Platform, Services, and Utilities

Generated on: 2024-11-07 21:37:05 GMT+0000

SAP Commerce | 2205

PUBLIC

Original content: https://help.sap.com/docs/SAP_COMMERCE/d0224eca81e249cb821f2cdf45a82ace?locale=en-US&state=PRODUCTION&version=2205

Warning

This document has been generated from the SAP Help Portal and is an incomplete version of the official SAP product documentation. The information included in custom documentation may not reflect the arrangement of topics in the SAP Help Portal, and may be missing important aspects and/or correlations to other topics. For this reason, it is not for productive use.

For more information, please visit the https://help.sap.com/docs/disclaimer.

SameSite Cookie Attribute Handler

SAP Commerce includes a cookie handler that you can use to configure the SameSite attribute of the Tomcat cookie processor to handle cookies differently depending on their origin.

The cookie handler works for cookies added only through standard Servlet API. It doesn't support headers that are assembled as raw values and sent as String values.

To use this feature, make sure that any Accelerator templates that you're using are generated from SAP Commerce version 1905 and later.

Configuration

You can enable or disable the SameSite cookie handler with the `cookies.SameSite.enabled` property in the `local.properties` file. The default value of `cookies.SameSite.enabled` is `false`, but if you install the `samlsinglesignon` extension, this value changes to `true` automatically.

Attribute Properties

The handler defines the value of the SameSite cookie attribute using the following four values, listed from most to least important:

1. `cookies.<domain>.<path>.<name>.SameSite`
2. `cookies.<domain>.<path>.SameSite`
3. `cookies.<domain>.SameSite`
4. `cookies.SameSite`

The handler first looks for a property that respects all the cookie attributes (domain, path, and name). If the handler doesn't find it, it looks for a property that considers the domain and path attributes. If such a property doesn't exist, the handler considers a property that respects the domain attribute. If the handler doesn't find it, it uses a property with the global default value (`cookies.SameSite`).

The property uses the following values:

- Unset
- None
- Lax
- Strict

For definitions of these values, see SameSite cookies .

Note

If you want to set the value of `cookies.SameSite` to `None`, `Secure` must be enabled as the following:

```
cookies.SameSite=None; Secure
```

Troubleshooting

If you notice that the handler has problems with finding a valid property name for a given cookie, set the `cookies.samesite` logger level to `DEBUG` to enable debug logging for the handler.

Scripting Engine

Scripting Engine support allows you to execute logic written in scripting languages in run time without restarting the SAP Commerce Server. The scripting engine thus saves time and makes it possible to improve existing logic like cron job, task engine, and other similar tools.

Scripts may be stored in various repositories like a database (based on the `Script` model), classpath, file system, or even in a remote repository (for example Gists at GitHub) or can be executed on the fly as a snippet without being stored.

What Can I Use Scripts For?

These are just some example applications of scripts:

- Scripts as Event Listeners
- Cronjob Scripting
- Task Scripting
- The SAP Commerce processengine
- ImpEx API

Supported Languages

Scripting Engine implementation in Platform is based on the JSR-223 standard. At the moment, three languages are supported out-of-the-box:

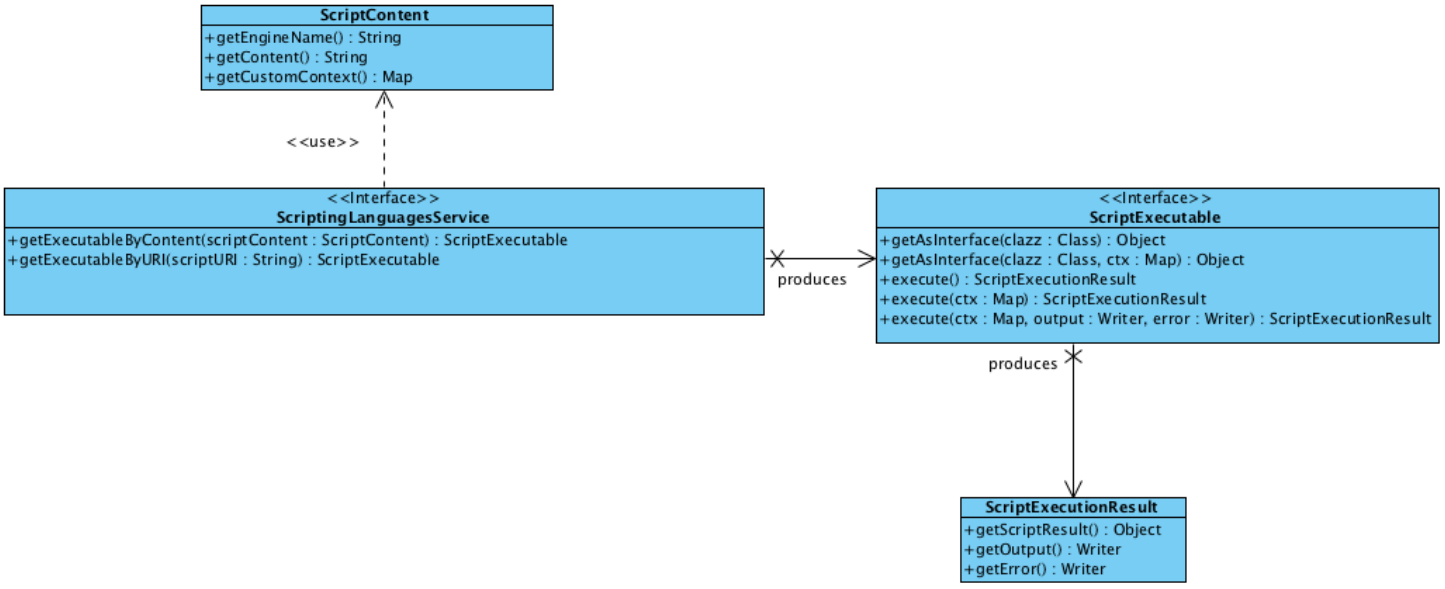
- Groovy
- BeanShell
- JavaScript

It is possible to add more languages from the JSR-223 support list, see Adding New Languages.

API Overview

From the user perspective, the API is very simple. It is based on four main interfaces:

- `ScriptingLanguagesService` provides methods for obtaining a `ScriptExecutable` object.
- `ScriptExecutable` represents an executable object, which provides methods for executing it.
- `ScriptContent` represents an object that wraps script content in a specific language type.
- `ScriptExecutionResult` represents the script execution result.



Creating Scripts

There are three ways of creating scripts:

- programmatically
- using SAP Commerce Administration Console. For details, see Administration Console
- using Backoffice. For details, see section Managing Script Versions in Backoffice.

Let's create a Groovy script, which searches for all `Media` types that have the `mime` attribute empty, and sets the value of this attribute based on their `realFilename` attribute.

```
import de.hybris.platform.servicelayer.search.FlexibleSearchQuery

flexibleSearchService = spring.getBean "flexibleSearchService"
mimeService = spring.getBean "mimeService"
modelService = spring.getBean "modelService"
def findMediasWithoutMime() {
        query = new FlexibleSearchQuery("SELECT {PK} FROM {Media} WHERE {mime} IS NULL")
```

```
        flexibleSearchService.search(query).result
}
findMediasWithoutMime().each {
        it.mime = mimeService.getMimeFromFileExtension(it.realfilename)

        modelService.save it
}
```

ℹ Note

Note how the interaction with the spring context is done. You can access any Spring bean from the SAP Commerce application context by referring to a special Spring global variable. This topic is covered later on in this document.

Executing Scripts

→ Tip

Scripts are run with the same rights that the current user has.

⚠ Caution

Scripts are not secured in any way, so you can use any language-specific structures; hence, calling `System.exit(-1)` is possible.

Now that you have this code sample, let's execute it directly by using the `ScriptingLanguagesService`. To do that, prepare a special wrapper for the real script content.

Executing Scripts from String - SimpleScriptContent

This implementation allows you to execute the script content directly as a String. Let's imagine that you have the previously mentioned script content in the String variable `content`:

```
final String content = ".... content of the script ...";
final String engineName = "groovy";

// Let's assume we have scriptingLanguagesService injected by Spring
final ScriptContent scriptContent = new SimpleScriptContent(engineName, content);
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByContent(scriptContent);

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

Executing Scripts from Resources - ResourceScriptContent

Writing scripts directly in the Java code and keeping them in variables is not the most convenient way to do it. It is much better to write them in your favorite editor with syntax highlighting. Script Engine allows you to execute scripts that are resource based, stored for instance on the classpath or directly in the file system.

Executing Scripts from the File System

Assuming that our sample script is located on the disk, with the path `/Users/zeus/scripts/setMimesForMedias.groovy`, you can execute it as follows:

```
import org.springframework.core.io.Resource;
import org.springframework.core.io.FileSystemResource;

final Resource resource = new FileSystemResource("/Users/zeus/scripts/setMimesForMedias.groovy");

// Let's assume we have scriptingLanguagesService injected by the Spring
final ScriptContent scriptContent = new ResourceScriptContent(resource);
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByContent(scriptContent);

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

Executing Scripts by Using Classpath

Or, if you have the same script but in the classpath in the folder `scripts`:

```
import org.springframework.core.io.Resource;
import org.springframework.core.io.ClassPathResource;

final Resource resource = new ClassPathResource("scripts/setMimesForMedias.groovy");

// Let's assume we have scriptingLanguagesService injected by the Spring
final ScriptContent scriptContent = new ResourceScriptContent(resource);
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByContent(scriptContent);

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

Executing Scripts Stored Remotely

You can also store the script content remotely, for instance as a gist at github.com under the URL `https://gist.githubusercontent.com/zeus/testMimesForMedias.groovy`. In this way, it is easy to get hold of and execute:

```
import org.springframework.core.io.Resource;
import org.springframework.core.io.UrlResource;

final Resource resource = new UrlResource("https://gist.githubusercontent.com/zeus/setMimesForMedias.groovy");

// Let's assume we have scriptingLanguagesService injected by the Spring
final ScriptContent scriptContent = new ResourceScriptContent(resource);
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByContent(scriptContent);

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

ℹ Note

Keep in mind that `ResourceScriptContent` determines the script engine name by the proper file extension, thus all scripts stored in files must have valid extensions for the language in which they are written.

Handling Exceptions

The scripting engine implementation uses only unchecked exceptions. This is a list of all exceptions that you can encounter:

Exception	Description
`de.hybris.platform.scripting.engine.exception.ScriptingException`	Main exception class for all scripting engine related exceptions
`de.hybris.platform.scripting.engine.exception.ScriptExecutionException`	Exception thrown when the execution of a script has failed (for instance, a script contains errors in the body of the script)
`de.hybris.platform.scripting.engine.exception.ScriptCompilationException`	Exception thrown when the compilation of a script has failed.
`de.hybris.platform.scripting.engine.exception.ScriptNotFoundException`	Exception thrown when a persisted script cannot be found in the repository.
`de.hybris.platform.scripting.engine.exception.ScriptURIException`	Exception thrown when the Script URI contains an error.
`de.hybris.platform.scripting.engine.exception.DisabledScriptException`	Exception thrown when a given `ScriptExecutable` is disabled due to previous errors.

Logging

To turn on logging for the scripting engine, set the logging level to debug by using the property `log4j.logger.de.hybris.platform.scripting.engine=DEBUG`.

Scripts Backed by Models - ModelScriptContent

SAP Commerce also comes with the model-based `ScriptContent` that is backed by the `Script` type. This special type is a container for storing scripts in the database. Let's now create a `Script` instance that keeps the media maintenance script:

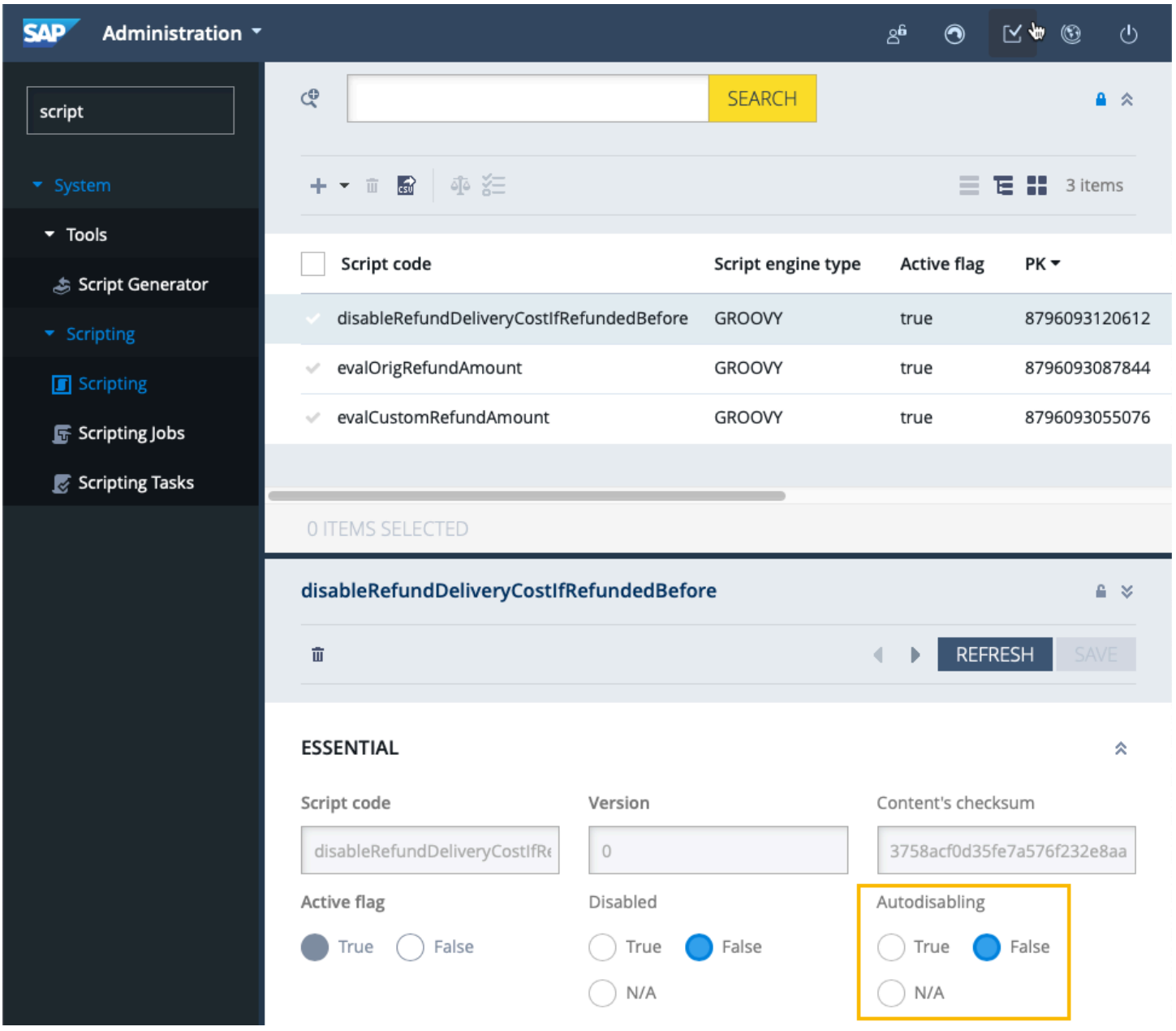
```
import de.hybris.platform.scripting.enums.ScriptType;

// Let's assume we have modelService injected by the Spring
final ScriptModel script = modelService.create(ScriptModel.class);
```

```
script.setScriptType(ScriptType.GROOVY);
script.setContent(".... content of the script ...");
// code must be unique
script.setCode("setMimesForMedias");
modelService.save(script);

// now having our model we can wrap it using ModelScriptContent
final ScriptContent scriptContent = new ModelScriptContent(script);
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByContent(scriptContent);

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

Autodisabling of Model-Based Scripts

Model-based scripts can be autodisabled. This comes in handy if a script throws an execution exception. In that case, the status of an autodisabling script changes to **disabled.** Otherwise, a faulty script would endlessly throw execution exceptions.

`ScriptModel` has two boolean properties: `autodisabling` and `disabled`. Both are by default set to `false`. To enable the auto-disabling feature, set `ScriptModel#autodisabling` item property to `true`. You can to set `Autodisabling` to `true` in Backoffice.



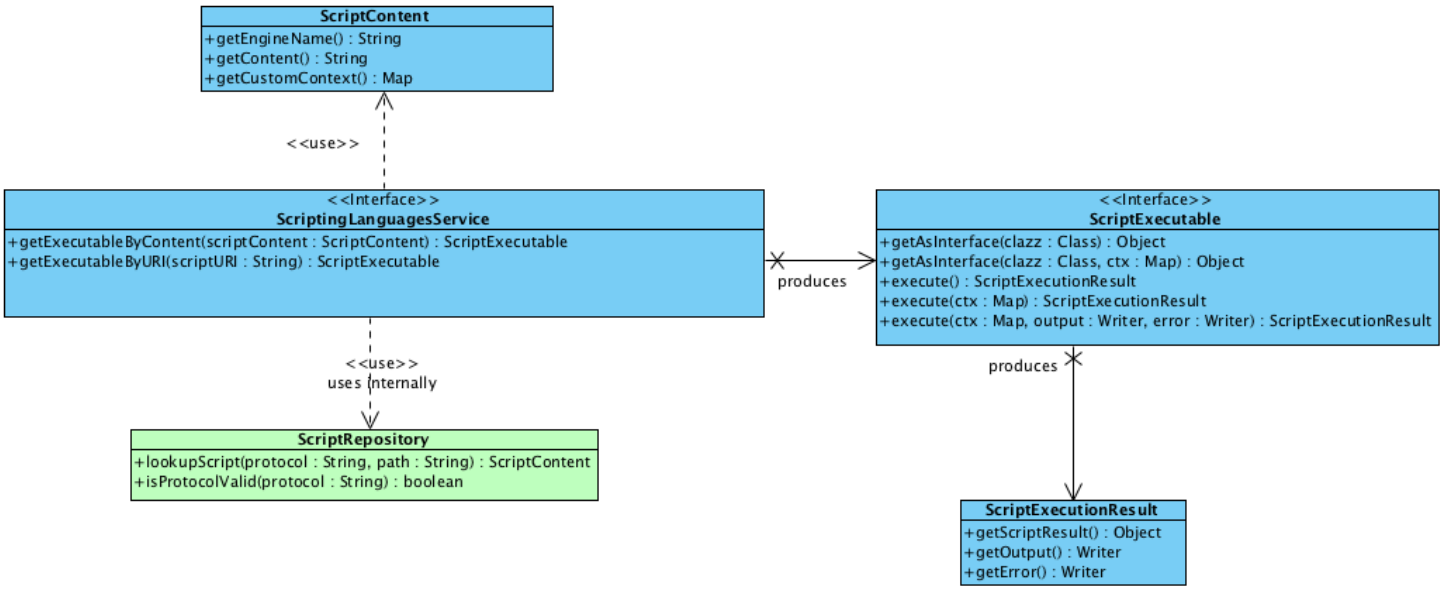
Scripts are marked as disabled on first throw of `de.hybris.platform.scripting.engine.exception.ScriptExecutionException`. Each next call to `ScriptExecutable#execute` on a script which was already marked as disabled throws a `de.hybris.platform.scripting.engine.exception.DisabledScriptException`

If you want to re-enable the script for execution, go to Backoffice, find the script and change the `disabled` flag to `false`.

Getting Script Executables by scriptURI

Playing with the ScriptContent interface for something persisted and whose exact address you know is not very handy. You always need to create a proper `ScriptContent` object. It is better to use the address of the script and get hold of the executable directly. Hence the concept of Script Repositories. A script repository allows you to look for a script in a particular storage - it may be a database, a local disk, or some remote storage. SAP Commerce comes with four implementations that are used internally by the `ScriptingLanguagesService` to resolve script addresses transparently. These are called `scriptURIs` and prepare appropriate `ScriptContent` objects behind the scenes to return a `ScriptExecutable` to the user.

- `ModelScriptsRepository`
- `ClasspathScriptsRepository`
- `FileSystemScriptsRepository`
- `RemoteScriptsRepository`



ModelScriptsRepository

This repository looks for scripts in the database. It supports the following type of `scriptURI`:

```
model://uniqueCodeOfScript
```

To get hold of a script using this repository, the user needs to use the `ScriptingLanguagesService` as follows:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("model://setMimesForMedias");

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

ClasspathScriptsRepository

This repository looks for scripts in the database. It supports the following type of `scriptURI`

```
classpath://path/to/uniqueCodeOfScript.groovy
```

To get hold of a script using this repository, the user needs to use `ScriptingLanguagesService` as follows:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("classpath://scripts/setMimesForMedias.gr

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

ℹ Note

Keep in mind that scripts in this repository must contain a valid file extension according to the language they are written in.

FileSystemScriptsRepository

This repository looks for scripts in the database. It supports the following type of `scriptURI`

```
file:///absolute/path/to/uniqueCodeOfScript.groovy
file://c:/absolute/path/to/uniqueCodeOfScript.groovy
```

To get hold of a script using this repository, use the `ScriptingLanguagesService` as follows:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("file:///Users/zeus/scripts/setMimesForMe

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

ℹ Note

Scripts in this repository must contain a valid file extension according to the language in which they are written.

RemoteScriptsRepository

This repository looks for scripts in the database. It supports the following type of `scriptURI`

```
http:///server.com/path/to/uniqueCodeOfScript.groovy
https:///server.com/path/to/uniqueCodeOfScript.groovy
ftp:///server.com/path/to/uniqueCodeOfScript.groovy
```

To get hold of a script using this repository, use the `ScriptingLanguagesService` as follows:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("http:///server.com/scripts/setMimesForMe

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

ℹ Note

Scripts in this repository must contain a valid file extension according to the language in which they are written.

Executing a Script Using Arguments - ScriptExecutable and ScriptExecutionResult

So far you have seen a simple usage of `ScriptExecutable` that shows how to execute a script without any arguments. Let's now look at a more advanced scenario, where you want to fix all mime types in medias whose `realFilename` has a specific extension:

```
import de.hybris.platform.servicelayer.search.FlexibleSearchQuery

flexibleSearchService = spring.getBean "flexibleSearchService"
mimeService = spring.getBean "mimeService"
modelService = spring.getBean "modelService"

// the query finds all medias for which mime type is null and whose realfile nasme has a specific pattern.
def findMediasWithoutMime() {
        query = new FlexibleSearchQuery("SELECT {PK} FROM {Media} WHERE {mime} IS NULL AND {realfilename} LIKE ?realfilena
        query.addQueryParameter("realfilename", realfilename);
        flexibleSearchService.search(query).result
}
findMediasWithoutMime().each {
        it.mime = mimeService.getMimeFromFileExtension(it.realfilename)

        modelService.save it
}
```

Now you can execute the script as follows (let's suppose the script is on the classpath) for all medias with a filename with the extension `xml`:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("classpath://scripts/setMimesForMedias.gr
final Map<String, Object> params = new HashMap<>();
// here we pass arguments into the hashmap.
params.put("realfilename", "%.xml");
// now we can execute script
final ScriptExecutionResult result = executable.execute(params);

// to obtain result of execution
System.out.println(result.getScriptResult());
```

The example above is written in Groovy. This means that the last line of the script always affects the result. If it returns something, you get a script result `Object`. If it does not, the call to the `ScriptExecutionResult#getScriptResult()` returns `null`. A script may also yield some output. Let's modify the script a little bit to print at the end the number of fixed mime types in Medias:

```
import de.hybris.platform.servicelayer.search.FlexibleSearchQuery

flexibleSearchService = spring.getBean "flexibleSearchService"
mimeService = spring.getBean "mimeService"
modelService = spring.getBean "modelService"

def findMediasWithoutMime() {
        query = new FlexibleSearchQuery("SELECT {PK} FROM {Media} WHERE {mime} IS NULL AND {realfilename} LIKE ?realfilena
        query.addQueryParameter("realfilename", realfilename);
        flexibleSearchService.search(query).result
}

def mediaHit = 0
findMediasWithoutMime().each {
        it.mime = mimeService.getMimeFromFileExtension(it.realfilename)

        modelService.save it
        mediaHit++
}

println "Num of fixed mimetypes for Medias - ${mediaHit}"
```

Now the last line executes the `println` function, which means that it does not return anything but prints something to the standard output. Now you can read the message.

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("classpath://scripts/setMimesForMedias.gr
final Map<String, Object> params = new HashMap<>();
params.put("realfilename", "%.xml");
// now we can execute script
final ScriptExecutionResult result = executable.execute(params);

// to obtain result of execution
System.out.println(result.getOutputWriter());// The message is: Num of fixed mimetypes for Medias - 5
```

ℹ Note

Note that `ScriptExecutionResult` contains two methods for getting output and error writers - `getOutputWriter()` and `getErrorWriter()`. When calling methods on `ScriptExecutable` both, by default, return standard `StringWriter` objects. If you want to pass your own `Writer` implementation, you may call the method `ScriptExecutable#execute(Map<String, Object> context, Writer outputWriter, Writer errorWriter)`

Using Returned Objects as Interfaces

It is also possible to get an object of a class from a script and call methods on it directly in the Java code. Let's rewrite our Medias-related script to something more object oriented:

```
class GroovyMimeFixer implements MimeFixer {
        final static FIND_ALL_QUERY = "SELECT {PK} FROM {Media} WHERE {mime} IS NOT NULL"
        final static FIND_FOR_EXT_QUERY = FIND_ALL_QUERY + " AND {realfilename} LIKE ?realfilename"

        def flexibleSearchService
        def mimeService
        def modelService
        def int fixAllMimes() {
                def query = new FlexibleSearchQuery(FIND_ALL_QUERY)
                def counter = 0
                flexibleSearchService.search(query).result.each {
                        it.mime = mimeService.getMimeFromFileExtension(it.realfilename)
                        modelService.save it
                        counter++
                }
                counter
        }

        def int fixMimesForExtension(String extension) {
                def query = new FlexibleSearchQuery(FIND_FOR_EXT_QUERY)
                query.addQueryParameter("realfilename", "%.${extension}");
                def counter = 0
```

```
                flexibleSearchService.search(FIND_FOR_EXT_QUERY).result.each {
                        it.mime = mimeService.getMimeFromFileExtension(it.realfilename)
                        modelService.save it
                        counter++
                }
                counter
        }
}

flexibleSearchService = spring.getBean "flexibleSearchService"
mimeService = spring.getBean "mimeService"
modelService = spring.getBean "modelService"

new GroovyMimeFixer(flexibleSearchService: flexibleSearchService, mimeService: mimeService, modelService: modelService)
```

i Note

Note that an instance of the class is returned in the last line of the script.

The script above defines a new Groovy class `GroovyMimeFixer` that implements the Java interface `MimeFixer`, which you have in the code base and which looks as follows:

```
public interface MimeFixer
{
        /**
         * Fix all empty mimes.
         *
         * @return num of fixed mimes
         */
        int fixAllMimes();

        /**
         * Fix mimes for particular file extension.
         *
         * @param extension
         * @return num of fixed mimes
         */
        int fixMimesForExtension(String extension);
}
```

Now you can execute this script a bit differently to obtain the object returned by the script:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("classpath://scripts/setMimesForMedias.gr

final MimeFixer mimeFixer = executable.getAsInterface(MimeFixer.class);

mimeFixer.fixMimesForExtension("xml");
mimeFixer.fixAllMimes();
```

Accessing Spring Beans in Groovy

In all of the examples, the Spring application context was referred to using the global variable `spring`. This variable is available only at the top level of the script, thus you need to pass it into the classes and other structures that have a limited scope. You can also access spring beans directly by their bean names but remember that top level script scope also applies here. So the following example works:

```
import de.hybris.platform.scripting.model.ScriptModel

modelService.create(ScriptModel.class)
```

But this one does not:

```
import de.hybris.platform.scripting.model.ScriptModel

class ScriptModelCreator {
        def create() {
                modelService.create(ScriptModel.class)
        }
}

new ScriptModelCreator().create()
```

You must pass `ModelService` into the `ScriptModelCreator` class:

```
import de.hybris.platform.scripting.model.ScriptModel

class ScriptModelCreator {
        def modelService
```

```
        def create() {
                modelService.create(ScriptModel.class)
        }
}

new ScriptModelCreator(modelService: modelService).create()
```

Accessing Beans in Javascript

Javascript is much less strict, so you can access Spring beans directly:

```
FlexibleSearchQuery = Packages.de.hybris.platform.servicelayer.search.FlexibleSearchQuery;

var query = FlexibleSearchQuery("SELECT {PK} FROM {Media} WHERE {mime} IS NULL")
var found = flexibleSearchService.search(query).getResult()

for each (var media in found.toArray()) {
  media.setMime(mimeService.getMimeFromFileExtension(media.getRealFileName()));
  modelService.save(media);
}
```

Note how the import of Java `FlexibleSearchQuery` class is done - via a special `Packages` object.

Adding New Languages

Adding a new language is simple, but you need to know which libraries are required in the classpath. Here you'll see how to add Ruby. JRuby implementation has built-in JSR-223 support.

First, you need to define library dependencies in appropriate `external-dependencies.xml` file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>de.hybris.platform</groupId>
    <artifactId>scripting</artifactId>
    <version>5.0.0.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <dependencies>
        <dependency>
            <groupId>org.jruby</groupId>
            <artifactId>jruby-complete</artifactId>
            <version>1.7.11</version>
        </dependency>
        </dependencies>
</project>
```

Next, declare bean of type `de.hybris.platform.scripting.engine.internal.ScriptEngineType` in the Spring context as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:c="http://www.springframework.org/schema/c"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"

        <bean id="jrubyEngine" class="de.hybris.platform.scripting.engine.internal.impl.DefaultScriptEngineType"
          c:name="ruby" c:fileExtension="rb" c:mime="text/x-ruby" />
</beans>
```

i Note

Note the `c` namespace in Spring file definition (http://www.springframework.org/schema/c) - this is a handy shortcut for passing constructor arguments to the bean.

This bean generally represents a new type of the scripting engine and keeps its name, file extension, and mime.

Finally, add a new value to the existing `ScriptType` enumtype in the SAP Commerce type system. This one is required by the `Script` type mentioned above - if you need to store your scripts into a database. The value is the name of the scripting engine.

```
<items xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="items.xsd">
<enumtypes>
        <enumtype code="ScriptType">
            <value code="GROOVY"/>
```

```
            <value code="BEANSHELL"/>
            <value code="JAVASCRIPT"/>
                        <!-- new value comes here -->
            <value code="RUBY"/>
        </enumtype>
    </enumtypes>
</items>
```

That's it - your new language is ready to use.

> ⚠ **Caution**
> Methods of interface ScriptExecutable are not synchronised, so the underlying ScriptEngine must provide thread-safety. Before adding a new script language, test that the new language behaves as expected in a multithread environment.

Script Versioning

1) Let's create a simple print script with the code `testScript` in the Scripting Language Console in SAP Commerce Administration Console.

```
println "Groovy Rocks"
```

Save the script.

2) Now modify this script such that it prints:

```
println "Groovy Wows"
```

Save the modified script.

3) Modify the script again:

```
println "Groovy Amazes"
```

4) Now go to the **Browse** tab. You can see only one instance of the `testScript` in the directory tree.

However, if you query the database using the following query, you will see three versions of the `testScript`.

```
select * from scripts where p_code = 'testScript' order by p_version;
```

You can call revision 1 of this script in the following manner:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutablebyURI("model://testScript/1");
final ScriptExecutionResult result = executable.execute();
```

If you do not specify any revision number, the latest version will be used.

Managing Script Versions in Backoffice

You can create, edit, and search for your scripts in Backoffice.

Searching for existing scripts only shows the latest script version. To see all versions, change the `Active flag` attribute to `False`.

Related Information

Scripts as Event Listeners
Cronjob Scripting
The SAP Commerce processengine
Task Scripting
scripting Extension

Cronjob Scripting

SAP Commerce includes a built-in scripting API for using dynamic typed languages in the Platform at runtime. An example of its usage is for cronjob scripting.

Benefits of Using Dynamic Scripting to Create Cronjobs

Traditionally, creating a new cronjob was time-consuming and entailed many manual steps, for example, you had to create a new java class, take care of spring bean definition, rebuild the Platform, restart the server, and so on, and so forth.

Using dynamic scripting, creating cronjobs becomes much easier and, most importantly, it can be done dynamically at runtime.

How It Works

Let's look at some basic concepts:

- `Script` - the item type where the script content is stored (a separate deployment table)
- `ScriptingJob` - a new ServicelayerJob item, which contains additionally the scriptURI (consequently, the stored script can be found at runtime from different locations (classpath, db, etc..)
- `ScriptingJobPerformable` - the spring bean assigned to every `ScriptingJob` instance; it implements the usual `perform()` method (like for any other cronjob). This is where the "scripted" cronjob logic is executed

You can execute the cronjob by java API (`CronjobService.perfom(Cronjob)`) or by defining a trigger. The execution is delegated to the `ScriptingJobPerfomable`.

Related Information

Scripting Engine
The Cronjob Service

Using Scripts Stored in Database

See the example to learn how to use a script stored in the database.

Define a Script

The scripting language used in this example is Groovy. Use the impex import console to import a script that will print out the 'hello world' text with the current date.

```
INSERT_UPDATE Script; code[unique=true];content
;myGroovyScript;println 'hello groovy! '+ new Date()
```

This stores your script in the database.

Define a ScriptingJob

Use the impex import console to import the following:

```
INSERT_UPDATE ScriptingJob; code[unique=true];scriptURI
;mydynamicJob;model://myGroovyScript
```

This stores your `ScriptingJob` (aptly named `mydynamicJob`) in the database. Regarding the scriptURI, if you want to point to a script stored in database, you can use the convention: `model://myscriptreference`

Define a CronJob

Use the impex import console to import the following:

```
INSERT_UPDATE CronJob; code[unique=true];job(code);singleExecutable;sessionLanguage(isocode)
;mydynamicCronJob;mydynamicJob;true;en
```

This stores the new cronjob holding the reference to our previously defined job.

Execute the Cronjob

Go to the scripting languages console in SAP Commerce Administration Console, make sure that you have the groovy language selected, enable the mode, and`commit` try the following:

```
def dynamicCJ = cronJobService.getCronJob("mydynamicCronJob")
cronJobService.performCronJob(dynamicCJ,true)
```

This executes your cronjob - just look at the console logs, where you should see the result - `"hello groovy!"`, followed by the current date.

Optional Step - Use a Trigger

Use the impex import console to import the following:

```
INSERT_UPDATE Trigger;cronjob(code)[unique=true];cronExpression
;mydynamicCronJob; 0 * 16 * * ?
```

This executes the cronjob every minute between 16:00 and 17:00. Examine the logs to verify.

Cronjob Scripts Advanced Features

See some advanced features of cronjob scripts.

The previous example showed a simple one-line script that only printed out a message. However, it is possible to do much more, for example, you can influence the result, pass information to logs, and access the cronjob that is being executed.

Returning Cronjob Results

i Note

Returning a script result is optional.

`ScriptingJobPerformable` handles returning the `PerformResult()`, holding the status of the cronjob, which is the final result of the cronjob perform logic.

For example, returning a `String` 'My script is almost finished', will only print the message out, but the final `PerformResult` is set "outside".

The following example shows how to influence the `PerformResult`.

```
import de.hybris.platform.servicelayer.cronjob.PerformResult
import de.hybris.platform.cronjob.enums.CronJobStatus
import de.hybris.platform.cronjob.enums.CronJobResult
println 'hello groovy! '+ new Date()
new PerformResult(CronJobResult.UNKNOWN, CronJobStatus.PAUSED)
```

This sets the Cronjob status to `Paused`. Of course, it is possible to set any kind of `PerformResult`, for example depending on some conditions set earlier.

Cronjob Item as Context Parameter

It is possible to access the currently executed cronjob (as a `CronJobModel` instance). It is passed as a context parameter (`key="cronjob"`).

```
println 'hello groovy! '+ new Date()
println cronjob.code
println cronjob.status

//etc...
```

Logging Inside Cronjob Scripts

The cronjob API offers many logging possibilities, including storing logs in the database.

Proper logger is available inside scripts as context parameter (`key="log"`). You can add your own logs:

```
println 'hello groovy! '+ new Date()
log.info('my custom info log')
log.warn('my custom warn log')
log.error('my custom error log')

//etc...
```

Use the `logtodatabase` property to create a cronjob that will use a specific log level.

Logs outside the script (`ScriptingJobPerformable`), will also be considered in such case. Might be useful, if, for example, the script lookup fails. In this case, the script internal logs will not be visible, but the error log with `ScriptNotFoundException` will be stored.

Scripts as Event Listeners

SAP Commerce provides a scripting API for dynamic typed languages such as groovy or any JSR-223 compliant script engine. One of the advantages offered by this API is the ability to create script-based event listeners.

Benefits of Using Scripts to Create Event Listeners

Using the traditional SAP Commerce event system, the user is forced to rebuild the system and restart the server, because it is necessary to extend the the `AbstractEventListener` class or, in some cases, to set up a Spring bean. For more information, see Event System.

Using dynamic scripting it is all done at runtime.

How It Works

The `Script` item contains the script content. `ScriptingEventService` is a dedicated event service that allows registering and unregistering the dynamic event listeners by `scriptURI` at runtime. The `scriptURI` points to the stored script in a given location such as the classpath or database. Finally, the `ScriptListenerWrapper` wraps the dynamic listener, which is necessary to always get the same listener instance for a given `ScriptURI`.

Registering and Unregistering Listeners

To store the script with the custom event listener as content, use the existing scripting languages console or Backoffice. Bear in mind, though, that they are not registered dynamically - you need to do it yourself. To register the event listener, use the `scriptingEventService` . Based on a given `ScriptURI,` this service can find the script and return a proper listener instance to be registered.

ℹ Note

Registering is **not persistent**. If you decide to restart the server at some point, dynamic scripts are not registered. However, the script items with the content are kept in the database. The user is responsible for registering scripting listeners again, if necessary.

Example

The following example uses a Groovy script that is stored in the database. Neither a server restart nor a Platform rebuild is necessary in any of the steps.

Prepare Your Script Content (EventListener)

Use the scripting languages console in SAP Commerce Administration Console to execute the following:

```
import de.hybris.platform.servicelayer.event.impl.AbstractEventListener
import de.hybris.platform.servicelayer.event.events.AbstractEvent
import de.hybris.platform.scripting.events.TestScriptingEvent
class MyScriptingEventListener extends AbstractEventListener<AbstractEvent>{

@Override
void onEvent(AbstractEvent event)
{
  if(event instanceof TestScriptingEvent){
    println 'hello groovy! '+ new Date();
  }
  else{
   println 'another event published '
   println event
  }
}
}
new MyScriptingEventListener();
```

Run the script to make sure that there are no syntax errors. If the script is fine, Groovy shows the `ScriptingEventListener` instance in the **Result** tab. This test script reacts to every event published in the system and prints out relevant logs. The `ScriptingEvent` is just a test event.

Store Your Script

Stay in the the scripting languages console and do not remove the previous content. Enter `myEventListenerScript` in the code input field and click **Save**. The script item gets persisted with the given code and content.

Register Your Listener

To register your listener, use the `ScriptingEventService` in the scripting languages console, as follows:

```
scriptingEventService.registerScriptingEventListener('model://myEventListenerScript')
```

The system returns `true` in the **Result** tab, which means that your listener is registered successfully. You can do an additional check by retrieving the list of the currently registered listeners:

```
eventService.eventListeners

//or getting more readable output:
//eventService.eventListeners.each{ println it}
```

Your listener is triggered by already-existing events. Examine the logs to verify that this is happening.

Create and Publish Your Custom Event

Assuming that there is a custom `ScriptingEvent` class, create an instance of that class and publish it.

```
import de.hybris.platform.scripting.events.TestScriptingEvent
event = new TestScriptingEvent('myEvent')
eventService.publishEvent(event);
```

This publishes your event and this way the scripted event listener also reacts to your custom event. As a result, you should see `hello groovy!`, followed by the current date printed in the logs.

Unregister Your Listener

```
scriptingEventService.unregisterScriptingEventListener('model://myEventListenerScript')
```

The listener is unregistered and stops reacting to the published events. You can verify it by publishing the previous event again.

Optionally Use Typed Events

Thanks to java generics, it is also possible to allow the listener to listen to only one type of events. The script can look like the following:

```
import de.hybris.platform.servicelayer.event.impl.AbstractEventListener
import de.hybris.platform.servicelayer.event.events.AbstractEvent
import de.hybris.platform.scripting.events.TestScriptingEvent
class MyScriptingEventListener extends AbstractEventListener<TestScriptingEvent>{

@Override
void onEvent(TestScriptingEvent event)
{
    println 'hello groovy! '+ new Date();
}
}
new MyScriptingEventListener();
```

This way you can easily simplify the code by avoiding the `instanceof` checks, because the listener will only react to a specific type of event. Registering and publishing events works the same as in the previous steps.

Related Information

Scripting Engine
Event System

Task Scripting

You can use the SAP Commerce scripting API to easily and quickly create dynamic tasks based on scripts.

Benefits of Using Dynamic Scripting to Create Tasks

Using the conventional approach, implementing the `taskRunner` interface and setting up the runner bean forces you to rebuild the Platform and restart the server, see The Task Service.

With dynamic scripting, there is no need to rebuild the Platform or restart the server, because we do not need to add and compile an implementation of the `taskRunner`. Using the new, all-in-runtime approach, you can go to the scripting console, create and store a script that is an implementation of the `taskRunner`, and execute the script; all this is done without ever having to rebuild the Platform.

How It Works

`Script` is the item type where the script content is stored. On the database level, it is a separate deployment table. `ScriptingTask` is the new Task item, which additionally contains the `scriptURI`. Consequently, the stored script can be found at runtime from various locations (for example from classpath, database). `ScriptingTaskRunner` is the spring bean assigned to every `ScriptingTask` instance. It implements the `run()` and `handleError()` method, like for any other task. That is the place where the "scripted" task logic is executed.

All you need to do is create your own `taskRunner` in any supported dynamic typed language. The `ScriptingTaskRunner` then takes care of finding the script and delegating the execution to the `taskRunner` specific methods that the user needs to implement, that is `run()` and `handleError()` methods.

→ Tip

You can execute a task via java API using `taskService`.

Creating and Storing Scripts as Tasks in the Database

In this example, we use the Scripting console (see Administration Console) to create a Groovy script that is an implementation of `taskRunner`. Then we execute it. Finally, we show how to handle errors in scripts.

Prepare Script Content (`TaskRunner`)

Use the scripting languages console to execute the following:

```
import de.hybris.platform.task.TaskRunner;
import de.hybris.platform.task.TaskService;
import de.hybris.platform.task.TaskModel;
class MyScriptRunner implements TaskRunner<TaskModel>{

@Override
void run(TaskService taskService, TaskModel task)
{
        println 'hello groovy! '+ new Date();
}

 @Override
void handleError(TaskService taskService, TaskModel task, Throwable error)
{
        println 'my groovy has errros! '+ new Date();
}
}
new MyScriptRunner();
```

This is the script content that is stored in the next step. Now execute the script to validate that there are no syntax errors. If the script is fine, groovy shows the `Runner` instance in the **Result** tab.

Store Your Script

Stay in the scripting languages console (don't remove the previous content), enter 'myGroovyScript' in code input field and click **save**. The script item should be persisted with given code and content.

Create and Execute Your Task

Stay in the the scripting languages console and remove the previous content (your script is already stored). Make sure you are executing the following in the commit mode:

```
import de.hybris.platform.task.model.ScriptingTaskModel;
import de.hybris.platform.task.TaskService;

task = modelService.create(ScriptingTaskModel.class);
task.scriptURI='model://myGroovyScript'

taskService.scheduleTask(task)
```

This runs your task; as a result you should see the '***hello groovy!***' text followed by the current date printed out in the logs.

ⓘ Note

The task is not persisted before scheduling by the `taskService`. It is enough to prepare it (via `modelService.create()`) and then the task engine handles the whole process - persisting the task, executing it, and removing from database. There were no errors during the execution of this task, so only the `run()` method was processed. Otherwise - it would be redirected to the `handleError()` method.

Try with Error Handling (Optional)

Prepare your script such that it throws an Exception in the `run()` method

```
import de.hybris.platform.task.TaskRunner;
import de.hybris.platform.task.TaskService;
import de.hybris.platform.task.TaskModel;
class MyScriptRunner implements TaskRunner<TaskModel>{

@Override
void run(TaskService taskService, TaskModel task)
{
        println 'hello groovy! '+ new Date();
        throw new NullPointerException("Error")
}

 @Override
void handleError(TaskService taskService, TaskModel task, Throwable error)
{
        println 'my groovy has errors! '+ new Date();
}
}
new MyScriptRunner();
```

Save the script with a new name, for example as `MyErrorGroovyScript`. Finally, execute it, pointing to the error script `'model://myGroovyScript'`

Examine the logs. A `NullPointerException` should be thrown and finally '***my groovy has errors***!' followed by the current date should be printed out. This is because the logic has been redirected to the `handleError()` method

Related Information

Scripting Engine
Administration Console

Search Mechanisms

Both the FlexibleSearchService and the GenericSearch API allow developers to construct and execute queries against the SAP Commerce database, by focusing on SAP Commerce items rather than raw SQL. However, the way in which the queries are constructed is based on two completely different and complimentary approaches.

The topics covered include:

FlexibleSearch

SAP Commerce comes with a built-in query language of an SQL-based syntax, **FlexibleSearch**. FlexibleSearch enables searching over the items in SAP Commerce.

GenericSearch

While SAP FlexibleSearch offers a powerful search API to developers, some of them may prefer the GenericSearch API that is similar to Hibernate Criteria Queries. Hibernate is a collection of related projects, enabling developers to utilize POJO-style domain models in their applications in ways extending well beyond Object and Relational Mapping.

FlexibleSearch

SAP Commerce comes with a built-in query language of an SQL-based syntax, **FlexibleSearch**. FlexibleSearch enables searching over the items in SAP Commerce.

FlexibleSearch is a powerful retrieval language built into SAP Commerce. It enables searching for SAP Commerce types and items using an SQL-based syntax. The execution of a FlexibleSearch statement takes place in two phases: pre-parsing into an SQL-compliant statement and running that statement on the database. During the pre-parsing phase, the FlexibleSearch framework resolves the FlexibleSearch syntax into SQL-compliant syntax. For example, the following two code snippets show a FlexibleSearch query and the statement that results from the FlexibleSearch query, which is executed on the database:

- FlexibleSearch query:

```
select {pk}, {code}, {name[de]} from {Product}
```

- SQL statement (executed on the database):

```
SELECT  item_t0.PK , item_t0.Code , lp_t0.p_name
        FROM products item_t0 JOIN productslp lp_t0 ON item_t0.PK = lp_t0.ITEMPK AND lp_t0.LANGPK= 9013632135395968
        WHERE (item_t0.TypePkString IN  ( 23087380955301264 , 23087380955663520 , 23087380955662768 , 23087380955661
        23087385363574432 , 23087380955568768 , 23087380955206016 ) )
```

FlexibleSearch abstracts the SAP Commerce type system from the actual database tables so that you can run searches on the type system level. Unlike on conventional SQL statements, in a FlexibleSearch query, you do not have to specify explicit database table names. The FlexibleSearch framework resolves type and database table dependencies automatically and specifies `UNIONS` and `JOINS` where necessary. The entire conversion process between type system and database representation takes place automatically. To access a type in a FlexibleSearch query, surround the type code with curly braces **{** and **}**, as in:

```
SELECT * FROM {Product}
```

ℹ Note

FlexibleSearch queries are executed on the database directly using SQL statements. By consequence, it is not possible to run FlexibleSearch queries on jalo-only attributes as these attributes are not written into the database. Find more information in the Jalo-only Attributes document.

SAP Commerce executes FlexibleSearch queries in the context of a certain user account, using a session. As different user accounts have access to different items on SAP Commerce, the number of search results depends on the user account. The number of search results is defined by type access rights (these affect the Backoffice search results only), restrictions, catalog versions, and categories, for example. The more privileged a user account is, the more search results a FlexibleSearch yields in the context of that user account. By default, the user account assigned to a session is **anonymous**, so any FlexibleSearch query returns the search results matching the **anonymous** account by default. To run FlexibleSearch queries in the context of a user account different from **anonymous**, the session needs to be assigned to a different user account, such as:

```
import de.hybris.platform.servicelayer.user.UserService;
...
// Injected by Spring
userService.setCurrentUser(userService.getUserForUID("myUserAccount"));
...
```

Related Information

The Type System

Syntax Overview

See the overview of the FlexibleSearch syntax.

The basic syntax of a FlexibleSearch query looks like this:

```
SELECT <selects> FROM <types> ( WHERE <conditions> )? ( ORDER BY <order> )?
```

A FlexibleSearch query consists of:

- The mandatory `<selects>` parameter for the `SELECT` clause.
- The mandatory `<types>` parameter for the `FROM` clause.
- An optional `<conditions>` field for the `WHERE` clause.
- An optional `ORDER BY` clause.

SQL Command / Keyword	Description / Comment	Code Example
ORDER BY {alias:attribute}	Display results ordered by the value of attribute.	`SELECT ... FROM ... ORDER BY...`
ASC	Sort results in ascending order (a...z, 0...9).	`SELECT ... FROM ... ORDER BY ... ASC`
DESC	Sort results in descending order (z...a, 9...0).	`SELECT ... FROM ... ORDER BY ... DESC`
DISTINCT	Eliminates double entries in the query result.	`SELECT DISTINCT ... FROM ...`
OR	Performs a logical `OR` compare between two queries.	`... WHERE ... OR ...`

SQL Command / Keyword	Description / Comment	Code Example
AND	Performs a logical AND compare between two queries.	`... WHERE ... AND ...`
IS [NOT] NULL	Returns the results that are [not] `null`	`... WHERE ... IS [NOT] NULL`
[NOT] IN	Returns the results that are [not] part of the following statement ... WHERE ... [NOT] IN ...	
[NOT] EXISTS	Returns the results that are [not] matching a given subquery.	`...WHERE ... EXISTS ( {{ SELECT ... }} )`
LIKE	Compares to a pattern.	`... WHERE ... LIKE '...'`
%	Wildcard matching any number of characters.	`... WHERE ... LIKE '%...'||'...%...'||'...%'`
_	Wildcard matching a single character.	`... WHERE ... LIKE '...' || '......' ||'..._'`
LEFT JOIN ON	Merges two tables into one.	`... LEFT JOIN ... ON ... = ...`
=	True if results are equal.	
!=, <>	True if results are not equal.	
<	True if result 1 is less than result 2.	
<=	True if result 1 is equal to or less than result 2.	
>	True if result 1 is greater than result 2.	
>=	True if result 1 is equal to or greater than result 2.	
CONCAT	Concatenates two results - the example on the right hand side would return the string `result`. ⚠️Note that **SQL Server 2012** doesn't provide a CONCAT function. Strings are connected via '+', for example 'foo' + 'bar' instead.	`NCONCAT('resul', 't')`

`:o`	Outer join parameter is used to include matches with missing rows in the **ProductsLP** table (=. the table that holds localized products) as well. Otherwise, the example query would only return products with an existing row in **ProductsLP** table, because it would only use JOIN.	`SELECT {p:PK}`
		`FROM {Product AS p}`
		`WHERE {p:description[en]:o} LIKE '%text%'`
		`OR {p:description[de]:o} LIKE '%text%'`

`{locAttr[ANY] }`	A special version of a localized attribute condition, where an item is returned if **any** localization record holds a value for that attribute which fulfills the given condition **regardless** of the actual language. Technically, this means that the LP table is joined without a language parameter, which means that the **same item** may occur **multiple** times in the search result! You should use DISTINCT to compensate.	`SELECT DISTINCT {p:PK}`
		`FROM {Product AS p}`
		`WHERE {p:description[ANY]} LIKE '%hybris%'`

The <selects> Field

The values for the `<selects>` field specify the database columns to be returned.

The asterisk (*) returns all database columns, as by SQL convention. To search for an attribute, specify the attribute identifier in curly braces, such as: `SELECT {code} FROM {Product}`.

To retrieve values of localized attributes, use the language identifier as a suffix in the attribute name, enclosed in squared brackets (**[** and **]**), such as:

```
SELECT {name[de]}, {name[en]} FROM {Product}
```

Find two examples of different queries:

- `SELECT * FROM {Category}`

This query returns every database column from the **Category** table.

- `SELECT {pk},{code},{name[de]} FROM {Product}`

  This query returns the database columns `pk`, `code`, and the German localized entries of the `name` column `[de]` from the **Product** table.

The <types> Field

The values for the `<types>` field in the `FROM` clause specify SAP Commerce types.

The values for the `<types>` field are nested in curly braces **{** and **}** which are to be searched, such as:

```
SELECT * FROM {Product}
SELECT * FROM {Category JOIN Catalog}
```

You can specify an alias to be used for distinguishing attribute fields, using the `AS` operator:

```
SELECT {p.code} FROM {Product AS p} ORDER BY {p.code}
```

You may also run `JOIN` and `LEFT JOIN` queries, as in:

```
SELECT {cj.code}
FROM {SyncItemCronJob AS sicj
         JOIN SyncItemJob AS sij
           ON {sicj:job} = {sij:pk}
     }

SELECT {p1.PK},{p2.PK}
FROM {Product AS p1
         LEFT JOIN Product AS p2
            ON {p1.code} = {p2.code}
     }
WHERE {p1.PK} <> {p2.PK}
ORDER BY {p1.code} ASC
```

Always remember that it is most important that the whole `<types>` block must be enclosed by **{** and **}** no matter how many types are actually inside it. Do not try to put in multiple `<types>` blocks in the `FROM` clause. Even though this may appear to be working, it may cause unpredictable errors.

Searching Subtypes

Specifying a type to search causes a FlexibleSearch query to search that type and any subtypes.

The following code snippet returns the codes and the PKs of all instances of `Product` and `VariantProduct`:

```
SELECT {code},{pk} FROM {Product}
```

By adding a trailing exclamation mark (`!`), the FlexibleSearch query searches only the specified type and omits all subtypes. For example, the following code snippet searches only instances of `Product`, not of `VariantProduct`:

```
SELECT {code},{pk} FROM {Product!}
```

When searching for subtypes, the FlexibleSearch first retrieves the subtypes to search, for example in the case of `Product`, types to search are `Product` and `VariantProduct`. As a type definition in SAP Commerce is an item and therefore has a Primary Key (PK), the FlexibleSearch retrieves the PK of all types to search. The list of the PKs of the types to search is put into an `IN` clause within the `WHERE` clause.

FlexibleSearch Query	SQL Statement

`SELECT {p:code}, {p:pk}`
`FROM {Product AS p}`
	`SELECT item_t0.Code , item_t0.PK`
	`FROM products item_t0`
	`WHERE (item_t0.TypePkString IN (`
	`23087380955301264 , 23087380955663520 , 23087380955662768 ,`
	`23087380955661760 , 23087385363574432 , 23087380955568768 ,`
	`23087380955206016 )`
	`)`

FlexibleSearch Query	SQL Statement

`SELECT {p:code}, {p:pk}`
`FROM {Product! AS p}`
	`SELECT item_t0.Code , item_t0.PK`
	`FROM products item_t0`
	`WHERE (item_t0.TypePkString = 23087380955206016 )`

i Note

In the code above, the use of ':' is the legacy alternative to '.' and is still supported for backward compatibility and a few special features.

Changing FlexibleSearch Subtype Handling Behavior

Queries to types that have subtypes with separate deployment tables are joined using the `UNION ALL` clause. Sometimes this behavior can lead to duplicated records. You can change this behavior by setting the `unionAllTypeHierarchy` property to `false` in a current session. FlexibleSearch will use the `UNION` clause instead of `UNION ALL` as a result.

