business oriented access layer to the content, that is managed in *CoreMedia CMS* and third-party systems. Technically, a content bean is a Java object that encapsulates access to any content, either to CoreMedia CMS content items or to any other kind of third-party systems. Various CoreMedia components like the CAE Feeder or the data view cache are built on this layer. For these components the content beans act as a facade that hides the underlying technology.
Content Delivery Environment	The *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 Feeder	The *Content Feeder* is a separate web application that feeds content items of the CoreMedia repository into the *CoreMedia Search Engine*. Editors can use the *Search Engine* to make a full text search for these fed items.
Content item	In *CoreMedia CMS*, content is stored as self-defined content items. Content items are specified by their properties or fields. Typical content properties are, for example, title, author, image and text content.
Content Management Environment	The *Content Management Environment* is the environment for editors. The content is not visible to the end user. It may consist of the following modules: • *CoreMedia Content Management Server* • *CoreMedia Workflow Server* • *CoreMedia Importer* • *CoreMedia Site Manager* • *CoreMedia Studio* • *CoreMedia Search Engine* • *CoreMedia Adaptive Personalization* • *CoreMedia Preview CAE*
Content Management Server	Server on which the content is edited. Edited content is published to the Master Live Server.
Content Repository	*CoreMedia CMS* manages content in the Content Repository. Using the Content Server or the UAPI you can access this content. Physically, the content is stored in a relational database.
Content Server	*Content Server* is the umbrella term for all servers that directly access the CoreMedia repository: *Content Servers* are web applications running in a servlet container. • *Content Management Server* • *Master Live Server* • *Replication Live Server*
Content type	A content type describes the properties of a certain type of content. Such properties are for example title, text content, author, ...
Contributions	Contributions are tools or extensions that can be used to improve the work with *CoreMedia CMS*. They are written by CoreMedia developers - be it clients, partners or CoreMedia employees. CoreMedia contributions are hosted on Github at https://github.com/coremedia-contributions.
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.
CORBA (Common Object Request Broker Architecture)	The term *CORBA* refers to a language- and platform-independent distributed object standard which enables interoperation between heterogenous applications over

	a network. It was created and is currently controlled by the Object Management Group (OMG), a standards consortium for distributed object-oriented systems. CORBA programs communicate using the standard IIOP protocol.
CoreMedia Studio	*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.
DTD	A Document Type Definition is a formal context-free grammar for describing the structure of XML entities. The particular DTD of a given Entity can be deduced by looking at the document prolog: `<!DOCTYPE coremedia SYSTEM "http://www.coremedia.com/dtd/coremedia.dtd"` There're two ways to indicate the DTD: Either by Public or by System Identifier. The System Identifier is just that: a URL to the DTD. The Public Identifier is an SGML Legacy Concept.
Elastic Social	*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.
EXML	EXML is an XML dialect used in former CoreMedia Studio version for the declarative development of complex Ext JS components. EXML is Jangaroo 2's equivalent to Apache Flex (formerly Adobe Flex) MXML and compiles down to ActionScript. Starting with release 1701 / Jangaroo 4, standard MXML syntax is used instead of EXML.
Folder	A folder is a resource in the CoreMedia system which can contain other resources. Conceptually, a folder corresponds to a directory in a file system.
Headless Server	CoreMedia Headless Server is a CoreMedia component introduced with CoreMedia Content Cloud which allows access to CoreMedia content as JSON through a GraphQL endpoint. The generic API allows customers to use CoreMedia CMS for headless use cases, for example delivery of pure content to Native Mobile Applications, Smart-