```
sessionService.executeInLocalViewWithParams(ImmutableMap.of(de.hybris.platform.jalo.flexiblesearch.FlexibleSearch.UNION_AL
                new SessionExecutionBody()
                {
                    @Override
                    public List<Object> execute()
                    {
                        return r.get();
                    }
                });
```

Excluding Types from a Search

You can omit certain types from a FlexibleSearch query run.

If you want to make sure that certain types are omitted from a FlexibleSearch query run, there are two approaches at your disposal:

- Using the `itemtype` operator and a parameter.

  This approach is feasible if you can prepare and pass a `Map` with references to the types you want to exclude as a FlexibleSearch parameters, such as:

  ```
  final Set<ComposedTypeModel> excludedTypes = new HashSet<ComposedTypeModel>();
  excludedTypes.add(getComposedType("mySuborderType"));

  StringBuilder queryString = new StringBuilder("SELECT {").append(OrderModel.PK).append("} ");
  queryString.append("FROM {").append(OrderModel._TYPECODE).append("} ");
  queryString.append("WHERE {").append(OrderModel.ITEMTYPE).append("} NOT IN (?excluded)");

  final FlexibleSearchQuery query = new FlexibleSearchQuery(queryString.toString(), Collections.singletonMap("excluded
  ```

- Using a `JOIN` clause

  This approach is feasible if you cannot pass parameters, for example, because you need to enter a FlexibleSearch statement directly:

  ```
  SELECT {o.PK} FROM {Order AS o JOIN ComposedType AS t ON {o.itemtype}={t.PK} }
  WHERE {t.code} NOT IN ( 'Foo', 'Bar' )
  ```

The <conditions> Field

The values for the `<conditions>` field in the optional `WHERE` clause narrow down the number of matches by specifying at least one condition that is matched by all search results.

i Note

Make sure to avoid spaces at the beginning and end of the search condition term, as `= 'al'` and `= 'al '` are not identical search conditions and cause different search results.

Avoid Spaces in Search Condition Terms

- `SELECT * FROM {Product} WHERE {code} LIKE '%al%'`
- Using the common SQL boolean operators (`AND`, `OR`) you can connect conditions, such as:

  ```
  SELECT * FROM {Product} WHERE {code} LIKE '%al%' AND {code} LIKE '%15%'
  ```

- Use the `IS NULL` operator to find all entries that have no value:

  ```
  SELECT * FROM {Product} WHERE {code} IS NULL
  ```

- Negating a condition is possible using the SQL boolean operators `NOT`:

  ```
  SELECT * FROM {Product} WHERE {code} NOT LIKE '%al%'
  ```

- It is possible to combine negating and connecting conditions:

  ```
  SELECT * FROM {Product} WHERE {code} LIKE '%al%' AND {code} NOT LIKE '%15%'
  ```

- The negation of the `IS NULL` operator is `IS NOT NULL`:

  ```
  SELECT * FROM {Product} WHERE {code} IS NOT NULL
  ```

- The `WHERE` clause also allows subselects using double curly braces (`{{` and `}}`), such as:

  ```
  SELECT {cat:pk} FROM {Category AS cat} WHERE NOT EXISTS (
     {{ SELECT * FROM {CategoryCategoryRelation} WHERE {target}={cat:pk} }} /* Sub-select */
  )
  ```

The <order> Field

The FlexibleSearch complies with the SQL syntax in terms of ordering results. By specifying an attribute in an `ORDER BY` clause, the list of search results are sorted according to the specified type.

You can also optionally specify `ASC` to sort the search results in ascending order (null, 0 through 9, A through Z) or `DESC` to sort the search results in descending order (Z through A, 9 through 0, null). `ASC` and `DESC` are mutually exclusive, `ASC` is default. Examples:

- The following FlexibleSearch query sorts the search results by the values of the `code` database column, in descending order:

  ```
  SELECT {code},{pk} FROM  {Product} ORDER BY {code} DESC
  ```

- The following FlexibleSearch query sorts the search results by the values of the `code` database column, in ascending order: (`ASC` is default order):

  ```
  SELECT {code},{pk} FROM  {Product} ORDER BY {code}
  ```

Parameters

A FlexibleSearch query optionally contains parameters, marked by a prefixed question mark.

Parameters enable you to pass values into the FlexibleSearch query. For example, in the following code snippet, the parameter `product` can be used to pass a search pattern:

```
SELECT {p:pk} FROM {Product AS p} WHERE {p:code} LIKE ?product
```

The following FlexibleSearch query has two parameters, `startDate` and `endDate`:

```
SELECT {pk} FROM {Product} WHERE {modifiedtime} >= ?startDate AND {modifiedtime} <=?endDate
```

Parametrized Queries

Parametrized queries, also called prepared statements or parametrized statements, allow you to create templates you can then use to repetitively run SQL statements in the database management system (DBMS) of your choice. The generic nature of parametrized queries improves the performance of database operations through the use of parameters. They also reduce the risk of SQL injections, because parameter values in the query templates are automatically escaped.

To use parametrized queries, first prepare a template statement of a generic query. Leave certain values of that query unspecified as in the following plain SQL example:

```
SELECT FROM <table name> VALUES (?, ?, ?);
```

? are the values the DBMS parses, then stores the template and its result without executing it. The stored template is ready for the application to supply the values and running the query as needed. Before using the parametrized queries, examine and determine advantages of using them. There are cases in which running a single query makes for a better database performance than using a parametrized query. Such situations include running a query only once, or not enough cache space.

The following code samples are an example of how to use a parametrized query in FlexibleSearch to return the list of inactive catalogs based on their ID. First, determine the parameters:

```
SELECT {catalogVersion.pk} FROM {CatalogVersion AS catalogVersion JOIN Catalog as catalog ON {catalog.pk} = {catalog}}WHER
```

Using it in a Groovy script in the Administration Console returns the list of inactive catalogs:

```
import de.hybris.platform.catalog.model.CatalogModel
import de.hybris.platform.catalog.model.CatalogVersionModel
import de.hybris.platform.servicelayer.search.FlexibleSearchQuery
import de.hybris.platform.servicelayer.search.FlexibleSearchService

final String catalogVersionQuery = "SELECT {catalogVersion." + CatalogVersionModel.PK + "}" +
        "FROM {" + CatalogVersionModel._TYPECODE + " AS catalogVersion " +
        "JOIN " + CatalogModel._TYPECODE + " as catalog " +
        "ON {catalog." + CatalogModel.PK + "} = {" + CatalogVersionModel.CATALOG + "}}" +
        "WHERE {catalog." + CatalogModel.ID + "} = ?id AND {catalogVersion.active} = ?isActive"

final FlexibleSearchQuery catalogVersionFlexibleSearchQuery = new FlexibleSearchQuery(catalogVersionQuery)

catalogVersionFlexibleSearchQuery.addQueryParameter("id", "testCat")
catalogVersionFlexibleSearchQuery.addQueryParameter("isActive", false)

FlexibleSearchService flexibleSearchService = spring.getBean("flexibleSearchService")
List<CatalogVersionModel> inactiveCatalogVersions = flexibleSearchService.search(catalogVersionFlexibleSearchQuery).getRes

return inactiveCatalogVersions.collect { catalogVersion -> catalogVersion.getVersion() }.join(", ")
```

No-Cache Mode for FlexibleSearch

FlexibleSearch heavily uses cache for storing search results. In most cases, this behavior is desirable but sometimes it is better to skip cache completely.

When a query contains a frequently changing parameter such as current time in milliseconds, or a query result is huge and actually would pollute the cache, it's better to use the no-cache mode for Flexiblesearch. The FlexibleSearchService allows you to easily skip cache on demand. The following code example shows how to do that:

```
final FlexibleSearchQuery fQuery = new FlexibleSearchQuery("SELECT {PK} FROM {Foo} WHERE {modificationTime}=?modificationT
fQuery.addQueryParameter("modificationTime", Long.valueOf(System.currentTimeMillis()));
fQuery.setDisableCaching(true);

final SearchResult<FooModel> searchResult = flexibleSearchService.search(fQuery);
```

By simply using the `setDisableCaching` method on a `FlexibleSearchQuery` object, you are instructing FlexibleSearch to skip cache just for the current query.

Keep in mind that the type **Foo** used in the example above is an artificial one.

Using FlexibleSearch in Backoffice

Triggering FlexibleSearch queries within Backoffice is possible in two ways: using the Backoffice Saved Queries functionality or using `ViewType` instances.

A `ViewType` instance is SAP Commerce representation of a database view. The `ViewType` representation in Backoffice is called a Report Definition. The Backoffice Saved Queries feature provides a way to execute custom FlexibleSearch queries and browse the results. A `SavedQuery` instance is a means of using

a FlexibleSearch query to retrieve items in the SAP Commerce instead of using the GenericSearch.

Using FlexibleSearch Using the SAP Commerce API

Using FlexibleSearch queries using the SAP Commerce API takes place in two steps, both of which can be done in one Java statement: setting up and running the query.

Constructing a FlexibleSearch Query

A FlexibleSearch query is constructed as a string which contains the query, such as:

```
final String query = "SELECT {pk} FROM {Product}"
// Flexible search service injected by Spring
final SearchResult<ProductModel> searchResult = flexibleSearchService.search(query);
```

To refer to a SAP Commerce type attribute in the FlexibleSearch query such as the primary key (PK) of an item, you need to reference the attribute when constructing the query. In cases where the attribute is clear without ambiguity, specifying the attribute alone is enough. Still, it is recommended to reference the type of the attribute as well for unambiguity. SAP Commerce resolves and translates the attribute reference automatically into the FlexibleSearch query:

Examples:

Java Code	FlexibleSearch
`final String query = "SELECT {" + ProductModel.PK + "} FROM {" + ProductModel._TYPECODE + "}";`	`SELECT {pk} FROM {Product}`

`String query =`
`"SELECT {p:" + ProductModel.PK + "} FROM {" + ProductModel._TYPECODE + " AS p}\n"+`
`"WHERE {" + ProductModel.VARIANTTYPE + "} IS NOT NULL"`
	`SELECT {p:pk} FROM {Product AS p`
	`WHERE {variantType} IS NOT NULL`

Calling a FlexibleSearch

→ Tip

Use Paging on Queries with Many Results

If you run a FlexibleSearch query that potentially returns more than 50 search results, be sure to use the paging mechanism of the FlexibleSearch described in the **Hints** section.

To call a FlexibleSearch statement using the API use `flexibleSearchService`, which is always available through the Spring, and has to be properly injected to your service as follows:

```
<bean id="myFancyService" class="de.hybris.platform.foobar.MyFancyService" >
  <property name="flexibleSearchService" ref="flexibleSearchService"/>
</bean>
```

```
public class MyFancyService implements FancyService
{
...
  private FlexibleSearchService flexibleSearchService;

  @Required
  public void setFlexibleSearchService(final FlexibleSearchService flexibleSearchService)
  {
    this.flexibleSearchService = flexibleSearchService;
  }
...
}
```

The `flexibleSearchService search(...)` method returns a `de.hybris.platform.servicelayer.search.SearchResult` instance, which holds a List of the individual search results. To access this List, call the `SearchResult` class `getResult()` method, such as:

```
final String query = "SELECT {" + ProductModel.PK + "} FROM {" + ProductModel._TYPECODE + "}";
final SearchResult<ProductModel> searchResult = flexibleSearchService.search(query);
List<ProductModel> result = searchResult.getResult();
```

You can then process this `Collection` instance like any other `Collection` instances:

```
final String query = "SELECT {" + ProductModel.PK + "} FROM {" + ProductModel._TYPECODE + "}";
final SearchResult<ProductModel> searchResult = flexibleSearchService.search(query);
final ProductModel product = searchResult.getResult().iterator().next();
```

⚠ Caution

The Collection returned by `SearchResult.getResult()` uses the lazy translation approach. At first access to a collection element, the element is translated to an item.

In case the item was removed between gathering of the search result and translation of the specific element, the returned collection has a null value at this position.

Passing Parameters

To pass parameters, create a Map instance holding the parameters and pass the Map to the `search( ...)` method, as in:

→ Tip

If you do not need to pass parameters to the query, you can pass `null` for the parameter map.

```
final Map<String, Object> params = new HashMap<String, Object>();

String query =
"SELECT {" + PriceRowModel.PK + "} FROM {" + PriceRowModel._TYPECODE "} "+
"WHERE {" + PriceRowModel.PRODUCT      + "} = ?product AND "+
      "{" + PriceRowModel.NET          + "} = ?net AND "+
      "{" + PriceRowModel.CURRENCY     + "} = ?currency AND "+
      "{" + PriceRowModel.UNIT         + "} = ?unit AND "+
      "{" + PriceRowModel.UNIT_FACTOR  + "} = ?unitfactor AND "+
      "{" + PriceRowModel.UG           + "} = ?userpricegroup AND "+
      "{" + PriceRowModel.MIN_QUANTITY + "} = ?minquantity AND "+
      "{" + PriceRowModel.PRICE        + "} = ?price ";

params.put("product",           product);
params.put("net",               priceCopy.isNet());
params.put("currency",          priceCopy.getCurrency());
params.put("unit",              priceCopy.getUnit());
params.put("unitfactor",        priceCopy.getUnitFactor());
params.put("userpricegroup",    priceCopy.getUserPriceGroup());
params.put("minquantity",       priceCopy.getMinQuantity());
params.put("price",             priceCopy.getPriceValue());

final SearchResult<PriceRowModel> searchResult = flexibleSearchService.search(query, params);
```

ⓘ Note

If your query uses the statement `?product`, then the key in the parameter map has to be `product` (without the `?`).

Instantiating of Search Results

If you retrieve only the PK database column (that is, only the PKs of SAP Commerce items), and provide the kind of type as a Java class, you can immediately cast the models represented by the PKs into the actual model instances. In other words, executing the following code returns a **Collection** of `CatalogModel` instances, not a **Collection** of PKs:

```
final String query = "SELECT {" + CatalogModel.PK + "} FROM {" + CatalogModel._TYPECODE + "} ORDER BY {" + CatalogModel.PK
final SearchResult<CatalogModel> searchResult = flexibleSearchService.search(query);
```

If you retrieve more than one database column, you receive several individual entries per row of result and you will not be able to cast the search result into item instances directly, not even if one of the database columns retrieved is the PK column.

Paging of Search Results

Managing more than some 50 or 100 search results in one single Collection is complicated and performs comparably slow. For this reason, the FlexibleSearch framework offers a paging mechanism.

Some FlexibleSearch queries run the risk of returning a very large number of search results, such as `SELECT * FROM {Products} WHERE {code} LIKE ?search OR {name} LIKE ?search`, where `?search` is a parameter from a text field.

To use this paging mechanism, use the `search(...)` method with `FlexibleSearchQuery` object as parameter. You have to set on `FlexibleSearchQuery` the `setNeedTotal` to `true`. If this parameter is set to `true`, the FlexibleSearch framework splits the number of returned search

results into pages. Using the `start` and `range` parameters, you can retrieve pages of search results. The following code snippet, for example, iterates over all the search results of the FlexibleSearch query, three at a time:

```
int start = 0;
final int range = 3;
int total;

String query = "SELECT {" + UnitModel.PK + "} FROM {"+ UnitModel._TYPECODE + "} ORDER BY " + UnitModel._TYPECODE;
final FlexibleSearchQuery fQuery = new FlexibleSearchQuery(query);
fQuery.setCount(range);
fQuery.setNeedTotal(true);

do
{
  fQuery.setStart(start);
  final SearchResult<LanguageModel> searchResult = flexibleSearchService.search(fQuery);
  total = searchResult.getTotalCount();
  start += range;
}
while(start < total);
```

Be aware that every navigation, either backward or forward, through a paged search result triggers a new search query on the database. Internally, the FlexibleSearch runs the query in full and uses an offset parameter to specify the portion of all search results to return. The fact that every navigation causes a database query has three major consequences:

1. Complex queries cause heavy load on the database:

   Executing a simple `SELECT` statement is rather fast, even with millions of search results. However, if your FlexibleSearch query requires `JOIN` or `UNION` to execute, load on the database (and, by consequence, response times) increases rapidly. As a rule of thumb, remember that the more different items are involved, the longer the execution time is. For example, the following table gives some short examples of some rather basic FlexibleSearch statements and the actual SQL queries triggered:

FlexibleSearch statement	SQL statement	Description

`SELECT {pk}`
`FROM {Product}`
	`SELECT item_t0.PK`
	`FROM products item_t0`
	`WHERE (item_t0.TypePkString`
	`IN (`
	`23087950835790560 , 23087950835774560 ,`
	`23087950837855968 , 23087950837852464 ,`
	`23087950837859216 , 23087950837848976 ,`
	`23087950837843968 , 23087950835790306 ,`
	`23087950835765569`
	`)`
	`)`
		This FlexibleSearch statement results in no big surprises. Basically,\ SAP Commerce translates the type system related data into the matching SQL data. Note that the actual database table to be searched is `products`, and not `product` as you might expect.

`SELECT {description}`
`FROM {Product}`
	`SELECT lp_t0.p_description`
	`FROM productslp lp_t0`
	`WHERE`
	`(`
	`(lp_t0.LANGPK=?)`
	`)`
	`AND`
	`(lp_t0.ITEMTYPEPK`
	`IN (`
	`23087950835790560 , 23087950835774560 ,`
	`23087950837855968 , 23087950837852464 ,`
	`23087950837859216 , 23087950837848976 ,`
	`23087950837843968 , 23087950835790306 ,`
	`23087950835765569`
	`)`
	`)`
		Even though you run the FlexibleSearch on the `Product` type, the SQL statement is run on the `productslp` table instead of on the `products` table. This is because the `description` attribute is localized, and localized attributes for a type are stored in an individual database table.

`SELECT {code}, {description}`
`FROM {Product}`
	`SELECT item_t0.Code , lp_t0.p_description`
	`FROM`
	`products item_t0`
	`JOIN productslp lp_t0`
	`ON item_t0.PK = lp_t0.ITEMPK`
	`AND lp_t0.LANGPK=?`
	`WHERE`
	`(`
	`item_t0.TypePkString`
		The seemingly simple example, `SELECT {code}, {description} FROM {Product}`, requires a `JOIN` over the `products` table and the localized tables of the `products` table (`products_lp`) in order to get ahold of the localizations of the `description` attribute as well. Whenever you run FlexibleSearch statements on localized and non-localized

FlexibleSearch statement	SQL statement	Description
	`IN ( 23087950835790560 , 23087950835774560 , 23087950837855968 , 23087950837852464 , 23087950837859216 , 23087950837848976 , 23087950837843968 , 23087950835790306 , 23087950835765569 ) )`	attributes of the same type, a `JOIN` is necessary.

2. Search results may differ over time:

   When a FlexibleSearch query is run for the first time, the search results correspond to the dataset in the database by execution time. As every navigation through paged search results runs a new database query, the FlexibleSearch statement always receives a list of search results which correspond to the point of time when the statement is executed. This means, however, that the list of search results does not necessarily match the list of search results which were found when the FlexibleSearch query was initially executed. Items referenced in the initial query may have been updated, created, or removed in the meantime.

3. `ORDER BY` required:

   The order in which unordered database search results are returned may differ on every database statement execution, depending on the database implementation. By consequence, an unordered FlexibleSearch query might receive different search results on every navigation through paged search results. To avoid seemingly random search results, using paged search results requires an `ORDER BY` clause in the FlexibleSearch. Note, that ordering the paged search results does not solve the issue with search results differing over time.

Specifying a Minimum Time to Live for Cached FlexibleSearch Results

The system enables you to specify a minimum time to live for FlexibleSearch results. For more information, see Specifying a Minimum TTL for Cached FlexibleSearch Results.

Testing FlexibleSearch Queries Using The SAP Commerce Administration Console

The SAP Commerce offers a tool to test FlexibleSearch queries.

Procedure

1. Open the SAP Commerce Administration Console.

   For more information, see Accessing the Administration Console.

2. Go to the **Console** tab and select the **FlexibleSearch** option.
3. The **FlexibleSearch** page displays. If you do not have any queries ready, you can use **Query samples** located in the sidebar.
4. Enter the query in **FlexibleSearch query** or **Direct SQL query** field.
5. Click the **Execute** button.

Restrictions

Restrictions are rules obeyed by FlexibleSearch which allow to limit search results depending on which type is searched and which user is currently logged in.

This happens transparently and does not require any code adjustment on business layer.

A restriction is basically just a fragment of the **WHERE** clause of a FlexibleSearch statement - that includes other UserGroups (whose members are affected by the restriction as well). A restriction always applies to a specified type and a specified User or UserGroup. It automatically adds them to the **WHERE** clauses of all applicable FlexibleSearch statements and thereby restricts the number of search results of these statements due to these additional search conditions.

For example:

FlexibleSearch Statement	Restriction Statement	Effective FlexibleSearch Statement
`SELECT {p:pk} FROM {Product AS p} WHERE {p:code} LIKE '%test%' statement - that is, a restriction adds search conditions.`	`{p:description} NOT NULL`	`SELECT {p:pk} FROM {Product AS WHERE {p:code} LIKE '%test%' AND {p:description} NOT NULL`

Scope of Restrictions

The effect of restrictions is transparent, no interaction is necessary. Whenever a restriction is active and applies to the combination of restricted type and user, the search results are limited.

Unlike type access rights (which are only effective within Backoffice), restrictions apply to FlexibleSearch results throughout SAP Commerce. In other words, restrictions affect FlexibleSearch results in Backoffice, in SAP, and in an SAP Commerce-based web application. Type access rights only affect Backoffice.

Since restrictions work on FlexibleSearch queries only, restrictions do not affect the following use cases:

- External search engines

  Search results supplied by third-party search engines are not affected by restrictions. To get third-party search engine search results affected by restrictions, you need to filter these search results by running a FlexibleSearch statement over them.

- **Item.getProperty()**, **LocalizableItem.getLocalizedProperty()** fetches item references directly via PK - any stored item is returned no matter if it would have been filtered by a currently active restriction

SAP Commerce includes a mechanism that allows using restrictions for cron jobs. For details, please refer to cronjob - Technical Guide.

Using Parameters in Restrictions

Often it will do to specify restrictions having literal query texts like

IS NOT NULL

or **{hidden}=1**.

But sometimes it may also be required to specify a restriction which relies upon a session bound parameters instead of a fixed literal value. To do so use the **?session** FlexibleSearch parameter which is available inside every query.

Now we're able to write restrictions like **{user} = ?session.user** or **{country} IN (?session.countries)** which gives us the freedom to filter differently depending upon which session context settings have been made.

⚠ Caution

When referring to custom session context attribute (like **countries** above) make sure that the session context **does** contain this attribute. Otherwise any attempt to run a query which is affected by the restriction will throw a FlexibleSearch error!

Disabling Restrictions

For development, testing and debugging purposes, it may prove useful to disable restrictions as they **falsify** FlexibleSearchService query results.

Assigning the Session to an Admin User

By convention, restrictions do not apply to admin users (that is, users who are members of the **admingroup** user group). This means that a session assigned to a member of the **admingroup** user group is not affected by restrictions. This setting is in effect as long as the session is assigned to an admin user and applies to all restrictions.

ℹ Note

Using Filters

Assigning the session to an admin user has the side effect of granting the session access to every CatalogVersion in SAP Commerce. The default user assigned to a session (**anonymous**) requires that CatalogVersions be set for the session explicitly. To make sure that CatalogVersions are set for non-admin user sessions, you need to integrate filters in your Web application. You need to use the Platform filter chain. Use it to add proper filter for catalog version activation into the filter chain. Read hybris Platform Filters for more details.

Find an example on how to assign the session to the **admin** user below:

```
...
                                        import de.hybris.platform.servicelayer.user.UserService;
                                        ...
                                        @Autowired
                                        private UserService userService;
                                        ...
                                        userService.setCurrentUser(userService.getAdminUser())
                                        ...
```

The following code snippet shows how to get Products for a category available for **admin** user. The important part here is the fact that setting **admin** user in the session as well getting products is executed in local view by using **sessionService.executeInLocalView**.

```
...
import de.hybris.platform.product.ProductService;
import de.hybris.platform.servicelayer.session.SessionExecutionBody;
import de.hybris.platform.servicelayer.session.SessionService;
import de.hybris.platform.servicelayer.user.UserService;
...
@Autowired
private SessionService sessionService;
@Autowired
private UserService userService;
@Autowired
private ProductService productService;
...
public List<ProductModel> getProductsByCategory()
{
return (List<ProductModel>) sessionService.executeInLocalView(new SessionE
{
@Override
public List<ProductModel> execute()
{
userService.setCurrentUser(userService.getAdminUser());
return productService.getProductsForCategory(getSomeCategory());
}
});
}

private CategoryModel getSomeCategory()
{
...
return someCategoryHere;
}
```

Enabling or Disabling Search Restrictions

Find below an example of how to enable and disable a search restriction:

```
...
import de.hybris.platform.search.restriction.SearchRestrictionService;
...
// Disable search restrictions
searchRestrictionService.disableSearchRestrictions();
// some query goes here

// Enable search restrictions
searchRestrictionService.enableSearchRestrictions();
// some query goes here
```

Creating Restrictions

A restriction is represented by a **SearchRestriction** class instance, which has the following mandatory attributes:

SearchRestriction Attribute	Allowed Value	Description
`active`	`java.lang.Boolean`	Specifies whether the `SearchRestriction` is enabled (**true**) or disabled (**false**. Defaults to **true**.
`code`	`java.lang.String`	The identifier for the restriction. Must be unique throughout all `SearchRestriction` type instances.
`principal`	`Principal`	The user or user group to whom the restriction applies.
`query`	`java.lang.String`	The query of the `SearchRestriction`, that is, a WHERE clause of a `FlexibleSearch` statement that narrows down a `FlexibleSearch` statement.
`restrictedType`	`ComposedType`	The type on which `FlexibleSearch` queries are to be restricted when executed with the specified principal.
`generate`	`java.lang.Boolean`	Defines whether the build process should generate a jalo code for this type system element. `SearchRestriction` extends `TypeManagerManaged` that has the non-optional generate field.

You can create a restriction by using the SAP Commerce API, and with ImpEx.

To learn how to create a restriction in Backoffice, see Creating Restrictions in Backoffice.

Creating Restrictions via the SAP Commerce API

To create a restriction we need to use model service to create a new **SearchRestrictionModel** instance and then populate it with desired attributes, for example:

```
final ComposedTypeModel restrictedType = typeService.getComposedTypeForClass(LanguageModel.class);
                                                final PrincipalModel principal = userService.getUserForUID("ahertz");

                                                final SearchRestrictionModel searchRestriction = modelService.create(Searc
                                                searchRestriction.setCode("some code");
                                                searchRestriction.setActive(Boolean.TRUE);
                                                searchRestriction.setQuery("{active} IS TRUE");
                                                searchRestriction.setRestrictedType(restrictedType);
                                                searchRestriction.setPrincipal(principal);
                                                searchRestriction.setGenerate(Boolean.TRUE);
                                                modelService.save(searchRestriction);
```

Creating Restrictions via the ImpEx Extension

The following sample code shows how to create restrictions using an Impex-compliant CSV file:

```
INSERT_UPDATE SearchRestriction;code[unique=true];name[lang=de];name[lang=en];query;principal(UID);restrictedType(code);ac
                                                                                ;Frontend_Navigationelement;Navigation;Navigation;{active} IS TRUE
```

Related Information

ImpEx
Visibility Control
The Cronjob Service

Creating Restrictions in Backoffice

Define a new personalization rule in Backoffice Administration Cockpit to create restrictions for FlexibleSearch.

Procedure

1. Log in to Backoffice using an account with sufficient rights to create restrictions.
2. Go to **System** > **Personalization**.
3. Click the **+** icon to open the **Create New Personalization rule** window.
4. Enter your restriction query string into the **Filter** field.
5. Click the **...** icon next to the **Apply on** field and select a user or a user group for the restriction to be effective on.
6. Click the **...** icon next to the**Restricted Type** field and select a type to apply the restriction to.
7. Give your new restriction a unique identifier.
8. Click **Done** to save the new restriction.

FlexibleSearch Samples

This document discusses a number of FlexibleSearch samples. As the FlexibleSearch is a key component of SAP Commerce, reading this document is recommended for all developers.

This document does not discuss the FlexibleSearch in general, please refer to FlexibleSearch for general information on FlexibleSearch.

Basic SELECT Statements

This section discusses FlexibleSearch statements with one single **SELECT** operator. Some of these samples overlap in terms of operators, this is hard to avoid for such a subject.

SELECT Statements with Negation

The following FlexibleSearch statements are samples for using the negation operator **NOT**.

- **Getting all Products whose code is not empty**

  The following FlexibleSearch statement returns Primary Keys (PK) of every **Product** whose **code** attribute is different from **null**. Be aware that in SQL syntax the empty string "" is not considered to be **null**. In other words: the following FlexibleSearch statement finds **Products** whose code is set to "".

  ```
  SELECT {p.pk} FROM {Product AS p} WHERE {p.code} IS NOT NULL
  ```

- **Getting all Categories whose code does not contain a certain string**

  The following FlexibleSearch statement returns the PKs of every **Category** whose **code** attribute does not contain the string **test**.

  ```
  SELECT {c:pk} FROM {Category AS c} WHERE {c:code} NOT LIKE '%test%'
  ```

SELECT Statements with Several Return Columns

The arguments passed for the **SELECT** operator specify the columns from the database that are to be returned by the FlexibleSearch query.

- **Returning every database column of every Category**

  The following FlexibleSearch statement returns every database column of every Category in SAP Commerce. The list of returned database columns could be something along the lines of: **hjmpts**, **modifiedts**, **createdts**, **typepkstring**, **pk**, **ownerpkstring**, **aclts**, **propts**, **p_showemptyattributes**, **p_normal**, **p_thumbnails**, **p_revision**, **p_code**, **p_data_sheet**, **p_logo**, **p_catalogversion**, **p_picture**, **p_detail**, **p_catalog**, **p_others**, **p_order**, **p_thumbnail**, and **p_externalid**.

  ```
  SELECT * FROM {Category}
  ```

- **Getting the point of time when a Category was last modified, Category code and PK**

  The following FlexibleSearch statement returns three database columns of every **Category** in SAP Commerce. Note that the **code** attribute is enclosed by curly braces.

  ```
  SELECT {cat:modifiedtime}, {cat:code}, {cat:pk} FROM {Category AS cat}
  ```

SELECT Statements Over Several Attributes

A FlexibleSearch statement allows narrowing down the list of search results by specifying several attributes in a search condition. A SAP Commerce item must match the search condition in the respective attribute to become included in the search result list. For example, if the FlexibleSearch statement queries for the **code** and the **name** attribute, the list of search results contains only items that have a matching value in both the **code** and the **name** attribute.

- **Getting all Products whose code or name contains a certain string**

  The following FlexibleSearch returns the PKs of all Products whose **code** attribute or **name** attribute contains a string that ends with **myProduct**.

  ```
  SELECT {p:PK}
     FROM {Product AS p}
     WHERE {p:code} LIKE '%myProduct'
        OR {p:name} LIKE '%myProduct'
     ORDER BY {p:code} ASC
  ```

  The percent sign (**%**) works as a wildcard character:

  - **a%** finds all strings that start with an **a**,
  - **%a** finds all strings that end with an **a**,
  - **%a%** finds all strings that contain an **a**.

    By introducing a parameter (**?name** in the following FlexibleSearch query) into the query, you can search for any search string:

    ```
    SELECT {p:PK}
       FROM {Product AS p}
       WHERE {p:code} LIKE ?name
          OR {p:name} LIKE ?name
       ORDER BY {p:code} ASC
    ```

    Be aware that the search condition **`WHERE LIKE ?name`** only finds products whose **name** attribute matches the value of the **?name** parameter exactly. To find **Products** whose **name** attribute contains the value of the **?name** parameter only partially, you need to use wildcard characters, as in **`WHERE LIKE CONCAT('%', CONCAT(?name, '%'))`** or enclosing parameter by wildcard characters like:

    `...LIKE ?name; query.addQueryParameter("name", "%h%")`. **Both solutions are SQL-injection safe.**

SELECT Statements Over Several Languages

SAP Commerce allows for attributes to be localized, that is, to have an individual value for each language in SAP Commerce. By specifying the language code enclosed by square brackets (**[** and **]**, respectively) after the attribute name, such as **`{description[de]}`**. Not specifying a language code for a localized attribute causes SAP Commerce to use the default language for the current session.

- **Getting all Products with an empty name in the current SAP Commerce language**

  The following FlexibleSearch query returns the PKs of all **Products** whose **name** attribute is not set (**IS NULL**). The **name** attribute is localized, but the FlexibleSearch query does not specify a language explicitly, therefore the FlexibleSearch defaults to the current session language.

```
SELECT {p:PK}
   FROM {Product AS p}
   WHERE {p:name} IS NULL
```

- **Getting all Products with an empty name in German or an empty description in English**

  The following FlexibleSearch query returns the PKs of all **Products** and whose...

- **name** attribute is not set for the German language (**IS NULL**) or
- **description** attribute is not set for the English language (**IS NULL**).

Both the **name** attribute and the **description** attributes are localized. The FlexibleSearch query specifies the language explicitly (via de and en, respectively). If no language is specified, the FlexibleSearch would default to the current session language.

```
SELECT {p:PK}
   FROM {Product AS p}
   WHERE {p:name[de]} IS NULL
      OR {p:description[en]} IS NULL
```

- **Searching several languages at once**

  By specifying different language codes, you can search localized attributes in various languages at a time. The following FlexibleSearch statement searches both the English and German values of the **description** attribute:

```
SELECT {p:PK}
   FROM {Product AS p}
   WHERE {p:description[en]:o} LIKE '%text%'
   OR {p:description[de]:o} LIKE '%text%'
```

  Here, **:o** (outer join) parameter is used to include matches with missing rows in the **ProductsLP** table (=. the table that holds localized products) as well. Otherwise , the query would only return products with an existing row in **ProductsLP** table, because it would only use JOIN.

→ Tip

Add OR Clause to Search Additional Attributes or Languages

To search another language, add the attribute to be searched with an explicit specification of the language to be searched to the **WHERE** clause via an **OR** clause. For example, the following FlexibleSearch statement searches the **description** attribute of the **Product** in the three languages English, German, and French (**en**, **de**, and **fr**, respectively) and the **name** attribute in German:

```
SELECT {p:PK}
   FROM {Product AS p}
   WHERE {p:description[en]:o} LIKE '%text%'
   OR {p:description[de]:o} LIKE '%text%'
   OR {p:name[de]:o} LIKE '%text%'
   OR {p:description[fr]:o} LIKE '%text%'
```

It is also possible to replace the hard-coded search string with a parameter. Please refer to the SELECT statements with parameters section below for details.

SELECT Statements with Parameters

A parameter in a FlexibleSearch query allows inserting varying search patterns. This is a common field of use for applications where one single query is intended to be used for various searches, such as:

-
  - Search fields in the store frontend.
  - Search fields in Backoffice.
  - Item retrieval in the application business code.

- **Using one parameter in a FlexibleSearch statement**

  The following FlexibleSearch statement queries the **description** attribute in three SAP Commerce languages (**en**, **de**, **fr**) for one single parameter, **?param**. In other words, the FlexibleSearch finds all **Product** instances whose description in English, German, or French contains the search pattern specified by **?**

param.

```
SELECT {p:PK}
   FROM {Product AS p}
   WHERE {p:description[en]:o} LIKE ?param
   OR {p:description[de]:o} LIKE ?param
   OR {p:description[fr]:o} LIKE ?param
```

Note, that there are **:o** characters in **WHERE** clause. They are used to force the related table to be outer-joined (in case of localized properties the xxxLP table).

- **Using two parameters in a FlexibleSearch statement**
  - Getting every **Product** that is in at least one of two **Categories**:

```
SELECT {cpr:target}
   FROM {CategoryProductRelation AS cpr}
   WHERE {cpr:source} LIKE ?param1
      OR {cpr:source} LIKE ?param2
```

  - Getting every **Product** that was changed between two dates:

```
SELECT {pk}
   FROM {Product}
   WHERE  {modifiedtime} >= ?startDate
      AND {modifiedtime} <= ?endDate
```

SELECT Statements with Concatenation

The FlexibleSearch feature allows concatenating strings within a statement. Be aware that each **CONCAT** operator call allows only two parameters. To concatenate more than two parameters, you need to run more than one **CONCAT** operator calls.

- **Enclosing a search string by percent signs %**

  The following FlexibleSearch statement gives an example on the **CONCAT** operator by concatenating the leading **%** character with the concatenation of **myProduct** and the **%** character for a total of **%myProduct%**:

```
SELECT {p:PK}
   FROM {Product AS p}
   WHERE {p:description[de]}
      LIKE
         CONCAT(
            '%',
            CONCAT(
               'myProduct',
               '%'
            )
         )
   OR {p:description[en]}
      LIKE
         CONCAT(
            '%',
            CONCAT(
               'myProduct',
               '%'
            )
         )
   ORDER BY {p:code} ASC
```

  This function is useful in combination with parameters, as in the following FlexibleSearch statement:

```
SELECT {p:PK}
   FROM {Product AS p}
   WHERE {p:description[de]} LIKE
      CONCAT(
         '%',
         CONCAT(
            ?param,
            '%'
         )
      )
   OR {p:description[en]} LIKE
      CONCAT(
         '%',
         CONCAT(
            ?param,
            '%'
         )
      )
   ORDER BY {p:code}
```

SELECT Statements with DISTINCT Operator

The **DISTINCT** operator makes sure that duplicate results are returned only once. Duplicate return results may occur from sub-selects, **JOIN** clauses or from identical parameters, for example.

- **Finding every Product that is in at least one of two given Categories**

  The following FlexibleSearch statement returns every **Product** that is assigned to at least one of the two **Categories** provided by the two parameters **?param1** and **?param2**. The **DISTINCT** operator ensures that every **Product** is returned only once even if it were assigned to both **Categories**.

```
SELECT DISTINCT {cpr:target}
   FROM {CategoryProductRelation AS cpr}
   WHERE {cpr:source} LIKE ?param1
      OR {cpr:source} LIKE ?param2
```

SELECT Statements with GROUP BY Operator

- **Getting every Product which has been ordered, grouped by the Product**

```
SELECT {oe:product}
FROM {OrderEntry AS oe}
GROUP BY {oe:product}
```

Subselects

A subselect is a **SELECT** statement within a **SELECT** statement. Via a subselect, a **SELECT** statement can affect (narrow down or expand, for example) a search result list. The basic syntax looks as follows:

```
SELECT *
   FROM ${type}
   WHERE

   {{
      SELECT *
         FROM ${other_type}
         WHERE ${subselect_search_condition}
   }}
```

Note

Subselects in **FROM** clause of the query are also allowed (the example above presents subselect in the WHERE clause of the query). See Subselect with Parameters section below for more details.

Subselect Over Several Types

- **Getting every Product that has a directly or indirectly assigned PriceRow**

  The following FlexibleSearch statement lists every Product that:

  - has at least one **DiscountRow** directly assigned to it (**subselect 1**) or
  - is assigned to a **ProductDiscountGroup** that thas a **DiscountRow** assigned to it (**subselect 2**)

```
SELECT DISTINCT {p:PK}, {p:name}, {p:code}
   FROM {Product AS p}
   WHERE {p:PK} IN
   (
      {{
         -- subselect 1
         SELECT {dr:product}
            FROM {DiscountRow AS dr}
      }}
   )
   OR {p:PK} IN
   (
      {{
         -- subselect 2
         SELECT {prod:PK}
            FROM
               {
                  Product AS prod
                     LEFT JOIN DiscountRow AS dr
                     ON {prod:Europe1PriceFactory_PDG} = {dr:pg}
               }
            WHERE {prod:Europe1PriceFactory_PDG} IS NOT NULL }}
   )
   ORDER BY {p:name} ASC, {p:code} ASC
```

- **Getting every Product that is in at least 3 Categories**

The following FlexibleSearch statement returns every **Product** that is in more than three **Categories** (specified by the **WHERE howmany > 3** clause in **subselect 1**). The **subselect 2** returns the number of categories a **Product** is in (via searching the **CategoryProductRelation**). The **subselect 1** returns only those products which are in more than three **Categories** (**WHERE howmany >3**).

```
SELECT {p:PK}
   FROM {Product AS p}
   WHERE {p:PK} IN
   (
      -- subselect 1
      SELECT prod
         FROM
            (
               {{
                  -- subselect 2
                  SELECT {cpr:target} AS prod, count({cpr:target}) AS howmany
                     FROM {CategoryProductRelation AS cpr}
                     GROUP BY {cpr:target}
               }}
            )
            temptable
            WHERE howmany > 3
   )
   ORDER BY {p:name} ASC, {p:code} ASC
```

ℹ Note

Add a Parameter for Flexibility

By replacing the hard-coded **3** with a parameter, you could use the statement to find every **Product** that is in a specified number of categories, such as:

```
SELECT {p:PK}
   FROM {Product AS p}
   WHERE {p:PK} IN
   (
      -- subselect 1
      SELECT prod
         FROM
            (
               {{
                  -- subselect 2
                  SELECT {cpr:target} AS prod, count({cpr:target}) AS howmany
                     FROM {CategoryProductRelation AS cpr}
                     GROUP BY {cpr:target}
               }}
            )
            temptable
            WHERE howmany > ?number
   )
   ORDER BY {p:name} ASC, {p:code} ASC
```

Please also refer to the Subselect with Parameters section below for additional information.

Subselect with Parameters

- **Getting all Products ordered on or after a certain date**

  The following FlexibleSearch statement returns every **Product** that was ordered on or after a certain date. Via **subselect 2**, the FlexibleSearch statement retrieves every **Order** that was created on or after the value specified by the **?date** parameter. Of these search results, **subselect 1** retrieves the OrderEntries that belong to these **Orders**. The outermost **SELECT** statement gets the **Products** referred by the **OrderEntries**.

```
SELECT {p:PK}
   FROM {Product AS p}
   WHERE {p:PK} IN
   (
      {{
         -- subselect 1
         SELECT DISTINCT {oe:product}
            FROM {OrderEntry AS oe}
            WHERE {oe:order} IN
            (
               {{
                  -- subselect 2
                  SELECT {o:PK}
                     FROM {Order AS o}
                     WHERE {o:date} >= ?date
               }}
            )
      }}
   )
```

- **Getting every Product without a PriceRow in a specified Currency**

The following FlexibleSearch statement returns every **Product** that does not have a **PriceRow** assigned for the specified **Currency**. The subselect returns every **PriceRow** for the specified currency. This search result is then **negated** via the outer FlexibleSearch statement, which returns every **Product** that is not included in the search results that are returned by the subselect.

```
SELECT {p:PK}
   FROM {Product AS p}
   WHERE {p:PK} NOT IN
   (
      {{
         -- subselect
         SELECT {pr:product}
            FROM {PriceRow AS pr}
            WHERE {pr:currency} = ?currency
      }}
   )
   ORDER BY {p:name} ASC, {p:code} ASC
```

- **Reporting query with subselect in FROM clause and SQL aggregate functions.**

  This query calculates average **Order** value and average **Order** unit count within specified date range.

  Result of this query is a pair of numeric values. The first one is the average **Order** value, and the second one is the average order unit count.

  The term **unit count** of the order means the sum of quantities of the order entries. For example if an order consists of:

  - 1 red T-shirt
  - 1 blue T-shirt
  - 2 yellow T-shirt

    Then the order unit count for this order is 4.

```
SELECT AVG(torderentries.totprice), AVG(torderentries.totquantity)
        FROM (
        {{
                SELECT SUM({totalPrice}) AS totprice, SUM({quantity}) AS totquantity FROM {OrderEntry}
                        WHERE {creationtime} >= ?startDate AND {creationtime} < ?endDate GROUP BY {order}
        }}
        ) AS torderentries
```

Combined SELECT Statements with UNION Operator

The following FlexibleSearch statement uses the **UNION** operator to retrieve the set of results for two **SELECT** statements:

```
SELECT x.PK
FROM
(
{{SELECT {PK} as PK FROM {Chapter}
WHERE {Chapter.PUBLICATION} LIKE 6587084167216798848
}}
UNION ALL
{{
SELECT {PK} as PK FROM {Page}
WHERE {Page.PUBLICATION} LIKE 6587084167216798848
}}) x
```

FlexibleSearch Tips and Tricks

You can use the FlexibleSearch to build advanced queries for reports, get information from collection attributes, use and format dates, or use conditional parts of the query.

Flexible Search and Collections

Let's say you have a subtype of `Order` that holds a collection of `VoucherCard` items. You wish to get all `VoucherCards` that are assigned to a particular order, and the prices assigned to the `VoucherCards`.

As long as the collection element type has a reference to the item that holds the collection, all is simple:

```
SELECT {vc.PK}, {vc.price}
FROM
{
   Order AS o JOIN VoucherCard AS vc
   ON {vc.order}= {o.pk}
}
WHERE {o.PK} = ?order
```

Things get more complicated if the reference between element and holder is missing. You can use some workaround based on the fact that a collection attribute is stored as a list of **PK**s:

```
SELECT {dm.code}, {pm.code}
FROM
{
   DeliveryMode AS dm JOIN PaymentMode AS pm
   ON {dm.supportedPaymentModeInternal} LIKE CONCAT( '%', CONCAT( {pm.PK} , '%' ) )
}
```

JOIN Clauses

In FlexibleSearch queries, only two kinds of **JOIN** clauses are available:

- **LEFT JOIN**
- **JOIN**

These clauses can be useful if you wish to select items that are connected to other items via a relation, for example:

```
SELECT {p:PK}, {c:code} FROM
{
   Product as p JOIN CategoryProductRelation as rel
   ON {p:PK} = {rel:target}
   JOIN Category AS c
   ON {rel:source} = {c:PK}
}
```

UNION Clauses

UNION clauses allow to unite the results of different queries. Here is a simple example of how to get all pages and chapters of a publication. `?pk` is a query parameter, for which a value must be specified at run time.

```
SELECT uniontable.PK, uniontable.CODE FROM
(
   {{
      SELECT {c:PK} as PK, {c:code} AS CODE FROM {Chapter AS c}
      WHERE {c:PUBLICATION} LIKE ?pk
   }}
   UNION ALL
   {{
      SELECT {p:PK} as PK, {p:code} AS CODE FROM {Page AS p}
      WHERE {p:PUBLICATION} LIKE ?pk
   }}
) uniontable
```

Using Temporary Tables

When you use temporary tables, you need to use a slightly different syntax to retrieve values from a temporary table. Instead of the FlexibleSearch syntax (`INNERTABLE:PK`), you have to use the native SQL syntax (`INNERTABLE.PK`), such as:

```
SELECT INNERTABLE.PK, INNERTABLE.CatCode FROM
(
   {{
      SELECT {p:PK} AS PK, {c:code} AS CatCode FROM
      {
         Product as p JOIN CategoryProductRelation as rel
         ON {p:PK} = {rel:target}
         JOIN Category AS c
         ON {rel:source} = {c:PK}
      }
   }}
) INNERTABLE
```

Conditional CASE Statements

If you create a complex query and you want to return different results of an attribute based on some condition, try the `CASE` statement.

See an example of getting all categories names, codes, and number of super categories. If category does not have any subcategories mark it as root category, otherwise mark it as normal category.

```
SELECT {c:name[en]} AS Name, {c:code} AS Code,
(
```

```
    CASE
       WHEN COUNT(DISTINCT{superCategory:PK}) <= 0
       THEN 'root category'
       ELSE 'normal category'
    END
) as TYPE,
COUNT(DISTINCT{superCategory:PK}) AS SuperCategories
FROM
{
    Category as c LEFT JOIN CategoryCategoryRelation as rel
    ON {c:PK} = {rel:target}
    LEFT JOIN Category AS superCategory
    ON {rel:source} = {superCategory:PK}
}
GROUP BY {c:PK}, {c:code}, {c:name[en]}
```

NAME	CODE	TYPE	SUPERCATEGORIES
AMD	HW2120	normal category	2
AMD	HW2120	normal category	2
ATI	HW2320	normal category	1
ATI	HW2320	normal category	1
Accessories	accessories	root category	0
Accessories	accessories	root category	0
Anti-Virus Software	antivirus	normal category	1
Apparel	apparel	root category	0
Apparel	apparel	root category	0

You can also put SELECT statements into CASE statements, as in this little more complicated query snippet below. The query basically counts super categories for each category and sub categories only for the root category. It may seem that such a query would never be used but it sometimes is the only solution for a report table filling query - a table that also is responsible to aggregate and sort results.

```
SELECT {c:name[en]} AS Name, {c:code} AS Code,
(
    CASE
       WHEN COUNT(DISTINCT{superCategory:PK}) <= 0
       THEN 'root category'
       ELSE 'normal category'
    END
) as TYPE,
(
    CASE
       WHEN ( COUNT(DISTINCT{superCategory:PK}) <= 0 )
       THEN
       (

          {{
             SELECT COUNT({innerC:PK}) FROM
             {
                Category as innerC LEFT JOIN CategoryCategoryRelation as innerRel
                ON {innerC:PK} = {innerRel:target}
             }
             WHERE {innerRel:source} = {c:pk}
          }}
       )
       ELSE 0
    END
 ) as RootSubCategories,
COUNT(DISTINCT{superCategory:PK}) AS SuperCategories
FROM
{
    Category as c LEFT JOIN CategoryCategoryRelation as rel
    ON {c:PK} = {rel:target}
    LEFT JOIN Category AS superCategory
    ON {rel:source} = {superCategory:PK}
}
GROUP BY {c:PK}, {c:code}, {c:name[en]}
```

NAME	CODE	TYPE	ROOTSUBCATEGORIES	SUPERCATEGORIES
Accessories	accessories	root category	0	0
Accessories	accessories	root category	0	0

NAME	CODE	TYPE	ROOTSUBCATEGORIES	SUPERCATEGORIES
Anti-Virus Software	antivirus	normal category	0	1
CPU	cpu	normal category	0	1
Content blocks	contentblocks	root category	3	0
Content blocks	contentblocks	root category	3	0
Digital photography	photography	normal category	0	1
Electronical Goods	electronics	root category	2	0
Hardware	hardware	normal category	0	1
Mainboards	boards	normal category	0	1
Memory	memory	normal category	0	1
Men's shoes	CL2100	normal category	0	1
Men's shoes	CL2100	normal category	0	1
Operating Systems	operating	normal category	0	1

Date Formatting

If you use dates in SQL, it is very likely that you would need to compare only parts of the date - for example, you would need to consider only months, dates, hours, but skip seconds and milliseconds. Nearly in every report you need to group your results by some date - months, hours, and so on. Here is how you do that on MySQL and Oracle.

Oracle

```
SELECT to_char({o:date},'mm/yyyy'), COUNT(DISTINCT{o:PK})
FROM {Order AS o}
GROUP BY to_char({o:date},'mm/yyyy')
```

MySQL

```
SELECT DATE_FORMAT({o:date},'%M/%Y'), COUNT(DISTINCT{o:PK})
FROM {Order AS o}
GROUP BY DATE_FORMAT({o:date},'%M/%Y')
```

String Formatting

In addition to date-specific operators, database system also offers String-specific operators.

MySQL Functions

Function	Description
ASCII()	Returns numeric value of left-most character.
BIN()	Returns a string representation of the argument.
BIT_LENGTH()	Returns length of argument in bits.
CHAR_LENGTH()	Returns number of characters in argument.
CHAR()	Returns the character for each integer passed.
CHARACTER_LENGTH()	A synonym for `CHAR_LENGTH()`.
CONCAT_WS()	Returns concatenate with separator.
CONCAT()	Returns concatenated string.
ELT()	Returns string at index number.
EXPORT_SET()	Returns a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string.

Function	Description
FIELD()	Returns the index (position) of the first argument in the subsequent arguments.
FIND_IN_SET()	Returns the index position of the first argument within the second argument.
FORMAT()	Returns a number formatted to specified number of decimal places.
HEX()	Returns a hexadecimal representation of a decimal or string value.
INSERT()	Inserts a substring at the specified position up to the specified number of characters.
INSTR()	Returns the index of the first occurrence of substring.
LCASE()	Synonym for `LOWER()`.
LEFT()	Returns the leftmost number of characters as specified.
LENGTH()	Returns the length of a string in bytes.
LIKE	Simple pattern matching.
LOAD_FILE()	Loads the named file.
LOCATE()	Returns the position of the first occurrence of substring.
LOWER()	Returns the argument in lowercase.
LPAD()	Returns the string argument, left-padded with the specified string.
LTRIM()	Removes leading spaces.
MAKE_SET()	Returns a set of comma-separated strings that have the corresponding bit in bits set.
MATCH	Performs full-text search.
MID()	Returns a substring starting from the specified position.
NOT LIKE	Negation of simple pattern matching.
NOT REGEXP	Negation of `REGEXP`.
OCTET_LENGTH()	A synonym for `LENGTH()`.
ORD()	Returns character code for leftmost character of the argument.
POSITION()	A synonym for `LOCATE()`.
QUOTE()	Escapes the argument for use in an SQL statement.
REGEXP	Pattern matching using regular expressions.
REPEAT()	Repeats a string the specified number of times.
REPLACE()	Replaces occurrences of a specified string.
REVERSE()	Reverses the characters in a string.
RIGHT()	Returns the specified rightmost number of characters.
RLIKE	Synonym for `REGEXP`.
RPAD()	Appends string the specified number of times.
RTRIM()	Removes trailing spaces.
SOUNDEX()	Returns a soundex string.
SOUNDS LIKE(v4.1.0)	Compares sounds.
SPACE()	Returns a string of the specified number of spaces.
STRCMP()	Compares two strings.
SUBSTR()	Returns the substring as specified.

Function	Description
SUBSTRING_INDEX()	Returns a substring from a string before the specified number of occurrences of the delimiter.
SUBSTRING()	Returns the substring as specified.
TRIM()	Removes leading and trailing spaces.
UCASE()	Synonym for `UPPER()`.
UNHEX()(v4.1.2)	Converts each pair of hexadecimal digits to a character.
UPPER()	Converts to uppercase.

Oracle Functions

Function	Definition
ASCII	The `ASCII` function returns the decimal representation in the database character set of the first character of char.
CHR	The `CHR` function returns the character having the binary equivalent to n as a `VARCHAR2` value in either the database character set.
COALESCE	The `COALESCE` function returns the first non-null expr in the expression list. At least one expr must not be the literal `NULL`. If all occurrences of expr evaluate to null, then the function returns null.
CONCAT	The `CONCAT` function returns the concatenation of 2 strings. You can also use the `||` command for this.
CONVERT	The `CONVERT` function converts a string from one character set to another. The datatype of the returned value is `VARCHAR2`.
DUMP	The `DUMP` function returns a `VARCHAR2` value containing the datatype code, length in bytes, and internal representation of expr. The returned result is always in the database character set.
INSTR	Returns the position of a String within a String. For more information, see Oracle instr function.
INITCAP	Transform String to init cap.
INSTRB	Returns the position of a String within a String, expressed in bytes.
INSTRC	Returns the position of a String within a String, expressed in Unicode complete characters.
INSTR2	Returns the position of a String within a String, expressed in `UCS2` code points.
INSTR4	Returns the position of a String within a String, expressed in `UCS4` code points.
LENGTH	The `LENGTH` function returns the length of char. `LENGTH` calculates length using characters as defined by the input character set.
LENGTHB	Returns the length of a string, expressed in bytes.
LOWER	The `LOWER` function returns a string with all lower case characters.
LPAD	Adds characters to the left of a string until a fixed number is reached. If the last parameter is not specified, spaces are added to the left.
LTRIM	`LTRIM` removes characters from the left of a string if they are equal to the specified string. If the last parameter is not specified, spaces are removed from the left side.
REPLACE	The replace function replaces every occurrence of a search_string with a new string. If no new string is specified, all occurrences of the search_string are removed.
REVERSE	Reverses the characters of a String.
RPAD	Adds characters to the right of a string until a fixed number is reached. If the last parameter is not specified, spaces are added to the right.
RTRIM	`RTRIM` removes characters from the right of a string if they are equal to the specified string. If the last parameter is not specified, spaces are removed from the

Function	Definition
	right side.
SOUNDEX	`SOUNDEX` returns a character string containing the phonetic representation of char. This function lets you compare words that are spelled differently, but sound alike in English.
SUBSTR	Returns a substring. For more information, see Oracle substring.
SUBSTRB	Returns a substring expressed in bytes instead of characters.
SUBSTRC	Returns a substring expressed in Unicode code points instead of characters.
SUBSTR2	Returns a substring using USC2 code points.
SUBSTR4	Returns a substring using USC4 code points.
TRANSLATE	`TRANSLATE` returns expr with all occurrences of each character in from_string replaced by its corresponding character in to_string. Characters in expr that are not in from_string are not replaced.
TRIM	The `TRIM` function trims specified characters from the left and/or right. If no characters are specified, the left and right spaces are left out.
(pipes)	With pipes, you can concatenate strings.
UPPER	Transform a string to all upper case characters.
VSIZE	The `VSIZE` function returns the byte size of a String.

Boolean Parameters in Queries

Since not all databases recognize `true` as a query parameter, `0` and `1` should be used instead of `false` and `true`.

Query and JDBC Hints

A hint is an optimization directive that you can embed into an SQL statement to instruct the database on how to execute the query. Platform allows you to influence query execution by using hints.

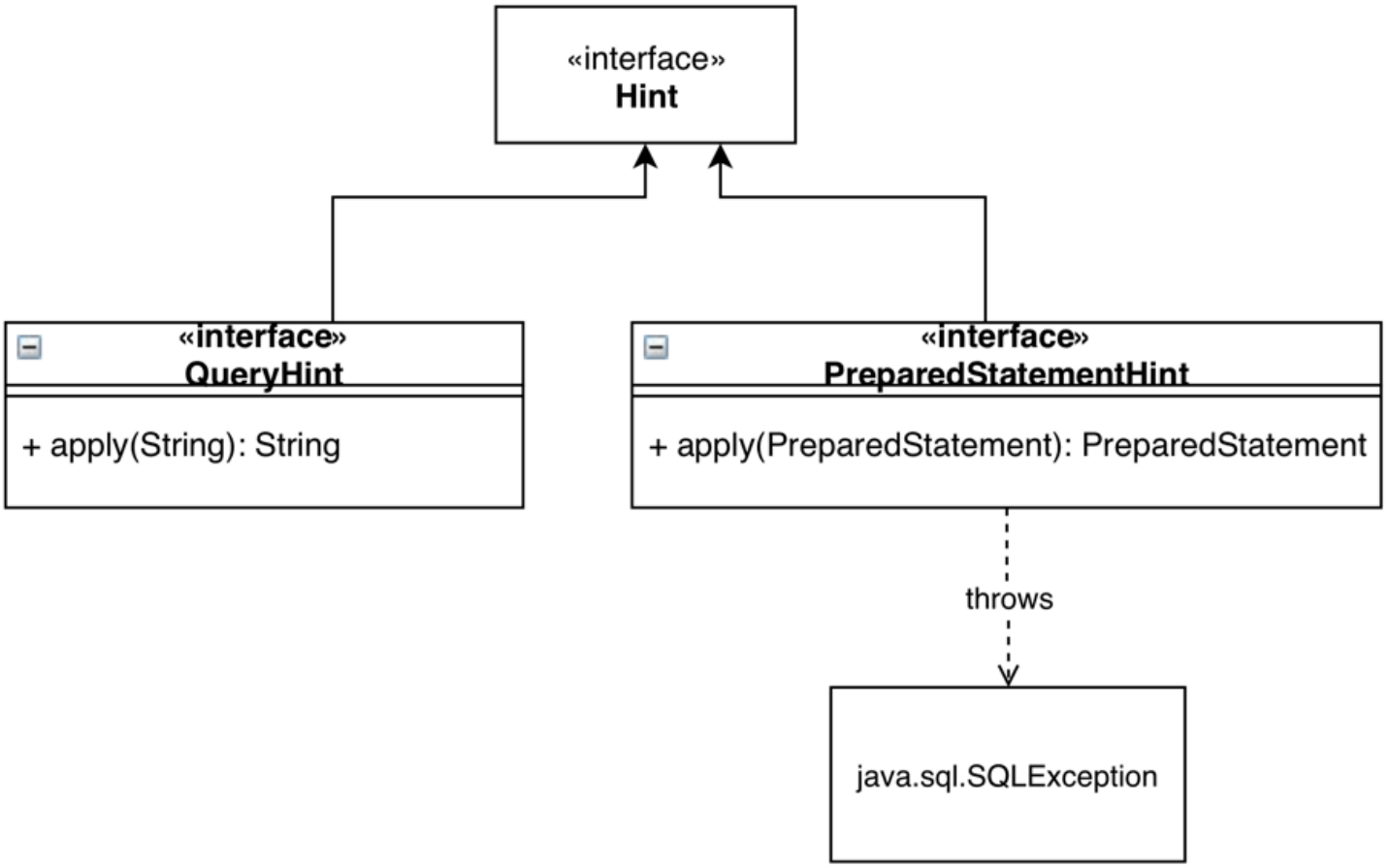
Some database engines allow you to set additional hints for an underlying optimizer that in certain circumstances may speed up query execution. In addition, the Java JDBC driver allows you to set some options on the `PreparedStatement` object that may also help in some situations.

ℹ Note

Keep in mind that providing query hints is an advanced topic and requires deep knowledge of the database engine.

Architecture

Platform provides a set of interfaces that allow you to define both the `Query` and the `PreparedStatement` hints:



The top level `Hint` interface acts only as a marker interface. The two sub-interfaces are responsible for modifying a query String and the `PreparedStatement` object respectively. The main `FlexibleSearchQuery` object has two additional methods to pass `Hint` instances to the FlexibleSearch engine:

```
public void addHints(final Hint... hints)
public void addHints(final List<? extends Hint> hints
```

QueryHint Implementation

Platform provides one implementation of the `QueryHint` interface for use in the SAP S/4HANA database. The `HanaHints` class acts as a factory builder for any hints SAP S/4HANA supports.

ℹ Note

Please keep in mind that the `HanaHints` class is a generic implementation so it doesn't provide any methods that would refer to the existing hints by name. You must exactly know the hint names to use them.

`HanaHints` provides the `create(String... hints)` method that allows you to create a `HanaHints` object as follows:

```
// will add IGNORE_PLAN_CACHE to the query
HanaHints hints = HanaHints.create("IGNORE_PLAN_CACHE");
// will add IGNORE_PLAN_CACHE and USE_OLAP_PLAN to the query
HanaHints hints = HanaHints.create("IGNORE_PLAN_CACHE", "USE_OLAP_PLAN");
```

If you need to add more hints to the existing `HanaHints` object, you can use the `add(String hint)` method. This can be useful for an instance in iterations:

```
HanaHints hints = HanaHints.create("IGNORE_PLAN_CACHE");
moreHints.forEach(hints::add);
```

After building the `HanaHints` object, it is easy to add it to the `FlexibleSearchQuery` object:

```
FlexibleSearchQuery fQuery = new FlexibleSearchQuery("SELECT {PK} FROM {User}");
fQuery.addHints(hints);
```

After the query is translated into a real SQL query, the hints are compiled into the proper hint directive and added to the very end of the query in the following form:

```
WITH HINT(HINT1,HINT2,....)
```

So the example from above would look like this:

```
SELECT  item_t0.PK  FROM users item_t0 WHERE (item_t0.TypePkString IN  (?,?,?,?) ) WITH HINT(IGNORE_PLAN_CACHE,USE_OLAP_PL
```

i Note

`FlexibleSearchQuery` objects allow you to add more than one hint to the query. In case of HANA-specific hints, it doesn't make sense since `HanaHints` compiles each separate hint into one `WITH HINT` directive.

The `HanaHints` class works in a safe way so that hints are applied only when the SAP S/4HANA database is really used.

PreparedStatement Hints

Platform provides the `JdbcHints` class that allows you to create hints that operate on the underlying `PreparedStatement` object. Use the `JdbcHints#preparedStatementHints()` factory method to produce an instance of the `PreparedStatementHint` interface to gain access to the `PreparedStatement` object.

To change the fetch size, use:

```
PreparedStatementHint = JdbcHints.preparedStatementHints().withFetchSize(50);
```

To change the timeout, use:

```
PreparedStatementHint hints = JdbcHints.preparedStatementHints().withQueryTimeOut(200);
```

It is also possible to provide your own code by implementing the `PreparedStatementHint` interface and using the generic method:

```
PreparedStatementHint hints = JdbcHints.preparedStatementHints().withHint(yourPreparedStatementHint);
```

or even using a JDK8 inline function as follows:

```
PreparedStatementHint hints = JdbcHints.preparedStatementHints().withHint(ps -> {
        ps.setCursorName("fooBar");
        returns ps;
    });
```

you can also chain all calls:

```
PreparedStatementHint hints = JdbcHints.preparedStatementHints().withFetchSize(50).withQueryTimeOut(200).withHint(ps -> {
        ps.setCursorName("fooBar")
        return ps;
    });
```

Since `PreparedStatementHint` is an instance of the `Hint` interface, you can easily add it to the `FlexibleSearchQuery` object:

```
FlexibleSearchQuery fQuery = new FlexibleSearchQuery("SELECT {PK} FROM {User}");
fQuery.addHints(JdbcHints.preparedStatementHints().withQueryTimeOut(200));
```

i Note

All `PrepatedStatementHint` objects are applied one by one in a chain on the same underlying `PreparedStatement` object

Polyglot Persistence Query Language

The polyglot persistence query language is a part of FlexibleSearch. You can use it to search for items inside an alternative storage, for example inside a document-type storage.

FlexibleSearch is used to search in a default, relational storage. The polyglot persistence query language is its counterpart for an alternative storage .

The polyglot query structure is a simplified version of the FlexibleSearch query structure. The FlexibleSearch `SELECT...FROM` is replaced with `GET`.

For example, to get:

- all `Title` instances, ordered by `code`, use:

```
GET {Title} ORDER BY {code}
```

- all `Title` instances with a given `code`, ordered by `code`, use:

```
GET {Title} WHERE {code}=?code ORDER BY {code}
```

Syntax Overview

A polyglot search query consists of the `GET` keyword followed by a type key and an expression:

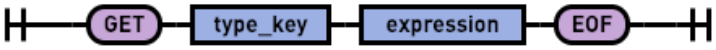


Figure: query

A type key is an item type identifier inside curly brackets, for example `{Title}`:

Figure: type_key

An identifier starts with a letter (a-Z), and then contains zero, or more of: letter (a-Z), digit (0-9), underscore ("_"), or the dot (".").

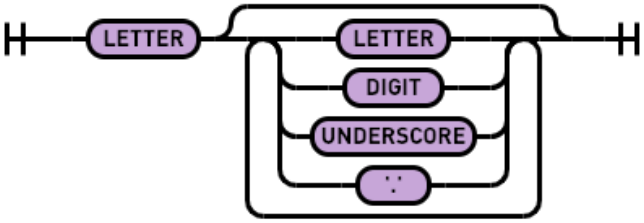


Figure: IDENTIFIER

An expression can start with the optional `WHERE` keyword.

`WHERE` can be followed by expr_or:

```
GET {Title} WHERE {code}=?code1 or {code}=?code2
```

`WHERE` can be also followed by optional order_by:

```
GET {Title} WHERE {code}=?code1 or {code}=?code2 ORDER BY {code}
```

An expression can contain order_by as the only element:

```
GET {Title} ORDER BY {code}
```

An expression can be empty - you don't have to include expressions in your queries at all:

```
GET {Title}
```

The diagram shows the described options:

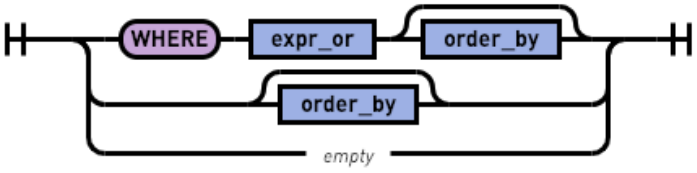


Figure: expression

order_by Structure

Here is an example of an order_by query, with search results ordered by a localized attribute:

```
GET {Product} WHERE {name}?=name ORDER BY {name[en]} ASC, {name[de]} DESC
```

The order_by element consists of the ORDER_BY keyword followed by one or more order_key elements separated by "," (a comma).

Figure: order_by

The order_key element consists of an attribute_key followed by the optional ORDER_DIRECTION element.

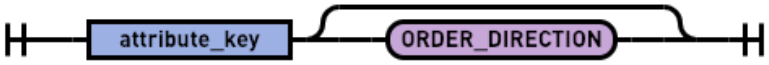


Figure: order_key

Here is an example of order_key - it consists of an attribute_key (includes a language identifier), and ORDER_DIRECTION:

```
{name[en]} ASC
```

attribute_key consists of the identifier element (described above) surrounded by curly brackets, with the optional localization identifier inside square brackets.

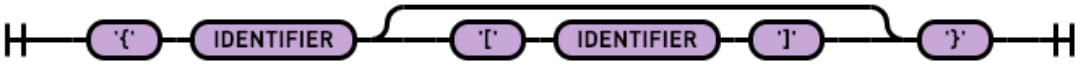


Figure: attribute_key

order_direction is: the ASC keyword for ascending, or the DESC keyword for a descending sort order.

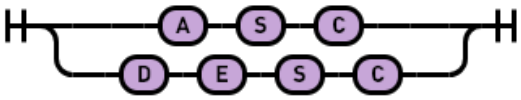


Figure: ORDER_DIRECTION

expr_or Structure

Here is an example of an expr_or query, with logical operations and sort order:

```
GET {Title} WHERE {code}=?code1 OR {code}=?code2 AND {code} IS NOT NULL ORDER BY {code[en]} DESC
```

To keep the operations in order, expr_or element consists of the expr_and element followed by the optional OR_OPERATOR and expr_and elements:

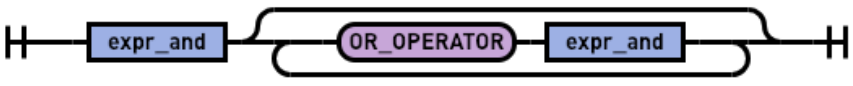


Figure: expr_or

This is the OR part of the query:

```
{code}=?code1 OR expr_and
```

The expr_and element consists of the expr_atom element followed by the optional AND_OPERATOR and expr_atom elements.

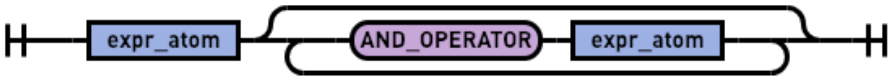


Figure: expr_and

This is the AND part of the query:

```
{code}=?code2 AND {code} IS NOT NULL
```

expr_atom consists of:

- attribute_key element followed by the CMP_OPERATOR element and '?' (question mark) followed by the IDENTIFIER element, or
- attribute_key element followed by NULL_OPERATOR, or
- expr_or element inside round brackets

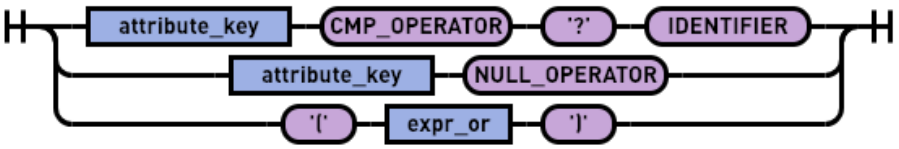


Figure: expr_atom

cmp_operator is one of the following: "=" (equal), "<>" (not equal), ">" (greater than), "<" (lower than), ">=" (greater or equal), "<=" (lower or equal):

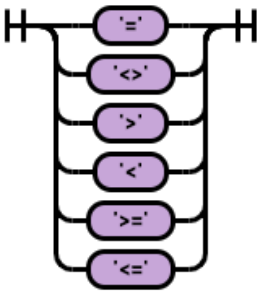


Figure: CMP_OPERATOR

null_operator consists of the `IS NULL` keywords divided by the optional `NOT` keyword:

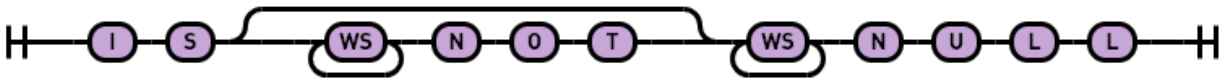


Figure: NULL_OPERATOR

ws is a white space (space, tabulator, new line, or carriage return) character:

Figure: ws

Related Information

Polyglot Persistence

GenericSearch

While SAP FlexibleSearch offers a powerful search API to developers, some of them may prefer the GenericSearch API that is similar to Hibernate Criteria Queries. Hibernate is a collection of related projects, enabling developers to utilize POJO-style domain models in their applications in ways extending well beyond Object and Relational Mapping.

Introduction

Both the **FlexibleSearchService** and the GenericSearch API allow developers to construct and execute queries against the SAP Commerce database, by focusing on SAP Commerce items rather than raw SQL. However, the way in which the queries are constructed is based on two completely different and complimentary approaches.

A **FlexibleSearchService** query is constructed by placing a FlexibleSearch query statement in a String, for example:

```
""SELECT {" + CategoryModel.PK + "} FROM {" + CategoryModel._TYPECODE + "} WHERE {" CategoryModel.ID "}=?id";
```

The **?id** placeholder in the query is the place for a real condition parameter that is replaced during query execution.

In contrast to this, a query, for the **GenericSearchService** is constructed by combining instances of **GenericField** and **GenericCondition** to form **GenericQuery**. **GenericQuery** is a Java-based object describing the search criteria, an example of which is shown below. Users of Hibernate ORM can see a close parallel between this approach and the Hibernate's Criteria Queries.

```
final String categoryID = "Foo";
final GenericCondition idCondition = GenericCondition.equals(
       new GenericSearchField(CategoryModel._TYPECODE, CategoryModel.ID ), categoryID);
final GenericConditionList gl = GenericCondition.createConditionList(idCondition);
final GenericQuery query = new GenericQuery(CategoryModel._TYPECODE, gl);
```

To execute the FlexibleSearch, use the **FlexibleSearchService**:

```
Collection result = flexibleSearchService.<CategoryModel> search("SELECT
    {" + CategoryModel.PK + "} FROM {" + CategoryModel._TYPECODE + "}
    WHERE {" CategoryModel.ID "}").getResult();
```

Likewise, to execute the GenericSearch, use the **GenericSearchService**:

```
Collection result = genericSearchService.search(query).getResult();
```

This document describes the latter approach, of performing GenericSearch calls by constructing **GenericQuery** instances. See FlexibleSearch for a detailed discussion on the former. Additionally, see the hybris Platform Search Mechanisms document for an overview of all search mechanisms available in SAP Commerce.

GenericSearch

The GenericSearch is a search framework that allows you to easily search on items stored within the Platform. Its functionality may be a bit limited, but on the other hand, it is easy to understand and to handle. Although it technically uses the FlexibleSearch, it encapsulates the FlexibleSearch syntax and usage. The functionality of the GenericSearch includes:

- Searching of items as well as raw data fields
- Unlimited number of conditions
- Inner joins and outer joins between item types possible
- Unlimited number of **order by** clauses
- Subselects.

The GenericSearch is the tool of choice for a typical **Find me all products** sort of query that the real **FlexibleSearchService** would be an overkill to work with. If you do not have extensive experience with the Platform, it is recommended to get familiar with the GenericSearch at first and use it as a ladder to work yourself upwards to the FlexibleSearch.

Before an actual GenericSearch statement is executed, it is run through a syntax checker. In other words, an incorrect statement is never executed. On top of that, you may assign names to GenericSearch statements so you may save them in the database.

To run a GenericSearch statement, set up a query and then have the current session run that query. Technically, you create a **GenericQuery** object and add fields to it, that contain the parts of your query that you want. An example for that:

```
// create a query to find products
GenericQuery query = new GenericQuery(ProductModel._TYPECODE);
// run the search
genericSearchService.search(query).getResult();
```

First, it creates a query that looks for products. Then it runs that query. Since the query is not limited in any way, it will find all products within the platform. Now, refine the query:

```
// create a query to find products
final GenericQuery query = new GenericQuery(ProductModel._TYPECODE);

// create a new field as a container for the search results
// the field searches in ProductModel.NAME
// roughly translated into SQL terms: SELECT FROM ProductModel.Name
final GenericSearchField nameField = new GenericSearchField(ProductModel._TYPECODE, ProductModel.NAME);

// add a search condition to the field created above
// corresponds about to the SQL statement:
// WHERE (nameField) IS LIKE 'display'
final GenericCondition condition = GenericCondition.createConditionForValueComparison(nameField, Operator.LIKE, "display")

// Add that search condition to query
query.addCondition(condition);

// order by name - ascendingly
```

```
query.addOrderBy(new GenericSearchOrderBy(nameField, true));

// run the search
genericSearchService.search(query).getResult();
```

The listing above searches for all products with names containing the string **display** and presents them in an ascendant order by their name. The next listing extends that even more:

```
// create a query to find products
final GenericQuery query = new GenericQuery(ProductModel._TYPECODE);

// create a new field as a container for the search results
// the field searches in ProductModel.NAME
// roughly translated into SQL terms: SELECT FROM ProductModel.Name
final GenericSearchField nameField = new GenericSearchField(ProductModel._TYPECODE, ProductModel.NAME);

// add a search condition to the field created above
// corresponds about to the SQL statement:
// where (nameField) IS LIKE 'display'
final GenericCondition condition = GenericCondition.createConditionForValueComparison(nameField, Operator.LIKE, "display")

// Add that search condition to query
query.addCondition(condition);

// order by name - ascendingly
query.addOrderBy(new GenericSearchOrderBy(nameField, true));

//create a join with all medias within the platform
GenericCondition joinCondition = GenericCondition.createJoinCondition(productMediaField, mediaField);

// add the join to the previous search so that only entries that
// match both searches are returned

query.addInnerJoin(joinCondition);

// run the search
genericSearchService.search(query).getResult();
```

This listing returns all products with names containing the string **display** that have a media assigned to them. If you need to, you can also assign a parameter to the search. The following listing shows how to do it:

```
// create a query to find products
final GenericQuery query = new GenericQuery(ProductModel._TYPECODE);

// create a new field as a container for the search results
// the field searches in ProductModel.NAME
// roughly translated into SQL terms: SELECT FROM ProductModel.Name
final GenericCondition condition = GenericCondition.createConditionForValueComparison(nameField, Operator.LIKE, "display",

// Add that search condition to query
query.addCondition(condition);

// order by name - ascendingly
query.addOrderBy(new GenericSearchOrderBy(nameField, true));

//create a join with all medias within the Platform
final GenericCondition joinCondition = GenericCondition.createJoinCondition(productMediaField, mediaField);

// add the join to the previous search so that only entries that match both searches are returned
query.addInnerJoin(joinCondition);

// run the search
genericSearchService.search(query).getResult();

//reset the value
query.getCondition().setResettableValue("myNameQualifier", "floppy"); // (2)

// find products with "floppy" in a name
genericSearchService.search(query).getResult();
```

The main difference is marked by (1) - it is the name of a parameter that you may hand over, for example in the URL. In the (2) this parameter is set to a new value, in this case String **floppy**. The search string to look for is then no longer **display**, but **floppy**. All other search parameters remain the same.

It is also possible to search for raw data instead of Item instances. To search for a code of a product, instead of for the **Product** itself, you should to do the following:

```
// the same code as above
// get only a code of products
final GenericSelectField codeField = GenericSelectField(ProductModel._TYPECODE, ProductModel.CODE, String.class);
query.addSelectField(codeField);
// find codes of products with "display" in their name
genericSearchService.search(query).getResult();
```

GenericSearch in Action

The most helpful resource for learning and seeing the GenericSearch in action is to take a look at the JUnit test for it in. The code sample below contains a commented example taken from **GenericSearchTest.java** illustrating a simple GenericSearch:

GenericSearchTest.java

```
public void testGenericConditions() throws Exception{
    //Create a GenericCondition
    final GenericSearchField codeField = new GenericSearchField(ProductModel._TYPECODE, ProductModel.CODE);
    GenericCondition condition = GenericCondition.createConditionForValueComparison(codeField, Operator.EQUAL, product1.ge
    // Add this to a new GenericConditionList
    final GenericConditionList conditionList = GenericCondition.createConditionList(condition);
    // Create another GenericCondition and append it to this GenericConditionList
    final GenericSearchField nameField = new GenericSearchField(ProductModel._TYPECODE, ProductModel.NAME);
    conditionList.addToConditionList(GenericCondition.createConditionForValueComparison(nameField, Operator.STARTS_WITH,
                                "pRoduCt", /* upper */true));
    // Create a GenericQuery
    GenericQuery query = new GenericQuery(ProductModel._TYPECODE);
    // Add the condition list to the GenericQuery
    query.addCondition(conditionList);
    // Also add two GenericSelectField to the GenericQuery
    final GenericSelectField codeSelectField = new GenericSelectField(ProductModel._TYPECODE, ProductModel.CODE, String.cl
    final GenericSelectField nameSelectField = new GenericSelectField(ProductModel._TYPECODE, ProductModel.NAME, String.cl
    query.addSelectField(codeSelectField);
    query.addSelectField(nameSelectField);
    // Run the query
    Collection result = genericSearchService.search(query, new StandardSearchContext(ctxDe)).getResult();
    ...
}
```

GenericSearchQuery as Container for GenericQuery

GenericSearchQuery is a container object that allows to configure some additional search criteria and other options, like a user, for which the query is executed or pagination. The base **AbstractQuery** class keeps common functionality for both **GenericSearchQuery** and **FlexibleSearchQuery**. One of the **GenericSearchQuery** constructors provides a possibility to pass **GenericQuery** object as follows:

```
final GenericCondition idCondition = GenericCondition.equals(new GenericSearchField(CatalogModel._TYPECODE, CatalogModel.I
final GenericConditionList gl = GenericCondition.createConditionList(idCondition);
final GenericQuery query = new GenericQuery(CatalogModel._TYPECODE, gl);
final GenericSearchQuery gsq = new GenericSearchQuery(query);
gsq.setCount(5);
gsq.setUser(userService.getUserForUID("ahertz"));
gsq.setCatalogVersions(catalogVersionService.getCatalogVersion("clothes", "Online"));

final SearchResult<CatalogModel> searchResult = genericSearchService.search(gsq);
```

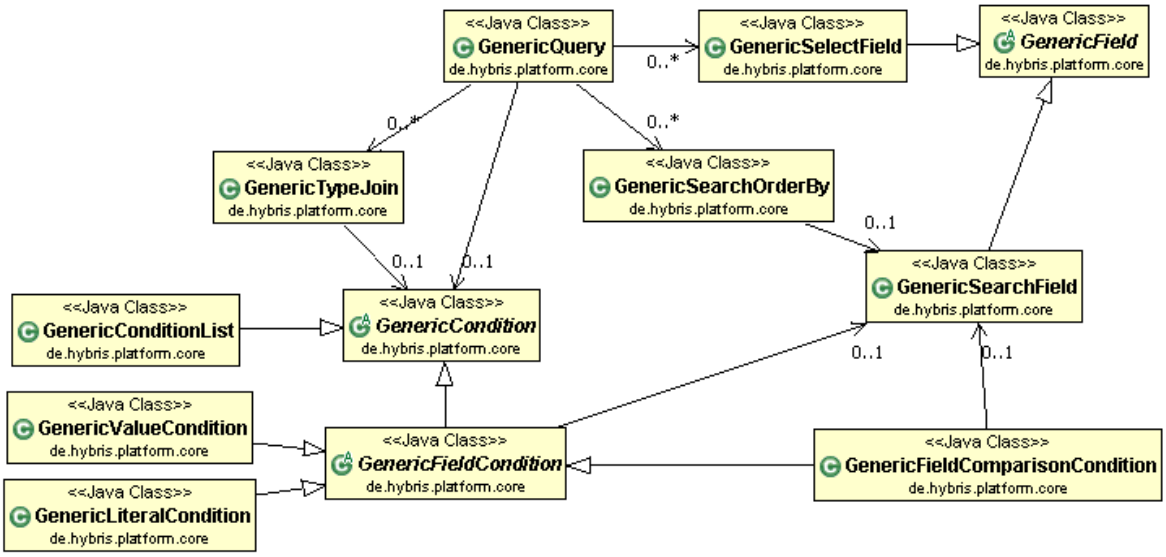
Behind the Scenes

Behind the scenes, the **GenericQuery** is ultimately converted into a **FlexibleSearchQuery** by the **toFlexibleSearch** method and than executed as FlexibleSearch query:

```
public String toFlexibleSearch(final Map valueMap){
      final StringBuilder sb = new StringBuilder();
      toFlexibleSearch(sb, null, valueMap);
      return sb.toString();
}
```

GenericSearch in UML

The UML diagram below illustrates the relationship between the main GenericSearch objects.



Related Information

How to Implement a Custom Search Provider
http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/querycriteria.html

Access Rights

Permissions services for SAP Commerce provide a framework that you can use to define and implement your own access rights policy. By using permissions services, you can define specific access rights for users and the user groups. You can manage the access right for types, attributes, or you can assign them globally by defining access rights for the users without referring to any existing types, items, or attributes.

Further in the document, both users and user groups are also called Principals. Also, the term permissions has the same meaning as user access rights or access rights.

Introduction

Permission is an abstract concept, sometimes also referred to as user right. Permission services don't define what a permission is. This means that the instructions for permissions services and results from permissions services need to be further processed by your own application. The meaning of the permission is something that must be defined by implementing a security policy for a particular system. Such particular security policy covers many more topics than just permissions. It also includes, for example, data encryption, type of credentials used, security auditing, and so on.

Permissions services don't automatically restrict access to items or types, rather they provide a framework for your own application to implement the correct response that reflects your particular access right security policy.

Permissions support you in deciding what kind of access users or user groups can have to the objects. Groups can contain other groups or other users. The groups or users that belong to the higher-level group inherit the access rights from the superior group that has the permissions assigned.

By defining access rights - in other words, by assigning permissions, you can specify a very detailed level of access to the types, items, and their attributes for a particular user or user group. Complex grading of access rights for an individual user or user group can be represented through a hierarchical structure. This is due to the fact that the access rights can be inherited from the groups higher in the hierarchy by the more specific principals lower in the hierarchy.

Permissions system determines how the access rights inheritance is going down through the several hierarchy levels. This way you can assign access rights to a high-level general group from where they're propagated down the hierarchy through the lower-level principals. It also helps you in predicting what kind of effective access rights is granted to the particular user or user group and how to resolve possible conflicts in case of mutually excluding access rights inherited from more than one superior group.

About Permissions Scope

Generally, you can assign permissions to types, items, and attributes. Additionally, you can assign global permissions to users.

Global Permissions

Related to the user or user group and is meant to have the lowest priority. It means that the global permissions assigned to the particular user/group can be overwritten by assigning permissions to the specific item, type, or attribute. The global permissions apply in case there are no other permissions defined.

Type Permissions

Apply to the entire type's data. Access rights to the type can be granted in several stages, for example Read, Change, Create, and Delete. However, permissions framework lets you define and use your own structure of permissions.

Item Permissions

Can be assigned to the particular instance of the previously defined type. Simply speaking, if you can create many items of some type, this could allow you to override type-related access rights and assign permissions for a certain user to the particular item.

Attribute Permissions

Can be granted or denied explicitly for the individual attributes holding the informational content of a type or an item. This can be used for even more refined access control.

It's also important to notice that permission assignments can be either positive (allowed or granted) or negative (disallowed or denied).

About Permissions Stages

There are a few access rights - in other words, permissions stages mentioned in the previous section. These stages used as an example are: Read, Change, Create, and Delete. However, the permissions services aren't limited to those listed stages. The framework is able to handle any kind of permissions that you may have created and implemented into your own application. The names and the number of permissions stages only depend on your particular needs for having a customized access rights policy for your users. Permissions services provide you a framework ready to support you with this task.

Depending on your security policy, it's possible to build a front-end application that can enforce certain limitations in using the available access rights. For example, you may wish to create such application where it isn't possible to perform any action on an attribute that belongs to the type to which the user has been assigned deny permission. In such a case, if user has no access to the type, then it isn't possible to perform any action to its attributes.

Access rights, however, aren't enforced by default by any of the Platform services. As a result, it's up to the particular application that uses permissions services framework to define such restrictions. It's therefore also possible to build such application that doesn't have such limitations.

The choice between these two options depends only on the security policy that the application owner wants to apply for the principals.

Permissions Priority

By creating access right policy, it's important to think about priority and in which order the permissions are applied to the principals: users and user groups. The general idea is that the most general permission has the lowest priority while the most specific permission has the highest priority. For example, if the user has been granted type permission: Read to the type while attribute permission: Read has been denied, then the user is able to see all the type attributes except the particular one that has been denied.

Another example would be a user that belongs to two different groups, one of which is granted a read permission to a type, while the other group is denied the same permission for the same type. In result of permissions inheritance the user inherits both permissions, however, the effective permission for the user is deny. Simply speaking, while on the same level in the inheritance hierarchy, deny settings rank higher in priority than grant settings.

ℹ Note

Access rights work differently in the Platform and the Backoffice Framework. A key difference is that in the Backoffice Framework, attribute level permissions don't override type-level permissions, which is possible in Platform.

Global Permissions

Global Permissions are assigned to the particular user or user group - in other words, to principals. As they don't relate to any particular type, object, nor attribute, they provide the most general kind of access rights to the users. Any further and more specific permission assignment like type-, item-, or attribute permission overrides the global permissions regarding the particular type, item, or attribute.

Type Permissions

Type-related permissions define the user access rights to the type. The user with assigned permissions to the type also has default access to all type attributes. This is valid unless more specific permissions have been assigned. Type permissions can be assigned to grant or deny certains actions, for example: Read, Change, Create, and Delete Type-level permissions should be a starting point in building security policy, since they're simple to define and cheap to evaluate (low-performance penalty for checking).

Attribute Permissions

Using attribute permissions allows you to explicitly assign permission to selected attributes of a type. Attribute-related permissions override type-related permissions. For example, if the type permission for a type is set to grant: *<Change>*, you can deny permission: *<Change>* to any of the attributes of that type.

Item Permissions

Using item permission assignments allows you to define very fine-grained access control in that you're able to grant or deny permissions to item instances. However, it's best to assign permission at a type level using principal groups as managing and checking permissions at item level can considerably degrade performance if a huge number of items is involved. On the other hand, such mechanism might be useful in many scenarios, such as overriding defaults or securing particularly important item instances.

Note

Item permissions are based on permissions defined for catalog versions: even if two catalog-version-aware items are of the same type, they might have different item permissions if they belong to different catalog versions with different read or write restrictions.

Assigning Permissions

You can assign permissions - in other words, create access rights for a principal by:

- Using the Permissions Service API: You can use Permissions Service API for enriching your application with user access rights management features.
- Using ImpEx: For details, see Legacy Permissions in ImpEx Scripts.

Note

Using ImpEx scripts is now restricted only to legacy permissions: Read, Change, Create, and Delete. There's no general-purpose syntax for importing arbitrary permissions. You can overcome this limitation by providing BeanShell scripts in the ImpEx file that can directly call permissions services. However, this solution is neither elegant nor easy to write. Read the next section to investigate the legacy permissions that are compatible with the ImpEx scripts syntax.

Legacy Permissions in ImpEx Scripts

With the permissions services framework, you can design your own access rights names for your front-end application. However, due to the legacy dependency from the ImpEx scripting feature, it's worth to mention the following access right names: Read, Change, Create, Delete. These names were used to provide compatibility with the syntax of the ImpEx scripts used to import data to the SAP Commerce type system.

You can still use the same terminology, however, permissions services system doesn't depend on any particular front-end application and doesn't enforce any particular naming convention for your access rights. It depends entirely on you what kind of access rights names you create to use in your application.

The Read, Change, Create, Delete access rights names are clear and self-describing, and are used in this document as an example.

For more information on user rights in ImpEx, see ImpEx API.

Read

This permission specifies whether the user account is granted access to the respective type. The type's attributes are displayed, but no actions are possible at this access level. If this permission is set to: deny, then the user account cannot:

- See instances of this type
- Read the values for the type's attributes

The type permission: Read applies to all attributes of the type unless explicitly overridden.

Change

This permission specifies whether the user account is allowed change the value of the type - in other words, modify all attributes of the type. The Change permission is different from the Create and Delete. Being assigned the Change permission doesn't allow users to create or delete instances of a type, such as new products or delete them.

You can also deny a user the type-related Change access right. The permission check result provides you with a feedback information that you can use to forbid users the access to the particular type.

The type-related Change permission applies to all attributes of the type unless explicitly overridden.

Create

The create permission assignment to the type specifies whether the user is allowed to create instances of the type, for example, creating individual products, customers, or system languages. A user account with granted Create access right but denied Change access right can create instances of the type, but can't set any of the attributes of the type. In result, the user account can only create type instances with no attribute values set at all. By consequence, there's usually no reason to assign Create permission to the user but to deny the Change access right at the same time.

Delete

The delete permission specifies whether the user has the right to delete items of the type. For example, using this permission you can decide if deleting a product of a certain type is allowed or not. In some cases, type permission Delete may not suffice to delete SAP Commerce items of that type. Deleting may not be possible due to, for example:

- Other required access rights not being granted
- The user account not having sufficient overall rights
- Restrictions applying to the user account.

For example, to delete a product from the catalog user account needs the Delete access right and Change access right for catalog version.

Change_perm

The change_perm permission specifies whether the user has the right to grant permissions to other users.

Changing Permission Assignment During Runtime Operation

Changing permissions assignment for users and user groups takes immediate effect internally. If your application assigns permissions to a user, then the permissions services immediately return feedback information to you, ready to be handled by your application's user rights management system. As a result, even those users who are logged in into your application by the time of changes are affected and if a user is logged in while the user's access rights are changed, then the changed access rights are immediately effective.

However, if your application has, for example, browser-based user interface, then part of the pages may actually be kept in the browser memory (browser cache). If the user opens such a page before the related permissions have been assigned, then the state of the page may not change automatically, even if the new permissions assignment is already in effect. The similar effect is current for any web-based access interface to the application. After the permissions have been changed some elements may no longer be editable, but it's first clear to the users after they make an attempt to modify attribute values for an item or type through web interface. Usually, refreshing the application to let it apply newly assigned permissions solves this uncertainty.

Generally, it's advisable to avoid confusion and not to modify permission assignments when users are logged in. If you need to change user access rights when SAP Commerce is in live operation, use a scheduled maintenance time window to assign new permissions.

Effective Access Rights for a User Account in a User Group

Users can belong to user groups. If a user group is assigned permissions, then all the users belonging to that group inherit the same permissions. However, it may also happen that the user belongs to several groups having several, sometimes mutually exclusive, permissions. In this case, users inherit these permissions, but the mechanism is in place that tries to resolve possible conflicts and provide the proper information about the effective permissions. Effective permissions aren't those that are assigned but those that apply to the specific user or user group based on the permissions assigned directly and inherited from the groups located higher in the hierarchy.

It's a common approach that the permissions aren't assigned to user accounts directly, but to the user group instead. All members of that user group, both users or other user groups, automatically inherit those access rights. This way access rights can be inherited within a complete user group hierarchy. Subgroups and individual user accounts in such a hierarchy may override their inherited permissions assignments, expanding or restricting the range of effective access rights. Generally, if we consider the permissions of a user group, the permissions assigned to the groups closest and upwards in the hierarchy will be the ones that are valid for that point.

The final result depends also on what kind of permissions are assigned to the groups higher in the hierarchy. If, for example, access is explicitly granted by one user group and explicitly denied by another user group on the same hierarchy level, then the access is denied for:

- Subgroup inheriting those permissions,
- User that belongs to both groups.

> **Note**
>
> There are two exceptions in the system of access rights:
>
> - User accounts that belong to the admingroup user group.
>
>   This also applies to user accounts that are inherited members of admingroup, that is, members of a subgroup of admingroup.
> - User accounts with uid: admin.
>
> A user account that is a member of the admingroup user group or users with uid: admin always have full access rights. Read more in Restrictions.

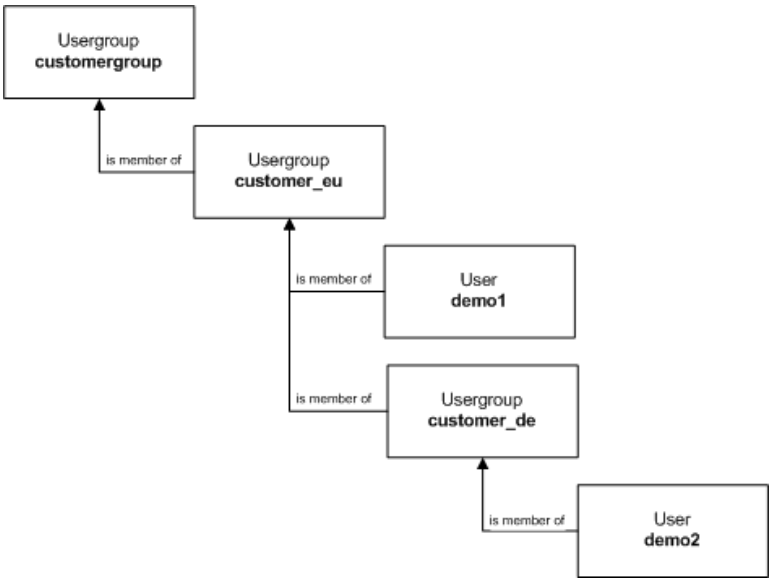
User Account as Member of Vertical User Group Hierarchy

The effective permissions for a particular user are those permissions that are assigned to the user group closest to that user account within a user groups hierarchy:

- If permissions are assigned to an individual user account, then those permissions are considered as closest ones to the user account in the inheritance hierarchy, and therefore are effective.
- If the user group has no permissions assigned, then the permissions assigned to the closest user group one level up in the user group hierarchy are valid.
- If no permission has been assigned throughout the user group hierarchy, then access right defaults to deny.

Example

The user group customergroup contains the user group customer_eu. The customer_eu usergroup contains the User demo1 and the usergroup customer_de. The customer_de usergroup contains the User demo2.



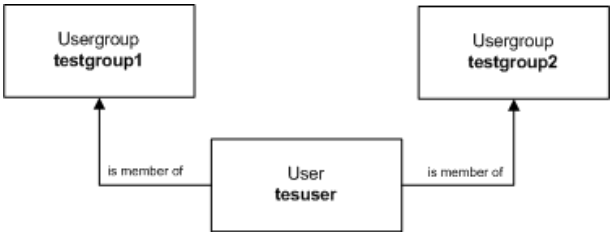
Permission Setting for customergroup	Permission Setting for customer_eu	Permission Setting for customer_de	Effective Access Right for demo1	Effective Access Right for demo2
granted	(no setting)	(no setting)	granted	granted
denied	(no setting)	(no setting)	denied	denied
granted	(no setting)	denied	granted	denied
denied	granted	(no setting)	granted	granted
denied	granted	denied	granted	denied
(no setting)	granted	denied	granted	denied
(no setting)	(no setting)	granted	denied (default)	granted

User Account as Member of Several Individual User Groups

If user account is a member of several user groups not forming a hierarchy, then all permissions assigned to these user groups apply by inheritance to the user account. Again, deny setting for access rights rank higher than grant setting for access rights. It means that if the same access right (for example: read, change) is defined in a conflicting manner by parent user groups being on the same hierarchy level, then the deny setting rank higher than grant setting.

Example

The user testuser is a member of both testgroup1 and testgroup2.



Permission Setting for testgroup1	Permission Setting for testgroup2	Permission Setting for testuser	Effective Access Right for testuser
granted	(no setting)	(no setting)	granted
denied	(no setting)	(no setting)	denied
granted	(no setting)	denied	denied
denied	granted	(no setting)	denied
denied	granted	denied	denied
(no setting)	granted	denied	denied
(no setting)	(no setting)	(no setting)	denied (default)

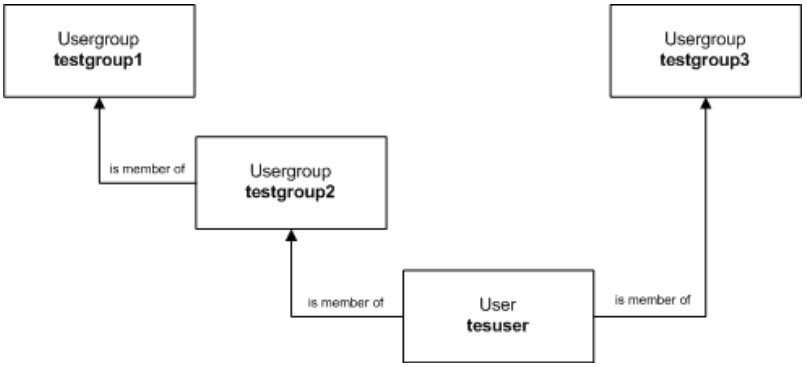
User Account as Member of User Groups Which Form More Than One Hierarchy

In this section, we have a combination of the two following situations:

- User account belongs to several user groups
- Some of the user groups form a hierarchy

Example

The example shows a user account as a member of two user groups, one of which is part of a user group hierarchy. Deny access rights rank higher than grant access rights. If the same access right is defined in a conflicting manner by user groups that are on the same hierarchy level, then the deny access rights rank higher than grant access rights.



Permission Setting for testgroup1	Permission Setting for testgroup2	Permission Setting for testgroup3	Permission Setting for testuser	Effective Access Right for testuser
granted	(no setting)	(no setting)	(no setting)	granted
denied	(no setting)	(no setting)	(no setting)	denied
granted	(no setting)	denied	(no setting)	denied
denied	granted	(no setting)	(no setting)	granted
denied	granted	denied	(no setting)	denied
(no setting)	granted	denied	granted	granted
(no setting)	(no setting)	granted	(no setting)	granted
(no setting)	(no setting)	(no setting)	(no setting)	denied (default)

ℹ Note

The inheritance mechanism considers the closest inheritance levels.

Related Information

Permissions in The Backoffice Framework

Managing and Checking Access Rights

You can use the permission management service to manage access rights in SAP Commerce. Additionally, the permission checking service allows you to check the existing permission assignments on a particular user or user groups.

Managing and Checking Access Rights

The permissions services is a framework that can be used to implement access rights system on the SAP Commerce Platform. By using the permissions services framework you can manage and check the access rights for users and user groups. You can assign, modify or remove permissions related to your users like employees and customers. You can also manage and check access rights assigned for the user groups. The framework has separate services for managing the access rights and checking what kind of access rights are assigned.

For example, you may assign the different access rights for your employees who manage SAP Commerce and for the customers who visit your web shop. The employees who administer SAP Commerce need to assign, modify, or remove permissions. For this task, use the `PermissionManagementService`. On the other hand, customers who visit your web shop do not need to manage permissions and there is no need to give them access to the `PermissionManagementService`. However, you may need to use the service that checks what the customer can see in your web shop. This is the task for `PermissionCheckingService`.

Access Rights should correspond to your security requirements, data model and organizational structure, and must be defined during design of your application, as a part of a security policy.

In other words, you need to design and implement access rights system used by your application.

Permission Services

Permission services described in this document focus on the permission assignment, which in most cases is a relation between:

- Permission: Distinguished by its name
- Principal: User or user group
- Object: Item, type or an attribute to which the permission is assigned

The only exception are the global permissions, that are only assigned for the users. They should apply in those situations when no other, more specific access rights, are assigned to the items, types or attributes.

The usual way to use permissions in the system is:

- Assigning permission to user groups
- Checking permissions for single users

However, permission services allow also to assign permissions for the single users and check permissions for the user groups, if needed. This is because the methods in permission services take a `PrincipalModel` argument, which can be either a user or usergroup.

PermissionManagementService

Responsible for managing permissions and permission assignments. Possible operations supported by this service are:

- Creating new permissions
- Listing available permissions
- Assigning permissions
- Replacing permissions
- Removing permissions

PermissionCheckingService

`PermissionCheckingService` is used to check the effective permission assignments, in other words: effective user access rights.

PermissionCRUDService

This is a convenience service that is a wrapper over `PermissionCheckingService` and provides checking operations for the four basic platform-defined permissions: Create, Read, Update, Delete (since CRUD).

Other

- `PermissionCheckResult`: Provides information whether the permission is granted or denied

- `PermissionCheckValue`: Simple enumeration of defined permissions used by other services
- `PermissionsConstants`: Contains a list of pre-defined permissions
- `PermissionAssignment`: Object used by `PermissionManagementService` to return the information about the permission for the particular user or users group.

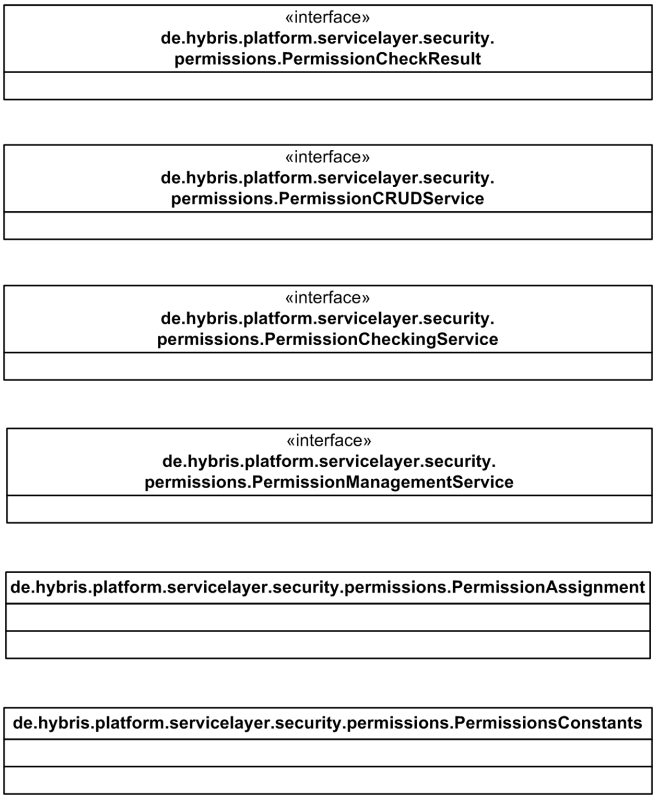


Figure: Permissions Services.

PermissionManagementService

PermissionManagementService is used to define and manage permission assignments. There are two basic operations for manipulating permissions without doing any assignments:

- Create a new permission

```
void createPermission(String permissionName);
```

- Return a collection containing names of all defined permissions:

```
Collection<String> getDefinedPermissions();
```

There are no means to delete the permission, once it is created. Such an operation would require checking the entire database to ensure that the permission is not used anywhere. This could also lead to a hole in the security system. Since permissions are only distinguished by name, removing a permission and then adding another one with the same name makes the existing security audit logs useless.

Creating Permission Assignments

To create the permission assignment, use `PermissionAssignment` object, which provides information whether the permission is granted for a principal, or denied. The principal is usually a usergroup, but it can be also a user. The following code snippet provides an example on how to grant permission A_PERMISSION to an item aCountry for a group aGroup.

```
//create a permission
permissionManagementService.createPermission("A_PERMISSION");

<!-- ... -->

final CountryModel aCountry = ...
final PrincipalModel aGroup = ...

//create a permission assignment object which grants A_PERMISSION for aGroup
final PermissionAssignment permissionAssignment = new PermissionAssignment("A_PERMISSION", aGroup);
```

```
//create permission assignment to aCountry.
permissionManagementService.addItemPermission(aCountry, permissionAssignment);
```

The next example shows how to deny A_PERMISSION to an item aCountry for an user anUser. The second constructor of `PermissionAssignment` class is used here to take explicit value for denied flag.

```
//create a permission
permissionManagementService.createPermission("A_PERMISSION");

<!-- ... -->

final CountryModel aCountry = ...
final PrincipalModel anUser = ...

//create a permission assignment object which denies A_PERMISSION for anUser
final PermissionAssignment permissionAssignment = new PermissionAssignment("A_PERMISSION", anUser, true);

//create permission assignment to aCountry.
permissionManagementService.addItemPermission(aCountry, permissionAssignment);
```

There is also a `addItemPermissions()` method that enables to add multiple permission assignments to an item in one invocation.

In the same way permission can be assigned to:

- Attribute descriptors: By using `addAttributePermission()` or `addAttributePermissions()` methods
- Item types: By using `addTypePermission()` or `addTypePermissions()` methods

Global permission assignments are even simpler. To create a global permission there is no need for any object. In the code sample, note `PermissionAssignment` constructor that takes explicit value of denied flag as false which means that permission is granted.

```
//create a permission
permissionManagementService.createPermission("A_PERMISSION");

<!-- ... -->

final PrincipalModel aGroup = ...
//create a permission assignment object which grants A_PERMISSION for aGroup
final PermissionAssignment permissionAssignment = new PermissionAssignment(TEST_PERMISSION_1, aGroup, false);

//assign the permission.
permissionManagementService.addGlobalPermission(permissionAssignment);
```

Retrieving and Removing Permission Assignments

The methods for getting and removing permission also use `PermissionAssignment` object. When retrieving this object provides the information about existing permission assignments. When removing, this object provides information which assignment is to be removed.

For details about the available methods, see API documentation for `PermissionManagementService` class.

PermissionCheckingService

Permission checking service provides the answer to questions of the following form:

- Is a permission to an object granted for a principal?
- Is a permission granted for a principal globally?

The answers is returned for:

- Given permission
- Principal
- Object
- Item
    - Item type
    - Attribute descriptor

The answer is calculated out of the existing permission assignments created with the `PermissionManagementService` using permission checking rules. This way the answer provides information about actual permission assignments, that means, the permissions that were directly assigned.

The combination of actual permission assignments and permission checking rules produce effective permission assignments. Effective permission assignments might not physically exist in the system, but are the effect of using checking rules. For example, if there is an actual permission assignment to a user group, then

every member of the group has that permission effectively assigned. This happens because user group permissions are inherited by users who belong to the group. Permission checking rules take into account permission inheritance within principal group hierarchy and inheritance within type and attribute descriptor hierarchy. The rules also prioritize different kind of permission assignments. For example, global assignments have the lowest priority. To see more detailed description, check the API documentation for `PermissionCheckingService`.

The result of checking permission assignment can be one of the following constants defined by `PermissionCheckValue`:

ALLOWED

Permission is granted for a principal

DENIED

Permission is explicitly denied for a principal

NOT_DEFINED

No actual permission assignment was found for the given permission checking operation

CONFLICTING

Situation when there are both DENIED and ALLOWED equal-priority assignments for a principal

The methods of `PermissionCheckingService` do not directly return `PermissionCheckValue` contants. When checking, it is much more convenient to get a single boolean `yes` or `no` answer. For that purpose a `PermissionCheckResult` interface is used as a result type. This interface gives acces to the raw checking results represented by a `PermissionCheckValue` enumeration. However, it also provides two convenience methods: **isGranted()** and `isDenied()`, which map `PermissionCheckValue` constants onto a boolean value. This mapping can be changed by using different `PermissionCheckValueMappingStrategy`. By default this strategy assumes that:

- Only ALLOWED maps to `true`
- Other values map to `false`

Use Cases

This section contains examples on checking permission for some most expected use cases. To fully understand these examples, check API documentation for `PermissionCheckingService` to find more details about permission checking rules.

Assume that:

- There is a permission `READ_CATALOG` defined
- User user1 is a member of group1
- The following objects are defined:

```
CatalogModel catalog1 = ...
CatalogModel catalog2 = ...
PrincipalModel user1 = ...
PrincipalModel group1 = ...
```

Checking Item Permissions within Principal Group Hierarchy

Code snippet shows how permissions to an item within principal group hierarchy can be checked.