	watches/Wearable Devices, Out-of-Home or In-Store Displays or Internet-of-Things use cases.
Home Page	The main entry point for all visitors of a site. Technically it is often referred to as root document and also serves as provider of the default layout for all subpages.
IETF BCP 47	Document series of *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.
Importer	Component of the CoreMedia system for importing external content of varying format.
IOR (Interoperable Object Reference)	A CORBA term, *Interoperable Object Reference* refers to the name with which a CORBA object can be referenced.
Jangaroo	*Jangaroo* is a JavaScript framework developed by CoreMedia that supports TypeScript (formerly MXML/ActionScript) as an input language which is compiled down to JavaScript compatible with Ext JS. You will find detailed descriptions on the Jangaroo webpage http://www.jangaroo.net. Jangaroo 4 is the ActionScript/MXML/Maven based version for CMCC 10. Since CMCC 11 (2110), Jangaroo uses TypeScript and is implemented as a *Node.js* and *npm* based set of tools.
Java Management Extensions (JMX)	The Java Management Extensions is an API for managing and monitoring applications and services in a Java environment. It is a standard, developed through the Java Community Process as JSR-3. Parts of the specification are already integrated with Java 5. JMX provides a tiered architecture with the instrumentation level, the agent level and the manager level. On the instrumentation level, MBeans are used as managed resources.
JSP	JSP (Java Server Pages) is a template technology based on Java for generating dynamic HTML pages. It consists of HTML code fragments in which Java code can be embedded.
Locale	Locale is a combination of country and language. Thus, it refers to translation as well as to localization. Locales used in translation processes are typically represented as IETF BCP 47 language tags.
Master Live Server	The *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.

MXML	MXML is an XML dialect used by Apache Flex (formerly Adobe Flex) for the declarative specification of UI components and other objects. Up to CMCC 10 (2107), CoreMedia Studio used the Open Source compiler Jangaroo 4 to translate MXML and ActionScript sources to JavaScript that is compatible with Ext JS 7. Starting with CMCC 11 (2110), a new, Node.js and npm based version of Jangaroo is used that supports standard TypeScript syntax instead of MXML/ActionScript, still compiling to Ext JS 7 JavaScript.
Personalisation	On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.
Projects	With projects you can group content and manage and edit it collaboratively, setting due dates and defining to-dos. Projects are created in the Control Room and managed in project tabs.
Property	In relation to CoreMedia, properties have two different meanings: In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content item depends on the content type. In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.
Replication Live Server	The aim of the *Replication Live Server* is to distribute load on different servers and to improve the robustness of the *Content Delivery Environment*. The *Replication Live Server* is a complete Content Server installation. Its content is an replicated image of the content of a *Master Live Server*. The *Replication Live Server* updates its database due to change events from the *Master Live Server*. You can connect an arbitrary number of *Replication Live Servers* to the *Master Live Server*.
Resource	A folder or a content item in the CoreMedia system.
ResourceURI	A ResourceUri uniquely identifies a page which has been or will be created by the *Active Delivery Server*. The ResourceUri consists of five components: Resource ID, Template ID, Version number, Property names and a number of key/value pairs as additional parameters.
Responsive Design	Responsive design is an approach to design a website that provides an optimal viewing experience on different devices, such as PC, tablet, mobile phone.
Site	A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In *CoreMedia CMS* a site especially consists of a site folder, a site indicator and a home page for a site. A typical site also has a master site it is derived from.
Site Folder	All contents of a site are bundled in one dedicated folder. The most prominent document in a site folder is the site indicator, which describes details of a site.

Site Indicator	A site indicator is the central configuration object for a site. It is an instance of a special content type, most likely `CMSite`.
Site Manager	Swing component of CoreMedia for editing content items, managing users and workflows. The Site Manager is deprecated for editorial use.
Site Manager Group	Members of a site manager group are typically responsible for one localized site. Responsible means that they take care of the contents of that site and that they accept translation tasks for that site.
Template	In CoreMedia, JSPs used for displaying content are known as Templates. OR In *Blueprint* a template is a predeveloped content structure for pages. Defined by typically an administrative user a content editor can use this template to quickly create a complete new page including, for example, navigation, predefined layout and even predefined content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes web application	The *User Changes* web application is a *Content Repository* listener, which collects all content, modified by *Studio* users. This content can then be managed in the *Control Room*, as a part of projects and workflows.
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