```
boolean result;

//The actual result is NOT_DEFINED, since no permission assignments has been done so far.
result = permissionCheckingService.checkItemPermission(catalog1, user1, "READ_CATALOG").isGranted(); //false
result = permissionCheckingService.checkItemPermission(catalog2, user1, "READ_CATALOG").isGranted(); //false

/*
 * Let's deny "READ_CATALOG" globally for group1
 */
permissionManagementService.addGlobalPermission(new PermissionAssignment("READ_CATALOG", group1, true));

//The actual result is now DENIED because of global "READ_CATALOG" denial for group1 and principal group inheritance.
result = permissionCheckingService.checkItemPermission(catalog1, user1, "READ_CATALOG").isGranted(); //false
result = permissionCheckingService.checkItemPermission(catalog2, user1, "READ_CATALOG").isGranted(); //false

/*
 * Let's grant "READ_CATALOG" for group1 to catalog1
 */
permissionManagementService.addItemPermission(catalog1, new PermissionAssignment("READ_CATALOG", group1));

//The result is now ALLOWED, although there is global denial for group1. This is because global assignments
//have lower priority.
result = permissionCheckingService.checkItemPermission(catalog1, user1, "READ_CATALOG").isGranted(); //true

//The result is still DENIED, because no permission assignment has been done to catalog2 to override global
```

```
// "READ_CATALOG" denial for group1.
result = permissionCheckingService.checkItemPermission(catalog2, user1, "READ_CATALOG").isGranted(); //false
```

Assigning Type Permissions

Building on the example in the previous section, permission assignments to the type can be used as shown in the code sample below.

```
ComposedTypeModel catalogType = ...

permissionManagementService.addTypePermission(catalogType, new PermissionAssignment("READ_CATALOG", group1));

//The result is still ALLOWED, because of permission assignment to catalog1. Type permission assignment is
// of lower priority,
// so it does not change anything.
result = permissionCheckingService.checkItemPermission(catalog1, user1, "READ_CATALOG").isGranted(); //true

//The result is now ALLOWED, because permission assignment to catalog type overrides global "READ_CATALOG"
//denial for group1.
result = permissionCheckingService.checkItemPermission(catalog2, user1, "READ_CATALOG").isGranted(); //true
```

For more sophisticated examples, check the `PermissionCheckingServiceTest`, where the permission checking rules are tested for various scenarios.

Best Practices and Tips

- Permissions are not automatically enforced. This means that your code must invoke a method or two from **PermissionCheckingService** and then act accordingly in order to have a permission-based security.
- By default, everything is denied.
- Assign permissions for the user groups. Every member of the group inherits the permission assignments. This work across the group hierarchy.
- Assign permissions to the types. Every item of that type will inherit this permission assignments. This works across type hierarchy.
- Use attribute descriptor assignments for fine-grained control of the attribute values.
- Use global permission assignments as a `fall-back` or `default` assignment. The global permissions have the lowest priority and are always overriden by the item, type, or attribute permissions assignments.
- Design your permissions schema and keep it simple.

Known Limitations

- You cannot search for the effective permission assignments using Flexible Search like: return all items that have a permission XYZ granted for principal P. The reason is that the effective permission assignments are calculated, and there is no easy way for the database server to perform such calculations when retrieving item rows from the database. This also means that permissions cannot be used for filtering rows in the data access layer.
- Since permission assignments are currently not represented by models, you cannot directly import them using the ImpEx scripts. The only way is to use `PermissionManagementService` from ImpEx using scripting.

Related Information

Users in Platform

Cross-Origin Resource Sharing Support

SAP Commerce supports the Cross-Origin Resource Sharing mechanism. The CORS mechanism defines a way for a browser and a server to decide which cross-origin requests for restricted resources can or cannot be allowed.

Imagine a script on `http://www.example.com/somepage.html` wanting to access resources from `https://www.example.com:87/resources`. Your browser considers these two addresses as two different origins and it will prevent the `/somepage.html` script from fetching the resources from `/resources`. It is possible to safely relax this limitation by preparing CORS configurations in such a way that the `/somepage.html` request will be accepted.

Enabling CORS Support in SAP Commerce Extensions

To enable CORS support in SAP Commerce extensions, include these sections in the `[extname]-web-spring.xml` files of chosen extensions.

extname-web-spring.xml

```
<bean id="extnamePlatformFilterChain" class="de.hybris.platform.servicelayer.web.PlatformFilterChain">
  <constructor-arg>
    <list>
      <ref bean="corsFilter"/>
      [...]
```

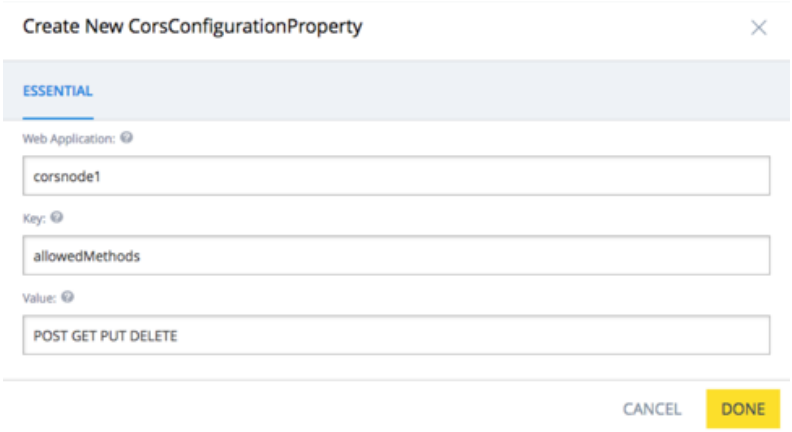
```
    </list>
  </constructor-arg>
</bean>
```

Configuring CORS in SAP Commerce Web Applications

Global Cluster Configuration

The CORS configuration is stored in the database inside `CorsConfigurationProperty` items. It is global and applies to all nodes connected to a cluster.

To edit your configuration, in Backoffice go to System > CORS Filter > CorsConfigurationProperty and create or modify instances of the `CorsConfigurationProperty` type.



As a context use the name of your extension - we use `webAppName` in this document as an example. As a key, use one of the available properties.

CORS Header	CorsConfigurationProperty	Default Value	Description
`Access-Control-Allow-Origin`	`allowedOrigins`	`null`	List of hosts that can receive CORS response from your extension.
`Access-Control-Allow-Methods`	`allowedMethods`	`GET HEAD`	List of the supported HTTP methods.
`Access-Control-Allow-Headers`	`allowedHeaders`	`null`	The names of the supported author request headers.
`Access-Control-Expose-Headers`	`exposedHeaders`	`null`	List of the response headers other than simple response headers that the browser should expose to the author of the cross-domain request.
`Access-Control-Allow-Credentials`	`allowCredentials`	`null`	Indicates whether the user credentials, such as cookies, HTTP authentication, or client-side certificates, are supported.
`Access-Control-Max-Age`	`maxAge`	`null`	Indicates how long the results of a preflight request can be cached by the web browser, in seconds. If `-1` - unspecified.
`Access-Control-Allow-Patterns`	`allowedOriginPatterns`	`false`	Alternative to `allowedOrigins` that supports more flexible origins patterns with "*" anywhere in the host name in addition to port lists.

ℹ Note

It is not allowed to use the combination of the `allowCredentials=true` and `allowedOrigins=*` attributes. Use `allowedOriginPatterns` instead.

Use xss filter to set the header

Add `Access-Control-Allow-Origin` in the response header as,

```
xss.filter.header.Access-Control-Allow-Origin=<#domain name#>
```

You must replace the domain name with the actual domain. Enter * to allow access from all domains.

Local Node Configuration

To configure CORS in a SAP Commerce extension, use these properties:

project.properties

```
corsfilter.webAppName.allowedOrigins=http://localhost:8080
corsfilter.webAppName.allowedMethods=GET PUT POST DELETE
```

> ⚠ **Caution**
>
> Properties from the database take precedence before local properties. If a property with the same context and key is configured both in the database and the `properties` file, the value is taken from the database.

Default Configuration

When no CORS properties are defined, all CORS requests are rejected.

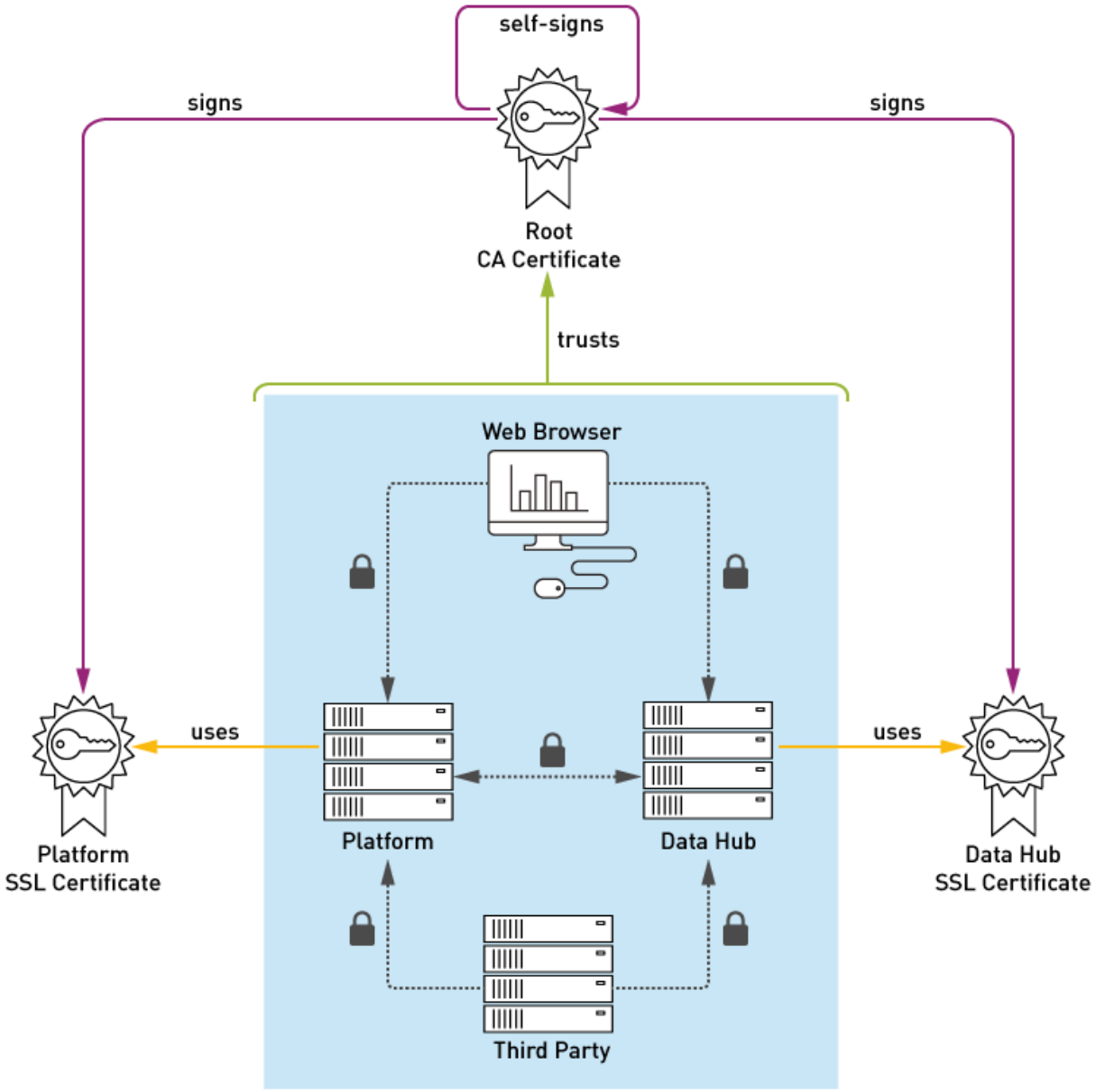
HTTPS Traffic Certificate

SAP Commerce handles the HTTPS traffic using a certificate signed by a self-signed RootCA.

> ⚠ **Caution**
>
> Since the private key is known, this approach is intended only for the development environments.

Platform trusts this RootCA by default so it is possible to access an HTTPS resource from Platform itself.

The following diagram shows how the approach works:



Sample SSL Certificates for Developer Environments

Sample SSL certificates are provided for developer environments and allow you to use localhost only. Those certificates should be disabled in production environments. You can disable them by setting empty values to the following properties in your `local.properties` file:

```
additional.javax.net.ssl.trustStore=
additional.javax.net.ssl.trustStorePassword=
```

Implementing Encoding Algorithms with the Generic Password Encoder

With the Platform, you can implement additional **PasswordEncoder** interfaces by using the **GenericSaltedPasswordEncoder** class. This feature enables you to add additional encoding algorithms into the Platform without the need to manually create a custom class for your algorithm.

Introduction

By default, the Platform implements the **MD5**, **SHA-256** and **SHA-512** algorithms. If you want, you can use the new feature to add any of the Java supported **MessageDigest** algorithms. For details, see Java Cryptography Architecture Standard Algorithm Name Documentation .

Implementing MessageDigest Algorithms

To implement an algorithm, in the **core-spring.xml** file configure the **GenericSaltedPasswordEncoder** class as a spring bean with the **algorithm** parameter. Use **genericPasswordEncoder** as a parent for that bean. As the parameter's value set one of the available **MessageDigest** algorithms.

Use the following example of implementing the **SHA-384** algorithm to implement other **MessageDigest** algorithms:

1. In the **`platform/ext/core/resources/core-spring.xml`** file, declare a bean and set "**SHA-384**" for the **value** attribute :

```
<bean id="sha384PasswordEncoder" parent="genericPasswordEncoder">
                  <property name="algorithm" value="SHA-384"/>
                  </bean>
```

2. Add this entry into the **encoders** map in the **core.passwordEncoderFactory** bean:

platform/ext/core/resources/core-spring.xml

```
<bean id="core.passwordEncoderFactory" class="de.hybris.platform.persistence.security.PasswordEncoderFactoryImpl">
                  <property name="encoders">
                  <map>
                  <!-- ... -->
                  <entry>
                  <key>
                  <value>sha-384</value>
                  </key>
                  <ref bean="sha384PasswordEncoder" />
                  </entry>
                  </map>
                  </property>
                  </bean>
```

You must restart the Platform to apply the changes.

⚠ Caution

The MD5 Message-Digest Algorithm is not considered strong anymore.

Password Autocomplete

The autocomplete function gives you suggestions while you type your login and password into the field. It allows you to quickly provide the credentials for different users, for example while testing your software.

Password Autocomplete in SAP Commerce

The login and password autocomplete feature is disabled on a global level in SAP Commerce. However, on the cockpit level it is turned on, meaning that the global settings are overwritten by these ones specified in the extension `project.properties` file. You can find an example of login and password autocomplete setting enabled in the `project.properties` file for SAP Commerce Product Cockpit. The `project.properties` file is located in the `${HYBRIS_BIN_DIR}/modules/cockpit-applications/deprecated/productcockpit/` folder.

```
productcockpit.default.login=<login>
productcockpit.default.password=<password>
```

Disabling Password Autocomplete

For security reasons, you might want to disable Password Autocomplete. Do it on the cockpit level by deleting the values for the following properties in your extension `project.properties` file:

```
# Default login and password for logging into your cockpit:
<yourcockpitname>.default.login=
<yourcockpitname>.default.password=
```

Password Change Auditing

The password change auditing feature lets you register all the changes introduced to your passwords.

You can track changes such as the time a password changed, or who changed it. Additionally, password change auditing lets you store the hash and the encoding type of the previous password. Data about changes is stored in `UserPasswordChangeAudit` items.

Enabling Password Change Auditing

To enable password change auditing, set the `user.audit.enabled` property to true:

```
# enables user audit of password changes
user.audit.enabled=true
```

On the model service side, it is the `UserPasswordChangeAuditPrepareInterceptor` that provides the logic responsible for recording password changes. On the Jalo side, the logic is executed inside the `setPassword` method of a `User` object.

Extending Scope of Audit

To extend the scope of audited information, you need to extend the `UserPasswordChangeAudit` class with the additional attributes you want to store. To set values for these attributes, you can use the `processAfterCreation` extension point of `UserAuditFactory`. The first step is to extend `de.hybris.platform.servicelayer.user.impl.UserAuditFactory`, and override the `processAfterCreation` method. Then, you must overwrite the `userAuditFactory` bean.

Password Security Policies

SAP Commerce Platform provides customizable extension points that enable you to have fine control of password handling. With password security policies, you can define requirements that your passwords must meet before you can set or change them. This feature tremendously contributes to increasing Platform security.

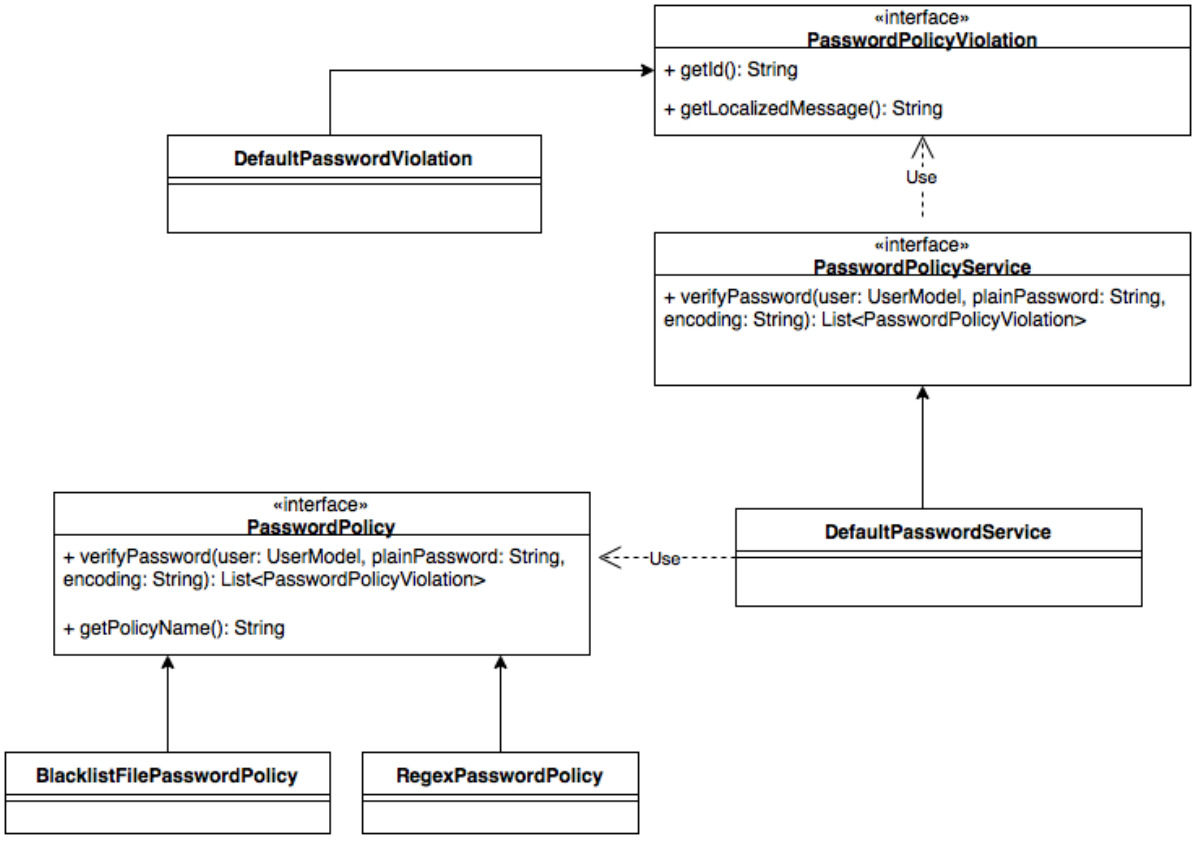
Regex and **blacklist** are two out-of-the-box password security policies delivered with Platform. Regex prevents users from using passwords that could be too easy to crack. Blacklist bans the most common passwords. You can implement your own policies too.

ℹ Note

SAP Commerce Platform automatically handles the password security policies for Backoffice. If you want to enable this feature for customers, implement appropriate password policy handling in your storefront.

Implementation

The following diagram shows classes used to implement password security policies:



The `PasswordPolicyService` interface provides the `verifyPassword` method that checks whether your password matches defined policy requirements. If your password doesn't violate the requirements, an empty collection is returned. If it does violate, `verifyPassword` returns a list with `PasswordPolicyViolation` objects. Each violation has its unique id. You can obtain those ids calling the `getId` method.

`PasswordPolicyService` is used every time you set a new password, no matter if via Service Layer, or Jalo. In Service Layer, you change a password by calling the `setPassword` method on the `UserService` bean. `DefaultUserService` delegates a check to `PasswordPolicyService`, and throws `PasswordPolicyViolationException` if a new password doesn't fulfill requirements. The same check takes place inside the `setPassword` method on a `User` Jalo object. Nevertheless, the recommended way to validate user input for a new password is to call `verifyPassword` explicitly, and set the password only if no violations are returned.

The `PasswordPolicyViolation` interface provides the `getLocalizedMessage` method that returns localized violation messages. You can use these messages to provide feedback for users in the view layer.

Assigning Password Policies to User Groups

Password policies aren't enabled for any user group unless you configure them. To enable an example policy named foo for user groups group1 and group2, set the following property as shown:

```
password.policy.foo.groups.included=group1,group2
```

You can also use the * character as a wildcard - it matches all user groups and thus enables a given policy for everyone in the system. To exclude users when using a wildcard, use the `excluded` property:

```
# match all groups except excluded_group for foo policy
password.policy.foo.groups.included=*
password.policy.foo.groups.excluded=excluded_group
```

Available Password Policies

Platform comes with two out-of-the-box implementations of password policies:

Password policy name	Implementation class	Bean	Description
regex	de.hybris.platform.servicelayer.user.impl.RegexPasswordPolicy	regexPasswordPolicy (user-spring.xml)	Matches a set of required and disallowed regular expressions defined in properties against a password
blacklist	de.hybris.platform.servicelayer.user.impl.BlacklistFilePasswordPolicy	blacklistPasswordPolicy (user-spring.xml)	Checks password against a list of prohibited passwords defined in a text file available on classpath - pointed to by the `password.policy.blacklist.file` property in platform/project.properties.

Regex Password Policy

The regex password policy allows you to configure a set of rules based on regular expressions. For example, you can decide that your passwords are at least 8 characters long and include at least one digit. These rules can either be required or disallowed:

Type	Property	Description
Required	`password.policy.regex.required.[rule_id]`	Regular expression must match against a new password
Disallowed	`password.policy.regex.disallowed.[rule_id]`	Regular expression must not match against a new password

Replace the `[rule_id]` placeholder with an assigned unique password violation id. This text is returned by the `getId` method. It's also used to find proper localization text for the `getLocalizedMessage` method.

See the example of defining regex rules:

```
# enforces passwords to be between 6 and 128 characters
password.policy.regex.required.certainsize=.{6,128}

# enforces passwords to contain digit
password.policy.regex.required.mustcontaindigit=.*\\d+.*

# prohibits from setting '12345678' as a password
password.policy.regex.disallowed.not12345678=12345678
```

To provide localization texts for these rules, create `password.policy.violation.regex.required.[rule_id]` or `password.policy.violation.regex.disallowed.[rule_id]` keys with translation in `resources/localization/[extension]-locales_en.properties` bundle.

See how you can provide localization for rules defined in the previous example:

```
password.policy.violation.regex.required.certainsize=Password must contain between 8 and 128 characters.
password.policy.violation.regex.required.mustcontaindigit=Password must contain at least 1 digit.
password.policy.violation.regex.disallowed.not12345678=Using '12345678' as the password is not allowed.
password.policy.violation.blacklist.blacklistedpassword=Selected password is a blacklisted.
```

Blacklist Password Policy

Blacklist password policy allows you to specify a file that contains a list of passwords (one password per line) that aren't allowed. To specify a blacklist file name, set the `password.policy.blacklist.file` property with a name of the file available on the classpath.

For example, this setting allows you to use the `password-blacklist.txt` file stored in an extension `resources` folder:

```
password.policy.blacklist.file=password-blacklist.txt
```

Creating Custom Password Policies

To create a custom password policy, create a bean that implements the `PasswordPolicy` interface. `DefaultPasswordPolicyServices` uses autowiring by type so it detects a custom implementation available to use. Assign your policy to appropriate groups. A policy name is a string returned by the `getName` method. Remember to obey the contract and provide password policy violations with correct translations.

Password Storage Strategies

When saving user passwords, it is important to protect them against unauthorized access. For that reason, Platform always stores passwords in an **encoded** format.

You can choose from multiple encoding strategies (known as **password encoders**) available. In addition, you can implement your own strategies.

Change the Default

Instead of requiring you to explicitly specify the strategy, some APIs allow you to set passwords using a **default** strategy. The default strategy in Platform is . You can change it if necessary through the *<default.password.encoding>* property.

```
#
# The code of the password encoder to use as default. These are the available options (from core-spring.xml -> 'core.passw
#
# '*' .. legacy 'default' mapping to 'plain' (not changeable before 5.7 - do not use anymore)
# 'plain' .. plain text
# 'sha-256' .. SHA 256
# 'sha-512' .. SHA 512
# 'md5' .. MD5 (for legacy reasons - do not use in production )
# 'pbkdf2' .. PBKDF2 (strong and configurable)
# 'bcrypt' .. (strong and configurable; a current default for FIPS compliance purposes)
# 'scrypt' .. (strong and configurable; one of the OWASP-recommended strategies)
# 'argon2' .. (strong and configurable; most recommended by OWASP)
#
#default.password.encoding=bcrypt
default.password.encoding=argon2
```

Note that changing the default strategy this way preserves previously stored passwords (using the **default** strategy) so that the user can still log in. ⚠ In case the previous strategy was considered to be unsafe, all previous passwords remain to be unsafe as well!

Configure New Generic Strategy

Adding a new password hashing strategy is now made easy by providing a **generic** implementation that just requires you to specify the hashing algorithm.

Refer to Implementing Encoding Algorithms with the Generic Password Encoder to find out how to do it.

Implement Your Own Strategy

It is also possible to implement and add a custom password hashing strategy:

```
public class DummyEncoder implements PasswordEncoder
                {
                @Override
                public String encode(final String uid, final String plain)
                {
                return Integer.toString(plain.hashCode());
                }
                @Override
                public boolean check(final String uid, final String encoded, final String plain)
                {
                return encoded.hashCode() == plain.hashCode();
                }
                @Override
                public String decode(final String encoded) throws EJBCannotDecodePasswordException
                {
                throw new EJBCannotDecodePasswordException(new Throwable("Dummy encoded passwords cannot be de-coded"), "Nc
                }
                }
```

Now add the implementation via Spring:

```
<bean id="dummy" class="foo.bar.DummyEncoder"/>
<bean id="core.passwordEncoderFactory" class="de.hybris.platform.persistence.security.PasswordEncoderFactoryImpl">
  <property name="encoders">
    <map>
      <!-- Attention: Ensure to preserve the built-in encoders.
If you remove or change them, the existing passwords will be lost !!!
-->
      <entry key="*" value-ref="${default.password.encoder}" />
      <entry key="plain" value-ref="core.plainTextEncoder" />
      <entry key="sha-256" value-ref="sha256PasswordEncoder" />
      <entry key="sha-512" value-ref="sha512PasswordEncoder" />
      <entry key="md5" value-ref="core.saltedMD5PasswordEncoder" />
      <entry key="pbkdf2" value-ref="pbkdf2PasswordEncoder" />

      <entry key="argon2" value-ref="argon2PasswordEncoder"/>
      <entry key="scrypt" value-ref="scryptPasswordEncoder"/>
      <entry key="bcrypt" value-ref="bcryptPasswordEncoder"/>
      <!-- the new one -->
      <entry key="dummy" value-ref="dummy" />
    </map>
  </property>
</bean>
```

See above to find out how to change the default so you can use the new strategy.

Localize Labels For Encoders

To localize a Backoffice label for a newly created encoder, add an appropriate entry to the `labels.properties` file located in the *<YOUR_EXTENTION>*`/resources/`*<YOUR_EXTENTION>*`-backoffice-labels` directory. This entry should match the following pattern:

```
hmc.encrypt_<encoder>=<localized encoding name>
```

The encoder should match the key property in the `core.passwordEncoderFactory` bean in the `ext/core/resources/core-spring.xml` directory, for example:

```
hmc.encrypt_bcrypt=BCrypt
```

If you need a quick and dirty solution, you can put this entry into the `resources/platformbackoffice-backoffice-labels/labels.properties` file in the `platformbackoffice` extension, however, we don't recommend modifying this file.

One-Time Password

SAP Commerce supports the functionality of creating and validating one-time passwords, which is a form of two-factor authentication (2FA). These passwords can be used in various scenarios.

Customer Login with One-Time Verification Token

You can enhance security using the one-time password functionality. To enable one-time passwords for customers when logging into composable storefronts:

- Update your composable storefront codebase and enable the one-time password functionality for customers. For more information on how to do this, see Patch Upgrade Notes.

When one-time verification token for customer login is enabled, the Customer User token can be retrieved from the Commerce Authorization Server only with the one-time password *<Token Id>* provided as *<username>* and one-time password *<Token Code>* provided as *<password>*. To be able to fetch a user token you first have to create a one-time login password for that Customer. After fetching one-time password *<Token Id>* from the `OCC API` and receiving *<Token Code>* in an email, you can request a Customer Token with the following command:

```
curl --location --request POST 'https://<COMMERCE_CLOUD_BACKOFFICE_URL>/authorizationserver/oauth/token' \
--header 'Accept: application/json, text/plain, */*' \
--header 'Authorization: Basic <BASE64_ENCODED_OAUTH_CLIENT_CREDENTIALS>' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--header 'X-Requested-With: XMLHttpRequest' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'scope=basic' \
--data-urlencode 'username=<TOKEN_ID>' \
--data-urlencode 'password=<TOKEN_CODE>'
```

ℹ Note

One-time password for customer login functionality only affects Customer type Users. Employee type Users are not affected and should authenticate/fetch tokens in a standard way using their credentials.

Preventing Brute Force Attacks

Using the `otp.customer.login.token.max.verification.attempts` property, you can set the maximum number of failed login attempts for a one-time token. If the maximum number of failed login attempts is reached, the specific one-time token can no longer be used to authenticate as it was removed. By default, the token is removed after the third failed attempt. You can override it in `local.properties`, as shown in the example:

```
otp.customer.login.token.max.verification.attempts = 5
```

The Customer login one-time password functionality can be adjusted by overriding the default configuration properties, described below:

```
# Specifies if the one time password (OTP) functionality is enabled for customer login
otp.customer.login.enabled = false
# Specifies the token encoder to be used for one time password code encoding functionality, see the available options in p
otp.customer.login.token.code.encoder = pbkdf2
# When OTP is enabled allows to specify the length of verification tokens for OTP login functionality
# the minimum allowed value is 6 characters, maximum allowed value is 34 characters
otp.customer.login.token.code.length = 8
# When OTP is enabled allows to set a specific character set with which the code for the OTP login functionality will be g
# The minimal charset length should be 10 characters. If the provided charset is shorter, the default charset value 12345
otp.customer.login.token.code.alphabet = 1234567890ABCDEFGHKLMNPRSTUWXYZabcdefghijklmnoprstuwxyz
# When OTP is enabled allows to specify time to live in seconds for the OTP login functionality
# the maximum allowed value is 900 seconds (15 minutes), minimum allowed value is 60 seconds
otp.customer.login.token.ttlseconds = 300
# Specifies the maximum number of failed verification attempts for a single token, after which the token is removed from t
# 1 means that the token is removed after the first failed verification attempt ( no retries allowed )
otp.customer.login.token.max.verification.attempts = 3
# Number of bits used to generate the random part of the token id for the one time password (OTP) login functionality
# The minimum allowed value is 128 bits. Its recommended to use at least 256 bits. Max value is 1024 bits
otp.customer.login.token.id.generator.bits = 256
# This short name is used to quickly identify the tokens generated for the OTP login functionality.
# The token id format is <{shortName}[{randomPart}]>
# The short name should be unique for initial verification purpose and kept short (max 10 characters) as it does not add a
# If the short name exceeds 10 characters, it will be fallback to default 'OTH' (other) short name.
otp.customer.login.token.purpose.short.name = LGN
```

Leveraging One-Time Password Feature

One-time passwords can be easily reused for other scenarios. To do so, you can define the new purpose of the one-time password by:

- Extending `de.hybris.platform.core.enums.SAPUserVerificationPurpose` in *-items.xml with a new purpose, for example REGISTRATION, ORDER or other.
- Use the `de.hybris.platform.servicelayer.user.UserVerificationTokenService` to create, lookup, and handle one-time token verification in your business logic.

Each `de.hybris.platform.core.enums.SAPUserVerificationPurpose` can have a custom set of additional configuration properties defined. Here's an example of a sample set of custom properties for the new REGISTRATION verification purpose type:

```
## The recipe for configuring new OTP purpose functionality is to create a set of properties according to rule: otp.custom
## The same safe defaults as for customer login OTP are used for other OTP functionalities.
## Lets see this on example set of properties for customer registration OTP after REGISTRATION code is added to SAPUserVer
#otp.customer.registration.enabled = false
#otp.customer.registration.token.code.encoder = pbkdf2
#otp.customer.registration.token.code.length = 8
#otp.customer.registration.token.code.alphabet = 1234567890ABCDEFGHKLMNPRSTUWXYZabcdefghijklmnoprstuwxyz
#otp.customer.registration.token.ttlseconds = 300
#otp.customer.registration.token.max.verification.attempts = 3
#otp.customer.registration.token.id.generator.bits = 256
#otp.customer.registration.token.purpose.short.name = REG
```

If some property value isn't provided for a new verification purpose type, the default value is applied.

Related Information

Flows

Verification Token

Protecting Against Cross-Site Scripting

Inadequately protected web applications are exposed to cross-site scripting (XSS) attacks.

XSS attacks are aimed at end users and are possible after a malicious code is sent to and executed by end users' web browsers. This code can be sent in the form of a web page link or through a web application into which it was previously injected. For example, an attacker can inject a malicious code into an unprotected online store and its database by leaving a comment about a product. If not stripped or encoded, the code will be executed after being displayed in end users' web browsers. Depending on the code's purpose, it might enable the attacker to access the end users' private information such as cookies or passwords they use in a given web application.

The XSS Encoder

The **`security.core.server.csi-1.00.8.jar`** library encodes different kinds of content and helps prevent XSS attacks from malicious data being stored inside the Platform database.

The library was added to the hybris `ext/core/lib` folder. The following methods are available to prevent XSS.

Context	Method
HTML / XML	`out = XSSEncoder.encodeHTML( in )` and `XSSEncoder.encodeXML( val )`
JavaScript	`out = XSSEncoder.encodeJavaScript( val )`
URL	`out = XSSEncoder.encodeURL( val )`
CSS	`out = XSSEncoder.encodeCSS( val )`

For more information, see:

- SAP Encoding Functions for AS Java and JavaScript
- Output Encoding Contexts

Web Security XSSFilter

`XSSFilter` is a simple generic cross-site scripting protection filter for the Platform.

Under normal operation, enabling the filter adds an average overhead of 50 ms per page requests in a performance testing environment. When the number of requests reaches the limit of the system capacity, the response time may increase significantly.

`XSSFilter` can be enabled or disabled. By default, the filter is enabled, and SAP recommends that it remains enabled.

ℹ Note

`XSSFilter` doesn't handle HTTP request bodies. It filters request parameters and headers only.

Configuring XSSFilter

You can find a set of modifiable properties included in `platform/project.properties`:

```
########################## WEB-SECURITY SETTINGS ######################################
#
# Here web related security settings can be found.
#
#############################################################################################
# enable globally
xss.filter.enabled=true

# define action on violation matching globally
# STRIP .. strips all text occurrences which match the patterns below but allow
#          processing the request (default)
# REJECT.. if any pattern matches the whole request gets rejected with the  BAD REQUEST
#          error code
xss.filter.action=STRIP

# our default rules
xss.filter.rule.script_fragments=(?i)<script>(.*?)</script>
xss.filter.rule.src=(?ims)[\\s\r\n]+src[\\s\r\n]*=[\\s\r\n]*'(.*?)'
xss.filter.rule.lonely_script_tags=(?i)</script>
xss.filter.rule.lonely_script_tags2=(?ims)<script(.*?)>
xss.filter.rule.eval=(?ims)eval\\((.*?)\\)
xss.filter.rule.expression=(?ims)expression\\((.*?)\\)
xss.filter.rule.javascript=(?i)javascript:
xss.filter.rule.vbscript=(?i)vbscript:
xss.filter.rule.onload=(?ims)onload(.*?)=
```

Disabling and Enabling XSSFilter

The filter can be disabled or enabled globally with just one parameter:

```
xss.filter.enabled=true
```

You can disable the filter if you have a web application firewall in front of SAP Commerce or if you have some other means of dealing with malicious input.

You can also disable or enable the filter per extension. Just add the **extension name** as a prefix to the parameter:

```
hac.xss.filter.enabled=false
```

Action on Match

The default reaction on matching any rule is that each occurrence of these text fragments is stripped, and the request proceeds:

```
xss.filter.action=STRIP
```

For testing purposes, you may want to change that setting to reject the request with the `BAD REQUEST` error code:

```
xss.filter.action=REJECT
```

Again, you can set the action for a specific extension:

```
hac.xss.filter.action=REJECT
```

Rules

> **i Note**
> Default rules are just a basic set and you should expand them within your projects.

Rules are globally defined as properties. A rule property must start with `xss.filter.rule` in its name, followed by a specific rule ID, which is used for information purposes only:

```
xss.filter.rule.script_fragments=(?i)<script>(.*?)</script>
xss.filter.rule.src=(?ims)[\\s\r\n]+src[\\s\r\n]*=[\\s\r\n]*'(.*?)'
xss.filter.rule.lonely_script_tags=(?i)</script>
xss.filter.rule.lonely_script_tags2=(?ims)<script(.*?)>
xss.filter.rule.eval=(?ims)eval\\((.*?)\\)
xss.filter.rule.expression=(?ims)expression\\((.*?)\\)
xss.filter.rule.javascript=(?i)javascript:
xss.filter.rule.vbscript=(?i)vbscript:
xss.filter.rule.onload=(?ims)onload(.*?)=
```

You may override rules globally or even per extension. Make sure to match the exact rule ID:

```
hac.xss.filter.rule.vbscript=
```

Configuration in web.xml

To enable the filter, change `web.xml` files for each web application. Make sure that the filter:

1. matches all requests
2. comes first in the overall filter chain

```
<filter>
   <filter-name>XSSFilter</filter-name>
   <filter-class>de.hybris.platform.servicelayer.web.XSSFilter</filter-class>
</filter>

<filter-mapping>
   <filter-name>XSSFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Sorting Filters and Rules

`XSSFilter` contains a list of patterns against which it filters dangerous content. With the default `XSSMatchAction.STRIP` action, the order of the items in the pattern list is important because it affects the filtering outcome. See these examples:

```
//Sorting Order 1
input: >><script>abc</script><<
patterns: {
(?i)<script>(.*?)</script>
(?i)</script>
(?ims)<script(.*?)>
}

output: >><<

//Sorting Order 2
input: >><script>abc</script><<
patterns: {
(?i)</script>
(?ims)<script(.*?)>
```

```
(?i)<script>(.*?)</script>
}

output: >>abc<<
```

You can see that the patterns are the same in both lists but the outcomes differ because the patterns are ordered differently. Without sorting, pattern definitions are provided in an undefined order.

You can, however, order filters and rules in your list by sorting them alphabetically. See the example:

```
xss.filter.rule.001_script_fragments=(?i)<script>(.*?)</script>
xss.filter.rule.002_src=(?ims)[\\s\r\n]+src[\\s\r\n]*=[\\s\r\n]*'(.*?)'
xss.filter.rule.003_lonely_script_tags=(?i)</script>
xss.filter.rule.004_lonely_script_tags2=(?ims)<script(.*?)>
```

In this way you make sure you get the outcome you aim for.

Ordering filter rules alphabetically is disabled by default. To enable it, set:

```
xss.filter.sort.rules=true
```

To override it per extension, set:

```
[extension_name].xss.filter.sort.rules=false
```

Servlet 3.0

In Servlet 3.0 containers (Tomcat 7) there may be servlets making use of asynchronous processing. In that case make sure that the filter is configured to support that by adding `<async-supported>true</async-supported>`:

```
<filter>
   <filter-name>XSSFilter</filter-name>
   <filter-class>de.hybris.platform.servicelayer.web.XSSFilter</filter-class>
   <async-supported>true</async-supported>
</filter>
```

Libraries

The filter is part of the platform global classpath. Therefore, you do not need to add any library to your platform web application.

Injecting Static HTTP Response Headers

You can configure SAP Commerce to inject static headers into HTTP response.

To inject headers globally, that is for all SAP Commerce web applications, use:

```
xss.filter.header.[header name]=[header value]
```

This property can also be set for a **specific** web application only. To set it, prefix the parameter with a web application name.

```
[webapp name].xss.filter.header.[header name]=[header value]
```

For example, you may want to add X-Frame-Options to all SAP Commerce web applications to prevent clickjacking. The X-Frame-Options header is used only as an example - it is already set to `SAMEORIGIN` by default. You need to add the following parameter to the `local.properties` file.

```
xss.filter.header.X-Frame-Options=SAMEORIGIN
```

However, if you would like to add this header only for a web application "foo", you would have to add the parameter with the key prefixed with the application name.

```
foo.xss.filter.header.X-Frame-Options=SAMEORIGIN
```

Remember Me Feature

Platform supports the Remember Me feature so that users can log on to the application without the need to provide their credentials every time they access it.

Platform uses a cookie-based login token (`LoginToken`) to authenticate users whose credentials have been stored by the Remember Me feature. The token uses the `CoreRememberMeService` class that implements the Spring `RememberMeServices` class interface.

Enhanced Login Token Generation

Using cookie-based login tokens has been enhanced as the following information is added to their value:

- their TTL (Time to Live) timestamps that are further verified on the server side. If the value of the timestamp has expired, the user isn't authenticated.
- a randomly generated salt that is used for password rehashing.
- a randomly generated value that is stored in the database for each user

The randomly generated value can be revoked. As a result of such revocation, all cookie-based login tokens generated for users become invalid and users can't be authenticated through the old token. To revoke user tokens, use the `de.hybris.platform.jalo.user.TokenService` interface and the dedicated `revokeTokenForUser(final String userId)` method. The default implementation (`DefaultTokenService`) replaces random values with new ones.

If you want to turn off the default behavior of the enhanced login token generation, add the following property to your `local.properties` file:

```
login.token.extended=false
```

⚠ Caution

Due to security reasons, it's not recommended to disable this behavior.

Accepting Login Tokens as URL Parameters

Login tokens aren't accepted as URL parameters. However, if you want to enable such login tokens, add the following property to your `local.properties` file:

```
login.token.url.enabled=true
```

⚠ Caution

Due to security reasons, it's not recommended to enable this behavior.

Disabling Authentication Through Login Tokens

You can disable authentication through login tokens by adding the following property to your `local.properties` file:

```
login.token.authentication.enabled=false
```

If you want to disable login token authentication for given web applications, use the `login.token.authentication.[extensionName].enabled=false` property, for example:

```
login.token.authentication.backoffice.enabled
```

If the property is not set for given web applications, the value of the `login.token.authentication.enabled` is used. See an example of a setting that enables authentication with login tokens for Backoffice and disables it for Administration Console:

```
login.token.authentication.hac.enabled=false
login.token.authentication.backoffice.enabled=true
login.token.authentication.enabled=false
```

Setting Basic Authentication in Core Plus Services

This document explains how to set up basic authentication between SAP Commerce and Core+ services, and, more specifically, between SAP Commerce and Core+-based REST clients.

Suggested Reading

For more information on basic authentication, refer to the documentation provided on the Tomcat website at http://tomcat.apache.org/tomcat-7.0-doc/realm-howto.html .

Transparent Attribute Encryption (TAE)

To encrypt sensitive data transparently, SAP Commerce supports you in declaring a `String-Attribute` as encrypted just by adding the modifier `encrypted="true"` in the `items.xml` file of your extension.

Why Encryption?

You built a secure system, encrypted the most sensitive data, and built a firewall around the database servers. But the thief took the easy approach: They took the backup tapes of your database.

Protecting the database data from such theft is not just good practice; it's a requirement for compliance with most laws, regulations, and guidelines. Consequently, encryption lets you protect your database from this vulnerability.

How It Works

The first layer of defense is the firewall around the whole information infrastructure of your organization, which keeps outsiders from accessing any of the information sources inside your company.

If an intruder gets past the external firewall, that person will be required to supply a password to access the server or perhaps be asked to provide other authentication credentials such as security certificates. This is the second layer of security.

After being authenticated, the legitimate user must only be allowed to access those assets that person is supposed to access.

If a user gets into the database but has no authority to see any table, view, or any other data source, the information is still protected. This mechanism is the next layer of security.

Some security-related compliancy regulations stress that it is possible for an intruder to somehow defeat all of the protective measures and get to the enterprise data. From a planning perspective, this possibility must be accepted, analyzed, and accounted for.

The only option left for defending against an intruder at this point, the last layer of security, is to alter the data, via a process known as encryption, in such a way that the intruder will not find it useful.

Encryption alters data to make it unreadable to all except those who know how to decipher the information.

In a symmetric cipher, the key for encryption and decryption is the same. The Caesar cipher is a simple example of such a symmetric cipher.

Modern symmetric cryptographic algorithms are much more advanced than the Caesar cipher, but they operate in similiar fashion - one passes the plaintext into a cipher with a given key and out comes the ciphertext. The same procedure in reverse produces the original (plaintext) message.

Encryption and Decryption Process

Encrypting a value involves passing the original data and the encryption key to the encryption algorithm to create encrypted data.

Element	Sample
Original Data	1234 1234 1234 1234
Key	nKZkqm!3~!;r^L#td,t!h1+;!vwM]Ipy"lEZ+{kKA:UkjYn_:M)IDcIj)NYm|^1
Encryption Algorithm	AES
Encryption Data	LFGjqOi4quc=K8p+AIJJBoIQTPBh7yaSojr97rHES7YaTkYO6SqBA/M=

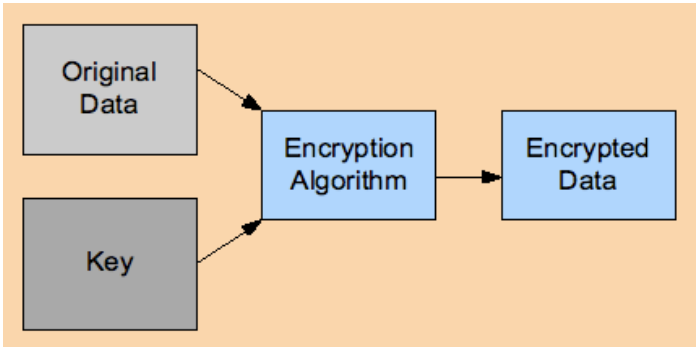


Figure: Encryption mechanism.

During decryption, the logic is reversed, producing the original value. As explained above, since the same key is used to encrypt and decrypt, this scheme is also known as symmetric encryption.

Encryption Algorithm Used by TAE

TAE uses the AES (Advanced Encryption Standard) as default encryption algorithm, which was chosen in October 2000 by the National Institute of Standard and Technology (NIST). They selected an algorithm called Rijndael, developed by Joan Daemen and Vincent Rijman. AES allows key lengths of 128, 192 and 256 bits.

Limitations of Symmetric Encryption

All symmetric algorithms will work in roughly the same fashion. The same key is used to both encrypt and decrypt messages. This means that in order to send a message to someone you must agree on a key beforehand.

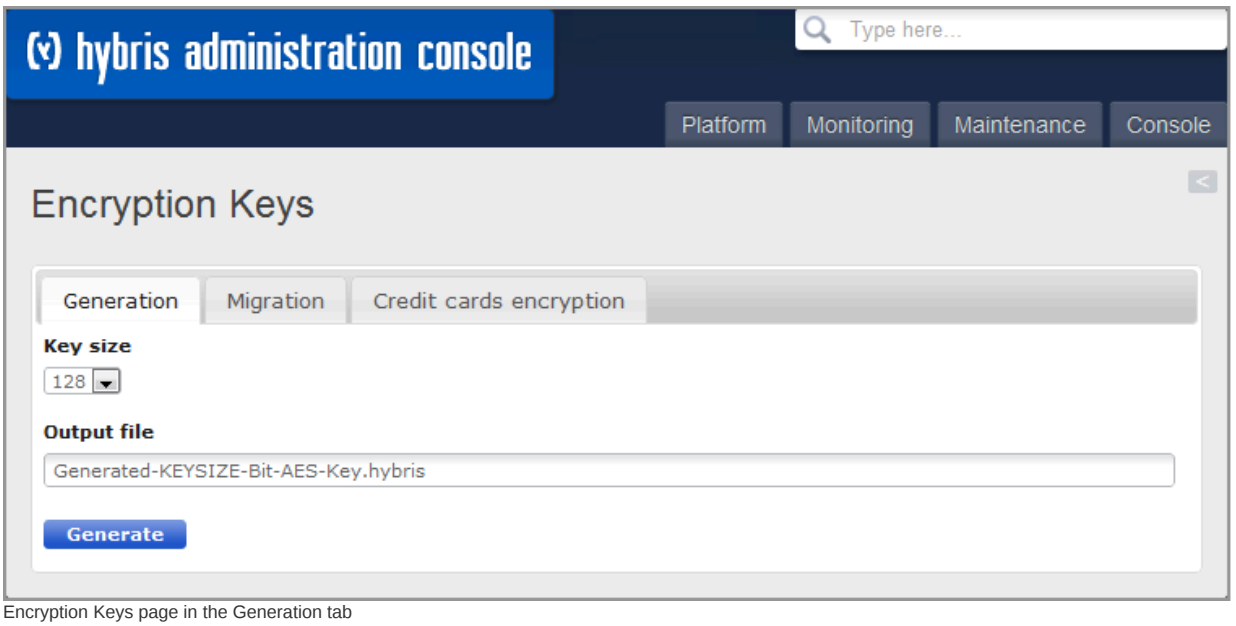
This is the problem with symmetric, or secret-key encryption - the key needs to be kept secret, and any exposure of the key compromises the secrecy of any ciphertext, such as the encrypted credit card number created with that key.

Key Generation

You can find an AES key generator for 128, 192 and 256 bits key in SAP Commerce Administration Console.

1. Open SAP Commerce Administration Console.
2. Go to the **Maintenance** tab and select **Encryption Keys** option.
3. The **Encryption Keys** page in the **Generator** tab displays.

For more information, see Maintenance Tab.



Encryption Keys page in the Generation tab

TAE and Salt

Encryption is all about hiding data, but sometimes it is easier to guess the value of encrypted data if there is repetition in the original plain text value of the data. (This type of attack is called Known Plaintext Attack). For instance, a salary information table may contain repeated values. In that case, the encrypted values will be the same, too, and an intruder could determine all entries with the same salary.

To prevent such an occurrence, an additive, also called salt, is added to the data. It that makes the encrypted value different, even if the input data is same. TAE, by default, applies a salt.

TAE stores the salt with the data that is encrypted. Each time a new piece of data is encrypted, a new salt is generated. This means that the same text encrypts to a different value each time it is encrypted, even if the same key is used every time. To decrypt, the salt must be extracted from the encrypted data, and then combined with the key to create the decryption key.

Key Management and Key Rotation

Key management is the foundation of any solid encryption implementation. Unless an organization establishes a systematic approach to generate, rotate and store its keys, its encryption activities will be largely futile.

Unfortunately, while data encryption itself can be reasonably easy to achieve, efficient management of encryption keys across their lifecycle continues to be a problem.

Annual rotation of encryption keys is required by data security regulations such as PCI DSS while security best practices indicate that rotation should be performed far more frequently. (The PCI Security Standards Council has indicated more frequent rotations will be required in a soon-to-be released revision of the standard).

⚠ Caution

Key Rotation: SAP Commerce TAE supports re-encryption of all historical data with the new key on the fly. The sample below explains how this works. A poorly implemented rotation process can create new data security vulnerabilities and may make critical data inaccessible even to authorized users. We strongly recommend that you never ever delete any encryption key you used for securing your productive data and that you will keep track of the related master password (symmetric.key.master.password).

`project.properties/local.properties`

```
# all keys have to be stored under: ${platform_config_dir}/security/
symmetric.key.file.1=weak-symmetric.key
symmetric.key.file.default=1
symmetric.key.master.password=w427tg3uy73uioomc1fohx1w6pew00n124mlt8ksplpm6ynz55z6305w2nwtj22
```

In the sample above all sensitive attributes (items.xml, encrypted=true) will be encrypted with the key that is specified by the setting `symmetric.key.file.default`.

In our sample this will be the key `weak-symmetric.key` (key id = 1). To make this encryption key unique for a specific SAP Commerce installation this one is protected by the `symmetric.key.master.password`.

ℹ Note

The format of the key definition is: symmetric.key.file.<key id> = <name of the key file>

⚠ Caution

All key files have to be stored in `${platform_config_dir}/security/` and every key file which uses `.hybris` as its file extension is expected to be encrypted with the configured master password. It is recommended that you use our AES key generator because this one will place the file in the right place, create secure random keys and use the proper file extension, too. For more information, see section Key Generation.

As a result of this, configuration encrypted attributes will be stored like:

1:LFGjqOi4quc=K8p+AIJJBoIQTPBh7yaSojr97rHES7YaTkYO6SqBA/M=

Here the prefix '1:' indicates that the key id '1' will be used for decrypting this value.

Changing the Used Encryption Key

In the new sample configuration shown below we specify a new encryption key (id=2) and by setting `symmetric.key.file.default=2` this key will be used for all new encryption operations.

`project.properties/local.properties`

```
# all keys have to be stored under: ${platform_config_dir}/security/
symmetric.key.file.1=weak-symmetric.key
symmetric.key.file.2=Generated-256-Bit-AES-Key.hybris
symmetric.key.file.default=2
symmetric.key.master.password=w427tg3uy73uioomc1fohx1w6pew00n124mlt8ksplpm6ynz55z6305w2nwtj22
```

As a result of this configuration encrypted attributes will be stored like:

2:m2FF0RUDs04=3QiJ+1QJG89AWB3slRZwqBrP65SUh/gOgIsZESrXuKs=

Here the prefix '2:' indicates that the key id '2' will be used for decrypting this value.

At this time we will have the following encrypted entries in our database:

1:LFGjqOi4quc=K8p+AIJJBoIQTPBh7yaSojr97rHES7YaTkYO6SqBA/M=

2:m2FF0RUDs04=3QiJ+1QJG89AWB3slRZwqBrP65SUh/gOgIsZESrXuKs=

At the point of time when an already encrypted entry will be rewritten, the key with id=2 will be used for this operation.

So an entry like

1:LFGjqOi4quc=K8p+AIJJBoIQTPBh7yaSojr97rHES7YaTkYO6SqBA/M=

will be (re)stored like

2:yJ2jg1s37Js=i3/Lwy6PXHrIN4LL3dUxIWaqW1/JTVY5d4CtyvP75Dc=

And now the encrypted database entries will look like:

2:yJ2jg1s37Js=i3/Lwy6PXHrIN4LL3dUxIWaqW1/JTVY5d4CtyvP75Dc=

2:m2FF0RUDs04=3QiJ+1QJG89AWB3slRZwqBrP65SUh/gOgIsZESrXuKs=

Here encrypted attributes will only be re-encrypted with the new key if the corresponding instance will be loaded and restored. To encrypt all encrypted attributes with the new key, use SAP Commerce Administration Console. A poorly implemented rotation process can create new data security.

See also Encryption Keys Migration.

More about Encryption Keys

Platform uses AES symmetric algorithm to encrypt any value for an attribute that is configured as `encrypted=true` in `items.xml`. If not configured differently by the user, Platform uses the default AES key named `default-128-bit-aes-key.hybris` located in the `bin/platform/ext/core/resources/security` directory. This file is pre-generated with use of the default salt `1234567`.

→ Remember

Change this salt to something unique and longer than the default value, and generate your own unique AES key.

The whole encryption key configuration is stored in properties and follows a simple schema:

```
symmetric.key.file.<ID>=filename
symmetric.key.file.default=<ID>
```

The `symmetric.key.file.<ID>` property points to the file name of the key. The `<ID>` is an ID of the key. You can have more than one key configured and have them stored under unique IDs that are integer values. The `symmetric.key.file.default` property points to the ID of the key considered as the default one in the system, and every instance of an Item that is created uses this key to encrypt data. Take a look at example configuration:

```
symmetric.key.file.1=default-128-bit-aes-key.hybris
summetric.key.file.default=1
```

Here is another configuration with more than one key:

```
symmetric.key.file.1=default-128-bit-aes-key.hybris
symmetric.key.file.2=my-own-256-bit-aes-key.hybris
symmetric.key.file.default=2
```

The above example shows a bit more complicated configuration where you have two keys. The key with ID=2 is the default one but the data is stored in the database encrypted with key ID 1 and key ID 2 may be used simultaneously. It results from how encrypted data is stored in the database.

How encrypted data is stored in the database

Encrypted data is stored in the database using following pattern:

```
keyID:encrypted data
```

The `keyID` before the colon is the ID of the key that is used to encrypt data that you see after the colon. This way you can have data in your database encrypted by many keys. One condition must be fulfilled though - you can't shuffle your keys in the configuration. Otherwise underlying logic may choose different key to decrypt data from the one which was used initially to encrypt it.

Encryption Keys Migration

Encryption key migration importance and frequency depends on your security policy. It may also become necessary as an emergency measure in the case where existing keys have become unsecure or were compromised.

Context

Procedure

1. Open SAP Commerce Administration Console.
2. Go to the Maintenance tab and select Encryption Keys option.
3. The Encryption Keys page in the Generator tab displays.

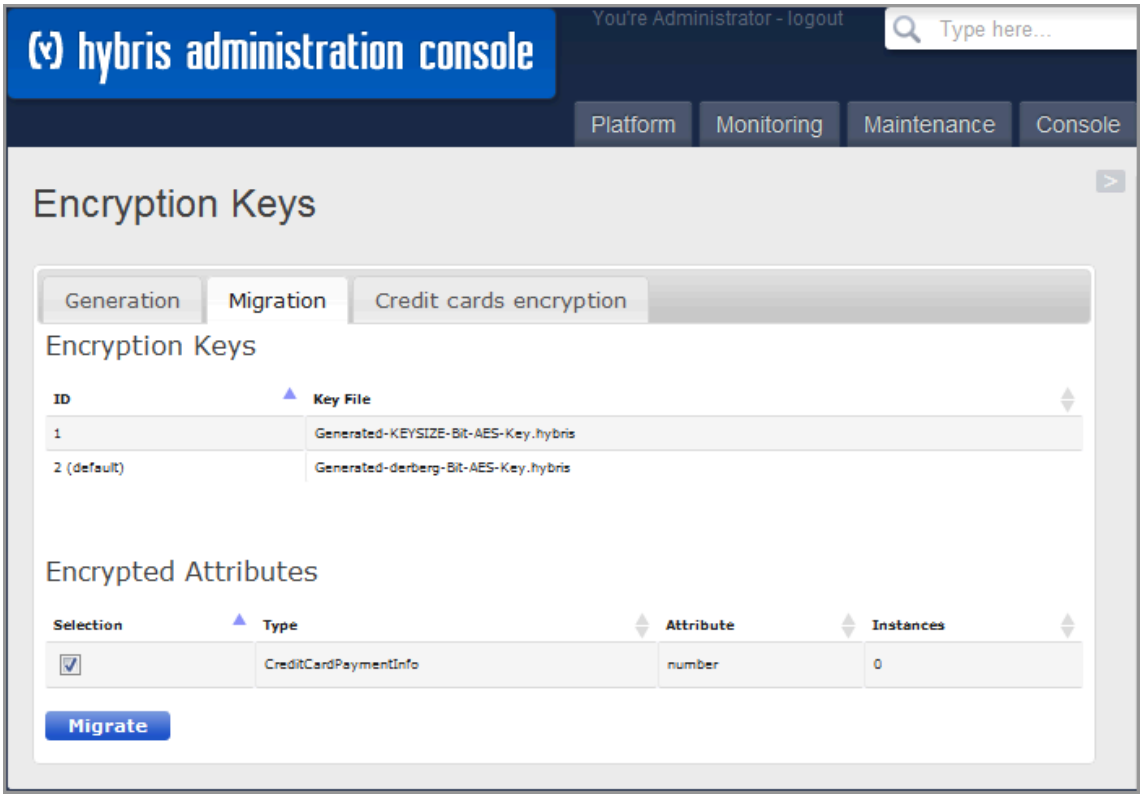


Figure: Encryption Keys page in the Migration tab.

If you did not configure your migration key in `project.properties` or `local.properties` file, then instructions on how to do it would display. Otherwise you can see below fields:

- Encryption Keys: List of the configured encryption keys
- Encrypted Attributes: All types with their encrypted attributes are listed. You see the total count of encrypted instance (#15) and how often every key is used (id=1, #15). Every selected row is encrypted with the new default key after clicking the Migrate button.

Tips and Pitfalls

Here are some tips and pitfalls related to TAE.

Encrypting Search-Relevant Values

As it's impossible to decrypt any values at run time, TAE shouldn't be used for fields that need to be searched for, for example first name or email address.

dontOptimize=true

Because you will not be able to search for encrypted values it could be a good idea to add the modifier **"dontOptimize=true"** in your `items.xml` for the encrypted attribute.

DB Persistence Type

Encrypting a string value increases its length, so be sure that you have chosen the right persistence type **(VARCHAR, CLOB, TEXT)** for storing your encrypted values.

Cluster Configuration

For guaranteeing the integrity of your data, you have to be sure that every node is using the same encryption key.

Since version 4.03 a general default key will be part of the release and is stored at $platformhome/bin/platform/ext/core/resources/security/default-128-bit-aes-key.hybris.

But we highly recommend, that you replace this weak and unsecure keyfile by your own one, which has to be placed in $platform_config_dir/security/.

Related Information

Administration Console

User Account

SAP Commerce Platform enables you to manage the user account and its security with special properties.

Preventing Password Brute-Force Attacks

It is possible to set a maximum number of unsuccessful login attempts in SAP Commerce to prevent brute-force attacks. After exceeding this number, the user isn't able to log in to the system anymore.

You can set the maximum number of unsuccessful login attempts on per-group basis. If the user belongs to many groups that have this property set to different values, the minimum (most strict) value is used.

Configuration

You can set your maximum number of unsuccessful login attempts for a given group in Backoffice, in the **User** > **User Groups** tab. After you select your target group, you have access to the **Max brute force login attempts** property field in the **Administration** tab.

Logs

When a user from a target group enters invalid credentials on the login screen, the following log is recorded:

```
INFO [hybrisHTTP36] [DefaultUserAuditLoginStrategy] user:foo failed 1 logins. Max failed logins allowed:5
```

and if they exceed the maximum number of login attempts:

```
INFO [hybrisHTTP37] [DefaultUserAuditLoginStrategy] user:foo has reached the max number of failed logins (5)
```

As a result, the user cannot log in to the account anymore.

Details

The number of unsuccessful login attempts for each user is stored in the database in `BruteForceLoginAttempts` items. Those items are persisted between system restarts and work correctly in the cluster environment. When the number of unsuccessful login attempts exceeds your configured value, the `User.loginDisabled` field is set to `true`. This prevents the user from logging in to the system until the value is reverted to `false`. In addition, a `BruteForceLoginDisabledAudit` item is created to mark the fact that the user account has been disabled because of too many unsuccessful login attempts.

You can customize this mechanism by implementing `de.hybris.platform.servicelayer.user.UserAuditLoginStrategy` and registering a `userAuditLoginStrategy` bean.

When disabling user accounts, the configuration also takes into account the number of unsuccessful authentication attempts in OAuth flows where user credentials are provided in request parameters.

Access tokens generated for user accounts disabled as part of brute force attack prevention can no longer be used to access resources as they get deleted from the database. The credentials of disabled users can also no longer be used in OAuth flows, which means that it's impossible to issue a new access token or refresh an existing one by using these credentials in the request. To get a new access token for disabled user accounts, re-enable the accounts first.

For disabled user accounts, every new request for an access token, even with correct client and user credentials, fails with a 400 HTTP "Bad request" error response:

```
{
  "error" : "invalid_grant",
  "error_description" : "User is disabled"
}
```

Deactivate User Accounts

The `User.deactivationDate` property allows you to deactivate a user account. If the deactivation date you set is earlier than the current date and time, you deactivate the account instantly. If the deactivate date is in the future, the account is active up to this date and time, and then it gets deactivated.

Re-Enabling User Accounts

User accounts that have been disabled can be re-enabled through Backoffice in the **Password** tab for each user.

Client and User Brute Force Attack Prevention

Brute force attack prevention is configured and performed separately for users and OAuth clients. However, in OAuth flows that require both an OAuth client and user credentials, user authentication is not performed in case of unsuccessful client authentication and brute force attempts are counted only for the client, not for the user.

For information on client brute force attack prevention, see OAuth2.

Users in Platform

Users and user groups in SAP Commerce all descend from the generic **Principal** type that is the foundation for all other user-related, more specific sub-types.

The **Principal** type is a base for the **user** type and, indirectly, the **usergroup** type. These are the starting points for you to use factory default **user** and **usergroup** accounts or to create your own user accounts and usergroup accounts.

You can create your own **employee** sub-types to reflect the company structure and the roles your employees play or structure. You can also create accounts for customers, or create the front end application that let your customers create and manage their own **customer** accounts.

Overview on Principals

Principal is the main abstract class for **user** and **usergroup** types. However, in common informal use both user and user groups are also referred to as principals. The following diagram gives an overview of **Principal** types in SAP Commerce.

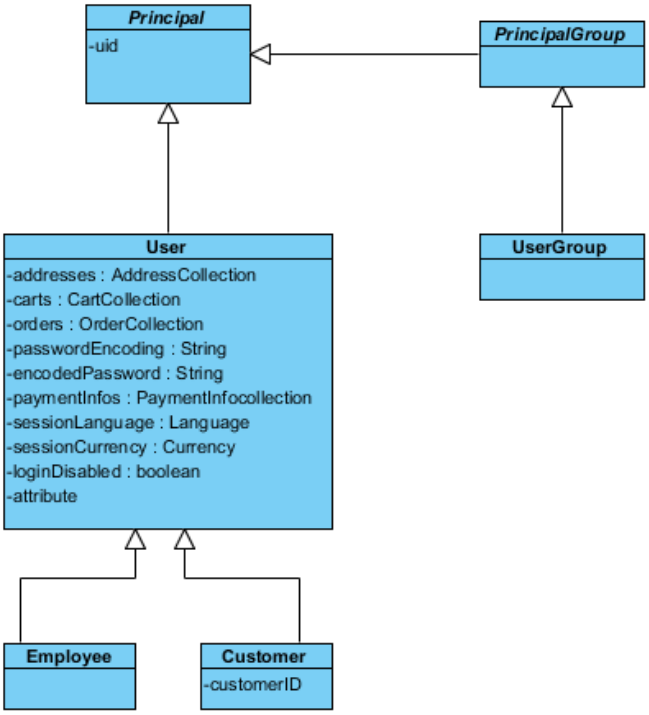


Fig. Principal types hierarchy

ℹ Note

Please, bear in mind that the information presented here may differ when it comes to the Backoffice Application. To know more, see Business Roles in Backoffice

Unique Identifier

In the **Principal** class you can see **uid**. This stands for Unique Identifier that must be defined for every single **user** and **usergroup** item. The main purpose for this is to provide a differentiation factor for users and groups, and to identify each item. As a result, the **uid** must be **globally unique** for all types that descend from **Principal**. This means you cannot have the user and group with the same name. Also, as attribute **uid** is unique both for customers and for employees, employee and customer cannot have the same identifier.

System Accounts

There are three special system user items, that are essential to the platform and cannot be modified nor deleted. These are:

- **employee**: admin
- **customer**: anonymous
- **usergroup**: admingroup

Both the **anonymous** and **admin** users are crucial for the internal processes of SAP Commerce. Therefore, both accounts are protected from removal and against renaming. SAP Commerce blocks all attempts of removing or renaming these user accounts.

You can also add other users to **admingroup**. Those users would have the same rights as **admin** user. The only difference is that the Rule Framework is evaluated for all users, while it is not evaluated for **admin** user, as administrator has full access rights to everything within the system by default.

ℹ Note

Please, bear in mind that the information regarding users may differ when it comes to the Backoffice Application. To know more, see Business Roles in Backoffice

Security

The security system concept in SAP Commerce is based on permissions that can be assigned to users and user groups, as well as global unique identifiers used to identify and authenticate users. Additionally, encryption mechanisms are in place to protect the user passwords. Using available tools you can create the complex security system based on, for example:

- Type related permissions applied to catalogs, products
- Roles of your employees and restrictions

Users

You can use the **user** type to create more specific types that define accounts for your employees and customers. Technically it is also possible to create an item of a **user** type, however from functional point of view it is pointless and should be avoided.

customer Type

Extended from a **user** type, the **customer** type comes with SAP Commerce. This is designed to be used for customers who visit your front end application like web shop, create their accounts for placing orders, manage their profiles, add and modify payment informations, and address data. Depending on your business context customers can create their own accounts, and have full or limited access to all their data.

Example of such situation can be when **customer** creates a new account, adds name and other information together with address and payment information. By creating a new account user also creates **uid**, that is a unique login, for further authorization and authentication purposes. As a result customer can modify the name or family name, address data or payment information, however, the login, the global identifier remains the same. You may configure your front end application to allow users changing data or to prevent them from modifying certain information.

Crucial **customer** data for ordering process, like address or payment information, are kept safe by being duplicated to the **Order** item. When customer makes an order, the delivery address and payment information are copied to the **Order** item and are stored separately to secure that the shipment arrives at the correct address and is correctly paid. Even, if a customer changes payment information or delivery address after the order was made, the shipment is paid and delivered according to the information stored with the **Order** item.

You can also use customer data to provide a support for localized service in your front end. For example, when a customer logs on to the system and starts a session, you could provide a localized version of the page for your customer.

The **customer** type has one extra attribute not present in other types descending from Principal: **customerID**. This mainly serves as an example how the **customer** type can be extended with new attributes to better reflect your business approach for collecting, using, and storing customer data.

Customers can be member of:

- **Usergroup**: For example, cockpit usergroup, VIP customergroup, frequent buyer group
- **Tax Group**: For details, see europe1 Extension
- **Price Group**: For details, see europe1 Extension
- **Discount Group**: For example, discount 5%. For details, see europe1 Extension

Customers can be affected by:

- **Restrictions**: For details, see Restrictions.
- **Access Rights**: Permissions services framework defines access rights for users to the catalogs and other content. For details, see Access Rights.

> **Note**
>
> **Customers are not Allowed to Manage SAP Commerce.**
>
> However, it is technically possible to grant **customer** account access to SAP Commerce management tools, this is not recommended and should not be allowed.

By factory default, SAP Commerce has one customer: **anonymous**. Other customers may be created as the need emerges. You can let the customers create their own account through the front end application, or you can do it by yourself, manually or by using ImpEx script to automate the process. For details on ImpEx, see ImpEx Import - Best Practices.

employee Type

Extended from **user** type. This user can usually do the same as the **customer**. Additionally, the **employee** user can have access to the back office management tools and can perform several management actions, defined by the system of permissions and other regulations.

User **employee** is a representation of a member of your company. This account is used to - depending on assigned permissions - manage objects in SAP Commerce.

The special case of employee user is **admin**. This account cannot be removed, renamed, or restricted. Access rights framework checks if the user is **admin**, if yes, then no evaluation rule is checked to defined permissions for **admin**. User **admin** has access to everything within SAP Commerce. The **admin** user is special in that extent that it is member of the **admingroup** user group. This groups has no factory default limitations, nor restrictions.

You can create other employee users later as the need arises, manually or by using ImpEx scripts. For details on ImpEx, see ImpEx Import - Best Practices. Also, by installing new cockpits, you also receive some factory default users that come together with particular cockpit extensions.

Employees can be affected by:

- **Restrictions**: For details, see Restrictions documentation.
- **Access Rights**: Permissions services framework defines access rights for users to the catalogs and other content. For details, see Access Rights.

There are several factory default employee accounts that come with the SAP Commerce system. For few examples of such user accounts, see Factory Default User Accounts.

User Groups

Unlike the user accounts, **usergroup** cannot be used to authenticate in SAP Commerce. The **uid** serves mainly as a group name. The group contain the collections of user accounts that belong to the group. The main purpose of that is to assign the permissions to the user group, that are further inherited to all group members. This enables easier permissions management, as they do not need to be assigned to particular user but to the set of users groups by the user groups.

Member of a user group inherits all settings from that user group, unless explicitly overridden. That way, you can combine permissions to the types, items, and attributes for a large number of users in one single place. All the users who are members of a certain user group will be affected by these settings. If the exception is needed, you can than override groups settings by assigning specific permissions to the user account in the group.

Groups can also contains other user groups. This let you build the complex hierarchical structures for easier management of your users and better reflecting of your company structure.

For a basic discussion of how inheritance with user groups in SAP Commerce works, see Access Rights documentation.

company Type

The user group **company** is a particular case of a **usergroup** type that by factory default comes together with SAP Commerce. This is a representation of commercial entity. You can add here other user groups and **employee** accounts in order to cover your company structure in the required scope.

Profiles and Roles

Customer can be defined by a broad set of data like name, address, age, title, language, sex, etc. There are some attributes in the **customer** type that are factory default. However, you can also extend the type in order to store more details about your customers and let them create personalized profiles. Such extended profiles may serve as a digital representation of your customer personality enabling you better understanding of customers behavior, needs, requirements, and expectations. Customer profile can help you to identify some characteristics about people who actually come to your web shop, create customer accounts, visit several pages, browse products, fill in the carts, and eventually make orders. The knowledge acquired from user profiles can be applied to predict the preferences of certain customer or customer groups. This is the step toward personalization of interactions between the customers and your company.

Advanced Personalization Module can serve as an example, where such kind of information can be very useful to enrich your business relations with customers. By extending the customer type with new attributes to hold customer data, you can collect information about your target groups for marketing and sale purposes. Depending on the customer group certain customer qualifies to, you could display different content, special offers, or hide certain areas of your web shop in order to prevent customers from accessing them. Static user profiles based on the data provided by customers can be extended with the information based on the customers behavior and shopping history in order to better adapt to customer needs and provide additional tools like suggestions or preferences.

Employee can also be treated as a personalized unit. Unlike the customer, the data related to the employee is rather back-office oriented and describes the administrative role that the employee plays within SAP Commerce. There is no way to predict those roles, as they depend on your specific business context. However, the SAP Commerce system is very flexible and allows you to create several employee accounts, assign different permissions to them or even combine permissions for **employee** users and **company** user group on an item or attribute level. Additionally, there are several factory default employee accounts that come with several cockpits, for details see the Overview of Factory Default Employee Accounts document.

Related Information

ServiceLayer Security
Restrictions
Access Rights

Managing Users and User Groups

Users and **user groups** in SAP Commerce have the same origin: **Principal**. They can be created, updated, and removed.

The process is similar for both users and user groups. There are some issues to take into consideration, such as uniqueness of the **uid**, or **system accounts** that cannot be removed.

Working with Users and User Groups

If you need to work with the user accounts, you can create them by instantiating a **UserModel** and saving it with the **ModelService**. For details, see Models.

For most actions, you need to get the unique user identifier, **uid,** which also serves as unique login for users.

```
UserModel getUserForUID(String userId);
<T extends UserModel> T getUserForUID(String userId, Class<T> class);
```

The **uid** is usually used as the login for a user. If you know the user's **uid**, you can check if the user of the specified login already exists in the system.

```
boolean isUserExisting(String uid);
```

You can get the user group by a specific **uid**, which in this case is the unique name of the group, **getUserGroupForUID**. You can have many groups in your system defined. The groups may belong to other groups. Also, they may contain other groups as well. This is how to create a hierarchical tree structure. Normally, a user is a leaf in such a structure. However, the group not containing other groups can also be considered a leaf.

If you want to check the groups to which a user belongs, you can use the following methods:

```
Set<UserGroupModel> getAllUserGroupsForUser(UserModel user);
getAllUserGroupsForUser(UserModel user, Class<T> class);
boolean isMemberOfGroup(UserModel member, UserGroupModel groupToCheckFor);
boolean isMemberOfGroup(UserGroupModel member, UserGroupModel groupToCheckFor);
```

You can specify a **Title** for a user. For example, Mr., Ms., or Dr. The **Title** itself is a separate type. Because it is a minor type related to the user and does not usually change, the **Title** is also handled in the UserService by the **getTitleForCode** method.

There is a set of methods to help you in checking the system user accounts, such as admin, anonymous, and admingroup.

```
EmployeeModel getAdminUser();
UserGroupModel getAdminUserGroup();
CustomerModel getAnonymousUser();
boolean isAdmin(UserModel user);
boolean isAnonymousUser(UserModel user);
```

To check if a user is an admin or has the administrator right, which it inherited from the admingroup, you can generate a list of all **admin** users or check all groups for the specific user to verify if any of them is **admingroup**.

You can use **UserService** methods to check the user in session and then, based on the result, to perform certain actions, for example:

```
UserModel currentuser = userService.getCurrentUser();

if(userService.isAnonymousUser(currentuser))
{
  System.out.print("You need to login to proceed");
}
```

Unique Identifiers for Users and User Groups

Users and user groups are identified by a unique identifier, a **uid**. The **uid** is an attribute of the **Principal** class; it is used in any type that extends **Principal**. The **uid** is used to uniquely identify all principals in the SAP system. It is used identify users such as **customer** and **employee**, or user groups such as **company**.

The **uid** must be unique. It cannot be duplicated. You can avoid duplicating a **uid** by adding the suffix "group" to every **uid** that is used in the **usergroup** type, for example, **admin** and **admingroup**.

Users

Creating Users

Creating user is a process similar to creating any other model instance in the service layer architecture. To create a user, you call specific methods, as shown in the following code samples.

- **Create an employee user**

  ```
  final EmployeeModel empl = new EmployeeModel();
  empl.setUid("test");
  modelService.save(empl);
  ```

- **Create a customer user**

  ```
  final CustomerModel cust = new CustomerModel();
  cust.setUid("test");
  modelService.save(cust);
  ```

In the previous examples, we used the user called test. However, the user **uid** that serves as login must be globally unique within the system. You cannot have a **customer** and **employee** with the same **uid**. If a **uid** is already used, the system throws a model exception and you receive information about **uid** duplicate similar to the example shown below. You must change the **uid** and save it again.

```
de.hybris.platform.servicelayer.exceptions.ModelSavingException:
ambiguous unique keys {uid=test} for model CustomerModel (<unsaved>) - found 1 item(s) using the same keys
```

How to name the methods depends on your business context. For example, your customers can create new accounts by visiting your web shop. In the web shop, you can implement a business application that:

- Handles customer data
- Calls UserService on the SAP Commerce side
- Creates an account for a customer

During this process, the customer is prompted to provide a unique **login** that serves as **uid**. If the login is already taken by another user, the SAP system returns the relevant information. You can use this information to display a more meaningful message to your user and prompt the user to pick another login. The login could be a string or email address. You can also assign specific permissions to anonymous users that allows them to view selected parts of your content or allows them to perform actions that do not require a unique login.

Generating uid with Key Generator

Another method of obtaining a unique identifier for your customers within the system is to use a number series key generator. This method creates a unique customer ID for each customer. To use the key generator, go to the **`servicelayer/user-spring.xml`** file and enable the following beans:

- **customerIDGenerator**: This is the key generator that creates the key for the user found in the **customer_id** field. In the code example, it defines an eight-digit alphanumeric key, starting from 00000000. To generate a key with only numerical characters, you must set the **numeric** property to **true**.
- **customerIDPrepareInterceptor**: This prepares the interceptor. The interceptor is activated only for the object with the number that is defined as the uid for the user.
- **InterceptorMapping**: This connects the interceptor with the particular customer. For more information on interceptors, see the Interceptors document.

```
<!--
 <bean id="customerIDGenerator" class="de.hybris.platform.servicelayer.keygenerator.impl.PersistentKeyGenerator"
                       init-method="init" >
  <property name="key" value="customer_id"/>
  <property name="digits" value="8"/>
  <property name="start" value="00000000"/>
  <property name="numeric" value="false"/>
 </bean>

 <bean id="customerIDPrepareInterceptor" class="de.hybris.platform.servicelayer.user.interceptors.CustomerIDPrepareI
                       >
  <property name="keyGenerator" ref="customerIDGenerator"/>
 </bean>

 <bean class="de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping" >
  <property name="interceptor" ref="customerIDPrepareInterceptor"/>
  <property name="typeCode" value="Customer"/>
 </bean>
  -->
```

Updating Users

Customer Type

Action	Description
Updating Address	The UserModel, as well as the Cartmodel and the OrderModel, can have an address. You use the **AddressService** to create the address for a user. If the customer creates a cart, the address is cloned to the cart related to the customer. If the cart changes to order during a later step, then the order also keeps the user address. The address in the cart and order are copies made by cloning the customer address value. This can be useful. if, for example, a user places the order, but later changes the address before the order is placed. In such a situation, the shipment from the order is to be delivered to the address created when the order was created.
Updating Payment Info	Each customer has his/her own **PaymentInfo.** It holds the payment details for user.
Saving Carts	For a specific user, you can save the cart-related data. This allows the user to stop browsing the web shop any time and return later to continue from the the point the user left the web shop.

Action	Description
Deactivating User	You can modify the **loginDisabled** attribute to deactivate the user account. SAP recommends that you deactivate user accounts instead of removing them, as it is possible that there are still orders holding the address for the users. Once you have removed a user account, there is no way to link the existing order with the user who placed it. To deactivate a user, use **user.setLoginDisabled(true);**

Employee Type

Action	Description
Changing User Roles	Users of the **employee** type can play different roles within a company. You can define roles by creating specific user groups and assigning users to them. To change the role of your employee, you remove the user from one group and assign it to another group. For example, by assigning a user to the **admingroup**, you define administrator as the role for this user . From a functional point of view, such a user has the same rights as an **admin** user. As one group can contain many users, and one user can belong to many groups, it is possible to define several roles for one user.

Removing User

You can remove any user account, except system users such as **admin** and **anonymous**. Removing a user account is done in the same as any other model. To remove a user account, you must first get the current user and then user a method that removes the user account.

```
UserModel user = UserService.getUserForUID("theUserId");
ModelService.remove(user);
```

If the user you want to remove is not found in the system, then then the **UnknownIdentifierException** exception is thrown. If you try to remove a system user account, you also get an exception because you cannot remove a system user account.

Historical User Data

Once the order is created, the **Address** and **paymentInfo** are cloned from the user type to the order. Cloning makes a copy of the data. The is done because the user can modify the payment information after placing an order, but the order should use the address given at the time of placing the order. The same is valid for the information about payment: **paymentInfo**.

After a user has been removed, historical data related to user, order, and payment can remain in the system. For example, the user's address is stored in two locations. If you remove a user, you can have an order with an address but without a user name. This can happen because orders are not removed with the user. If the user is removed, then the order loses its reference to the user. This is why SAP recommends that you deactivate the user instead of removing the user account. Deactivated users are not automatically removed from the system. If you want to remove a deactivated user account from the system, you must do so manually.

User Groups

Usergroup types are similar to the **user** types. Both **user** and **usergroup** types are extended from the **Principal** type. You can create user groups the same way as you would create the users. For more information, see the API Documents.

Creating User Groups

User groups are containers that hold users and user groups. This allows you to create complex hierarchical structures. When you create user groups, you must ensure that no cyclic references are added in the user groups structures.

For example, you can create group A that contains group B, that holds group C. However, you cannot create a cyclic reference in which group A contains group B, that contains group C, that contains group A.

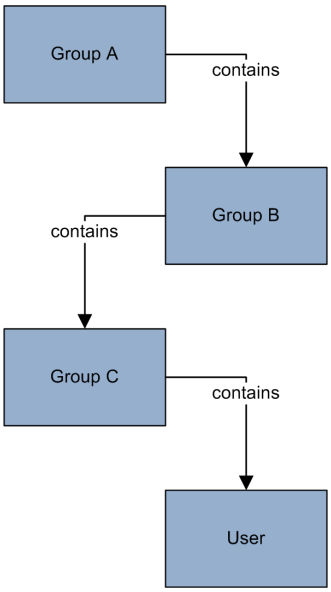


Fig. User Group Structure Allowed

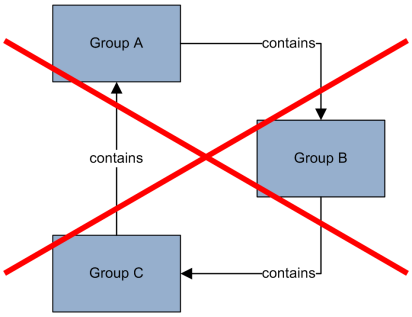


Fig. User Group Structure Not Allowed

The purpose of a user group can be:

- **Roles**: With the group hierarchy you can have a representation of the roles that your users play in the system.
- **Permissions management**: You can use the user groups to assign access rights to groups of your employees. User groups make it easier to assign and remove access rights to a number of users.
- **Marketing targeting**: You can define your target groups and assign users to the groups that reflect this kind of assignment. Customers could then have a different type of access to particular areas of your web shop content. Depending on the target group, you could show or hide the content from some customer groups.
- **Different access layers**: Generally, you can use the groups to separate those users who can perform management functions like **employee** users from those who can access your web shop without any management rights like **customer** users.
- **Cockpit customization**: Several cockpits come with the pre-defined user accounts that can have a specific access right scope to the particular cockpit management functions.
- **Catalog visibility**: With the assigning specific permissions to the different groups you can allow your users to see different catalogs, different products or different catalog versions depending on several criteria. For example, you can gather all users from a particular country into one group and display the product offer that is specifically designed for that group.

Updating User Group

You can update the attributes of the **usergroup**. For example, you can add users to the group or remove users, or change other attributes. Updating the usergroup can be done the same way as updating any other model in the Service Layer architecture. You need to get the user by the **uid**, and then proceed with other actions depending on what kind of information you want to update. For details on the operations on models, see Models documentation.

```
UserGroupModel usergroup = UserService.getUserGroupForUID("theUserGroupId");
```

It is also possible to change the **uid** of the user group. As the **uid** is the same as the user group name, it just means that you change the name of the user group.

Removing User Group

It is not possible to remove the system user group: **admingroup**. Assuming you have the sufficient permissions assigned you can remove all other groups easily:

```
UserGroupModel usergroup = UserService.getUserGroupForUID("theUserGroupId");
ModelService.remove(usergroup);
```

Related Information

Removing Model Instance
Creating Model Instance
How to Display a User
Models

Factory Default User Accounts

There are several factory default user accounts that come with the SAP Commerce system.

The following table list all the default user acounts included in SAP Commerce.

Factory Default Employee Accounts

User Name
aarav.devi@acme.com
aaron.customer@hybris.com
abra.christensen@sapfsa.com
abraham.mclane@acme.com
acctmgra
acctmgrb
acctmgrc
acctmgrd
admin
adrian.kent@hybris.com
aiko.abe@sapfsa.com
akiro.nakamura@pronto-hw.com
akiro.nakamura@rustic-hw.com
aladdin.gentry@sapfsa.com
alan.martin@hybris.com
albert.decastro@acme.com
alda.kamaka@acme.com
alejandro.navarro@rustic-hw.com
alistair@hybris.com
amanda.smith@stateofrosebud.com
amelia.hill@acme.com
amos.adkins@sapfsa.com
analyticsmanager
anamaria.coots@acme.com
andrea.customer@hybris.com
andrew.customer@hybris.com
anette.customer@hybris.com
angelyn.lobaugh@acme.com
anil.gupta@pronto-hw.com
anil.gupta@rustic-hw.com

User Name
annabel.golder@acme.com
anne.customer@hybris.com
anonymous
anthony.customer@hybris.com
anthony.lombardi@rustic-hw.com
antonio.ferrari@sapfsa.com
arjun.sewant@acme.com
arnold.customer@hybris.com
asagent
asagentmanager
asagentsales
aubrey.baxter@sapfsa.com
axel.krause@rustic-hw.com
ayumi.nakamura@acme.com
BackofficeIntegrationAdministrator
BackofficeIntegrationAgent
BackofficeProductAdministrator
BackofficeProductManager
BackofficeWorkflowAdministrator
BackofficeWorkflowUser
BedfordWarehouseAgent
BerlinDomWarehouseAgent
BerlinHospitalWarehouseAgent
BerlinMuseumWarehouseAgent
BerlinZooWarehouseAgent
bernard.customer@hybris.com
bernardo.coelho@acme.com
blossom.welch@sapfsa.com
bobby.customer@hybris.com
brandon.leclair@acme.com
brian.customer@hybris.com
bridget.customer@hybris.com
burton.franco@sapfsa.com
burtonlover@hybris.com
byung-soon.lee@rustic-hw.com
CajonWarehouseAgent
calvin.citizen@stateofrosebud.com
cameralenslover@hybris.com
canonlover@hybris.com

User Name
carla.torres@rustic-hw.com
CarltonWarehouseAgent
carol.citizen@stateofrosebud.com
carson.shepherd@sapfsa.com
chandni.devaraju@acme.com
chelsie.steck@acme.com
chen.gao@acme.com
chris.rumple@ehost.com
christmascustomer@hybris.com
cmseditor
cmsmanager
cmsmanager-apparel-de
cmsmanager-apparel-uk
cmsmanager-electronics
cmsmanager-electronics-de
cmsmanager-electronics-eu
cmsmanager-electronics-euzone
cmsmanager-electronics-uk
cmsmanager-electronics-us
cmsmanager-powertools
cmspublisher
cmsreader-apparel
cmsreviewer
cmstranslator
cmstranslator-Annette
cmstranslator-Seb
colt.ford@sapfsa.com
csagent
customer.support.1@sap.com
customer.support.2@sap.com
customer.support.3@sap.com
customer.support.4@sap.com
customer.support.5@sap.com
customer.support.6@sap.com
customer.support.7@sap.com
customer.support.8@sap.com
customer.support@chicago.com
customer.support@ichikawa.com
customer.support@nakano.com

User Name
customer.support@sanfrancisco.com
CustomerSupportAdministrator
CustomerSupportAgent
CustomerSupportManager
cxmanager
cxmanager-apparel-de
cxmanager-apparel-uk
cxmanager-electronics
cxmanager-electronics-de
cxmanager-electronics-eu
cxmanager-electronics-euzone
cxmanager-electronics-uk
cxmanager-electronics-us
cxmanager-powertools
cxuser
cxuser-apparel-de
cxuser-apparel-uk
cxuser-electronics
cxuser-electronics-de
cxuser-electronics-eu
cxuser-electronics-euzone
cxuser-electronics-uk
cxuser-electronics-us
cxuser-powertools
dagmar.fischer@acme.com
daisy.smith@irc.uk
dan.cameron@siteb.com
daniel.customer@hybris.com
daniele.sorber@acme.com
darrin.hesser@acme.com
deacon.fuller@sapfsa.com
dean.barton@sapfsa.com
debera.spiller@acme.com
DefaultWarehouseAgent
diana.best@sapfsa.com
dietrich.brand@acme.com
dionne.siguenza@acme.com
dipti.customer@hybris.com
diptiman.customer@hybris.com

User Name
donna@moore.com
dorthy.geoghegan@acme.com
dot.cohan@acme.com
elena.bulav@internet.ru
elena.petrova@sapfsa.com
elizabeth.juhlin@acme.com
elizabeth.miller@stateofrosebud.com
emily.bennett@acme.com
etta.berg@hybris.com
eusebio.scharff@acme.com
evangeline.jefferson@sapfsa.com
fern.henline@acme.com
francie.wildman@acme.com
fsintegrationadmin
gi.sun@pronto-hw.com
gi.sun@rustic-hw.com
GlasgowWarehouseAgent
glen.hofer@acme.com
granny.citizen@stateofrosebud.com
h.williams@peabody.ca
hac_editor
hac_viewer
hanna.andresen@acme.com
hanna.schmidt@pronto-hw.com
hanna.schmidt@rustic-hw.com
harold.wine@asite.org
hedley.mayer@sapfsa.com
hermelinda.cusick@acme.com
homer.citizen@stateofrosebud.com
importmanager
InboundConsentTemplateUser
IndianapolisWarehouseAgent
indira.duffy@sapfsa.com
IntegrationAdministrator
IntegrationAgent
integrationapi_adminuser
integrationapi_createuser
integrationapi_viewuser
integrationmonitoringtestuser

User Name
integrationservicetestuser
isabella.jackson@acme.com
isha.customer@hybris.com
jack.smith@stateofrosebud.com
james.bell@pronto-hw.com
james.bell@rustic-hw.com
james.davis@acme.com
jamey.sowa@acme.com
jane.citizen@stateofrosebud.com
janetta.estepp@acme.com
jason.citizen@stateofrosebud.com
JerseyWarehouseAgent
jill.citizen
joana.oliveira@sapfsa.com
joey.citizen
john.citizen@stateofrosebud.com
john.li@sapfsa.com
john.miller@sapfsa.com
john.russel@hybris.com
josephine.citizen@stateofrosebud.com
jules.hasson@acme.com
juliane.tickle@acme.com
kadeem.gamble@sapfsa.com
kai.ratliff@sapfsa.com
karleen.holub@acme.com
kathy.liu@sapfsa.com
keenreviewer0@hybris.com
keenreviewer1@hybris.com
keenreviewer10@hybris.com
keenreviewer11@hybris.com
keenreviewer12@hybris.com
keenreviewer13@hybris.com
keenreviewer14@hybris.com
keenreviewer15@hybris.com
keenreviewer16@hybris.com
keenreviewer17@hybris.com
keenreviewer18@hybris.com
keenreviewer19@hybris.com
keenreviewer2@hybris.com

User Name
keenreviewer20@hybris.com
keenreviewer21@hybris.com
keenreviewer22@hybris.com
keenreviewer23@hybris.com
keenreviewer24@hybris.com
keenreviewer25@hybris.com
keenreviewer26@hybris.com
keenreviewer27@hybris.com
keenreviewer28@hybris.com
keenreviewer29@hybris.com
keenreviewer3@hybris.com
keenreviewer30@hybris.com
keenreviewer4@hybris.com
keenreviewer5@hybris.com
keenreviewer6@hybris.com
keenreviewer7@hybris.com
keenreviewer8@hybris.com
keenreviewer9@hybris.com
keita.tanaka@acme.com
kelsie.spencer@sapfsa.com
kirti.customer@hybris.com
KotoWarehouseAgent
kritika.customer@hybris.com
lael.garibay@acme.com
lars.bauer@rustic-hw.com
latisha.latimer@acme.com
lavone.dupler@acme.com
LeedsWarehouseAgent
linda.wolf@pronto-hw.com
linda.wolf@rustic-hw.com
liu.yang@acme.com
lucas.kowalski@rustic-hw.com
mai.matsumoto@acme.com
manager
mara.martino@acme.com
marco.rossi@sapfsa.com
maria.stevens@hybris.com
marie.dubois@rustic-hw.com
marie.kempner@acme.com

User Name
mark.farrel@hybris.com
mark.rivers@pronto-hw.com
mark.rivers@rustic-hw.com
marketingmanager
marvel.fargo@acme.com
matheu.silva@rustic-hw.com
MatsudoWarehouseAgent
matthew.miller@stateofrosebud.com
matthew.zhao@sapfsa.com
maycustomer@hybris.com
men@hybris.com
menover30@hybris.com
menshortslover@hybris.com
menvipbronze@hybris.com
menvipgold@hybris.com
merchantcontentmanager
merchantproductmanager
merchantvendormanager
michael.adams@sapfsa.com
michael.barton@sapfsa.com
michael.clarke@sapfsa.com
miguel.rodriguez@sapfsa.com
mingmei.wang@pronto-hw.com
mingmei.wang@rustic-hw.com
MisatoWarehouseAgent
monique.legrand@sapfsa.com
MunichMuseumWarehouseAgent
NakanoWarehouseAgent
natsumi.takahashi@acme.com
nishi.customer@hybris.com
noah.jenkins@acme.com
oliver.baker@acme.com
olivia.ann@hybris.com
ossie.dilks@acme.com
patricia.anderson@sapfsa.com
pedro.dasilva@sapfsa.com
piedad.holdren@acme.com
powerdrillslover@pronto-hw.com
pradeepthi.customer@hybris.com

User Name
pritika.customer@hybris.com
productmanager
punchout.customer@punchoutorg.com
punchout.customer2@punchoutorg.com
ravi.pandey@sapfsa.com
RegApproverA
reggy.ray@hybris.com
revenueCloudCustomerApiUser
reviewer1@hybris.com
reviewer10@hybris.com
reviewer11@hybris.com
reviewer12@hybris.com
reviewer13@hybris.com
reviewer14@hybris.com
reviewer15@hybris.com
reviewer16@hybris.com
reviewer17@hybris.com
reviewer18@hybris.com
reviewer19@hybris.com
reviewer2@hybris.com
reviewer20@hybris.com
reviewer21@hybris.com
reviewer22@hybris.com
reviewer23@hybris.com
reviewer24@hybris.com
reviewer25@hybris.com
reviewer26@hybris.com
reviewer27@hybris.com
reviewer28@hybris.com
reviewer29@hybris.com
reviewer3@hybris.com
reviewer30@hybris.com
reviewer4@hybris.com
reviewer5@hybris.com
reviewer6@hybris.com
reviewer7@hybris.com
reviewer8@hybris.com
reviewer9@hybris.com
rewati.customer@hybris.com

User Name
richard.martin@acme.com
richard.wilson@sapfsa.com
richard@wilson.com
ronnie.ray@hybris.com
sade.mcdougall@acme.com
salome.levi@rustic-hw.com
sandesh.patwary@acme.com
sapCpqQuoteApiUser
sapInboundB2BCustomerUser
sapInboundB2CCustomerUser
sapInboundClassificationUser
sapInboundMDMB2CCustomerUser
sapInboundOMMOrderUser
sapInboundOMMReturnRequestUser
sapInboundOMSOrderUser
sapInboundPriceUser
sapInboundProductUser
sapInboundRCConfigUser
sapInboundSubscriptionPriceUser
sbgadmin
screwdriverslover@pronto-hw.com
searchmanager
SecondaryNakanoWarehouseAgent
selfserviceuser2@hybris.com
selfserviceuser3@hybris.com
selfserviceuser4@hybris.com
selfserviceuser5@hybris.com
selfserviceuser6@hybris.com
sheilah.duffin@acme.com
ShinbashiWarehouseAgent
shortslover@hybris.com
shun.watanabe@acme.com
stefan.bosch@sapfsa.com
summercustomer@hybris.com
surabhi.customer@hybris.com
TacomaWarehouseAgent
tag.demph@sapfsa.com
takahiro.suzuki@sapfsa.com
TampaWarehouseAgent

User Name
temeka.meekins@acme.com
teresa.citizen@stateofrosebud.com
teresa.ruiz@sapfsa.com
test-user-with-coupons@ydev.hybris.com
test-user-with-orders@ydev.hybris.com
thanksgivingcustomer@hybris.com
thomas.schmidt@sapfsa.com
tilda.prisbrey@acme.com
tim.james@hybris.com
tom.ziebarth@acme.com
TranslatorGSW
tuaInboundSimpleProductOfferingUser
Tulsa1WarehouseAgent
Tulsa2WarehouseAgent
Tulsa3WarehouseAgent
Tulsa4WarehouseAgent
Tulsa5WarehouseAgent
ula.barragan@acme.com
ulf.becker@rustic-hw.com
ulysses.head@sapfsa.com
vada.rahm@acme.com
vendor1vendoradministrator
vendor1vendorcontentmanager
vendor1vendorproductmanager
vendor1vendorwarehousestaff
vendor2vendoradministrator
vendor2vendorcontentmanager
vendor2vendorproductmanager
vendor2vendorwarehousestaff
vendor3vendoradministrator
vendor3vendorcontentmanager
vendor3vendorproductmanager
vendor3vendorwarehousestaff
vendor4vendoradministrator
vendor4vendorcontentmanager
vendor4vendorproductmanager
vendor4vendorwarehousestaff
vendor5vendoradministrator
vendor5vendorcontentmanager

User Name
vendor5vendorproductmanager
vendor5vendorwarehousestaff
vendor6vendoradministrator
vendor6vendorcontentmanager
vendor6vendorproductmanager
vendor6vendorwarehousestaff
vendor7vendoradministrator
vendor7vendorcontentmanager
vendor7vendorproductmanager
vendor7vendorwarehousestaff
vendor8vendoradministrator
vendor8vendorcontentmanager
vendor8vendorproductmanager
vendor8vendorwarehousestaff
vipbronze@hybris.com
vipgiold@hybris.com
vipgold@hybris.com
vipsilver@hybris.com
vjdbcReportsUser
wang.lei@acme.com
WarehouseAdministrator
WarehouseAgent
WarehouseEAgent
WarehouseManager
WarehouseNAgent
WarehouseSAgent
WarehouseWAgent
wei.liu@homemail.ch
wfl_marketing
wfl_marketing_DE
wfl_marketing_EN
wfl_marketing_ES
wfl_marketing_FR
wfl_marketing_GB
wfl_marketing_GSW
wfl_marketing_IT
wfl_marketing_SWE
wfl_productApproval
wfl_productManagement

User Name
wfl_purchase
wfl_translator_DE
wfl_translator_EN
wfl_translator_ES
wfl_translator_FR
wfl_translator_GB
wfl_translator_GSW
wfl_translator_IT
wfl_translator_SWE
william.hunter@pronto-hw.com
william.hunter@rustic-hw.com
women@hybris.com
womenvipgold@hybris.com
womenvipsilver@hybris.com
xaviera.crawford@sapfsa.com
yan.shehorn@acme.com
yformsmanager
yoshie.dority@acme.com
yu.yamamoto@acme.com
yuka.kobayashi@acme.com
yuri.chandler@sapfsa.com
zhang.wei@acme.com

Related Information

Users in Platform

Visibility Control

You can configure SAP Commerce to allow or deny access to items, catalogs, and features. Some of these controls take effect in the front end, while others are specific to Backoffice Administration Cockpit.

There are three types of visibility controls in SAP Commerce:

- Those that are effective all across SAP Commerce, such as restrictions
- Those that are effective in a web front end only, such as product approval status
- Those that are effective in Backoffice only, such as access right settings

ℹ Note

The following table only lists controls that are available in SAP Commerce by default. If you have built custom extensions, some of these controls may not take effect, or may work differently from what is described here.

Available controls in the shop frontend	Available controls in Backoffice
• Restrictions. • Category version visibility.	• Restrictions. • Category version visibility.

Available controls in the shop frontend	Available controls in Backoffice
• Product approval status. • Catalog version `active` attribute. For more information, see Front End Visibility Controls.	• Backoffice access rights. • Usergroup-specific Backoffice configuration. For more information, see Backoffice Visibility Controls.

Restrictions

Restrictions affect Flexible Search results. If restrictions are in place, a user may not receive all search results that would be available without those restrictions.

For more details, see Restrictions.

Duration of Effect

Visibility rights are bound to a session. If you change a visibility setting and the change appears to have no effect, try closing the session and logging in again. Some visibility settings are valid for the entire session lifetime and are not affected retroactively by changes made when the session has already been created. For an overview of catalog-related setting lifetimes, see Catalog Guide.

Visibility Control Checklist

Quickly find answers to problems related to the visibility of items in the storefront or Backoffice Administration Cockpit.

The following table lists a number of possible visibility issues and their potential causes. The potential causes are discussed in more detail in the related topics.

Problem	Location	Potential cause
Changes I have made seem to have no effect.	Front end, Backoffice	Session lifetime has expired.
Not all catalog versions are visible.	Front end	Restrictions are preventing visibility. The catalog version is not set to active. No category is visible for the current user.
No categories, or not all categories are visible.	Front end	Restrictions are preventing visibility. The user or user group has no read access to the category.
Individual product is not visible.	Front end	Restrictions are preventing visibility. The product status is not set to `approved`.
No catalogs are visible.	Backoffice	Restrictions are preventing visibility. Usergroup-specific Backoffice configuration is preventing visibility for this group. Appropriate Backoffice access rights are not granted for this group.
Not all catalog versions are visible.	Backoffice	Restrictions are preventing visibility.
An attribute is invisible or not present on the expected tab.	Backoffice	Appropriate Backoffice access rights are not granted for this group. Attribute has no explicit Backoffice configuration and is therefore moved to the **Unbound** section.
I get an "Attribute not readable" error.	Backoffice	Appropriate Backoffice access rights are not granted for this group.
Greyed-out editor.	Backoffice	Appropriate Backoffice access rights are not granted for this group.
Greyed-out Delete button.	Backoffice	Appropriate Backoffice access rights are not granted for this group.
Greyed-out Create context menu.	Backoffice	Appropriate Backoffice access rights are not granted for this group.

Problem	Location	Potential cause
An entry (node) in the Explorer Tree is missing.	Backoffice	Usergroup-specific Backoffice configuration is preventing visibility for this group. Appropriate Backoffice access rights are not granted for this group.
A tab is not showing. .	Backoffice	Usergroup-specific Backoffice configuration is preventing visibility for this group.
A section is not showing.	Backoffice	Usergroup-specific Backoffice configuration is preventing visibility for this group.
Non-default attribute editor.	Backoffice	Usergroup-specific Backoffice configuration is preventing visibility for this group.

Front End Visibility Controls

SAP Commerce includes tools to control the visibility of items for customers in the storefront.

If a customer can not see the full list of items you expect (only a number of products, for example), there may be an issue with either your custom extension, or a visibility setting. A good means of finding out whether there is a visibility issue or a source code problem is by assigning the `admin` employee to the session. For more information, see Restrictions.

- If the full list of items appears, it is a visibility issue.
- If only a part of the list of items appears, it is not a visibility issue. Instead, the cause lies somewhere else.

If it is a visibility issue, you can check the following possible causes.

Catalog Version Visibility

By default, only the catalog version marked **active** is visible in the frontend.

Category Visibility

Each category has an attribute that specifies the users or user groups who are allowed to see and browse the category. You can find this under**Category Visibility** on the **General** tab in Backoffice Administration Cockpit, or `allowedPrincipals` in the type system). Users who are not specified for this attribute are not allowed to access the category. Hierarchically structured usergroups inherit visibility settings from their parent.

If no category of a catalog version is visible to a user, then the catalog version will not be visible either: Having a catalog version with no category displayed would result in a catalog version with nothing useful to display in the web front end.

Product Approval Status

Only products with an approval status of **approved** are visible by default in the front end. Products whose status is **unapproved** or **check** are not visible.

Backoffice Visibility Controls

Backoffice has settings to control the visibility of items and features to certain groups within Backoffice Administration Cockpit.

An admin user can restrict the visibility of Backoffice features and functions to employee users and user groups. Such restrictions have no effect on customer users or user groups.

Catalog Version Visibility

You can assign read and write permissions to a catalog version in the Permissions tab of the catalog version details in Backoffice.

A catalog version is accessible to a user if the user has read access to the catalog version. Whether the catalog version is active or not has no effect on this visibility. The user account who creates a catalog version is given read and write access by default. Read or write permissions must be set explicitly for all other users and groups.

Backoffice Access Rights

You can grant or deny user accounts access to Backoffice.

Usergroup Specific Backoffice Configuration

Backoffice allows setting up a specific configuration of features for individual user groups. This makes it possible to hide Backoffice elements from user groups, both in the Explorer Tree and in the Organizer. This does not affect access rights.

Unbound Attributes

Backoffice moves attributes that are not explicitly placed anywhere onto the **Administration** tab of the Type details. Such movement will result if no explicit configuration for the attribute was specified.

Related Information

Restrictions
Catalog Guide
FlexibleSearch

Web Application Endpoint Control

Platform allows you to disable endpoints within web apps. This functionality may be useful especially for REST-oriented applications.

The example below shows how to disable endpoints in a web app.

The example web app offers the following endpoints:

```
/platform
/platform/init
/platform/update
/platform/system
/console/impex/export
/console/impex/import
```

Assume that you want to set up Platform to permanently disable the `/hac/platform/init` and `/hac/platform/update` endpoints. You can do it through a configuration that uses appropriate properties.

The configuration property responsible for disabling endpoints uses the following naming convention:
`endpoint.extensionName.endpointPath.disabled=true|false`, where:

- `endpoint` is a constant part and must be always included
- `extensionName` is the name of the extension that provides the web app; in the example, it is the `hac` extension
- `endpointPath` is a dotted version of the endpoint path, for example, `foo.bar.baz` is the dotted version of the `/foo/bar/baz` path.
- `disabled` is a constant part and must be always included

Here is the property configured to disable the `/hac/platform/init` and `/hac/platform/update` endpoints:

```
endpoint.hac.platform.init.disabled=true
endpoint.hac.platform.update.disabled=true
```

ℹ Note

Any configuration for this functionality is only read during system startup. It is not possible to change it during runtime.

ResourcesGuardService

`ResourcesGuardService` enables you to check whether an endpoint in a web app is configured to be disabled or enabled. It also enables you to manage configuration propagation for endpoint paths and subpaths. You can inject it through Spring by using the `ResourcesGuardService` ID.

Checking Whether Endpoints Are Disabled

`ResourcesGuardService` provides two public methods that enable you to check whether endpoints are disabled or enabled:

- `boolean isResourceDisabled(String extensionName, String resourcePath)`

- `boolean isResourceEnabled(String extensionName, String resourcePath)`

This example shows how to use them:

```
// assuming resourceGuardService was injected by Spring
boolean platformInitDisabled = resourcesGuardService.isResourceDisabled("hac", "/platform/init");
boolean platformUpdateDisabled = resourcesGuardService.isResourceDisabled("hac", "/platform/update");

// assertions
assertThat(platformInitDisabled).isTrue();
assertThat(platformUpdateDisabled).isTrue();
```

Propagating Configuration for Paths and Subpaths

By default, each configuration respects the parent-child hierarchy of paths and subpaths. It means that a configuration set for a given path propagates into all subpaths of that path.

For example, the `/platform` path has these three subpaths: `/platform/init`, `/platform/update`, and `/platform/system`. Disabling only the `/platform` path makes all the three subpaths disabled too - by default and without any further configuration:

```
endpoint.hac.platform.disabled=true
```

To prevent your configuration from propagating from a path into its subpaths, explicitly configure those subpaths. Also, configure `ResourcesGuardService` by setting the `endpoints.guardservice.respect.parents` property value to `false`.

For example, here are some available endpoints:

```
/platform
/platform/init
/platform/update
/platform/system
```

And here is some configuration:

```
endpoints.guardservice.respect.parents=false
endpoint.hac.platform.disabled=true
endpoint.hac.platform.update.disabled=false
```

As a result:

- `/platform/init` and `/platform/system` endpoints inherit the configuration settings from their top-level parent, `/platform`; these endpoints are disabled because their paths aren't explicitly configured otherwise
- you prevent the `/platform` configuration settings from propagating into the `/platform/update` subpath; the `/platform/update` endpoint is not disabled

ResourcesGuardFilter

You need a tool to use all the presented configurations and settings to actually make a particular endpoint disabled. Platform provides a new `Filter` class called `ResourcesGuardFilter`. You can inject it into the standard SAP Commerce filter chain. Configure it on the web app level as in this example:

```
<bean id="myWebAppResourcesGuardFilter" class="de.hybris.platform.servicelayer.web.ResourcesGuardFilter">
    <property name="resourcesGuardService" ref="resourcesGuardService"/>
    <property name="extensionName" value="myWebApp" />
</bean>

<bean id="myWebAppFilterChain" class="de.hybris.platform.servicelayer.web.BackOfficeFilterChain">
    <constructor-arg>
        <list>
            <ref bean="myWebAppResourcesGuardFilter"/>
            <!-- Possible other filters here... -->
        </list>
    </constructor-arg>
</bean>
```

By default, this filter sends the HTTP 404 code whenever it finds that a particular request URI matches a configured endpoint and that endpoint is disabled. It may also send a redirect to a particular page or endpoint if you configure it to do so, for example:

```
<bean id="myWebAppResourcesGuardFilter" class="de.hybris.platform.servicelayer.web.ResourcesGuardFilter">
    <property name="resourcesGuardService" ref="resourcesGuardService"/>
    <property name="extensionName" value="myWebApp" />
    <property name="reditectTo" value="/404.jsp" />
</bean>
```

Hiding Tabs in SAP Commerce Administration Console

If you disable endpoints that are URLs to Administration Console tabs, the tabs become invisible.

Considerations

The provided tools read a configuration and help you determine whether some endpoints in your web app are configured to be disabled. Your web app, however, may have, for example, navigation links to those endpoints on some jsp pages. You need to implement your own logic to use `ResourcesGuardService` if you don't want the end user to see such links.

ServiceLayer

When implementing new business logic, you separate the business code into java classes called services. Each service implements a specific, well-defined requirement. The ServiceLayer is an API for developing services for SAP Commerce.

The topics include:

ServiceLayer Architecture

The ServiceLayer uses a variety of different architecture concepts. Some of these are optional, others are mandatory.

Working with the ServiceLayer

We provide a series of procedures covering various aspects of the ServiceLayer. Use these procedures to get you started quickly with common ServiceLayer user cases.

Key Services Overview

The ServiceLayer offers a variety of key services. These are built in to SAP Commerce and comprise System, Infrastructure, and Platform Services.

Implementing Services

While you may be able to build your entire business application using SAP Commerce services alone, you will no doubt need to implement services yourself. Before you do this, there are certain princples and best practices you should make yourself familiar with.

Transitioning to the ServiceLayer

Previously, all persistence and business logic was written in the Jalo Layer. With the introduction of the Service Layer, all existing business logic in the Jalo Layer has moved to the Service Layer. As the Jalo Layer no longer contains business logic, the public API is significantly smaller.

After Save Event

After each database operation (whether caused by a committed transaction or not), an event is created. This event contains information about the item that was edited and the type of a database operation performed. You can collect these events and handle them according to your needs.

ServiceLayer Architecture

The ServiceLayer uses a variety of different architecture concepts. Some of these are optional, others are mandatory.

Structure Overview

The ServiceLayer can be described as a layer of services on top of the persistence layer. The services themselves can be divided into subcomponents.

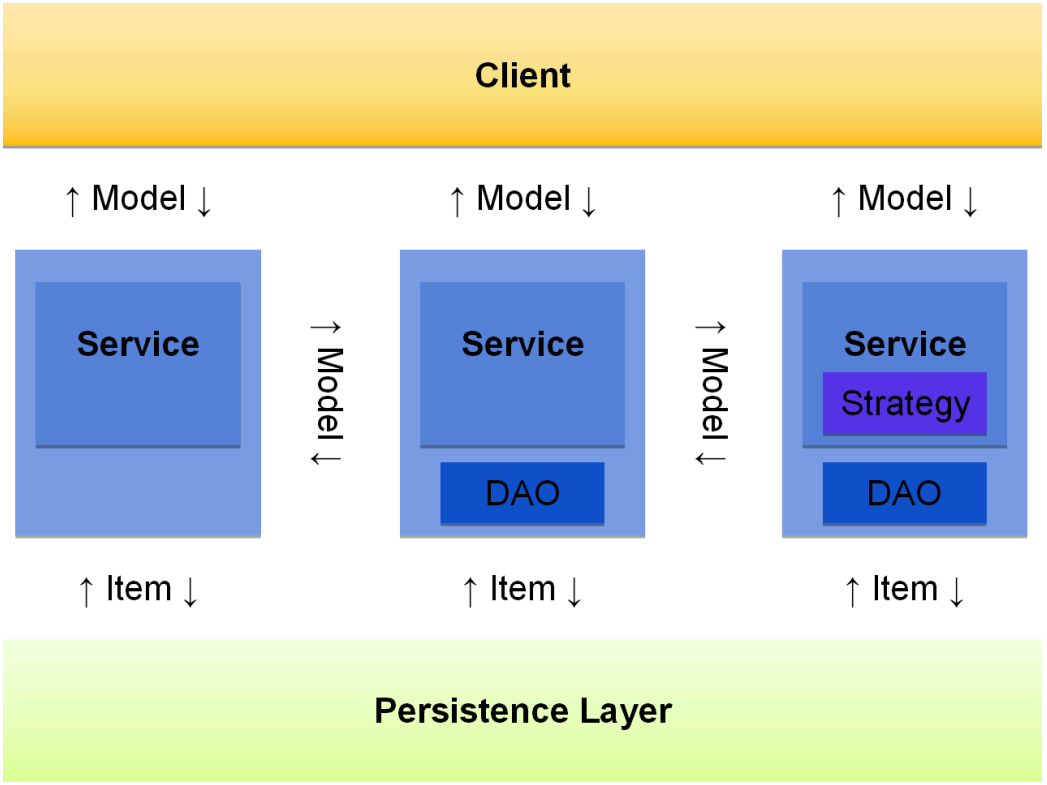


Figure: The ServiceLayer (blue) interconnects the persistence layer with the client. Arrows indicate exchange of data objects.

Architectural Components

Client

A client in this context is any software component that uses the ServiceLayer, such as:

- Page Controllers of an MVC framework
- Web Service clients
- Scripts
- Other services

Services

A service holds the logic to perform business processes and provides this logic through a number of related public methods. These public methods usually are defined in a Java interface. Most often these methods operate on the same kind of model object, for example product, order, and so on.

Services are expected to abstract from the persistence layer, that is, to only contain functional logic and no persistence-related code. That means if you implement a service, make sure to implement it in a way that the underlying implementation is as loosely coupled to the persistence layer as possible.

For more information, see Wikipedia on Services .

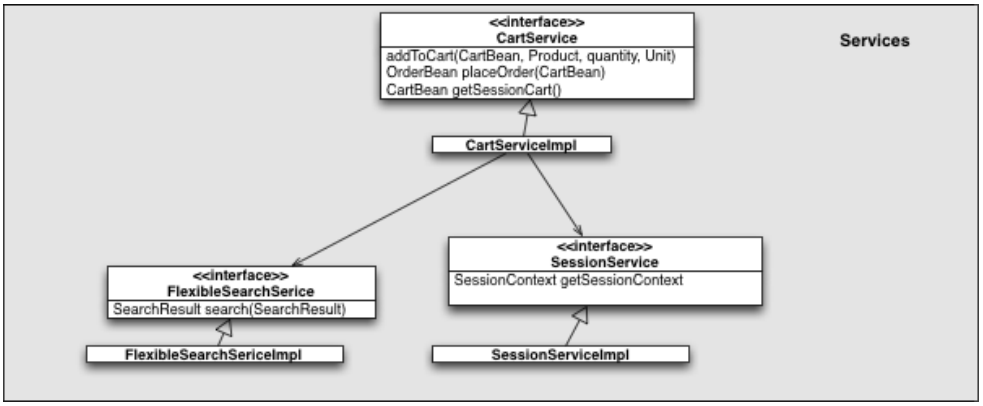


Figure: Sample of relations between services. Note the pattern of having an interface and an implementation per service.

SAP Commerce exposes all of its functionality through services. The following kinds of services are available:

- **Business Services** implement business use cases, such as cart handling or back order.
- **Infrastructure Services** provide the underlying technical foundation, such as internationalization, import, export, and so on.
- **System services** provide functionality required by the ServiceLayer, such as model handling and session handling.

The service methods should be as fine-grained as possible to enable reuse.

Extensions must provide their functionality as services. Per extension you may provide as many services as you deem necessary, not just one.

Services may use other services to perform their tasks but should keep their interdependencies to a minimum to avoid overly tight coupling with other components.

In a project, you may write your own services to either provide unique functionality or to aggregate other services' functionality.

Although technically not necessary, SAP Commerce recommends implementing services in terms of interfaces. See Spring Framework in SAP Commerce for reasons why.

Services interact with other components through models. For details see section Models below.

Strategies

A service may delegate parts of its tasks to smaller micro-services, called strategies. The service then serves as a kind of façade to the strategies. Clients still use the service and its stable API. But under the hood the functionality is split into multiple parts. Because these parts are smaller and very focused to their task, it is easier to adapt or replace them. Strategies therefore help to further encapsulate behavior and make it more adaptable.

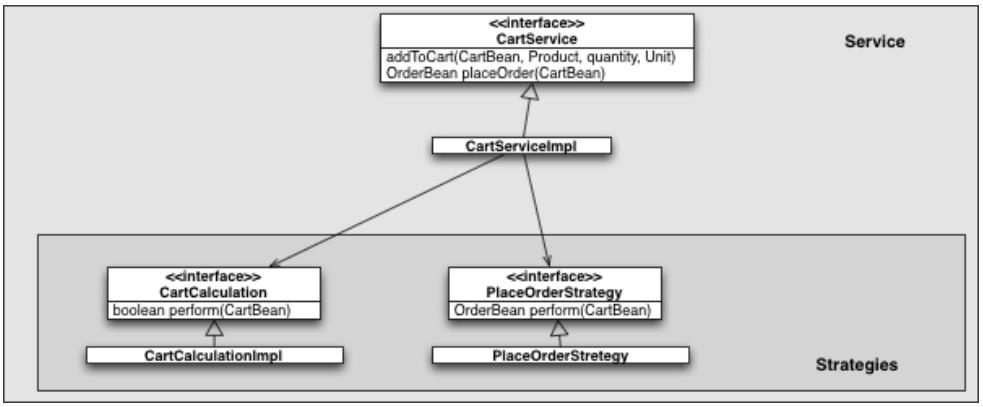


Figure: Sample of a service (consisting of an interface definition and the related implementation) relying on strategies. Note how the strategies also follow the pattern of interface definition and related implementation.

DAOs

A DAO (Data Access Object) is an interface to the storage back end system. DAOs store and retrieve objects. You use DAOs to save, remove, and find models. DAOs are the place to put SQL or `FlexibleSearch` statements and nowhere else. This is to ensure further decoupling from the underlying storage facility. DAOs interact with services via models and with the database via `FlexibleSearch` and SQL statements.

> **Note**
> In SAP Commerce, DAOs use the SAP Commerce Type System for persistence. This means that SAP Commerce DAOs do not implement any individual logic and simply call the underlying persistence layer.

Models

Models are a new way to represent SAP Commerce items. Each model contains all item attributes from all extensions thus unifying access to an item's data. Models are generated from the type system of SAP Commerce, see Type System Documentation. Furthermore they are more or less simple POJOs (Plain Old Java Objects) that can be used without any storage facility. Thus, it's pretty easy to mock them up, for example for testing and debugging.

Models are used by DAOs, services, strategies, converters, and facades.

Related Information

Adding a New Service
Models

Models

Models are a way to represent SAP Commerce items, whereby each of these items is represented by a Model class. Each Model contains all item attributes from all extensions thus unifying access to an item's data.

Models are generated from the SAP Commerce type system. Furthermore, they are similar to POJOs (Plain Old Java Objects) and are used without any storage facility. Consequently, it is easy to mock them for testing and debugging purposes.

Models are used by application developers working with the ServiceLayer and extension developers who create new items with corresponding Model classes.

Basic Model Aspects

There are two basic aspects of Models, depending on the phase of SAP Commerce:

- Model class generation, that is, at compile time. See topic Model Class Generation.
- Model life cycle, that is, at run time. See topic Model Life Cycle.

Model Class Generation

During a SAP Commerce build, the build framework generates Model classes and configuration files for each item type. Models are generated across all extensions, no matter whether the extension is available in source code or in binary only. The Model generation process also ignores the value of the `generated` attribute of the `<coremodule>` element in the `extensioninfo.xml` file of the extension.

Models are generated into the `bootstrap/gensrc` directory.

ⓘ Note

Why Are Models Generated into the gensrc directory?

You might ask why the generated Models end up in the `platform` and not in the extension where the type is defined? This is due to the way the SAP Commerce Build Framework operates.

For example, let's take the `cms` extension. In this extension, the `Catalog` type is extended with an attribute called `Store`. The `Store` type is defined in the `cms` extension but the `Catalog` type is defined in the `catalog` extension. In the build order of SAP Commerce, the `catalog` extension precedes the `cms` extension. Therefore, the `catalog` extension is unaware of the attributes defined in the `cms` extension.

By consequence, putting the generated `CatalogModel` into the `catalog` extension causes a build failure: `CatalogModel` has an attribute of type `StoreModel` that is defined in the `cms` extension but has to be available in the `catalog` extension already.

For every type, an individual Model class is generated. A Model owns the following:

- The name of the type from which the Model is generated, plus the suffix `Model`. For example, the Model for the `Product` type has the name `ProductModel`.
- A similar package as the type from which the Model was generated :
  - The string `model` is added to the package after the extension root, and `jalo` is eliminated from the package.
  - For example, `de.hybris.platform.europe1.jalo.TaxRow` has the Model `de.hybris.platform.europe1.model.TaxRowModel`.
- All attributes that the type has, represented as private fields.
- Getter and setter methods for all attributes

As Models are generated during a early phase in the SAP Commerce build process - before actually building any extension - the Models are available by compile time.

→ Tip

Allows Checking the Data Model More Easily

As Models are generated to match the type system definitions in the `items.xml` files, you can use them to check the data model you have defined. This is faster than updating or initializing the SAP Commerce system and checking the data model in Backoffice, for example.

For more information, see topic Build Framework, section Phases of a Platform Build.

Example: Assume you define an item type like this:

`items.xml`

```
<itemtype
  generate="true"
  code="ContactRequest"
  jaloclass="de.hybris.springmvcdemo.jalo.ContactRequest"
  extends="GenericItem"
  autocreate="true"
  >
    <attributes>
      <attribute qualifier="message" type="java.lang.String">
        <persistence type="property"/>
      </attribute>
    </attributes>
</itemtype>
```

The Model class is then generated into the `modelclasses` folder of the `platform` like this:

- Model as generated:

```
package de.hybris.springmvcdemo.model;

import de.hybris.platform.core.model.ItemModel;

/**
 * Generated Model class for type ContactRequest first defined at extension *springmvcdemo*
 */
@SuppressWarnings("all")
public class ContactRequestModel extends ItemModel
{
        /** <i>Generated type code constant.</i>*/
        public final static String _TYPECODE = "ContactRequest";

        /**
         * <i>Generated constant</i> - Attribute key of <code>ContactRequest.message</code> attribute defined
         * at extension <code>springmvcdemo</code>.
         */
        public static final String MESSAGE = "message";

        /** <i>Generated variable</i> - Variable of <code>ContactRequest.message</code> attribute defined
         * at extension <code>springmvcdemo</code>.
         */
        private String _message;

        /**
         * <i>Generated constructor</i> - for all mandatory attributes.
         * @deprecated Since 4.1.1 Please use the default constructor without parameters
         */
        @Deprecated
        public ContactRequestModel()
        {
                super();

        }

        /**
         * <i>Generated constructor</i> - for all mandatory and initial attributes.
         * @deprecated Since 4.1.1 Please use the default constructor without parameters
         * @param _owner initial attribute declared by type <code>Item</code> at extension <code>core</code>
         */
        @Deprecated
        public ContactRequestModel(final ItemModel _owner)
        {
                super(
                        _owner
                );

        }

        /**
         * <i>Generated method</i> - Getter of the <code>ContactRequest.message</code> attribute defined
         * at extension <code>springmvcdemo</core>.
         * @return the message
         */
        public String getMessage()
        {
                if( !isAttributeLoaded(MESSAGE))
                {
                  this._message = getAttributeProvider() == null ? null : (String) getAttributeProvider().getAttribu
                  getValueHistory().loadOriginalValue(MESSAGE, this._message);
                }
                throwLoadingError(MESSAGE);
                return this._message;
        }

        /**
         * <i>Generated method</i> - Setter of <code>ContactRequest.message</code> attribute defined
         * at extension <code>springmvcdemo</code>.
         *
         * @param value the message
         */
```

```
                public void setMessage(final String value)
                {
                        this._message = value;
                        markDirty(MESSAGE);
                }

        }
```

Note the constructor methods and the generated getter and setter methods for the `Message` attribute.

Modifying the Model Generation

Attributes for Models are generated automatically by default based on attributes of the type, including getter and setter methods for those attributes. You can explicitly exclude attributes of a type from the generation process, or exclude entire types from the generation so that no Model is created for the type.

- To exclude an entire type (including all subtypes) from the generation:

```
<itemtype
  generate="true"
  code="ContactRequest"
  ...  >
    <model generate="false"/>
    <attributes>...</attributes>
</itemtype>
```

- To exclude an attribute from the Model generation process, you have to explicitly define the exclusion. The result is that neither the private field nor getter and setter methods are generated. To exclude an attribute, use `<model generate="false" />` in the attribute definition within the `items.xml` file, such as:

```
<attribute qualifier="message" type="java.lang.String">
  <persistence type="property"/>
  <model generate="false"/>
</attribute>
```

You can specify a constructor to be generated by specifying the attributes part of the constructor signature:

```
<itemtype
  generate="true"
  code="ContactRequest"
  ...  >
    <model>
        <constructor signature="message"/>
    </model>
    <attributes>...</attributes>
</itemtype>
```

This results in a constructor at the model:

```
/**
 * <i>Defined constructor from items.xml</i>
 * @param _message mandatory attribute declared by type <code>ContactRequest</code> at
 * extension <code>impex</code>
 */
public ContactRequestModel(final String _message)
{
        super();
        setMessage(_message);
}
```

You can add alternative getter and setter methods for an attribute:

```
<attribute qualifier="message" type="java.lang.String">
  <persistence type="property"/>
  <model>
    <getter name="myMessage"/>
  </model>
</attribute>
```

An additional getter method `getMyMessage()` is then generated for the corresponding Model:

```
public String getMessage()
{
  ...
}

public String getMyMessage()
{
```

```
    return this.getMessage();
  }
```

Furthermore, you can replace the original getter and setter methods by using `default` flag:

```
<attribute qualifier="message" type="java.lang.String">
  <persistence type="property"/>
  <model/>
    <getter name="myMessage" default="true"/>
  </model>
</attribute>
```

As the result, `getMyMessage` method is generated and the original `getMessage` method does not exist in the Model anymore:

```
public String getMyMessage()
{
  // now executes logic of former getMessage() method
  ...
}
```

Another option is to mark alternative getter and setter methods as deprecated:

```
<attribute qualifier="message" type="java.lang.String">
  <persistence type="property"/>
  <model/>
    <getter name="myMessage" default="deprecated"/>
  </model>
</attribute>
```

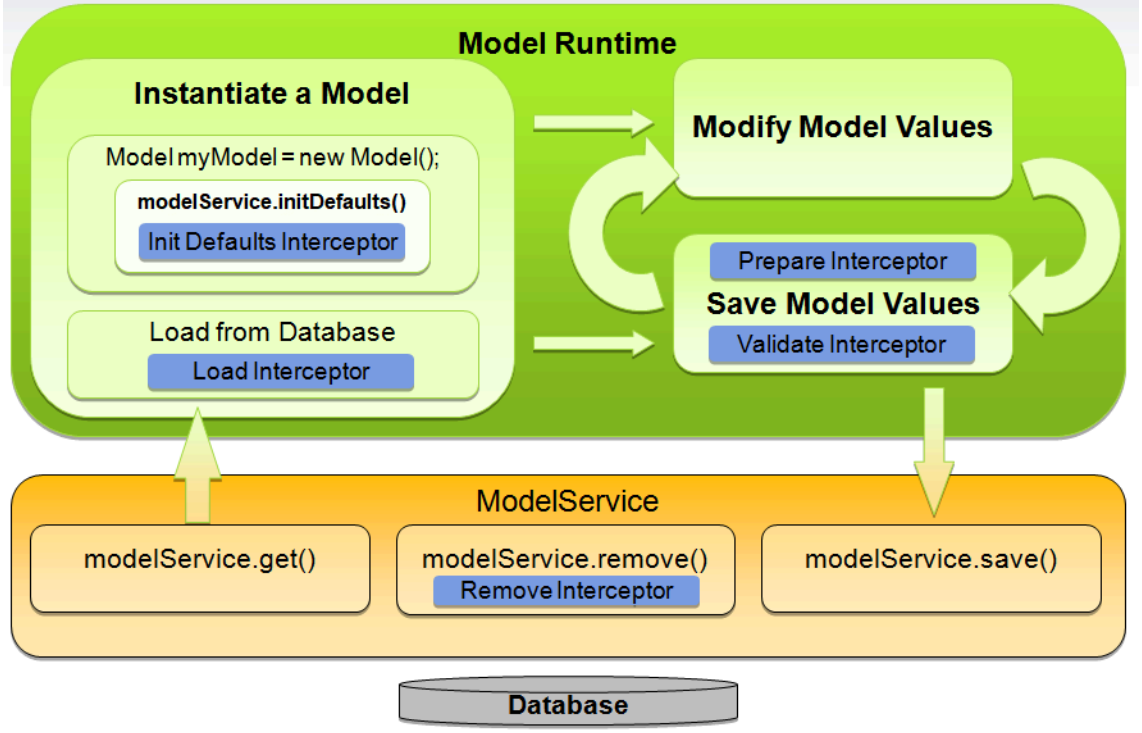
The generated `getMyMessage` method in the Model has `@deprecated` annotation:

```
public String getMessage()
{
  ...
}
/**
 * @deprecated use {@link #getMessage()} instead
 */
@Deprecated
public String getMyMessage()
{
  return this.getMessage();
}
```

Model Life Cycle

A Model represents a state in the database. The representation is not live, that means that modified Model values are not written to the database automatically. Instead, when you modify a Model, you must explicitly save it to the database to have its state reflected there.

Life Cycle of a Model



Life cycle of a model.

Figure: Schematic discussion of a Model's life cycle.

The corresponding phases in a Model's life cycle include:

- **Instantiating the Model**

  This can be done by either creating a new Model instance or by loading a Model from the database.

  - Creating a Model instance

    This can be done by either of these ways. See topic Creating a Model Instance.

    - Through its constructor.
    - Through the factory method in the `ModelService`.
  - Loading an existing Model from the database is possible either by using the **pk** or by using a query expression. See topic Loading a Model.
- **Modifying Model Values** if desired: Set the properties of a Model.
- **Saving Model Values** if created or modified: You save back the Model to update the database. If you have used a new Model, a new record is created in the database; otherwise the existing record is updated.
- **Removing the model**: If the Model is no longer needed, the database record is deleted.

You can use interceptors to hook into the Model's life cycle.

Lazy Loading

Lazy loading is the process of not setting all values of an object right away on the object's instantiation. The Model loading mechanism uses lazy loading.

Once you have loaded a Model, it contains all primitive values. However, at this state, relationships are not filled. As soon as you access such a relationship by calling the appropriate getter, the relationship is loaded on demand. That means that once you have loaded a Model, you do need to worry about loading any dependent Model.

When a Model is loaded, no Model values are loaded right away. The Model consists only of a blank Java object instance with no values set. All Model values are only loaded when any value of the Model is retrieved. This mechanism increases performance during Model initialization.

To change the loading mechanism, set the `servicelayer.prefetch` property in your `local.properties` file to `all` or `literal`. 'literal' behavior means prefetching only atomic attribute values and not prefetching reference attribute values. 'all' means that all attribute values are prefetched upon model loading.

ℹ Note

Loading All Attribute Values

It is not recommended to set the loading mechanism to `All` to avoid loading too much data or, in case of circular dependencies, stack overflow exceptions.

Lazy Loading Model Relations

Since the relationships of a Model are loaded on demand, in certain circumstances it is advisable to avoid calling their getter and setter methods. Consider the following basic example and two approaches, in which you would like to determine the total amount of all orders placed by the user.

To determine the cumulative value of the user's orders from within the ServiceLayer, a simple call to `getOrders()` could be made on the user Model and you could iterate over each order, accumulating the total price. In scenarios where you know the number of orders is relatively small, this is an effective strategy and results in clean code:

`MyServiceImpl.java`

```
...
public Double getTotal(UserModel user ) {
   double cumulativeTotalPrice = 0.0d;
   for (final OrderModel order : user.getOrders())
   {
      cumulativeTotalPrice += order.getTotalPrice().doubleValue();
   }
}
...
```

However, consider a scenario where users may have thousands of orders. For example, in a B2B scenario, an entire company might be represented by one single user account. The first call to getOrders() results in a SQL query that fetches all orders associated with that user. Ideally, the query and the results would be cached, but in a worst case, the query is executed and returns thousands of order rows. For each row in the query result, an order Model is instantiated and then cached.

An alternative approach is to perform this operation in a DAO using a FlexibleSearch query. The query aggregates the total price of each order into a single result with only one row being fetched from the database:

`MyDaoImpl.java`

```
public Double getTotal()
    {
        final FlexibleSearchQuery fq = new FlexibleSearchQuery("SELECT SUM(totalPrice) AS CUMULATIVE_PRICE FROM {Order} WH
        fq.setResultClassList(Lists.newArrayList(Double.class));

        final SearchResult<Double> search = fs.search(fq);
        final List<Double> result = search.getResult();
        if (!result.isEmpty())
        {
            return result.iterator().next();
        }

        return Double.valueOf(0);
    }
```

Lazy Loading Relation and Collection Attributes

You can enable lazy loading mode for loading the relation and collection attributes for Models in `LazyLoadModelList` and `LazyLoadItemSet` for Model getters by using the following property:

```
servicelayer.lazy.collections=true
```

The default value of the property is `false`. This global setting can be overwritten for a particular attribute, for example:

```
servicelayer.lazy.collections.Company.addresses.enabled=true
```

In the example, the property enables the lazy loading mode for the `addresses` collection that is defined in the `Company` Model.

Model Context

As soon as you load a Model or create it through the `modelService`, it is put in the Model context. The Model context keeps track of all changes to the Model, especially references to new, unsaved Models.

If you choose to save a Model individually, note that only unsaved Models are automatically saved. Models that are already created are not saved.

For example, let us assume you have a `CategoryModel`, that holds a reference to a `ProductModel`. You modify the `CategoryModel`:

- If the `ProductModel` is created and not yet saved, then saving the `CategoryModel` saves the `ProductModel` as well.

  This is because the reference to the `ProductModel` is new and new references are always saved.

- If the `ProductModel` is already saved, then saving the `CategoryModel` does not save the `ProductModel`.

  This is because the reference to the `ProductModel` points to an existing Model, and existing Models are not saved.

Because the Model context keeps track of your changes, it is possible to save all changes at once. You do not have to save each Model separately. You can save all Models at once. For details on saving Models, please refer to topic Saving a Model.

If you create a Model by using its constructor, note that the Model is not attached to the Model context. See topic Using a Constructor.

You can manually modify the Model context in order to:

- Add a Model to the context:

  ```
  modelService.attach(model)
  ```

- Remove a Model from the Model context, if you do not want your modifications to be saved automatically:

  ```
  modelService.detach(model)
  ```

Model context is bind to the `HybrisRequestScope`, which is similar to the standard request scope but belongs to a single thread only.

The Model context is automatically cleared if the related session or the related request is closed or times out. By a consequence, unsaved Models are removed and can't pose a threat of memory leak.

The Model created or modified during request is stored in the context. If you load this kind of model from database, for example by using flexible search, you get the same instance of the Model that is stored in the context. You must be aware that this behavior is not guaranteed for loaded but unmodified Models. If you try to display the same object twice without modifying it, you usually get the same object but you cannot depend on this. Better keep reference to this object in your own code.

Keep in mind that the Model context is thread local and Models are not thread safe. You should synchronize access to the setters if you are passing your Models to different threads, for example by using `SessionService`.

You should not use the Model context after transaction rollback. If you try to use Models saved during a transaction which is rolled back, you may get an exception. It depends on your usage pattern. It happens because Models can be in an inconsistent state with their database representation. For example, the Model was saved, its primary key was generated and set but corresponding record in database was not persisted because of rollback operation. Therefore, after rollback you should not use any references to Models retrieved or created during this transaction.

ModelService

The `ModelService` is a service that deals with all aspects of a Model's life cycle. It is available in Spring under the ID `modelService` and implements the `de.hybris.platform.servicelayer.model.ModelService` interface. Its main tasks include the following:

- Loading Models by **pk**
- Loading Models from items
- Creating Models
- Updating Models
- Deleting Models

Creating a Model Instance

There are two ways to create a new Model instance:

- Using a constructor
- Using a factory method

Using a Constructor

You do not need a special create method or other kind of factory. You simply create the new instance using `new`, such as:

```
ProductModel product = new ProductModel();
```

Values of Models are not written to the database directly but only on explicit save. Due to this, you do not have to specify values for mandatory attributes when instantiating the Model. However, by the time you try to save the Model, the values for mandatory attributes must be set, except for the default values.

Model Constructor Methods

Constructor methods for mandatory attributes are deprecated. To instantiate Models, use only the non-argument constructor method, and set the values afterwards, such as:

```
ProductModel product = new ProductModel();
product.setCatalogVersion(catalogVersion);
product.setCode(code);
```

Furthermore, you can use constructor defined at `items.xml` as explained in topic Modifying the Model Generation.

When you create a Model like this, notice it is not attached to the Model context. (See also topic Model Context) There are two ways to do it:

- Using the `ModelService` `save(Object)` method: The Model is saved to the database and automatically attached.

  ```
  modelService.save(Object)
  ```

- Using the `ModelService` `attach(Object)` method: The Model is attached but not saved. Consequently, you either have to manually save it later or do a bulk save.

  ```
  modelService.attach(Object)
  ```

In addition, a Model created that way is not filled with default values as defined in the `items.xml` file. You can fill it as follows:

```
modelService.initDefaults(model);
```

If you do not explicitly call it, the default values are automatically applied during the save process.

Using a Factory Method

You also can use the `ModelService` to create a Model instance, for example by specifying the Model class:

```
ProductModel product = modelService.create(ProductModel.class)
```

Alternatively, you can specify the type's identifier (code):

```
ProductModel product = modelService.create("Product")
```

This is useful at runtime if you dynamically wish to determine the type of a Model to create. Also, this method immediately puts the Model in the Model context. Consequently, you do not have to manually add the Model to the Model context, and the default values are assigned automatically. See also section Model Context above.

Loading an Existing Model

To load an existing Model, you can look up by one of the following:

- Using the Model **pk**
- Using a FlexibleSearch query
- Using an example Model as search parameter

Loading by Primary Key

The simplest case to load a Model is based on its primary key (**pk**). You call the `get` method from `ModelService`:

```
ProductModel product = modelService.get(pk)
```

Loading by Query Expression

Commonly, you wish to look up Models based on a FlexibleSearch query. To do this use the `flexibleSearchService`. It implements the `de.hybris.platform.servicelayer.search.FlexibleSearchService` interface. It is available as a Spring bean with the ID `flexibleSearchService`:

```
FlexibleSearchQuery query = new FlexibleSearchQuery("SELECT {pk} FROM {Product} WHERE {code}
=?" + Product.CODE);
query.addQueryParameter(Product.CODE, code);
SearchResult<ProductModel> result = flexibleSearchService.search(query);
List<ProductModel> products = result.getResult();
```

If no Model is found, `search()` method may throw `ModelNotFoundException`.

You may use `searchUnique()` method from the `FlexibleSearchService` that is similar to `search()` method. The difference is that `searchUnique()` method returns exactly one model or throws one of two types of exceptions:

- `ModelNotFoundException`: If no Model is found
- `AmbiguousIdentifierException`: If more than one Model fulfilling search parameters is found, for example, if you search for `Products` without a `WHERE` clause.

```
FlexibleSearchQuery query = new FlexibleSearchQuery("SELECT {pk} FROM {Product} WHERE {code}
=?code");
query.addQueryParameter("code", code);
ProductModel result = flexibleSearchService.searchUnique(query);
```

Loading by Example Model

It is possible to use an example Model as a search parameter to load existing Models.

Loading by an example Model is moved from `ModelService` to `FlexibleSearchService` and is divided into two methods: `getModelByExample()` and `getModelsByExample()`.

Instead of searching for an existing Model using a FlexibleSearch query, as described in section Loading by Query Expression, you can also create a new example Model, change its attributes and search by this example for an existing Model in the system. Essentially, a Model of the same kind with matching values is returned. The functionality of `getModelByExample()` method does not allow to search for multiple Models. To search for multiple Models, use `getModelsByExample()` method.

To search for existing Models using an example Model:

1. Create a new Model.
2. Change its attributes to match the values you want to search for.
3. When you expect only one result, call the `flexibleSearchService.getModelByExample(...)` method, passing the Model.
4. When you expect more than one result, call the `flexibleSearchService.getModelsByExample(...)` method, passing the Model.

For example:

```
//search for a product with the unique code "test"
ProductModel exampleProduct = new ProductModel();
exampleProduct.setCode("test");
ProductModel foundProduct = flexibleSearchService.getModelByExample(exampleProduct);
```

`getModelByExample()` method can throw two kinds of exception:

- `ModelNotFoundException`: If no Model is found
- `AmbiguousIdentifierException`: If more than one search result is found, for example if you search for `ArticleApprovalStatus.APPROVED`.

To avoid getting `AmbiguousIdentifierException`, use `getModelsByExample()` method, for example:

```
//search for a products with the nonunique ArticleApprovalStatus
ProductModel exampleProduct = new ProductModel();
exampleProduct.setApprovalStatus(ArticleApprovalStatus.APPROVED);
List<ProductModel> foundProducts = flexibleSearchService.getModelsByExample(exampleProduct);
```

ℹ Note

`create()` Method vs `new` Operator

If you create an example Model using the `modelService.create()` method, the Init Defaults Interceptor of this Model is called. As a consequence, default values of the Model are set and used as search parameters, for example:

```
//search for a product with the unique code "test", ApprovalStatus="check", PriceQuantity=1.0
ProductModel exampleProduct = modelService.create(ProductModel.class);
exampleProduct.setCode("test");
ProductModel foundProduct = flexibleSearchService.getModelByExample(exampleProduct);
```

The effective search parameters for this code sample are:

Parameter	Value	Set By
`ProductModel.code`	`test`	Explicit call
`ProductModel.approvalStatus`	`CHECK`	Default value from the `LoadInterceptor` for `ProductModel`
`ProductModel.priceQuantity`	`1.0`	Default value from the `LoadInterceptor` for `ProductModel`

If you create an example Model using `new ProductModel()`, no default values are set and only specified values of parameters are used for search.

It is also possible to search for localized attributes. However, if you do so, be sure to attach the example Model to the `ModelContext`. Without it, no `LocaleProvider` is set, which would result in `java.lang.IllegalStateException: got no locale provider - cannot access default localized getters and setters`.

```
//search for a product where the english name is "uniqueName_english" and the german name "uniqueName_deutsch".
ProductModel exampleProduct = new ProductModel();
exampleProduct.setName("uniqueName_deutsch", Locale.GERMAN);
exampleProduct.setName("uniqueName_english", Locale.ENGLISH);
modelService.attach(exampleProduct); // <- important
```

→ Tip

For additional code samples, please refer to the `FlexibleSearchServiceGetModelByExampleTest.java` file.

Saving a Model

There are two basic means of saving Models:

Saving an Individual Model with Referenced Models, Under Certain Circumstances

To save a Model, call the `modelService` `save(...)` method:

```
modelService.save(model);
```

If a Model holds a reference to another Model and is to be saved, the referenced Models are also saved if they have not been saved before. The referenced Models that have already been saved before are not saved. Other, non-referenced Models are not saved.

For example, if a catalog version holds a new, unsaved `CategoryModel` and the catalog version is saved, then the `CategoryModel` is also saved. This function relies on the Model context. See also section Model Context above.

Saving all Models at Once

```
modelService.saveAll();
```

This saves all modifications as registered with the Model context. See also topic Model Context.

Collections

There is a special behavior when using collections. You can't simply get a collection-based attribute of a Model, modify the collection's contents and call the `ModelService` `save(...)` method. The getter methods of Models return unmodifiable lists, so you can't modify the collection. Instead, you have to:

1. Create a new collection object.
2. Add existing, non-modified values.
3. Add new or modified values.
4. Set the collection to the attribute.
5. Store the attribute by calling the `save(...)` for the Model.

This is the intended behavior to ensure data consistency: You explicitly have to create a new Collection, set values to it, and save the attribute for the collection. Thus, you know which values have been added and stored, and which values have not.

Removing a Model

To remove a Model, call the `remove` method of the `modelService`:

```
modelService.remove(product)
```

Refresh a Model

```
modelService.refresh(product)
```

Note

Refreshing retrieves the Model's values from the cache of a given node, thus overriding the current values. Therefore, unsaved changes are lost.

Converting Between Models and SAP Commerce Items

In some cases, you need to convert a Model into an SAP Commerce item, or vice versa. That is, you might need to switch from a ServiceLayer-based data Model to a Jalo Layer-based data Model, or vice versa. This section discusses how to deal with conversion between Models and items.

Note

You should always avoid accessing Jalo items directly, unless there is no replacement in the ServiceLayer.

Converting a Model to an Item

Sometimes you need to get access to the underlying item of a Model, for example to perform some logic only available in the item class and not yet ported to a service. To convert a Model to an item, use the `getSource(...)` method of the `modelService`:

```
Product productItem = modelService.getSource(productModel)
```

Converting an Item to a Model

Sometimes, instead of using a Model, you get access to an item only. This typically occurs if legacy, Jalo Layer-based code is involved. To make use of this item in your service layer-related code, you have to convert the item to a Model. Use the special `get` method in the `modelService` that takes an item as parameter and returns a Model, such as:

```
final Cart cart = JaloSession.getCurrentSession().getCart();
final CartModel result = modelService().get(cart);
return result;
```

Defining Enums for Models

Models can optionally use Java Enums. That way, you can predefine potential values for the Model's attributes. Typical use cases for Enums are:

- days of the week (Monday, Tuesday, etc.)
- months of a year (January, February, etc.)
- colors of a t-shirt (red, green, blue, etc.)

To define an Enum value for a Model:

1. Define the enum values in the `items.xml` file, such as:

```
<enumtypes>
        <enumtype code="ArticleApprovalStatus" autocreate="true" generate="true">
                <value code="check"/>
                <value code="approved"/>
                <value code="unapproved"/>
        </enumtype>
<enumtypes>
```

2. Define an attribute of the enum type:

```
<attribute qualifier="approvalStatus" type="ArticleApprovalStatus">
        <modifiers read="true" write="true" search="true" optional="false"/>
        <persistence type="property"/>
</attribute>
```

3. Trigger the Model generation. The Model is generated with getter and setter methods for the enum, such as:

```
/**
*Generated method- Getter of theProduct.approvalStatus
* attribute defined at extensioncatalog.
* @return the approvalStatus
*/
public ArticleApprovalStatus getApprovalStatus()
{
if( !isAttributeLoaded(APPROVALSTATUS))
{
this._approvalStatus = (ArticleApprovalStatus) loadAttribute(APPROVALSTATUS);
}
throwLoadingError(APPROVALSTATUS);
return this._approvalStatus;
}

/**
*Generated method- Getter of theProduct.articleStatus
* attribute defined at extensioncatalog.
* @return the articleStatus
*/
public MapgetArticleStatus()
{
return getLocalizedValue(this._articleStatus, ARTICLESTATUS, getCurrentLocale());
}
/**
*Generated method- Getter of theProduct.articleStatus
* attribute defined at extensioncatalog.
* @param loc the value localization key
* @return the articleStatus
* @throws IllegalArgumentException if localization key cannot be mapped to data language
*/
public MapgetArticleStatus(final Locale loc)
{
return getLocalizedValue(this._articleStatus, ARTICLESTATUS, loc);
}

/**
*Generated method- Setter ofArticleApprovalStatus.approvalStatus
* attribute defined at extensioncatalog.
*
* @param value the approvalStatus
*/
public void setApprovalStatus(final ArticleApprovalStatus value)
{
this._approvalStatus = value;
markDirty(APPROVALSTATUS);
}

/**
*Generated method- Setter oflocalized:ArticleStatusMapType.articleStatus
* attribute defined at extensioncatalog.
*
* @param value the articleStatus
*/
public void setArticleStatus(final Mapvalue)
{
setLocalizedValue( this._articleStatus,ARTICLESTATUS, getCurrentLocale(), value );
}
/**
*Generated method- Setter oflocalized:ArticleStatusMapType.articleStatus
* attribute defined at extensioncatalog.
*
* @param value the articleStatus
* @param loc the value localization key
* @throws IllegalArgumentException if localization key cannot be mapped to data language
*/
public void setArticleStatus(final Mapvalue, final Locale loc)
{
setLocalizedValue( this._articleStatus,ARTICLESTATUS, loc, value );
}
```

Related Information

http://static.springframework.org/spring/docs/2.5.x/reference/index.html
Spring Framework in SAP Commerce
Interceptors

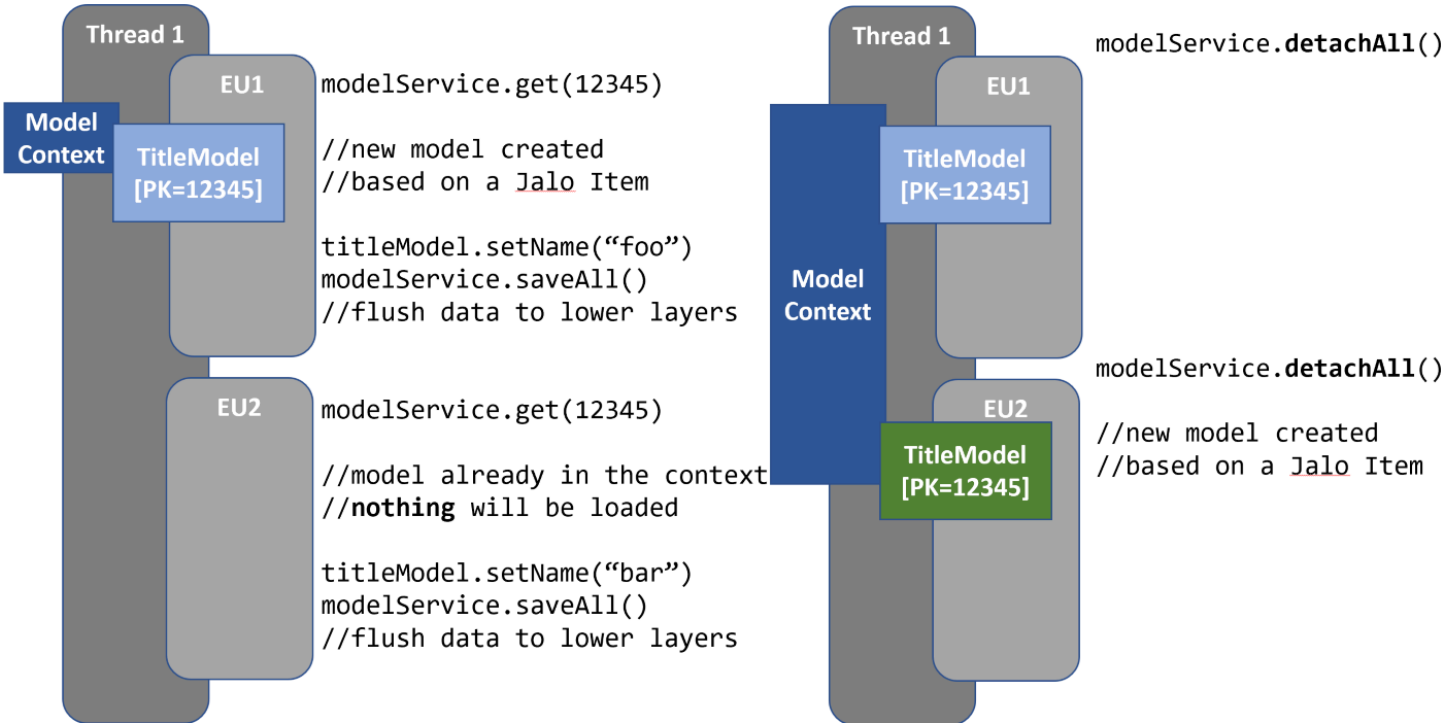
Threads in ModelService

Learn how to use threads in `ModelContext`.

`ModelContext` is an abstraction of the Unit of Work pattern. A Unit of Work:

- loads the state from a database to memory

- manipulates the state in memory
- flushes the effective state from memory to database



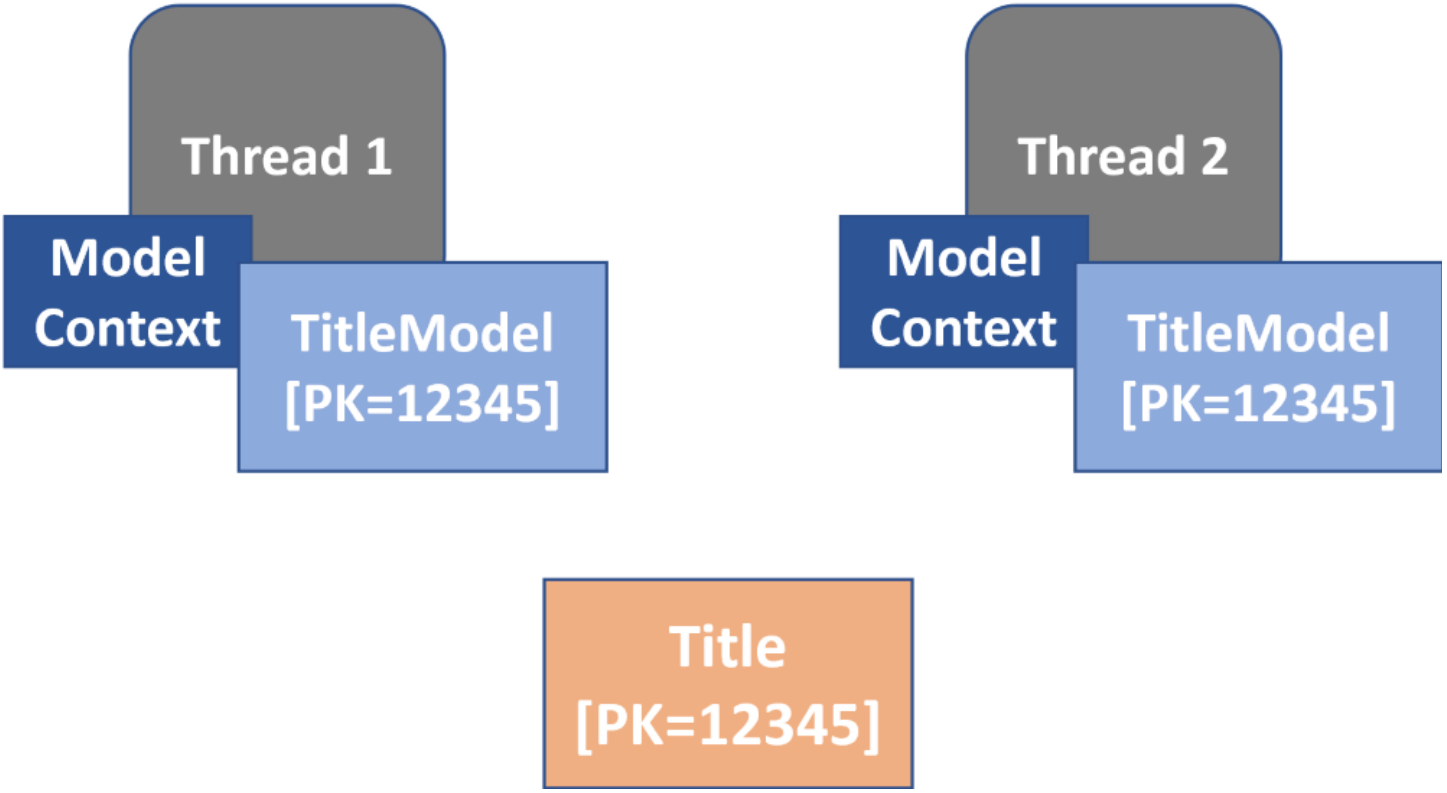
SAP Commerce ensures that `ModelContext` is unset after a new session is established. As a result:

- Activating JaloSession or HttpSession creates new ModelContext.
- Execution units managed by Platform start with clean `ModelContext`.
- Boundaries for the Unit of Work that start when `ModelContext` is created and end when the last operation is executed by the execution unit are provided.

As `ModelContext` is implemented in SAP Commerce as `ThreadLocal`, it provides isolation between Execution Units (threads) even without any transactions. If you use your own execution units, you need to provide such boundaries on your own. See an example of two methods executed one by one in the same Execution Unit:

```
public void setName()
{
TitleModel titleModel = modelService.get(PK.fromLong(12345L));
//new model created
//based on a Jalo Item
titleModel.setName("foo")
modelService.saveAll()
//flush data to lower layers
}

// in the meantime another thread modifies titleModel with 12345 PK

public String getName()
{
TitleModel titleModel = modelService.get(PK.fromLong(12345L));
//model already in the context
//nothing will be loaded

return titleModel.getName()
}
```

If you don't provide any boundaries, the `getName()` method always returns "foo", even if another Execution Unit has changed `titleModel` between the `setName()` and `getName()` calls.



One of the possible solutions is to use the `modelService.detachAll()` method before the `modelService.get()` call as in the following example:

```
public void setName()
{
modelService.detachAll();
TitleModel titleModel = modelService.get(PK.fromLong(12345L));
//new model created
//based on a Jalo Item
titleModel.setName("foo")
modelService.saveAll()
//flush data to lower layers
}

// in the meantime another thread modifies titleModel with 12345 PK

public String getName()
{
modelService.detachAll();
TitleModel titleModel = modelService.get(PK.fromLong(12345L));
//new model created
//based on a Jalo Item

return titleModel.getName()
}
```

The `getName()` method returns the name value from the database. Depending on what happens in other Execution Units between the call to set the `setName()` and `getName()` methods, the returned name could be anything, including "foo".

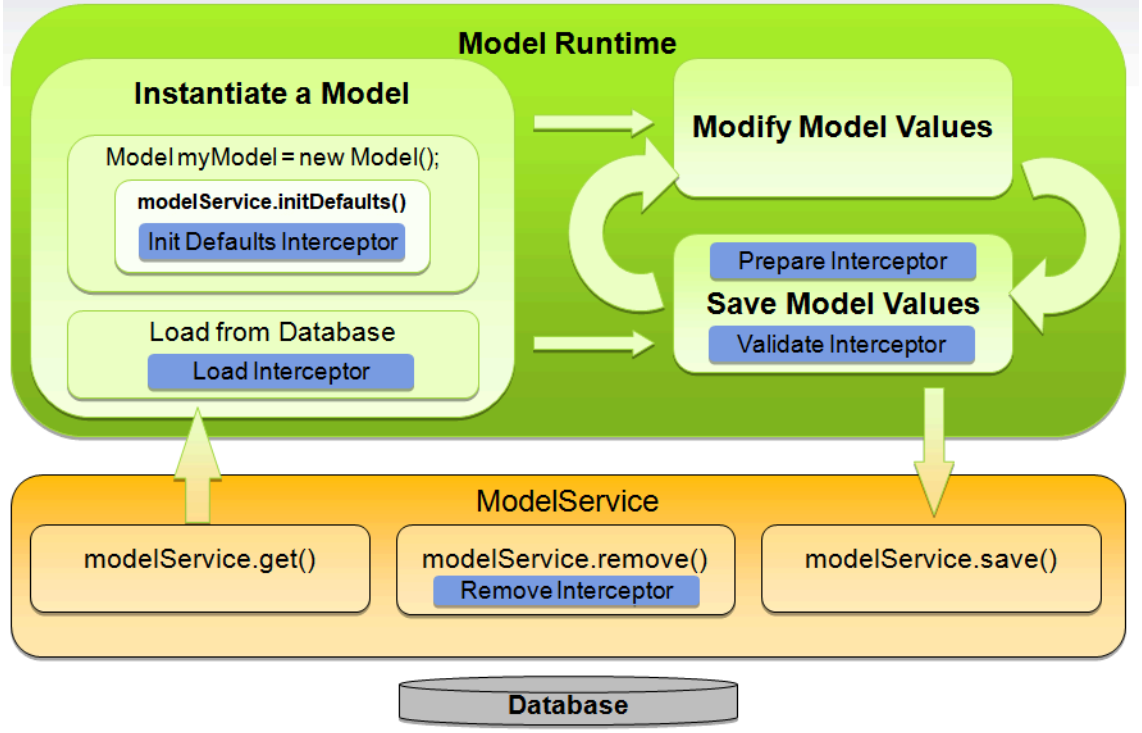
Not invoking the `modelService.detachAll()` method in combination with `ThreadPools` or `Executors` increases the probability of using stale data.

Interceptors

Interceptors check whether constraints set for the behavior of life cycles of models are fulfilled. Use interceptors to make sure you delete or persist the correct data **only**, and that you handle data properly.

To intercept the behavior of life cycles of models, various types of interceptors have been developed. Each such interceptor addresses a particular step of the life cycle. When the life cycle of a model reaches a certain step, a corresponding interceptor is activated. During the interception, it is possible to modify the model or raise an exception to interrupt the step. For example, you can check that certain values are set for a model before the model is saved. After implementing an interceptor, you need to register it as a Spring bean.

Life Cycle of a Model



Life cycle of a model.

Implement an Interceptor

To implement an interceptor you have to implement one of the following interfaces. Apart from the model that is being intercepted, a second argument is passed to the interceptor method, the interceptor context. It provides information about other models in the current context as well as useful utility methods.

→ Tip

Interceptors can do much more. For example, the `remove` interceptor can store removed items in a separate table, but only those items that are validated by an additional `validate` interceptor.

Type of Interceptor	Description	Interface to be Implemented

Load Interceptor	The Load Interceptor is called whenever a model is loaded from the database. You may want to use this interceptor if you want to change values of the model after load. An exception raised during execution prevents the model from being loaded.	`LoadInterceptor` interface (`de.hybris.platform.servicelayer.interceptor` package):
		`public interface LoadInterceptor extends Interceptor`
		`{`
		`void onLoad(Object model, InterceptorContext ctx) throws InterceptorException;`
		`}`

Init Defaults Interceptor	The Init Defaults Interceptor is called when a model is filled with its default values. This happens either when it is created via the `modelService.create` method or when the `modelService.initDefaults` method is called. You can use this interceptor to fill the model with additional default values, apart from the values defined in the `items.xml` file. For details, see items.xml.	`InitDefaultsInterceptor` interface (`de.hybris.platform.servicelayer.interceptor` package):
		`public interface InitDefaultsInterceptor extends Interceptor`
		`{`
		`void onInitDefaults(Object model, InterceptorContext ctx) throws InterceptorExcept`
		`}`

Prepare Interceptor	The Prepare Interceptor is called before a model is saved to the database **before** it is validated by Validate interceptors, see below. Use this to add values to the model or modify existing ones before they are saved. An exception raised during execution	`PrepareInterceptor` interface (`de.hybris.platform.servicelayer.interceptor` package):
		`public interface PrepareInterceptor extends Interceptor`
		`{`
		`void onPrepare(Object model, InterceptorContext ctx) throws InterceptorException;`
		`}`

Type of Interceptor	Description	Interface to be Implemented
	prevents the model from being saved. **Note** Prepare interceptor is called before the impex translators.	**Note** Do not use this interceptor to perform validation. Use the **Validate Interceptor** instead.

Validate Interceptor	The Validate Interceptor is called before a model is saved to the database **after** is been prepared by the Prepare interceptors, above. You can use Validate Interceptors to validate values of the model and raise an **InterceptorException** if any values are not valid.	`ValidateInterceptor` interface (`de.hybris.platform.servicelayer.interceptor` package):
		
		`public interface ValidateInterceptor extends Interceptor`
		`{`
		`  void onValidate(Object model, InterceptorContext ctx) throws InterceptorException;`
		`}`
		
		Note
		Do not use this interceptor to fill the model with values or otherwise prepare it for saving. Use the **Prepare Interceptor** instead.

Remove Interceptor	The Remove Interceptor is called before a model is removed from the database. You can use this interceptor, for example:
	• To remove models that are related to the model but are not in the model context.
	• To prevent the removal of the model by raising an **InterceptorException**.
		`RemoveInterceptor` interface (`de.hybris.platform.servicelayer.interceptor` package):
		
		`public interface RemoveInterceptor extends Interceptor`
		`{`
		`  void onRemove(Object model, InterceptorContext ctx) throws InterceptorException;`
		`}`

Register an Interceptor

After implementing an interceptor, register it as a Spring bean.

1. To register an interceptor, add it to the Spring application context XML:

   `myextension-spring.xml`

   ```
   <bean id="myValidateInterceptor" class="mypackage.MyValidateInterceptor"
             autowire="byName"/>
   ```

   The *<id>* is used in the following bean.

Add a `de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping` to the XML file:

`myextension-spring.xml`

```
<bean id="MyValidateInterceptorMapping"
      class="de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping">
        <property name="interceptor" ref="myValidateInterceptor"/>
        <property name="typeCode" value="MyType"/>
        <property name="replacedInterceptors" ref="uniqueCatalogItemValidator"/>

        <!-- The order property is only effective with 4.1.1 and later -->
        <property name="order" value="5000"/>
</bean>
```

Property in InterceptorMapping	Mandatory?	Description
<interceptor>	Yes	A reference to the actual interceptor.
<typeCode>	Yes	The code of the type to intercept. Note that all subtypes are intercepted, too.
<order>	No	The running order of all registered Interceptors for a given typecode and its subtypes can be ordered by this

Property in InterceptorMapping	Mandatory?	Description
		property. The Interceptor with the lowest number is called first. If this property is not set, the highest integer number is used by default. Any SAP Commerce Interceptor has no order number set by default. If two or more interceptors have the same order value (or no order value), then the exact order in which these interceptors trigger is not deterministic.
<replacedInterceptors>	No	A list of Interceptor bean id's (not the corresponding *<InterceptorMapping>* id) which are replaced by this current Interceptor.

Interceptor Enhancements

Interceptors used to be limited to the scenario in which they were executed. For example, `PrepareInterceptor` could only register new models to be saved, and `RemoveInterceptor` could only register new models to be removed.

Currently the `Prepare` and the `Remove` interceptors can both register models and specify a `PersistenceOperation` to be performed with these models. `PersistenceOperation` is an enumeration with the following values:

- *<SAVE>*
- *<DELETE>*

This simple functionality opens up many new possibilities like:

- Generating audit entries for removed items
- Custom handling of orphaned attribute values (*<partOf>*-like)
- etc.

From the interceptor point of view, the most important changes took place in the `InterceptorContext`

```
void registerElementFor(final Object model, PersistenceOperation operation);
    boolean contains(Object model, PersistenceOperation operation);
    Set<Object> getElementsRegisteredFor(PersistenceOperation operation); interface. There
                              are a few new methods:
```

These methods allow an interceptor to:

- register a model (specifying operation to be performed),
- check if the given model is already registered using the given operation,
- get all models registered for the given operation.

ℹ Note

To modify models within interceptors, use `registerElementFor()` instead of the `modelService.save()` and `modelService.remove()` methods.

To keep backward compatibility, and the behavior of existing methods without the *<operation>* parameter, `InterceptorContext` always has a *<default operation>*, which depends on the scenario in which the interceptor is executed. If the entry point to `modelService` was one of `save*` methods, the default operation for `InterceptorContext` is *<SAVE>*, which means that all models registered by interceptors without specifying a `PersistenceOperation` are registered with the *<SAVE>* operation. Accordingly, if the entry point to `modelService` is one of the `remove*` methods, the default operation for `InterceptorContext` is *<DELETE>*, which means that all models registered by interceptors without specifying `PersistenceOperation` are registered with *<DELETE>* operation.

Below you can find a list of deprecated methods with the recommended replacements:

Deprecated Method	Current Behaviour of Deprecated Method	Recommended Method	Justification
`Set<Object> getAllRegisteredElements()`	Returns all models registered for default persistence operation	`Set<Object> getElementsRegisteredFor(PersistenceOperation operation)`	The new method clearly states which models to return (registered for which operation)
`void registerElement(final Object model, final Object`	Registers element for default operation, providing	`void registerElement(final Object model)` or	Deprecated method uses Jalo "source" object

Deprecated Method	Current Behaviour of Deprecated Method	Recommended Method	Justification
`source)`	additionally its source	`void registerElementFor(final Object model, PersistenceOperation operation)`	
`boolean contains(Object model)`	Checks if the given model is already registered for the default persistence operation.	`boolean contains(Object model, PersistenceOperation operation)`	The new method clearly states which models to consider (registered for which operation)

ℹ Note

To avoid issues with thread or process synchronization, do not use the `eventService.publishEvent()` method from within interceptors.

Example - Creating a RemoveInterceptor that Stores Deleted Users in a Separate Table.

1) Define an item that stores the data of each deleted user:

`items.xml`

```
<itemtype code="UserAuditEntry" generate="true" autocreate="true">
                                <deployment table="UserAuditEntries" typecode="8998"/>
                                <attributes>
                                        <attribute qualifier="uid" type="java.lang.String">
                                                <persistence type="property"/>
                                        </attribute>
                                        <attribute qualifier="name" type="java.lang.String">
                                                <persistence type="property"/>
                                        </attribute>
                                        <attribute qualifier="displayName" type="java.lang.String">
                                                <persistence type="property"/>
                                        </attribute>
                                        <attribute qualifier="changeTimestamp" type="java.util.Date">
                                                <persistence type="property"/>
                                        </attribute>
                                </attributes>
                        </itemtype>
```

2) Create an interceptor that creates an instance of the above item each time a user is deleted:

```
public class AuditingUserRemoveInterceptor implements RemoveInterceptor
        {
                @Override
                public void onRemove(final Object o, final InterceptorContext ctx) throws InterceptorException
                {
                        if (o instanceof UserModel)
                        {
                                final UserModel user = (UserModel) o;
                                final UserAuditEntryModel auditEntryModel = ctx.getModelService().create(UserAuditEntryMod
                                auditEntryModel.setChangeTimestamp(new Date());
                                auditEntryModel.setDisplayName(user.getDisplayName());
                                auditEntryModel.setName(user.getName());
                                auditEntryModel.setUid(user.getUid());
                                ctx.registerElementFor(auditEntryModel, PersistenceOperation.SAVE);
                        }
                }
        }
```

3) Register the interceptor in the spring context using `InterceptorMapping`.

Example: Validating UserAuditEntryModels

Optionally, we can consider a more contrived scenario where we want to validate `UserAuditEntryModels` and only allow those where the username is not empty.

To achieve this goal, let's define the following `ValidateInterceptor`:

```
public class AuditEntryValidateInterceptor implements ValidateInterceptor
        {
                @Override
                public void onValidate(final Object o, final InterceptorContext ctx) throws InterceptorException
                {
                        if (o instanceof UserAuditEntryModel)
                        {
                                final UserAuditEntryModel auditEntry = (UserAuditEntryModel) o;
                                if (StringUtils.isEmpty(auditEntry.getName()))
```

```
                {
                        throw new InterceptorException("User audit entries cannot have empty username");
                }
            }
        }
    }
```

When this interceptor is registered, all *<UserAuditEntries>* created by the `AuditingUserRemoveInterceptor` are validated with the above interceptor. If the validation fails, all changes are rolled back (the user is not removed and *<UserAuditEntry>* is not created).

Related Information

Models

Disabling Interceptors

You can disable interceptors in a session if necessary. You can do it programmatically by using the `sessionService` class, or declaratively via ImpEx import.

You can disable interceptors for data integration scenarios to relax certain constraints imposed on a data model. Disabling interceptors may also be useful from the performance point of view. If you are sure that your data doesn't violate a particular interceptor, you can disable it for a given operation.

Disable Interceptors Programmatically

To disable interceptors in code, use the `sessionService.executeInLocalViewWithParams` method. It is an implementation of a template method pattern and takes a map of attributes that are set in a session before execution of the provided callback method, and reverted afterwards.

There are three attributes that allow you to disable specific interceptors during execution of callback:

- `disable.interceptor.beans` (`InterceptorExecutionPolicy#DISABLED_INTERCEPTOR_BEANS` constant) This attribute takes a set of Spring bean IDs.
- `disable.interceptor.types` (`InterceptorExecutionPolicy#DISABLED_INTERCEPTOR_TYPES` constant) This attribute takes a set of interceptor types (`de.hybris.platform.servicelayer.interceptor.impl.InterceptorExecutionPolicy.InterceptorType`) that you want to disable.
- `disable.UniqueAttributesValidator.for.types` (`InterceptorExecutionPolicy#DISABLED_UNIQUE_ATTRIBUTE_VALIDATOR_FOR_ITEM_TYPES`) This attribute takes a set of item types for which you want to disable `UniqueAttributesValidator`.

The following example of a currency model has the out-of-the box defined `ValidateInterceptor`. This interceptor checks if a currency digit is not a negative number. If it is, the following code fails:

```
final CurrencyModel currency = modelService.create(CurrencyModel.class);
currency.setSymbol("$");
currency.setIsocode("USD");
currency.setDigits(-1);
modelService.save(currency); // throws ModelSavingException caused by InterceptorException
```

The following code snippet shows how to disable all validate interceptors so that you can save the data successfully. Currently, this method only supports the validator type interceptors:

```
final Map<String, Object> params = ImmutableMap.of(InterceptorExecutionPolicy.DISABLED_INTERCEPTOR_TYPES,
                ImmutableSet.of(InterceptorExecutionPolicy.InterceptorType.VALIDATE));
sessionService.executeInLocalViewWithParams(params, new SessionExecutionBody()
{
    @Override
    public void executeWithoutResult()
    {
        final CurrencyModel currency = modelService.create(CurrencyModel.class);
        currency.setSymbol("$");
        currency.setIsocode("Dollar");
        currency.setDigits(-1);

        modelService.save(currency);    // save successful - all validate interceptors are disabled
```

```
    }
});
```

You can also choose to be more granular and disable specific interceptors on a per-bean basis:

```
final Map<String, Object> params = ImmutableMap.of(InterceptorExecutionPolicy.DISABLED_INTERCEPTOR_BEANS, ImmutableSet.of(

sessionService.executeInLocalViewWithParams(params, new SessionExecutionBody()
{
    @Override
    public void executeWithoutResult()
    {
        final CurrencyModel currency = modelService.create(CurrencyModel.class);
        currency.setSymbol("$");
        currency.setIsocode("Dollar");
        currency.setDigits(-1);
        modelService.save(currency);    // save successful - validateCurrencyDataInterceptor interceptor is disabled
    }
});
```

In case we know that there is no `Currency` with the `Dollar` IsoCode, we can disable `UniqueAttributesValidator` for the `Currency` type, as shown in the code:

```
final Map<String, Object> params = ImmutableMap.of(InterceptorExecutionPolicy.DISABLED_UNIQUE_ATTRIBUTE_VALIDATOR_FOR_ITEM

sessionService.executeInLocalViewWithParams(params, new SessionExecutionBody()
{
        @Override
        public void executeWithoutResult()
        {
                final CurrencyModel currency = modelService.create(CurrencyModel.class);
                currency.setSymbol("$");
                currency.setIsocode("Dollar");
                currency.setDigits(-1);
                modelService.save(currency);    // save successful - UniqueAttributesValidator not called
        }
});
```

Disable Interceptors via ImpEx

Similarly, the following import fails:

```
INSERT_UPDATE Currency;isocode[unique=true];digits;symbol
;EUR_Test;-2;€
```

with the following exception:

```
[de.hybris.platform.servicelayer.i18n.interceptors.ValidateCurrencyDataInterceptor@13097d1b]: Number of digits for
CurrencyModel must be greater than 0.;EUR;-2;€
```

To disable all validator type interceptors, use the `disable.interceptor.types=validate` header attribute:

```
INSERT_UPDATE Currency[disable.interceptor.types=validate];isocode[unique=true];symbol;digits;
;EUR_Test2;$;-2;
```

To disable specific interceptors, specify a comma-separated bean-IDs list for the `disable.interceptor.beans` header attribute. The following impex specifies only one such interceptor:

```
INSERT_UPDATE Currency[disable.interceptor.beans='validateCurrencyDataInterceptor'];isocode[unique=true];symbol;digits;
;EUR_Test;$;-2;
```

To disable `UniqueAttributesValidator`, specify a comma-separated item types for which you want to disable it:

```
INSERT_UPDATE Currency[disable.UniqueAttributesValidator.for.types='Currency'];isocode[unique=true];symbol;digits;
;EUR_Test;$;-2;
```

If you want to specify more than one ID, put the bean IDs in apostrophes: *<'beanID'>*.

Related Information

API Documentation and YAML Files
JavaDocs

Available Interceptors

See interceptors used in Platform.

JDBCInterceptor

`JDBCInterceptor` allows the database server some time after a recoverable exception on the database side occurs.

These properties allow you to get a list of exceptions considered as recoverable:

- `jdbc.recovery.common.recoverable.error.codes`
- `jdbc.recovery.sqlserver.recoverable.error.codes` (defaults to 4060, 40197, 40501, 40613, 49918, 49919, 49920, 4221)
- `jdbc.recovery.postgresql.recoverable.error.codes`
- `jdbc.recovery.mysql.recoverable.error.codes`
- `jdbc.recovery.hsqldb.recoverable.error.codes`
- `jdbc.recovery.oracle.recoverable.error.codes`
- `jdbc.recovery.sap.recoverable.error.codes`

These properties allow you to list unrecoverable exceptions:

- `jdbc.recovery.common.nonrecoverable.error.codes`
- `jdbc.recovery.sqlserver.nonrecoverable.error.codes`
- `jdbc.recovery.postgresql.nonrecoverable.error.codes`
- `jdbc.recovery.mysql.nonrecoverable.error.codes`
- `jdbc.recovery.hsqldb.nonrecoverable.error.codes`
- `jdbc.recovery.oracle.nonrecoverable.error.codes` (defaults to 17002, 17008)
- `jdbc.recovery.sap.nonrecoverable.error.codes`

`SQLRecoveryStrategy` waits up to 60 seconds and checks whether the database server responds.

You can configure timeouts using these properties:

- `jdbc.recovery.backoff.initial.seconds=10`
- `jdbc.recovery.backoff.increase.factor=1.3333333334`
- `jdbc.recovery.backoff.max.seconds=60`

Hooks for Initialization and Update Process

Use the `@SystemSetup` annotation in any ServiceLayer class to hook ServiceLayer code into the SAP Commerce initialization and update life-cycle events. In this way, you can provide a means for creating essential and project data for any extension.

By using annotations, no interfaces have to be implemented or parent classes extended, keeping the classes loosely coupled. Methods are called in an order that respects the extension build order. You can define when a method should be executed, and pass context information to it.

Spring Configuration of Java Class

The Java classes, in which the `@SystemSetup` annotation is used, must be registered as Spring beans in the `ApplicationContext`:

```
<bean id="someclass" class="de.hybris.sample.extension.SomeClass"  />
```

Related Information

ServiceLayer

Java Class Annotations

See how to use Java Class Annotations.

Annotate the classes whose methods are to be executed. In the following example, the method `createImportantModelDuringInitProcess` is executed during the initialization process when the essential data for extension `myextension` is created.

```
@SystemSetup(extension = MyExtension.EXTENSIONNAME)
public class SomeClass ... {
    @SystemSetup(extension = MyExtension.EXTENSIONNAME, process = Process.INIT, type = Type.ESSENTIAL)
    public void createImportantModelDuringInitProcess(){
       //create the model here
    }
...
```

Note that:

- The `@SystemSetup` annotation is recognized for public methods only.
- If the annotation is attached to a class, the defined values are used as the default for all annotated methods within the class.
- It is possible to annotate any number of methods for a class.
- If the annotation attribute `extension` is not defined, the default value is an empty String. It means that the annotated code is not executed as there is no extension with the name "".
- If the annotation attributes `process` and `type` are not defined, the default ALL is used.

Below are further examples:

```
//AnotherJavaClass.java
@SystemSetup(extension = MyExtension.EXTENSIONNAME, process = Process.UPDATE, type = Type.ESSENTIAL)
public class AnotherJavaClass
{
  @SystemSetup
  protected void ignoredMethodOne() {}

  public void ignoredMethodTwo() {}
  //Neither of the above will be executed because only public methods with the annotation will be recognized

  @SystemSetup(extension = "myOtherExtension") {}
  //Although this class is located in myextension, it is called during the update
  //process and creating essential data for the extension myOtherExtension
}


@SystemSetup(extension = "myextension")
public class DifferentJavaClass
{
  @SystemSetup(type = Type.ESSENTIAL)
  public methodOne() {}

  @SystemSetup(type = Type.PROJECT)
  public methodTwo() {}

  // methodOne will be executed during creation of essential data in myextension, in both init and update processes
  // methodTwo will be executed during creation of project data in the same extension, in both init and update
  // processes
}
```

Using the code above, you can, for example, import ImpEx files in a specific order during the initialization process:

```
@SystemSetup(extension = MyExtension.EXTENSIONNAME)
public class NaoCoreInitializer
{
        @SystemSetup(type = Type.ESSENTIAL)
        public static void setupEssential()
        {
                ImpexUtils.impex("essentialdata-1.impex");
                ImpexUtils.impex("essentialdata-2.impex");
        }

        @SystemSetup(type = Type.PROJECT)
        public static void setupProject()
        {
                ImpexUtils.impex("projectdata-1.impex");
                ImpexUtils.impex("projectdata-2.impex");
        }
```

```
}
```

Annotation Attributes

See the available annotation attributes.

The following annotation arguments are available:

- `extension`: Defines, in which extension the code should be executed. The order of extensions is given by the build order. Use the `extensionname` constant, for example `MyExtension.EXTENSIONNAME`.
- `process`: Defines, whether the code should be executed during the initialization or update process.
  - `INIT`: Code is executed only during the initialization process.
  - `UPDATE`: Code is executed only during the update process.
  - `ALL` (Default): Code is executed twice, during the initialization and during the update process.
- `type`: Defines if the code should be executed during the essential data creation or project data creation process.
  - `ESSENTIAL`: Code is executed during the essential data creation process.
  - `PROJECT`: Code is executed during the project data creation process.
  - `ALL` (Default): Code is executed twice, during the essential data creation and during the project data creation process.

The `@SystemSetup` annotation flags designed for the Patching Framework include:

- `patch`: marks a logic as one to be executed only once
- `required`: determines whether a logic is to be executed or not
- `name`: allows you to provide a name of a logic to be executed
- `description`: allows you to provide a longer logic description

Parameters

See how to use parameters.

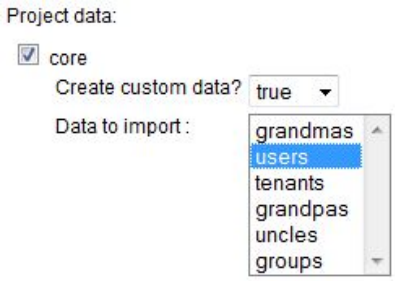
If you need any additional information in your annotated methods, you can use the `SystemSetupContext`.

The following code sample shows how to use the `SystemSetupContext`:

```
@SystemSetup(type = Type.PROJECT, method = Method.ALL, extension = "myExtension")
public void createData(SystemSetupContext context)
{
        if (context.getParameterMap().containsKey("createAdditionalStuff") &&
       "true".equalsIgnoreCase(context.getParameterMap().get("createAdditionalStuff").toString()))
        {
                if (context.getMethod().isInit())
                {
                        // for example, delete everything first
                }

                // create additional stuff
        }

        // create normal stuff
}
```

Usage of User-Defined Parameters

Sometimes it is helpful to ask the customer what to do, for example:



For defining user-specific parameters, you have to add a method annotated with `@SystemSetupParameterMethod` inside a `SystemSetup` annotated class that returns a list of parameters. Here is the example according to the screenshot above:

```
@SystemSetupParameterMethod
public List<SystemSetupParameter> getSystemSetupParameters()
{
        final List<SystemSetupParameter> params = new ArrayList<SystemSetupParameter>();

        final SystemSetupParameter customDataParameter = new SystemSetupParameter("createCustomData");
        customDataParameter.setLabel("Create custom data?");
        customDataParameter.addValue("true");
        customDataParameter.addValue("false", true);
        params.add(customDataParameter);

        final SystemSetupParameter imports = new SystemSetupParameter("imports");
        imports.setMultiSelect(true);
        imports.setLabel("Data to import : ");
        imports.addValue("users", true);
        imports.addValues(new String[]
        { "groups", "tenants", "grandmas", "grandpas", "uncles" });
        params.add(imports);

        return params;
}
```

Note that:

- `customData` parameter is a single selection list box.
- `imports` parameter is a multiple selection list box.

After the customer has made his decision according to the parameters you offered, you can access the result by evaluating the `SystemSetupContext`. The parameters are stored in the map with the prefix `<extension>_` to avoid having duplicate parameters. If you are using a single selection list box, then you can easily use the `SystemSetupContext.getParameter(key)` method. `SystemSetupContext.getParameters(key)` method returns a `String[]` with all selected options.

```
@SystemSetup(type = Type.PROJECT, process = Process.ALL)
public void createProjectData(final SystemSetupContext context) throws Exception
{
        LOG.info("-----> createCustomData : " + context.getParameter(CoreConstants.EXTENSIONNAME +
        "_createCustomData"));
        LOG.info("-----> imports :");
        for (final String imp : context.getParameters(CoreConstants.EXTENSIONNAME +
        "_core_imports"))
        {
                LOG.info("------------------> " + imp);
        }
}
```

Method Calling Order

The call order is controlled by your `classloader` and also depends on when the according bean is defined. Beans are normally created in the extension build order.

Manual Call for a Special Extension

For testing purposes, you might want to execute all those beans for your extension. This can be done by the `SystemSetupCollector`:

```
public class MyTest
{
  private ApplicationContext applicationContext;

  @Before
  public void setUp()
```

```
    {
     final SystemSetupContext systemSetupContext = new SystemSetupContext(null, Type.ESSENTIAL, "MyExtension");
     applicationContext.getBean(SystemSetupCollector.class).executeMethods(systemSetupContext);
    }

    public void setApplicationContext(ApplicationContext applicationContext)
    {
      this.applicationContext = applicationContext;
    }
  }
```

Patching Framework

You can annotate certain Spring beans with `@SystemSetup`. Logic annotated in this way is then executed during system initialization or update. The `@SystemSetup` annotation works with the Patching Framework, which prevents this annotated logic from being executed more than once. We refer to such logic as a patch due to this once-only application.

To achieve the same without the Patching Framework, you have to provide some extra logic that checks whether data in a database exists or not. Based on that, your annotated logic gets executed or not. This approach provides an extra level of unnecessary complication. The provided scenario proves that using the Patching Framework is much more efficient.

Example Scenario

Imagine an extension that comes with the following item definition:

```
<itemtype code="XmlDefinition"
    jaloclass="de.hybris.example.XmlDefinition"
    autocreate="true"
    generate="true">
    <deployment table="ExampleItems" typecode="3456" />
    <attributes>
        <attribute qualifier="code" type="java.lang.String">
            <persistence type="property" />
            <modifiers optional="false" initial="true" write="false" unique="true"/>
        </attribute>
        <attribute qualifier="content" type="java.lang.String">
            <persistence type="property" />
            <modifiers optional="false" write="true" read="true" />
        </attribute>
    </attributes>
</itemtype>
```

This item keeps a content of an xml.

In the next version of the extension another property is added. It is of type `boolean`, and named `valid`:

```
<attribute qualifier="valid" type="boolean">
    <persistence type="property" />
    <modifiers optional="true" write="true" read="true" />
</attribute>
```

Additionally, a new Interceptor is added that validates `content` of the `XmlDefinition`item, and puts the validation result in the newly introduced property.

The problem is you now have multiple old instances of the `XmlDefinition` item that haven't been validated. The usual way to fix this problem is to create a new class that validates all old items during the system update process. Here is an implementation of such a class:

```
@SystemSetup(extension = "sampleExtension", process = Process.UPDATE)
public class XmlDefinitionUpdater
{
   private XmlDefinitionValidationService validationService;
   private FlexibleSearchService flexibleSearchService;
   private ModelService modelService;

   @SystemSetup
   public void updateValidation()
   {
```

```
        findAllXmlDefinitions().forEach(d -> {
            d.setValid(validationService.isValid(d));
            modelService.save(d);
        });
    }

    private List<XmlDefinitionModel> findAllXmlDefinitions()
    {
       final FlexibleSearchQuery fQuery = new FlexibleSearchQuery("SELECT {PK} FROM {XmlDefinition} WHERE {valid} IS NULL")
       return flexibleSearchService.<XmlDefinitionModel> search(fQuery).getResult();
    }

    @Required
    public void setValidationService(final XmlDefinitionValidationService validationService)
    {
        this.validationService = validationService;
    }

    @Required
    public void setModelService(final ModelService modelService)
    {
       this.modelService = modelService;
    }

    @Required
    public void setFlexibleSearchService(final FlexibleSearchService flexibleSearchService)
    {
       this.flexibleSearchService = flexibleSearchService;
    }
}
```

You have to register this class as a Spring bean:

```
<bean id="xmlDefinitionUpdater" class="de.hybris.example.XmlDefinitionUpdater">
    <property name="validationService" ref="xmlDefinitionValidationService"/>
    <property name="modelService" ref="modelService"/>
    <property name="flexibleSearchService" ref="flexibleSearchService"/>
</bean>
```

The whole setup is enough to update all the old item instances during the system update process. Platform will search for all the instances with the `valid` property that have value `null`, ask `validationService` whether a given instance has a valid XML content, and finally save items using `modelService`.

The whole logic will be executed each time you execute the system update action - each time a query to the database will be executed.

You can, however, easily refactor the logic code to enable the Patching Framework and, as a result, prevent the system logic from executing your logic during another system updates. To enable the Patching Framework, use appropriate `@SystemSetup` annotation flags:

```
@SystemSetup(extension = "sampleExtension", process = Process.UPDATE)
public class XmlDefinitionUpdater
{
// .......
   @SystemSetup(patch = true, required = true)
   public void updateValidation()
   {
     // ......
   }

// .....
}
```

In the example we used two flags in the `@SystemSetup` annotation applied on a method level - `patch=true` and `required=true`.

For more information on patch-related flags, see Annotating Classes and Methods.

Annotating Classes and Methods

You can apply the `@SystemSetup` annotation on a class level and a method level. `@SystemSetup` provides a few flags that are exclusively designed to be used with the Patching Framework.

A class level annotation is required for the special `SystemSetupCollector` class to find all Spring Beans annotated as `@SystemSetup`. A method level annotation is required to determine which methods must be executed during the initialization or the update phase. When you look at `@SystemSetup` as a patch, each method of an annotated class becomes a separate patch.

The `@SystemSetup` annotation flags designed for the Patching Framework include:

- `patch`: marks a logic as one to be executed only once
- `required`: determines whether a logic is to be executed or not
- `name`: allows you to provide a name of a logic to be executed
- `description`: allows you to provide a longer logic description

A properly annotated logic is executed once. You can choose whether you want to execute it during system update or initialization - it depends on the option you provide for the `@SystemSetup` `process` flag (UPDATE, INIT, or ALL). For more information on `@SystemSetup`, see Hooks for Initialization and Update Process.

Annotation Examples

Using `@SystemSetup` flags for the Patching Framework gives you many possibilities. We provide you with examples showing different use cases.

Example 1

Use a class level annotation to mark an entire class as a patch. Each method in that class is treated as a patch:

```
@SystemSetup(extension = "sampleExtension", patch = true)
public class Example
{
   @SystemSetup(name = "patch1")
   public void foo()
   {
   }

   @SystemSetup(name = "patch2")
   public void bar()
   {
   }
}
```

Example 2

You can combine patches with regular `@SystemSetup` methods. In this example only the `foo` method is a patch. The `bar` method is a regular `@SystemSetup` method and gets executed during each initialization or update:

```
@SystemSetup(extension = "sampleExtension")
public class Example
{
   @SystemSetup(name = "patch1", patch = true)
   public void foo()
   {
   }


   @SystemSetup(name = "patch2")
   public void bar()
   {
   }
}
```

Note

Don't use `patch=true` on a class level for classes that have both patch methods and regular `@SystemSetup` methods.

Example 3

In this example you apply a patch only during the update phase:

```
@SystemSetup(extension = "sampleExtension")
public class Example
{
   @SystemSetup(name = "patch1", patch = true, process = Process.UPDATE)
   public void foo()
   {
   }
}
```

Example 4

In this example you apply a patch only during the essential data phase, or the project data phase if you adjust the code accordingly:

```
@SystemSetup(extension = "sampleExtension")
public class Example
{
   @SystemSetup(name = "patch1", patch = true, type = Type.ESSENTIAL)
   public void foo()
   {
   }
}
```

Note

Due to the nature of a patch, `type=Type.ALL` doesn't work as intended. You apply patches only once. It means the first execution in the **ESSENTIAL** data phase already applies a patch. That patch doesn't get applied again during the **PROJECT** data phase.

Example 5

If your class has only one method and acts as a patch, you may choose a class level annotation instead of a method level annotation:

```
@SystemSetup(extension = "sampleExtension", patch = true, name = "my Patch", description = "Patches Example item", process
public class Example
{
   @SystemSetup
   public void foo()
   {
   }
}
```

Patch Naming Convention

Understanding the naming convention is key to figuring out the order patches are applied in.

To enable the system to sort patches in an alphabetical order, use the following property with the value `true`:

```
system.setup.sort.legacy.mode=true
```

Patches are sorted by name within an extension and applied in that order. To give a patch a name, use the `name` flag. If you don't provide a name, the system automatically applies one using both the Spring Bean `ID` of a class and the name of a method.

In this example the `foo` method has `foo` as a value for the `name` parameter. The `bar` method doesn't have an explicitly provided name:

```
@SystemSetup(extension = "sampleExtension")
public class XmlDefinitionUpdater
{
    @SystemSetup(name = "foo", patch = true)
    public void foo()
    {
    //...
    }

    @SystemSetup(patch = true)
    public void bar()
    {
    //...
    }
}
```

In this case the name for the `bar` method is auto-generated. Assuming that a Spring Bean ID of the class is `xmlDefinitionUpdater`, the name of the patch will be `xmlDefinitionUpdater#bar`.

Tracking Patches

You can track data related to the patches you apply.

Data is tracked in a database thanks to the `SystemSetupAudit` item. You can track the following data:

- `extensionName`: a name of an extension that defines a `@SystemSetup` annotated logic
- `required`: information whether logic is required
- `user`: a user that executed logic
- `name`: a name of a patch; if not provided in a `@SystemSetup` annotation, then `Spring Bean ID plus name` of a method is used
- `description`: a description of a patch
- `className`: a class name of a patch
- `methodName`: a method name of a patch

Managing Patches in SAP Commerce Administration Console

SAP Commerce Administration Console enables you to select patches you want to apply during initialization or update.

At the bottom of the **Initialize** and **Update** pages Administration Console displays patches pending execution. You can select or deselect them, and decide in this way which patches will be applied and which won't. You cannot deselect patches marked as `required=true`.

Event System

The ServiceLayer provides a framework to send and receive events within SAP Commerce. Events can either be published locally or across cluster nodes.

A common way to communicate between software components is through events. A source publishes an event while any interested party registers a listener that is able to receive these events and then perform logic. SAP Commerce Service Layer uses this concept: Events are published using an event service. Listeners are registered with the event service and can then react to the events.

The SAP Commerce event system is based on the Spring event system, but we have added some intelligence to make it easier to handle SAP Commerce-related logic.

Examples of events:

- Java Swing button publishes an event when clicked.
- Item publishes the fact that one of its attributes changes, for example that a description or a thumbnail of a media item has been modified.
- SAP Commerce publishes the fact that a new session has been created or an order has been submitted.

Events can either be published locally or across cluster nodes.

Event Service

`DefaultEventService` allows you to register event listeners and publish events.

There is a `DefaultEventService` that implements the `EventService` interface (`de.hybris.platform.servicelayer.event` package). To use this service, add a Spring resource to your class.

`MyFantasticClass.java`

```
@Resource
private EventService eventService;
```

Use this service to:

- Register event listeners via the `registerEventListener(...)` method.

  `EventServiceTest.java`

  ```
  public class EventServiceTest extends ServicelayerTest
  {
    @Resource
    private EventService eventService;

    @Test
    public void testTxEvents()
    {
      try
      {
        final TestListener listener = new TestListener();
        eventService.registerEventListener(listener);
        ...
      }
      ...
    }
  }
  ```

- Publish events via the `publishEvent(...)` method.

  `MyService.java`

  ```
  class MyService
  {
    @Autowired
    private EventService eventService;

    protected void something()
    {
      if( something )
      {
        eventService.publishEvent( new MyEvent("i am so happy") );
      }
    }
  }
  ```

Related Information

API Documentation and YAML Files

JavaDocs

Event Listeners

Event listeners are objects that are notified of events and perform business logic depending on the kind of event.

To implement an event listener, extend from the abstract class `AbstractEventListener` (`de.hybris.platform.servicelayer.event.impl.` package), in particular implement the method `onEvent(...)`.

In this method, check for the type of event you are listening for. When you have made sure that you have received the proper type of event, run your logic.

Example:

`MyEventListener.java`

```
public class MyEventListener extends AbstractEventListener
{
   @Override
   protected void onEvent(final AbstractEvent event)
   {
      if (event instanceof MyEvent)
      {
          // Your logic here
      }
      if (event instanceof MyOtherEvent)
      {
          // Your logic here
      }
   }
}
```

→ Tip

Java Generics Allowed if Listening for One Type of Events

Using Java generics, you are able to pass the event class you want to listen for. With that, you do not have to do an `instanceof` check. The listener is only called for a specified event type.

However, note that this only works if you listen for one single type of event. To listen for two or more types of events, you still need to use an `instanceof` -based implementation.

For example, to listen for events of type `AfterItemCreationEvent`:

`MyEventListener.java`

```
public class MyEventListener extends AbstractEventListener<AfterItemCreationEvent>
{
   @Override
   protected void onEvent(final AfterItemCreationEvent event)
   {
         // Your logic here
   }
}
```

Note that, by default, event publishing is synchronous on the same SAP Commerce Multichannel Suite instance. That means that the current thread is locked until all listeners have reacted to an event. Therefore, keep an eye on the performance of your listener and make sure that your code scales well.

However, on a SAP Commerce Cluster, the event publishing runs asynchronously on other cluster nodes. Events on the same cluster node are still processed synchronously (unless it is a ClusterAware event, in which case the event is always processed asynchronously, see the section **Cluster-Aware Events**), but remote cluster nodes receive and process events in an asynchronous way. If you need to make sure that a certain node reacts to receiving a certain event, you will have to implement some sort of acknowledgment mechanism, for example by publishing another kind of event.

ⓘ Note

If you want your event to be processed asynchronously, implement ClusterAwareEvent interface.

Related Information

API Documentation and YAML Files
JavaDocs

Registering Event Listeners

You can register event listeners by extending the `AbstractEventListener` class or through the `eventService`.

There are two ways to register an event listener:

- The typical way is to write your event listener class by extending the `AbstractEventListener` class, and register the event listener class as a Spring bean. The event system scans the application context and automatically registers all event listeners.

  `myextension-spring.xml`

  ```
  <bean id="myListener" class="my.package.MyListener" parent="abstractEventListener">
  ```

  - The `MyEventListener` listener class implementation:

MyEventListener.java

```
public class MyEventListener .....
{
 private ModelService modelService;

 /**
  * @param modelService the modelService to set
  */
 public void setModelService(ModelService modelService)
 {
  this.modelService = modelService;
 }
```

- Corresponding configuration in Spring context:

  myextensions-spring.xml

```
<bean id="myListener" class="my.package.MyEventListener" >
        <property name="modelService" bean="modelService" />
    </bean>
```

- Alternatively you can register the listener via the `eventService`:

  MyFantasticClass.java

```
@Resource
EventService eventService;
...
eventService.registerEventListener(new MyListener());
```

  You want to do it this way when you have to dynamically add listeners and can't statically put them in the application context file, for example, when you have to configure the listeners at runtime.

Using Scripts as Event Listeners

Using the traditional SAP Commerce event system, the user is forced to rebuild the system and restart the server, because it is necessary to extend the AbstractEventListener class or, in some cases, to set up a spring bean.

Using dynamic scripting it is all done at runtime and is much easier. To find out how you can use scripts as event listeners, see Scripts as Event Listeners.

Publishing Events

Learn how to publish events.

An event is an instance of a subclass of `AbstractEvent` (`de.hybris.platform.servicelayer.event.events` package). It contains a `source` object that denotes the origin of the event.

In its most basic form, an event class would look like this:

MyEvent.java

```
public class MyEvent extends AbstractEvent implements ClusterAwareEvent
{
  Object source;

  public MyEvent(Serializable source)
  {
    super(source);
    this.source=source;
  }
}
```

This event only holds a reference to an object.

To publish an event:

MyFantasticClass.java

```
source = //any java object which might be interesting
MyEvent event = new MyEvent(source);
eventService.publishEvent(event);
```

With an event listener, you can now react on this event:

`MyEventListener.java`

```
public class MyEventListener extends AbstractEventListener<MyEvent>
{
   @Override
   protected void onEvent(final MyEvent event)
   {
     System.out.println("Got a MyEvent with the object "+event.getSource());
   }
}
```

Related Information

API Documentation and YAML Files

JavaDocs

Cluster-Aware Events

SAP Commerce supports cluster-aware events. With cluster-aware events, SAP Commerce can process events in separate threads, or on particular nodes of a cluster.

To send custom events from one node to other specific nodes or broadcast them across all nodes of a cluster, implement the `canPublish(final PublishEventContext publishEventContext)` method from the `ClusterAwareEvent` interface in your events class in a way to return `true` if you want to publish the event from the cluster node with the `publishEventContext.getSourceNodeId()` ID to the node with the `publishEventContext.getTargetNodeId()` ID or to the target `publishEventContext.getTargetNodeGroups()` node groups. Return the boolean value based on whether you wish to publish events from cluster node source node to cluster target node or a node group.

To ensure that a specific node processes an event, for example, a node that hosts an index service that needs to be informed of data changes, return `true` if the target node ID is equal to the ID of the node that hosts that index server or belongs to the target node group.

For more information, see Clustered Environment.

`MyEvent.java`

```
public class MyEvent extends AbstractEvent implements ClusterAwareEvent
{
  @Override
  public boolean canPublish(final PublishEventContext publishEventContext)
  {
    //decide which cluster node should process that event
    return publishEventContext.getSourceNodeId() != publishEventContext.getTargetNodeId();  //process on a different clust
    //return publishEventContext.getSourceNodeId() < 5 && publishEventContext.getTargetNodeId() >= 5; //process on any eve
    //return true; //allow processing on all cluster nodes
  }
}
```

Related Information

API Documentation and YAML Files

JavaDocs

Transaction Aware Events

You can publish events at the end of a transaction to get notified of model changes instead of attribute changes.

Sometimes you want an event to be published only at the end of a transaction. You can then implement the `TransactionAwareEvent` interface (`de.hybris.platform.servicelayer.event` package). Have the `getId` method return a unique ID. All events returning the same ID are only be published once at the end of the transaction if `publishOnCommitOnly` returns `true`.

This is useful when you want to publish events that notify of model changes. Without a transaction aware the event would be published on each attribute change. But you probably only want it to be published once. You then implement a `getId` method that returns the item PK of the model.

Synchronous vs. Asynchronous Events

`ClusterAwareEvents` is processed at receiving platform (as well as own platform) in an asynchronous way, no matter of used protocol. All other events not extending `ClusterAwareEvents` are published only locally where this is done synchronous. You can configure the publishing as asynchronous by setting n `java.util.concurrent.Executor` at used SAP Commerce `PlatformClusterEventSender` instance. This can be done easily by overriding the spring definition of the sender at a spring configuration file participating the global `ApplicationContext`, typically `myExtension/resources/[myExtension]-spring.xml`:

```
<bean id="platformClusterEventSender" class="de.hybris.platform.servicelayer.event.impl.PlatformClusterEventSender">
        <property name="serializationService" ref="serializationService"/>
  <property name="tenant" ref="tenantFactory"/>
        <property name="executor">
                <bean class="java.util.concurrent.Executors" factory-method="newCachedThreadPool"/>
        </property>
</bean>
```

ℹ Note

Publishing events asynchronously with `java.util.concurrent.Executors` occurs only after a tenant startup.

Related Information

Building SAP Commerce
ServiceLayer
Scripts as Event Listeners

Spring Framework in SAP Commerce

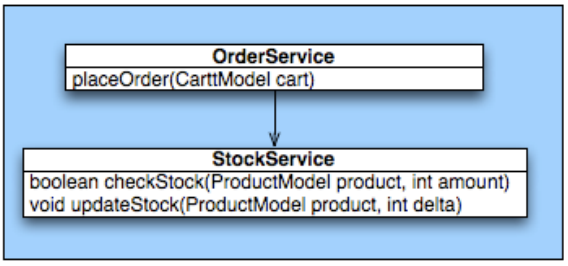
SAP Commerce integrates the Spring Framework, offering developers a familiar and flexible programming model. The Spring Framework is the foundation of the SAP Commerce Service Layer.

The Spring Framework insulates business objects from the complexities of platform services for application component management, web services, transactions, security, remoting, messaging, data access, aspect-oriented programming, and more.

Dependency Injection

Dependency injection is a software architecture pattern in which a component's dependencies are not managed by the component itself but are configured externally.

We use an `OrderService` as an example. During the ordering process, you want to check whether all of the items of the order are still in stock. Therefore you need a `StockService`; you would have the following:



To enable the `OrderService` to get the reference to the `StockService`, you could do use the following code:

```
StockService stockService = new StockService();
```

If you require a different implementation of a StockService, you could extend the original implementation. However such a modification would require that you go through the code and replace the old class names with the new ones. A simpler option would be to implement it as follows:

```
StockService stockService = StockService.getInstance();
```

This solution also has its limitations: in order to change the object that is returned by `getInstance()`, you would have to change the object's implementation, which requires access to the source code. It would also require that you write a `getInstance()` method for each class.

It would be better if the `OrderService` did not have to obtain the `StockService` reference and if you could maintain its dependencies by means of a configuration file. That is what dependency injection does. The `OrderService` simply needs to define a setter method:

```
public void setStockService(StockService stockService) {
        this.stockService = stockService;
}
```

In an XML file, you then can define the `OrderService` as follows:

```
<bean class="de.hybris.platform.order.OrderService">
        <property name="stockService">
                <bean class="de.hybris.platform.stock.StockService"/>
        </property>
</bean>
```

Here you define the `OrderService` and have the `StockService` injected (hence the name of the pattern). A so-called container (application context) reads the configuration file, resolves the dependencies, and puts together the objects. When the objects are ready, all the dependencies are already injected.

The XML fragment above is a snippet from a configuration file for the Spring Framework . One of the main features of Spring is to maintain and inject dependencies.

Interface-Driven Design

Using interfaces makes it far easier to later use a different implementation.

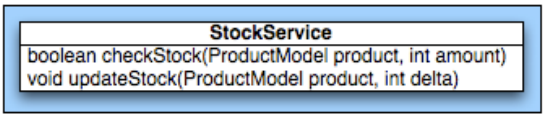
Let us once more have a look at the `OrderService`. It depends on the `StockService` and therefore defines an instance variable like this:

```
private StockService stockService;
```

You have written a setter for the Dependency Injection mechanism to set this instance variable. Now, `StockService` is a class and, because Java is a strongly typed language, it means that the object to be injected must be of type `StockService` or a subclass thereof.

Let's assume you want to replace the `StockService`. You might have to do this because you have a new fulfillment partner, or you want to mock the service to come up with test scenarios that the original `StockService` does not cover.

To do this, extend from `StockService` and implement the necessary methods:

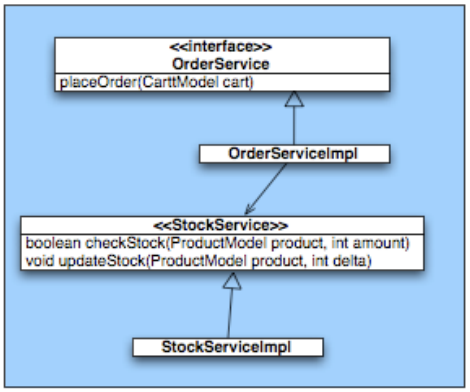


But what if you do not reuse anything from the original class? Your new subclass would unnecessarily carry around everything from its superclass. Or suppose that the superclass has dependencies that the subclass does not need? For example, the original `StockService` might need some kind of logic to perform web services while a mocking test `StockService` would not need such a web service.

Technically, this is not a problem but it makes the new class far less intuitive and harder to understand, because it may contain inherited concepts that are never needed.

You could overcome this by defining an abstract superclass with only the two methods, but then why not use Java interfaces right away? That is what they are made for in the first place. Using interfaces, you define the public methods for the outside world and completely hide implementation details.

So now it looks like this:



As you can see, the dependency between `OrderService` and `StockService` is defined not as a (concrete) class, but as the interface. By consequence, every class that implements the interface `StockService` may be used as a StockService for the `OrderService`.

Beans

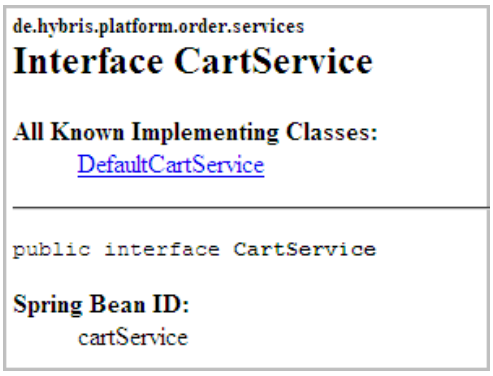
A bean is an object that is instantiated, assembled, and managed by a Spring container. Beans are created with configuration metadata that you supply to the container.

Lets look at the `OrderService` bean once again:

```
<bean class="de.hybris.platform.order.OrderService">
        <property name="stockService">
                <bean class="de.hybris.platform.stock.StockService"/>
        </property>
</bean>
```

These XML configuration files can be found in the `${HYBRIS_BIN_DIR}/platform/ext/core/resources` directory. There you find files with the name pattern `|component|-spring.xml`, where **<component>** is something like product, order, i18n, security, and so on. For the beans themselves, a name pattern such as `xyzService` is recommended, such as: `productService`, `catalogService`, and so on.

Service Layer exposes its Service API through interfaces. The implementations of these interfaces are available as Spring beans. You find the name of these beans in the API Doc of the respective interface. For example, in the screenshot below, the ID of the Spring bean is `cartService`:


de.hybris.platform.order.services

Interface CartService

All Known Implementing Classes:

DefaultCartService

`public interface CartService`

Spring Bean ID:

cartService

Figure: Screenshot from SAP Commerce API Doc, showing the Spring Bean ID name of the CartService interface: cartService.

For more information, see Adding a New Service.

Aliases and Overriding Beans

```
<alias alias="cartService" name="defaultCartService"/>
<bean id="defaultCartService"
        class="de.hybris.platform.order.services.impl.CartServiceImpl"
        parent="abstractBusinessService" >
        ....
```

In addition to the actual bean definition with an ID prefixed with **default**, there is an alias for the bean. The idea is that you should use the alias to override instead of using the actual bean ID in a hard-wired way. This is to ensure extensibility. You can override the alias ID and still have access to the original bean.

For more information, see the *Overriding the Bean Definition* section of Adding Logic to an Existing Service.

Bean Scopes

Apart from regular bean scopes, SAP Commerce features two special scopes: **yrequest** and the deprecated **tenant** scope.

yrequest Scope

The `global-core-spring.xml` file of the `core` extension adds a special scope named **yrequest** to the core `ApplicationContext`. The **yrequest** scope may be used for binding beans to the application context for a single request. Upon session deactivation, the beans are no longer referenced from the application context, allowing them to be garbage collected. This is similar to a standard Spring **request** scope but is used outside of a web application context.

As an example use case, assume that you have a third-party invoice processing server that requires pulling order data from a SAP Commerce system at periodic intervals. Instead of using web services or some other means of data transfer, you decide to use the SAP Commerce API directly from within the invoice processing server. You use the **yrequest** scope in this case, so that all beans bound for a request from the invoice server are released when the request completes.

An example of how to set a bean's scope to **yrequest**:

```
<bean id="myExtension.myBean" class="de.hybris.platform.myExtension.myClass" scope="yrequest" />
```

Spring Profiles

Profiles allow users to separate bean definitions for different environments.

Beans that use some external resources may be mocked for the testing environment. You can read more about Spring profiles in the official documentation .

SAP Commerce Implementation of Spring Profiles

SAP Commerce provides a convenient way of configuring active Spring profiles with the property `spring.profiles.active`. You can declare profiles as a list of comma-separated names. All the profiles are added as active to the `GlobalApplicationContext`. As a result, all beans declared in these profiles are loaded into the application context.

Additionally, SAP Commerce provides predefined profiles for each tenant-aware `ApplicationContext` named for the tenant ID and prefixed by `tenant_`. An example profile for the Junit tenant is **tenant_junit**. These profiles are active by default, no configuration is required, so if you know the naming convention, you can declare any bean in this profile and it is loaded automatically.

ⓘ Note

Keep in mind that tenant aware profiles are bound to the `ApplicationContext`, so any bean that is declared in such a profile needs to be configured in the proper xml file corresponding to the `ApplicationContext` and not **GlobalApplicationContext**.

SAP Commerce Recommendations

To avoid clashing with profiles declared by other developers, use unique names for profiles. The best way is to add an extension name prefix to the profile, for instance **b2b_develop**.

Introduction to Application Contexts

Beans are managed by containers, the so-called application contexts. The Core Application Context is constructed using all configuration files of the extensions enabled for a given tenant.

This is how application context mechanism works in SAP Commerce:

- SAP Commerce contains a mechanism that builds one global context and a number of application contexts. These contexts correspond to the number of tenants. This means that each tenant has its own application context.
- All application contexts have as the parent the global application context.
- Every web application has a separate web spring context, which has as the parent its tenant-specific application context.

This is the overview of application contexts in SAP Commerce.

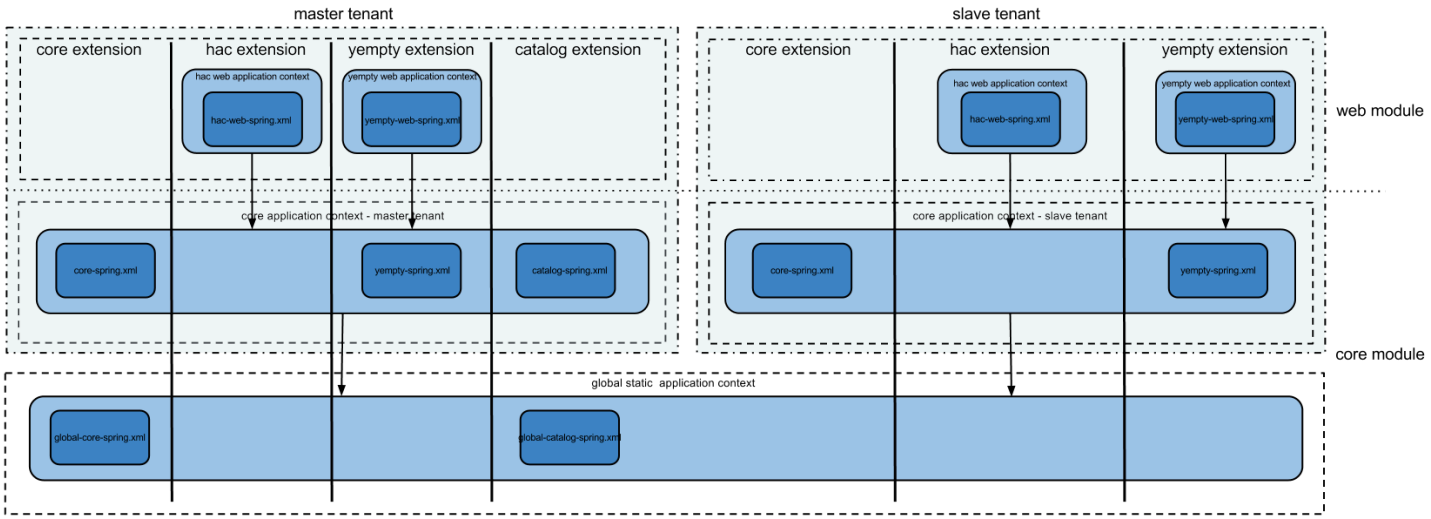


Figure: Hierarchy of Application Contexts.

Core ApplicationContext

1. The Core Application Context is constructed using all configuration files of the extensions enabled for a given tenant (each tenant can have a different set of extensions).

   The construction process uses the classloader of the `core` extension. It has, as its parent, the (singleton) global application context.

2. The Core Application Context is Loaded. As the bean's configuration is assembled by SAP Commerce upon start-up, the configuration is always as up-to-date as the configuration files. You do not have to run a SAP Commerce build, starting SAP Commerce is sufficient.

You can access the beans of other extensions at your configuration file or override other extension's beans' definitions.

Also, at `core` extension, you can override the beans of the global configuration file.

Adding Bean Definitions to Core ApplicationContext

By default, if a file that matches the convention extname-application-context.xml exists, its configuration is loaded. If it does not exist:

1. Make sure that a `|extname|-spring.xml` file exists in the `resources` directory of your extension, such as `myextension-spring.xml`.

   If your extension's `resources` directory does not contain a `spring.xml` file, refer to the **ysimple** or `yempty` extension templates for a sample file. For details on the `yempty` extension templates, refer to Creating a New Extension.

2. Configure the `project.properties` file in your extension:

   ```
   <extname>.application-context = <extname>-spring.xml
   ```

As already mentioned, you can access beans of other extensions or override them. Bear in mind that if you use the same bean id in two extensions, one bean overrides the other one. The files are loaded in build order of the installed extensions, so the last definition of the same bean id is used. Therefore, make sure that you use unique names, for example by using your extension name as prefix for each bean.

The order of loading the configuration is the same as in the build process, so if you have to ensure that the configuration of your extension will be loaded later than that of another extension, add a `<requires-extension>` entry to the `extensioninfo.xml` file of your extension.

You can check whether your configuration is used by checking the console log for:

```
INFO [] (master) [Registry] Loading Spring configuration for '<extname>' from classpath (/<extname>-spring.xml)
```

ℹ Note

If you want to use more than one configuration file, you can configure the list of configuration files in the extension's `project.properties` file using a comma-separated list of files (where it first tries to load them from classpath, then from file system). You might want to use different configuration files for a better structure and reusage of code, for example:

```
<extname>.application-context = <extname>-spring.xml,additional-classpath-file.xml,C:\\additional-filesystem-file.xml
```

Accessing Core ApplicationContext

- `getApplicationContext()`

  SAP recommends that you use this method because it is context-aware. It first tries to get a web application context set at the current `ServletContext`. Only if no web application context is set, the method returns the global `ApplicationContext`.

  ```
  MyBeanType myBean = (MyBeanType)Registry.getApplicationContext().getBean("<extname>.mybean");
  ```

  #### ℹ Note

  It is worth noting that this method may return, in web application environment, the spring web application context attached to a different tenant than the current one. This produces warnings.

- `getCoreApplicationContext()`.

  This method returns the core `ApplicationContext`. Use it only in case you explicitly need the core `ApplicationContext`.

- getGlobalApplicationContext().

  This method is deprecated. Use `getCoreApplicationContext()` instead.

For details about how to access the `ApplicationContext` within the Web Module of an extension, please refer to the Accessing the Web ApplicationContext section below.

TenantListener

Don't mess with the SAP Commerce beans during `ApplicationContext` events but use `TenantListener` instead.

You can use `afterPropertiesSet` or other `ApplicationContext` event to register a new `TenantListener`. Here is an example:

```
@Override
                public void afterPropertiesSet() throws Exception
                {
                                Registry.registerTenantListener(new TenantListener()
                                {
                                                @Override
                                                public void afterTenantStartUp(Tenant tenant)
                                                {
                                                                //do your logic here
                                                }
                                });
                }
```

Web Application Context

To define the beans part of the Web Module of an extension or with scopes not available to the global `ApplicationContext` or core application contexts, like **session** or **request**, you need to configure a web `ApplicationContext` in the `web.xml` file. You also need access to the core `ApplicationContext` to access beans of the Core Module, such as services.

For this access, SAP Commerce provides a `HybrisContextLoaderListener` for setting the core `ApplicationContext` automatically as parent of your web `ApplicationContext`. With that, you can use the core bean definitions, for example to inject them into your web `ApplicationContext` beans. If you make a `getBean` call to your web `ApplicationContext`, it is checked whether there is a definition available. If not, the parent `ApplicationContext` is used.

This hierarchical concept ensures that a web `ApplicationContext` can use core beans, but cannot modify them. If you try to redefine a core bean at your web `ApplicationContext`, no real overriding is done. Instead a second definition is created at the web `ApplicationContext`, which overlays the core bean definition. In the following you can see how to configure such an `ApplicationContext`, and how to connect it to the core `ApplicationContext` to use the beans defined there. The same applies to redefining the global singleton beans.

Configuration of Web Application Context

The configuration of a web `ApplicationContext` is done in the `web.xml` file of your extension. The simplest way to get an `ApplicationContext` in the Web Module is to define a `ContextLoaderListener` like:

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/<extname>-web-spring.xml</param-value>
</context-param>

<listener>
   <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

The `WebApplicationContext` is created from configuration files specified by `contextConfigLocation` parameter and saved at the `ServletContext`.

You can see that your web `ApplicationContext` is loaded when checking the console log for:

```
INFO [] (master) [[/<extname>]] Initializing Spring root WebApplicationContext
```

If you want to enable the usage of the web application-specific scopes **session** and **request**, you additionally have to add the Spring `RequestContextListener`, such as:

```
...
   <listener>
      <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
   </listener>
...
```

It is recommended you connect the web `ApplicationContext` with the core `ApplicationContext` to get access to core beans by simply accessing the web `ApplicationContext`. With that you can inject beans defined at the core `ApplicationContext` to your beans defined at your web `ApplicationContext`. To do so, replace the `ContextLoaderListener` with the `HybrisContextLoaderListener`. The recommended configuration in your `web.xml` file should look like:

```
<!--
   Enabling a Spring web application context with 'session' and 'request' scope.
   - The 'contextConfigLocation' param specifies where your configuration files are located.
```

```
    - The HybrisContextLoaderListener extends the usual ContextLoaderListener (which loads
      the context from specified location) by adding the core application context of
      the platform as parent context. With having the global context set as parent you can access
      or override beans of the global context.
    - The RequestContextListener is needed for exposing the 'request' scope to the context.
  -->
 <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>WEB-INF/<extname>-web-spring.xml</param-value>
 </context-param>

 <listener>
    <listener-class>de.hybris.platform.spring.HybrisContextLoaderListener</listener-class>
 </listener>

 <listener>
    <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
 </listener>
```

Accessing Web Application Context

While the `Registry.getCoreApplicationContext` method always returns the core `ApplicationContext`, the `Registry.getApplicationContext` checks first if there is a `ServletContext` currently holding a **WebApplicationContext**. If that is the case, this one is returned, which is indeed your web `ApplicationContext` configured at your `web.xml` file. If there is no current `ServletContext` or no `ApplicationContext` set at it, the global `ApplicationContext` is returned. With that you do not have to be aware if your context is a Core Module or a Web Module, you always get the correct `ApplicationContext`.

> **Note**
>
> Be aware that getApplicationContext may return - in the web application environment - the spring web application context attached to a different tenant than the current one. This produces warnings.

Be aware that if you do not connect your web `ApplicationContext` to the global or core `ApplicationContext`, you cannot access global or core beans via the `ApplicationContext` returned by `Registry.getApplicationContext`. If you have built up the connection by simply using the `HybrisContextLoaderListener`, you have access to all the beans of your web and the global and core `ApplicationContext`, unless you have defined overlaying bean definitions in your web `ApplicationContext`.

```
MyBeanType myBean = (MyBeanType)Registry.getApplicationContext().getBean("<extname>.mybean");
```

Using Spring MVC

For getting in touch with Spring MVC, the `spring-webmvc` library has to be located in the `WEB-INF/lib` directory of your extension.

The core of the Spring MVC framework is the `DispatcherServlet`. It has to be configured in your `web.xml` file to activate the dispatching of incoming requests to the special MVC controller. For each servlet configuration using a derivate of the `DispatcherServlet`, an own application context is created, whereas the root web application context is set as parent. So, if you have configured your root web application context using the `HybrisContextLoaderListener`, you are also able to access beans of the global or core `ApplicationContext` from within a dispatcher `ApplicationContext`.

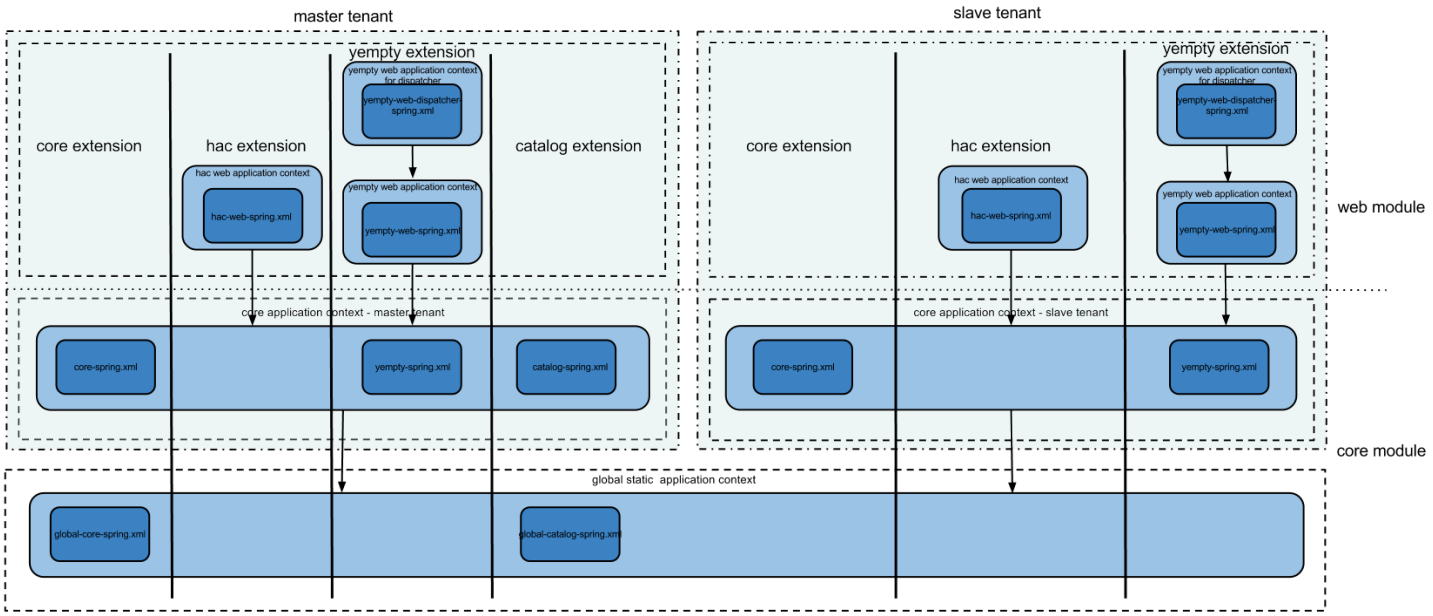


Figure: The Spring MVC `DispatcherServlet` in SAP Commerce and how it can access the web `ApplicationContext`, and, by consequence, the core, and global `ApplicationContext`.

Please be aware that the configuration of a `DispatcherServlet` implicates a new application context different to the root web application context. So the specification of a `contextConfigLocation` via the `<context-param>` tag is used by the root web application context, not for the `ApplicationContext` created by the `DispatcherServlet`. If you want to specify a configuration file for the `DispatcherContext`, you have to use the `<init-param>` tag inside the `<servlet>` tag.

You can configure your root web `ApplicationContext` and add a servlet entry to your `web.xml` file for each `DispatcherServlet`. For more information, see Spring Framework in SAP Commerce.

`web.xml`

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>classpath:/springtest-web-context.xml</param-value>
</context-param>

<listener>
   <listener-class>
      de.hybris.platform.spring.HybrisContextLoaderListener
   </listener-class>
</listener>

<listener>
   <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
</listener>

<servlet>
   <servlet-name>springtest</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>

   <load-on-startup>1</load-on-startup>

   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:/springtest-web-dispatcher-context.xml</param-value>
   </init-param>
</servlet>

<servlet-mapping>
   <servlet-name>springtest</servlet-name>
   <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

Spring Integration

To communicate with external systems, SAP Commerce needs to support the required data transport protocol and to convert to and from the required data format.

SAP Commerce needs to communicate with a number of external systems, for example:

- Warehouses: To place orders, receive updates of order status, stock updates, and order returns
- Product databases: To receive product data
- CRM software: To send information about customer transactions like order data.

To communicate with these systems, SAP Commerce needs to support the required data transport protocol (SOAP, CSV files, SQL, and so on), and to convert to and from the required data format (XML, text, and so on). Communication can be asynchronous when, for instance, SAP Commerce may submit an order to a warehouse, and the warehouse may notify the SAP Commerce System about the order dispatch a day later.

Spring Integration extends the Spring framework to support Enterprise Integration Patterns. It provides a simple way of communication within Spring-based applications and supports integration with external systems by using various adapters. By incorporating the Enterprise Integration Patterns in SAP Commerce, SAP Commerce provides Spring Integration libraries and a few helper classes that make it easier to implement common use cases.

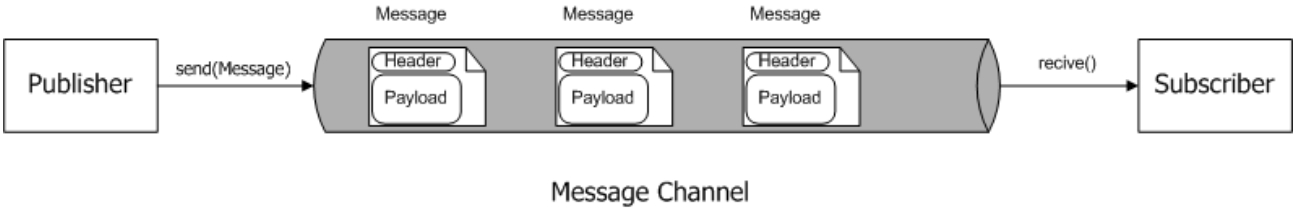
For more information, see:

- http://www.springsource.org/spring-integration : Spring Integration
- http://www.eaipatterns.com : Enterprise Integration Patterns
- http://static.springsource.org/.../expressions.html : Spring Expression Language (SpEL)

Enterprise Integration Patterns - Spring Integration

Although there are a number of EIP solutions, Spring Integration is the best fit for SAP Commerce, because it is seamlessly integrated with the Spring Container. The whole configuration is either done in the Spring `ApplicationContext` configuration files or through annotations. It is also very non-invasive. Your components do not need to know anything about the fact that they are triggered from within a Spring Integration pipeline. You do not have to implement any special interfaces to make them work. Spring Integration Framework can call the methods of your components by means of reflection or from within an SpEL expression.

Enterprise Integration Patterns are based on a message-driven architecture.



A publisher, which represents any component that is able to produce a Message, sends a Message to a channel. The channel transports the Messages, so that components remain loosely coupled. One or more subscribers pick up the Message from the channel and further process it. They might convert it to a different format, invoke a service method with the content of the Message (payload) as parameter, split it up into different pieces or aggregate several Messages into one. For the publisher of the Message, it does not matter what the subscribers are going to do with it. Moreover, any component from anywhere in the system might subscribe to a channel.

→ Tip

Enterprise Integration Patterns (EIP) are defined in a book of the same name, which includes a catalog of design patterns to integrate external systems in a flexible and decoupled way based on a message-driven architecture. This catalog of patterns provides a common vocabulary that can be referred to when designing integration scenarios.

Spring Integration in Practice - Submitting an Order

A practical example of using Spring Integration with external systems could be a situation when a customer submits an order and a message is sent to a special channel, the `orderSubmittedChannel`. Initially there are two subscribers to the channel:

- A component that sends out a confirmation email to the customer
- A component that triggers the fulfillment process

After the shop has been running for some time, you might want to integrate it with a CRM. The only thing you need to do, is to add a subscriber to the `orderSubmittedChannel` and have this subscriber call the CRM. This new subscriber can reside in a totally independent extension that does not have any dependency on any existing extension.

Process Engine

The Process Engine is used to execute automated processes, for example, user registration and order fulfillment. These processes often connect to external systems like email servers or warehouses. It is a good idea to use Spring Integration to connect to these systems. To do it, use the `MessageSendingAction`.

```
<bean id="action" class="de.hybris.platform.integration.processengine.MessageSendingAction">
  <property name="channel" ref="myChannel"/>
</bean>
```

This creates a message with the business process as payload, and send it to **myChannel**.

For more information, see:

- The SAP Commerce processengine
- Starting a processengine Process with a Service Activator

Spring AOP in SAP Commerce

Aspect-oriented programming (AOP) is a powerful paradigm that complements the traditional object-oriented (OO) approach to software design. It provides an elegant solution to cross cutting concerns, which OO doesn't. It enables us to apply new behavior to classes in a flexible, declarative fashion without touching the class itself.

Through the Aspect-Oriented Programming (AOP) integration to Spring, you can easily add aspects to beans, especially service beans. Be aware of Spring AOP limitations in contrast to a real AOP framework.

AOP offers elegant solutions to two problems:

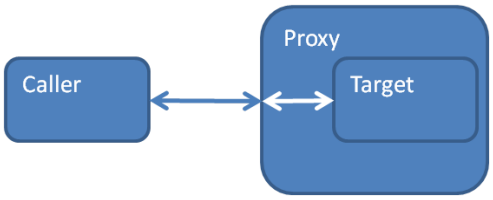
- How to avoid code scattering and duplication due to cross cutting concerns, which aren't well addressed by OO
- How to add new behavior to classes without having to modify their source code

AOP Terminology Basics

- The proxy is an **aspect**.
- The business logic performed before, after, around, or instead of the target's method is called **advice**.
- The business logic belonging to the **aspect** and representing new methods not found in the target are called **introductions**.
- The possible points at which an **advice** can be applied (**advised**) is called a **join point**. Example: All methods in all classes.
- A **pointcut** is a particular **join point** where **advice** is applied. Example: All methods starting with **get** in classes belonging to the package x.y.z.
- **Weaving** is the process of wiring up the advised object with an **aspect**

  To modify a class without access to its source code, we use so-called **aspects**. Aspects are like wrappers to a target class---this is in fact the way Spring implements its own AOP framework.

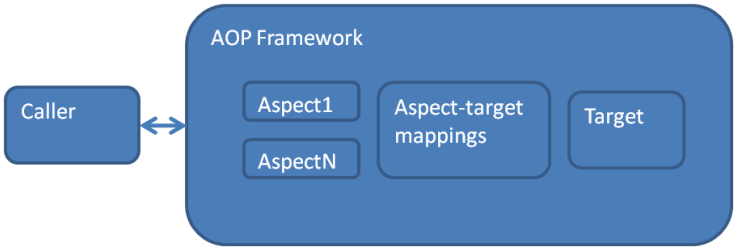
For more information, see Spring Documentation



When a caller invokes a method in the target class, the proxy intercepts the call, allowing it to perform its own behavior perhaps before, after, around, instead-of the target class's method.

In AOP, the concept of a proxy is called an aspect. In addition to offering this proxy-like behavior, the AOP framework has following characteristics:

- It provides a powerful means for declaring which methods the aspect (Proxy) should intercept (**advised** in AOP terminology) for example all methods starting with the string **get** within classes of the package x.y.z.
- It supplies the proxy with the context of the target and calling classes.
- It allows us not only to wrap an existing method but to provide entire new interfaces to target classes, a feature called **introductions**.



Registering and Using AOP Aspects

This is an example of how to register and use an AOP aspect in `extension-spring.xml`:

```
<bean id="yemptyProfBean" class="yemptypackage.aop.YEmptyProfilingAspect"/>
<aop:config proxy-target-class="true">
    <aop:aspect id="yemptyProfAspect" ref="yemptyProfBean" >
        <aop:pointcut id="profiledMethods" expression="execution(* toString(..))" />
        <aop:around pointcut-ref="profiledMethods" method="profile" />
    </aop:aspect>
</aop:config>
```

Some other examples of pointcuts that match everything:

```
<aop:pointcut id="profiledMethods" expression="bean(userService) && !execution(* toString(..))" />
<aop:pointcut id="profiledMethods" expression="execution(* *(..))" />
```

Weaving Aspects

Depending on the AOP framework chosen, weaving can be done at compile time, class-loading time, or at runtime:

- In **run-time weaving** the aspect is woven in during the execution of the application. This is how Spring implements its AOP solution, but performs slowly in comparison.
- In **compile-time weaving** the aspects are woven to the targets during compilation. This requires a special compiler, such as that from AspectJ.
- In **load-time weaving** (also called **class-loading weaving**) the aspects are woven to the targets when the target class is loaded into the JVM. This requires a special class loader to modify the byte code of a class, such as AspectJ's class loader.

AspectJ offers richer AOP support than Spring and focuses on compile-time and class-loading weaving.

Comparison of Load-Time Weaving and Compile-Time Weaving

Load-Time Weaving

Rating	Reason
➕	No impact for compilation process.
➕	Easy, detailed runtime configuration possibility through `aop.xml`
➕	Aspects are just regular java classes. No plain AspectJ source possible.
➕	Although not currently supported by SAP Commerce, the same binaries could be used for compile time weaving.
➕	Possibility to provide own concrete aspects with pointcuts.
➖	Impacts the class loader, it slows downloading classes in runtime.
➖	ITD isn't supported, it complicates inject custom interface.

Compile-Time Weaving

Rating	Reason
➕	Usage of InterTypeDedclarations simple and strong mechanism for declaring members (fields, methods, and constructors) that are owned by other types.
➕	Less runtime impact---no load class impact.
➖	Needs specifying the additional attributes (such as **inpath**) in order to weave aspects with Platform binaries.
➖	Great impact for compilation time, memory consumption, which is highly dependent of the inpath content.
➖	Problems with method conflicts between POJO methods and those introduced via **InterTypeDeclaration**
➖	Complicates the compilation process if using a separate **ajc** compiler for a `.aj` files. files).

Using Compile-Time Weaving

Binaries to be used as compile-time weaved should be compiled using AspectJ, which can be achieved as follows:

- Use **ajc**, the **AspectJ** compiler and bytecode weaver, for all classes.
- Use separate compilers:
  - For regular java classes use **javac**.
  - For an explicitly defined list of the aspect sources (`.aj` files) use the **ajc** compiler.

For more information, see Eclipse ajc Documentation

After an in-depth analysis of both options for SAP Commerce, it turned out that while compile-time weaving is very fast at run-time, the compilation process is extremely CPU and memory intensive.

Therefore, we chose to implement AspectJ's load-time weaving only. See Configuring Load-Time Weaving in SAP Commerce for more information on load-time weaving.

Configuring Load-Time Weaving in SAP Commerce

Prerequisites

In order to benefit from load-time weaving (also called class-loadingweaving), you need to provide the following:

1. The source of the aspect which looks very much like a regular class (plain Java class `.java` file, which additionally should have an **@Aspect** annotation. For the regular **javac** compiler, this class should be compilable (according to Java syntax). Logic in this class will be woven with related java code (which is configurable) during class load by an Aspect agent.

   ```
   @Aspect
   public class CoreAspect {
       ....
   }
   ```

2. Provide an XML configuration (`aop.xml`) containing declarations of Aspects to be used for weaving. This can also contain type patterns describing which types should be woven in addition to sets of options to be passed to the weaver. Since AspectJ5 it also supports the definition of concrete aspects in XML.

Configuring Load-Time Weaving

Load-time weaving is activated automatically when specific configuration files are detected in the class path of an extension during the build process.

That means that Java startup attributes are extended by AspectJ agent configuration parameters:

```
-javaagent:./bin/platform/ext/core/lib/aspectjweaver-1.6.9.jar -Dorg.aspectj.tracing.enabled=false -Daj.weaving.verbose=fa
```

This configuration defines an AspectJ agent that hooks into class-load process and weaves regular class binaries with configured aspect logic. Which java class is weaved with which aspect is determined by the `META-INF/aop.xml` file for each extension.

> **Note**
> The load-time weaving activation mechanism affects all extensions, and all binaries which are configured as woven. We recommend configuring it to affect as few binaries as possible in order to avoid performance problems.

project.properties

There are two `project.properties` available for load-time weaving customization

- **aspect.weaver.library** defines name and version of the aspect weaver library searched for in `${ext.core.path}/lib/` directory.
- **aspect.weaver.config** defines weaver options, in particular attributes for fine-tuning the AspectJ weaver.

For more information, see Eclipse AspectJ Weaver Documentation

aop.xml

The `aop.xml` is used by the agent for weaving configured binaries with defined aspects. It offers you powerful customization possibilities:

- Customize weaver options.
- It provides a tracing possibility.
- Define weaved class, for example by its package with wildcards.

- Define aspects mappings.
- Define concrete aspect mappings with pointcuts.
- Diagnose a class loading/weaving issues via dump mechanism.

For more information, see:

- Weaver Options Documentation
- Examples of Tracing
- Aspect Mappings Documentation
- Dump Mechanism Documentation

Because of the class loader scope issue, you have to provide different configurations for a core part or web part of the extension. You have to provide the `META-INF/aop.xml` file in one of the following directories:

- In the `/extension/resources` directory for Aspects in the core logic
- In the `/extension/web/resources/` directory for Aspects in the web-module logic

```
<aspectj>
    <weaver>
        <include within="de.hybris.platform..*"/>
    </weaver>
    <aspects>
        <aspect name="de.hybris.platform.aspect.CoreAspect" />
    </aspects>
</aspectj>
```

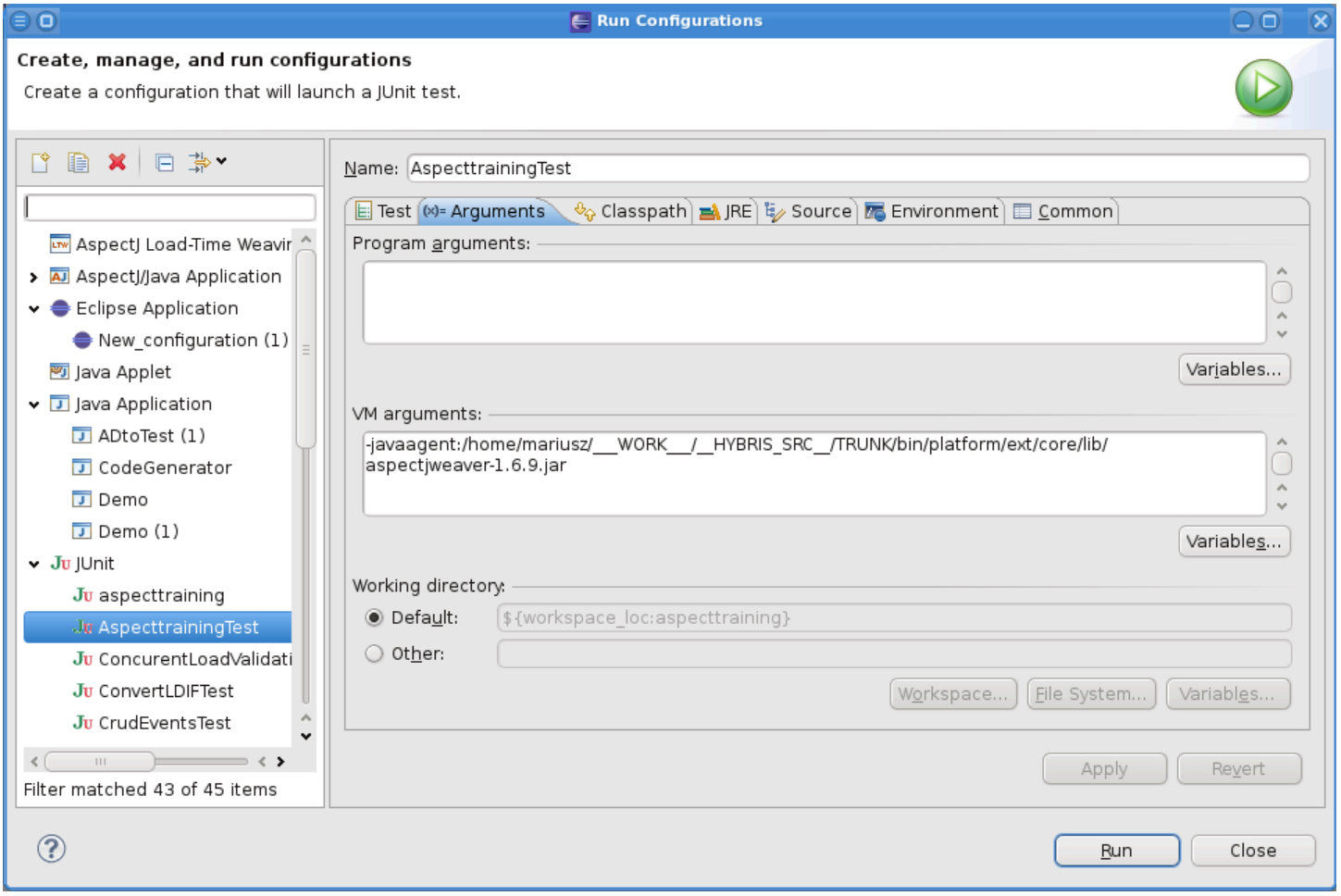
If during the prepare phase of the build process the Platform finds any `aop.xml` file in paths as above, it changes the start command (see **Configuration** section) while starting the Platform in order to start the load-time weaving agent. The agent intercepts the class loading, and it weaves classes with aspects as configured in the `aop.xml` file.

For more information, see Load-time Weaving Configuration Details

Configuring Eclipse IDE for Supporting Load-Time Weaving

In order to configure the Eclipse IDE to support load-time weaving, you should provide additional attributes to the JVM:

```
-javaagent:/bin/platform/ext/core/lib/aspectjweaver-1.6.9.jar
```



Using Load-Time Weaving

A configuration placed in the `aop.xml`, for example regarding weaver, overrides the global settings for an agent, which is provided within the `project.properties` file.

> **Note**
> **Recompilation**
>
> The `aop.xml` file for a core module is applied without recompilation of the extension. However, regarding the `aop.xml` for a web module, compilation is required to changes get applied.

Platform supports aspects as plain Java classes annotated as **@Aspect**. Because of the Platform classloading scope issue, we need to distinguish the following cases:

- Core Part of the Extension Logic
- Web Part of the Extension Logic

Core Part of the Extension Logic

This case presents:

- A simple profiling aspect
- Changing of the result of the **ProductModel.getDescription** method

```
@Aspect
public class CoreAspect {

    @Pointcut("execution( String de.hybris.platform.core.model.product.ProductModel.getDescription(..)) ")
    public void descAspect() {
    }

    @Before("descAspect()")
    public void beforeAnyDescriptionGetter(final JoinPoint jp) {
        System.out.println("Get getDescription invoked  [" + jp.toLongString() + "]");
    }

    @Around("descAspect()")
    public Object aroundDescriptionGetter(final ProceedingJoinPoint pjp) throws Throwable {
```

```
            final long startTime = System.nanoTime();
            final Object ret = pjp.proceed();
            final long endTime = System.nanoTime();
            System.out.println("Method " + pjp.getSignature().toShortString() + " took " + (endTime - startTime));
            return ret + "_aspected_description";
        }

    }
```

The test below illustrates that the aspect has been called, as **getDescription** returns a string containing the suffix **_aspected_decription**.

```
public class AspecttrainingTest extends ServicelayerTransactionalTest {

    @Resource
    private ModelService modelService;

    @Resource
    private ProductService productService;
    /** Edit the local|project.properties to change logging behaviour (properties log4j.*). */
    @SuppressWarnings("unused")
    private static final Logger LOG = Logger.getLogger(AspecttrainingTest.class.getName());

    @Before
    public void setUp() throws Exception {
        createCoreData();
        createDefaultCatalog();
    }

    /** This case presents core aspectJ injection placed in CoreAspect which adds some '_aspected_description' to descript

    @Test
    public void testAspecttraining() {

        final ProductModel product = productService.getProductForCode("testProduct0");

        assertEquals("null" + "_aspected_description", product.getDescription());

        product.setDescription("some_desc");

        modelService.save(product);

        assertEquals("some_desc" + "_aspected_description", product.getDescription());
    }
}
```

Web Part of the Extension Logic

This case presents a sample aspect for a LanguageDTO in a web context. The following code adds a custom interface on the fly to the **LanguageDTO**:

```
@Aspect
public class WebLanguageDTOAspect {

    @DeclareMixin("de.hybris.platform.core.dto.c2l.LanguageDTO")
    public CustomInterface aspectedLanguagedto() {
        //return new CustomInterfaceImpl();
        return new CustomInterface() {

            @Override
            public void setTime(final Long time) {
               ...
            }

            @Override
            public Long getTime() {

                return Long.valueOf(System.currentTimeMillis());
            }

            @Override
            public String getExtensions() {
                final StringBuffer buffer = new StringBuffer();
                for (final String s: Utilities.getExtensionNames()) {
                    buffer.append(s).append(","); //NOPMD
                }
                return buffer.toString();
            }

            @Override
            public void setExtensions(final String funny) {
                ...
            }

        };
    }
}
```

The result of this aspect is illustrated in the test below. The CustomInterface has been introduced to the **LanguageDTO**, resulting in new time and extension tags:

```
public class AspectedLanguageResourceTest extends AbstractWebServicesTest {

    private static final String URI = "languages/";
    private Language testLanguage;

    public AspectedLanguageResourceTest() throws Exception {
        super();
    }

    @Before
    public void setUpLanguages() throws ConsistencyCheckException {
        createTestsLanguages();
        testLanguage = C2LManager.getInstance().getLanguageByIsoCode("testLang1");
    }

    @Test
    public void testGetLanguage() throws IOException {

        final String text = webResource.path(URI + testLanguage.getIsoCode()).cookie(tenantCookie).
        header(HEADER_AUTH_KEY, HEADER_AUTH_VALUE_BASIC_ADMIN).
        accept(MediaType.APPLICATION_XML).get(String.class);

        assertTrue(text.indexOf("<time>") > 0);
        assertTrue(text.indexOf("</time>") > 0);
        assertTrue(text.indexOf("</extensions>") > 0);
        assertTrue(text.indexOf("<extensions>") > 0);
        final StringBuffer buffer = new StringBuffer();
        for (final String s: Utilities.getExtensionNames()) {
            buffer.append(s).append(","); //NOPMD
        }
        assertTrue(text.indexOf(buffer.toString()) > 0);
    }

}
```

The custom DTO also can be tested via a web service client:

```
http://localhost:9001/aspecttraining/rest/languages/de
```

which results in:

```
<language time="1283847062316" isocode="de" pk="8796093087776"
                                        uri="http://localhost:9001/aspecttraining/rest/languages/de/languages/de"
    xmlns:ns2="http://research.sun.com/wadl/2006/10">
    <comments/>
    <creationtime>2010-09-06T12:29:08.550+02:00</creationtime>
    <modifiedtime>2010-09-06T12:29:08.603+02:00</modifiedtime>
    <active>true</active>
    <name>German</name>
    <extensions>
                                       core
                                       ,paymentstandard
                                       ,deliveryzone
                                       ,commons
                                       ,impex
                                       ,validation
                                       ,variants
                                       ,ldap
                                       ,europe1
                                       ,category
                                       ,workflow
                                       ,processing
                                       ,catalog
                                       ,platformservices
                                       ,comments
                                       ,cockpit
                                       ,mcc
                                       ,advancedsavedquery
                                       ,admincockpit
                                       ,aspecttraining,
                                       </extensions>
    <fallbackLanguages/>
</language>
```

Working with the ServiceLayer

We provide a series of procedures covering various aspects of the ServiceLayer. Use these procedures to get you started quickly with common ServiceLayer user cases.

Start with Preparing an Extension for ServiceLayer Examples, as this is required for all other procedures. Afterwards, try any of the following:

Using the ServiceLayer:

- Displaying a User
- Displaying a Product

Adding a New Model:

- Adding a New Model

Implementing a Service:

- Adding a New Service (requires Adding a New Model)
- Extending a Service

Assumed Knowledge

Note that a certain amount of knowledge is assumed for these how-to documents including the following:

- The basic structure of an extension with core and web module; see About Extensions.
- The essentials of defining own types via the `items.xml` file; see items.xml.
- The essentials of the Spring framework; see http://www.springsource.org/ .
- Basics of the integration of Spring Framework in SAP Commerce, see Spring Framework in SAP Commerce.
- Basics of Spring MVC such as a solid idea on how request-oriented web MVC frameworks work and that you are acquainted with the vocabulary. All examples use the Spring MVC web framework; see http://static.springframework.org/spring/docs/2.5.x/reference/mvc.html . Spring MVC by nature integrates very well with the Spring Framework. Other frameworks such as Struts or Stripes (see http://struts.apache.org/ and http://www.stripesframework.org/) should work similarly, because they typically have a good Spring integration.

Preparations

These procedures require some preparatory setup steps, such as setting up and configuring an extension in a certain way. Thus, start with Preparing an Extension for ServiceLayer Examples.

Preparing an Extension for ServiceLayer Examples

The ServiceLayer example procedures require some preparatory setup steps, such as configuring your extension in a certain way. Once you have completed this task, you will then have the environment required for all subsequent ServiceLayer procedures.

Setting Up the Web Module

1. Create a new extension with the SAP Commerce extension generator `extgen` using the `yempty` template. This document assumes that the extension is named `training` and has the package root `org.training`. For detailed instruction on how to create extension, refer to Creating a New Extension.

   > **Note**
   >
   > Do not forget to reference the new extension in your `localextensions.xml` file.

   If you use Eclipse, you can then import the newly generated project into your eclipse workspace by choosing **File** > **Import** > **Existing Projects Into Workspace** and select root directory *<HYBRIS_BIN_DIR>*`/custom/training`.

2. To use the Spring MVC framework, you should add the latest release of Spring MVC library, which is part of the Spring distribution; see http://www.springsource.org/download .

3. Put the Spring MVC library JAR into the `web/webroot/WEB-INF/lib` folder of the `training` extension.

   If you use Eclipse, be sure to reference the Spring MVC library JAR in the project's build path, too. You need the `spring-webmvc.jar` in version 3.2.8

4. Add a servlet definition to the extension's `web.xml` file located in the `training/web/webroot/WEB-INF` directory. Add the following lines to the `web.xml` file:

   `web/webroot/WEB-INF/web.xml`

   ```
   <servlet>
     <servlet-name>springmvc</servlet-name>
     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
     <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
     <servlet-name>springmvc</servlet-name>
     <url-pattern>*.html</url-pattern>
   </servlet-mapping>
   ```

5. Modify the filter to handle HTML requests:

   - Modify the Platform Filters chain. The `yempty` extension template has its own filter chain, which you need to modify in the `web.xml` file:

     `web.xml`

     ```
     <filter-mapping>
             <filter-name>myyemptyPlatformFilterChain</filter-name>
             <url-pattern>*.html</url-pattern>
     </filter-mapping>
     ```

     For more details, see Platform Filters, section *Configuring Existing Filters*.

6. Furthermore, we have to add the servlet context XML file in which we can define the controller for the servlet. Create a new file with the name `springmvc-servlet.xml` in the `training/web/webroot/WEB-INF` directory.

7. Add the following snippet to the `springmvc-servlet.xml` file:

   - `springmvc-servlet.xml`

     ```
     <beans xmlns="http://www.springframework.org/schema/beans"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/b
     </beans>
     ```

   This file will be used by the servlet automatically according to the Spring naming convention.

8. Build SAP Commerce:

   - Open a command shell.
   - Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.
   - Make sure that a compliant Apache Ant version is used:
     - On Windows systems, call the *<HYBRIS_BIN_DIR>* `/platform/setantenv.bat` file. Do not close the command shell after this call as the settings are transient and would get lost if the command shell is closed.
     - On Unix systems, call the *<HYBRIS_BIN_DIR>* `/platform/setantenv.sh` file, such as: `. ./setantenv.sh`
   - Call `ant clean all` to build the entire SAP Commerce software package.

9. Start the SAP Commerce Server:

   Normal operation mode:

   a. Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.

   b. To start the SAP Commerce Server:

   - On Windows systems call the `hybrisserver.bat` file.
   - On Unix systems call the `hybrisserver.sh` file, such as: `./hybrisserver.sh`

   Debug operation mode, requiring **develop** configuration template:

   a. Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.

   b. To start the SAP Commerce Server:

   - On Windows systems run the `hybrisserver.bat` file with the debug parameter, such as `hybrisserver.bat debug`.
   - On Unix systems call the `hybrisserver.sh` file with the debug parameter, such as `./hybrisserver.sh debug`.

   For more information, see Configuration Templates.

10. Check the results: The setup is correct if no exceptions appear in the SAP Commerce Server log during start-up.

Other ServiceLayer Procedures

- Displaying a User: How to look up a user by using the ServiceLayer
- Displaying a Product: How to look up a product by using the ServiceLayer
- Adding a New Model: How to define a ServiceLayer model
- Extending a Service: How to extend an existing service
- Adding a New Service: How to define and implement a new, non-existing service (note: requires Adding a New Model)

Displaying a User

Learn how to write your own Spring MVC controller which uses the UserService for gathering a user and displaying it at a JSP view.

Creating a Frontend Controller and a View

Write a simple frontend controller that listens for requests of `user.html` and responds with a `user.jsp` view.

Prerequisites

Basically, this controller maps requests for the `user.html` to the `user.jsp` file. The effect is that both requests for `user.html` and `user.jsp` return the same file, `user.jsp`.

Procedure

1. Create a `org/training/web/controllers` package in the `training/web/src` directory.
2. Create a new class in the `training/web/src/org/training/web/controllers` folder that implements the `org.springframework.web.servlet.mvc.Controller` interface.

   This controller simply returns the name of the view.

   ```
   package org.training.web.controllers;

   import javax.servlet.http.HttpServletRequest;
   import javax.servlet.http.HttpServletResponse;

   import org.springframework.web.servlet.ModelAndView;
   import org.springframework.web.servlet.mvc.Controller;

   public class UserController implements Controller
   {

           public ModelAndView handleRequest(final HttpServletRequest request,
                   final HttpServletResponse response) throws Exception
           {
                   return new ModelAndView("user.jsp");
           }

   }
   ```

3. Create a new file named `user.jsp` in the `training/web/webroot` directory and add the following content:

   ```
   <html>
           <head>
                   <title>User</title>
           </head>
           <body>
                   User goes here
           </body>
   </html>
   ```

4. Reference the controller to the `springmvc-servlet.xml` file:

   ```
   <bean
           name="/user.html"
           class="org.training.web.controllers.UserController"/>
   ```

5. Build SAP Commerce.

   a. Open a command shell.

   b. Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.

   c. Make sure that a compliant Apache Ant version is used.

   - On Windows systems, call the *<HYBRIS_BIN_DIR>* `/platform/setantenv.bat` file. Do not close the command shell after this call as the settings are transient and would get lost if the command shell is closed.

- On Unix systems, call the *<HYBRIS_BIN_DIR>* `/platform/setantenv.sh` file, such as: `. ./setantenv.sh`

d. Call `ant clean all` to build SAP Commerce.

6. Start the SAP Commerce Server.

For normal operation mode:

a. Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.

b. To start the SAP Commerce Server:

- On Windows systems call the `hybrisserver.bat` file.
- On Unix systems call the `hybrisserver.sh` file, such as: `./hybrisserver.sh`

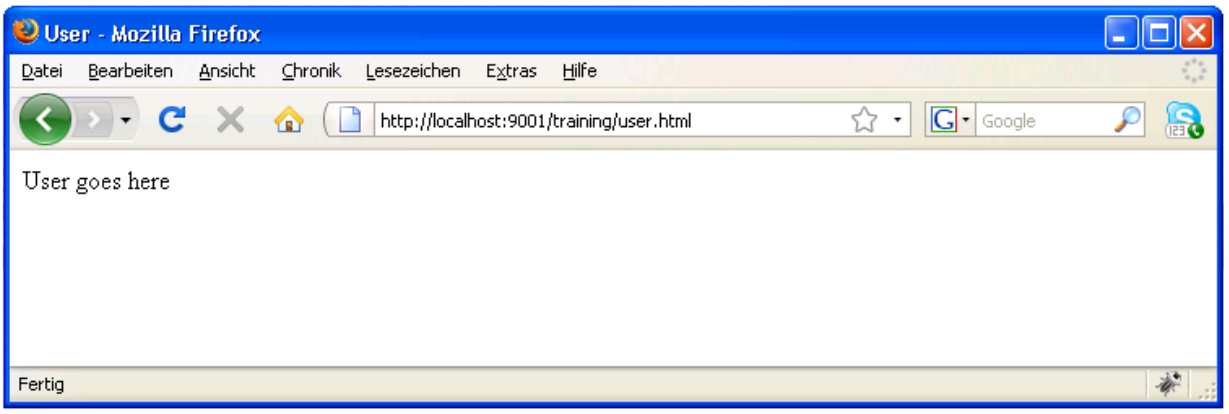
For debug operation mode, requiring **develop** configuration template:

a. Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.

b. To start the SAP Commerce Server:

- On Windows systems run the `hybrisserver.bat` file with the debug parameter, such as `hybrisserver.bat debug`.
- On Unix systems call the `hybrisserver.sh` file with the debug parameter, such as `./hybrisserver.sh debug`.

7. If you have set up everything correctly you invoke the URL `http://localhost:9001/training/user.html`.
8. Check the results.



Injecting the UserService

To display a user, you need to use the respective service. To use a service, you need to inject it using Spring.

Context

To find out the name of a service, please refer to Naming Conventions for Services. For user management, there is an instance of `de.hybris.platform.user.UserService` with the ID `userService`.

Procedure

1. Add an instance variable and a setter method to the `UserController` to hold this service.

```
import de.hybris.platform.servicelayer.user.UserService;

...

private UserService userService;

@Required
public void setUserService(final UserService userService)
{
	this.userService = userService;
}
```

2. Modify the Spring bean definition of the controller.

```
<bean name="/user.html" class="org.training.web.controllers.UserController">
	<property name="userService" ref="userService"/>
</bean>
```

3. The result is that a bean named `userService` is injected into the property of the same name.

Looking Up the Current User

To look up the current user, use the `getCurrentUser()` method.

Procedure

1. Import the `UserModel` class.

   ```
   de.hybris.platform.core.model.user.UserModel;
   ```

2. In the `handleRequest(...)` method of the `UserController`, add this line:

   ```
   UserModel user = userService.getCurrentUser();
   ```

3. For the user to be available in the view, add it to the model map and add this model map to the `ModelAndView`:

   ```
   Map<String, Object> model = new HashMap<String, Object>();
   model.put("user", user);
   return new ModelAndView("user.jsp", model);
   ```

   The entire `handleRequest(...)` method now looks like this:

   ```
   public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception
   {
           UserModel user = userService.getCurrentUser();

           Map<String, Object> model = new HashMap<String, Object>();
           model.put("user", user);
           return new ModelAndView("user.jsp", model);
   }
   ```

Displaying the Current User

Follow the steps to display the current user.

Context

The sample used here is straightforward and uses JSTL expressions (see also http://java.sun.com/products/jsp/jstl/) to access the model.

Procedure

1. Modify the `user.jsp` page.

   ```
   <html>
           <head>
                   <title>User</title>
           </head>
           <body>
                   <h1>${user.uid}</h1>
                   <b>Name:</b> ${user.name}
                   <br/>
                   <b>Description:</b> ${user.description}
           </body>
   </html>
   ```

2. Build SAP Commerce.

   a. Open a command shell.

   b. Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.

   c. Make sure that a compliant Apache Ant version is used.

   - On Windows systems, call the *<HYBRIS_BIN_DIR>* `/platform/setantenv.bat` file. Do not close the command shell after this call as the settings are transient and would get lost if the command shell is closed.
   - On Unix systems, call the *<HYBRIS_BIN_DIR>* `/platform/setantenv.sh` file, such as: `. ./setantenv.sh`

   d. Call `ant clean all` to build SAP Commerce.

3. Start the SAP Commerce Server.

For normal operation mode:

a. Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.

b. To start the SAP Commerce Server:

- On Windows systems call the `hybrisserver.bat` file.
- On Unix systems call the `hybrisserver.sh` file, such as: `./hybrisserver.sh`

For debug operation mode, requiring **develop** configuration template:

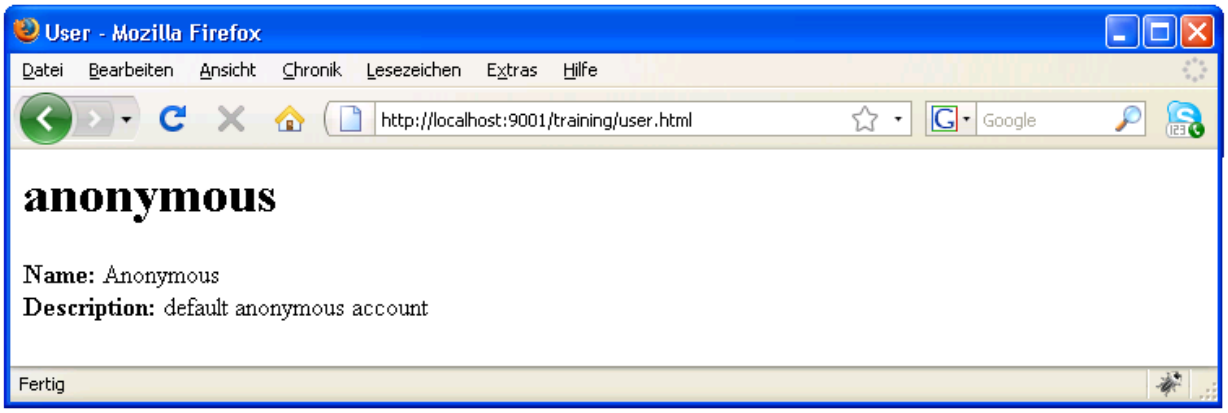
a. Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.

b. To start the SAP Commerce Server:

- On Windows systems run the `hybrisserver.bat` file with the debug parameter, such as `hybrisserver.bat debug`.
- On Unix systems call the `hybrisserver.sh` file with the debug parameter, such as `./hybrisserver.sh debug`.

4. Check the results.

You should see something like that:



Looking Up and Displaying a Custom User

Learn how to display a specific user instead of the current one.

Context

To display a specific user instead of the current one, use a URL parameter to pass the desired UID, such as: `http://localhost:9001/training/user.html?uid=admin`. That way, you only have to adjust the controller to take the passed UID and do the lookup by UID. In the frontend, you do not have to change anything.

Procedure

1. Modify the code:

- Get the URL parameter from the request object.
- Handle the case that the URL parameter may be null (because simply no parameter is passed). For example, you could use the current user.
- Look up the user by UID.

```
public ModelAndView handleRequest(final HttpServletRequest request, final HttpServletResponse response) throws Excep
{
        final String uid = request.getParameter("uid");
        UserModel user = null;
        if (uid == null)
        {
                user = userService.getCurrentUser();
        }
        else
        {
                user = userService.getUser(uid);
        }
        final Map<String, Object> model = new HashMap<String, Object>();
        model.put("user", user);
        return new ModelAndView("user.jsp", model);
}
```

2. Build SAP Commerce.

a. Open a command shell.

b. Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.

c. Make sure that a compliant Apache Ant version is used.

- On Windows systems, call the *<HYBRIS_BIN_DIR>* `/platform/setantenv.bat` file. Do not close the command shell after this call as the settings are transient and would get lost if the command shell is closed.
- On Unix systems, call the *<HYBRIS_BIN_DIR>* `/platform/setantenv.sh` file, such as: `. ./setantenv.sh`

d. Call `ant clean all` to build SAP Commerce.

3. Start the SAP Commerce Server.

   For normal operation mode:

   a. Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.

   b. To start the SAP Commerce Server:

   - On Windows systems call the `hybrisserver.bat` file.
   - On Unix systems call the `hybrisserver.sh` file, such as: `./hybrisserver.sh`

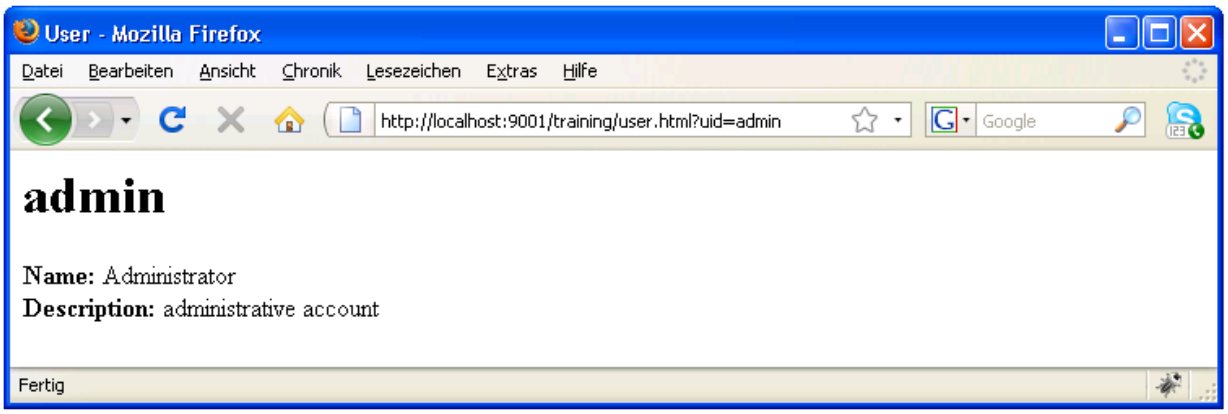
   For debug operation mode, requiring **develop** configuration template:

   a. Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.

   b. To start the SAP Commerce Server:

   - On Windows systems run the `hybrisserver.bat` file with the debug parameter, such as `hybrisserver.bat debug`.
   - On Unix systems call the `hybrisserver.sh` file with the debug parameter, such as `./hybrisserver.sh debug`.

4. Navigate to `http://localhost:9001/training/user.html?uid=admin`.
5. Check the results.



Displaying a Product

Use the `ProductService` for displaying a product on a frontend JSP. This differs to displaying a user as the product type is managed by catalogs. This implies additional logic for setting the correct catalog versions at the session context.

Prerequisites

This example requires a prepared extension to be set up in a certain way. If you have completed other tutorials of the ServiceLayer documentation, you typically have an extension available as needed.

For more information, see Preparing an Extension for ServiceLayer Examples.

Creating a Frontend Controller and a View

Write a simple frontend controller that listens for requests of `product.html` and responds with a `product.jsp` view. Basically, this controller maps requests for the `product.html` to the `product.jsp` file. The effect is that both requests for `product.html` and `product.jsp` return the same file, `product.jsp`.

1. In the `training/web/src` directory, create a `org/training/web/controllers` package.
2. Create a new class in the `training/web/src/org/training/web/controllers` folder that implements the `org.springframework.web.servlet.mvc.Controller` interface:

   `ProductController.java`

```
package org.training.web.controllers;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class ProductController implements Controller
{

	public ModelAndView handleRequest(final HttpServletRequest request,
		final HttpServletResponse response) throws Exception
	{
		return new ModelAndView("product.jsp");
	}

}
```

This controller simply returns the name of the view.

3. Create a new file named `product.jsp` in the `training/web/webroot` directory and add the following content:

   `web/webroot/product.jsp`

```
<html>
	<head>
		<title>Product</title>
	</head>
	<body>
		Product goes here
	</body>
</html>
```

4. Reference the controller to the `springmvc-servlet.xml` file:

   `web/webroot/WEB-INF/springmvc-servlet.xml`

```
<bean
	name="/product.html"
	class="org.training.web.controllers.ProductController"/>
```

5. Build SAP Commerce:
   - Open a command shell.
   - Navigate to the *<${HYBRIS_BIN_DIR}>* `/platform` directory.
   - Make sure that a compliant Apache Ant version is used:
     - On Windows systems, call the *<${HYBRIS_BIN_DIR}>* `/platform/setantenv.bat` file. Do not close the command shell after this call as the settings are transient and would get lost if the command shell is closed.
     - On Unix systems, call the *<${HYBRIS_BIN_DIR}>* `/platform/setantenv.sh` file, such as: `. ./setantenv.sh`
   - Call `ant clean all` to build SAP Commerce.
6. Start the SAP Commerce Server:

   Normal operation mode:

   a. Navigate to the *<${HYBRIS_BIN_DIR}>* `/platform` directory.

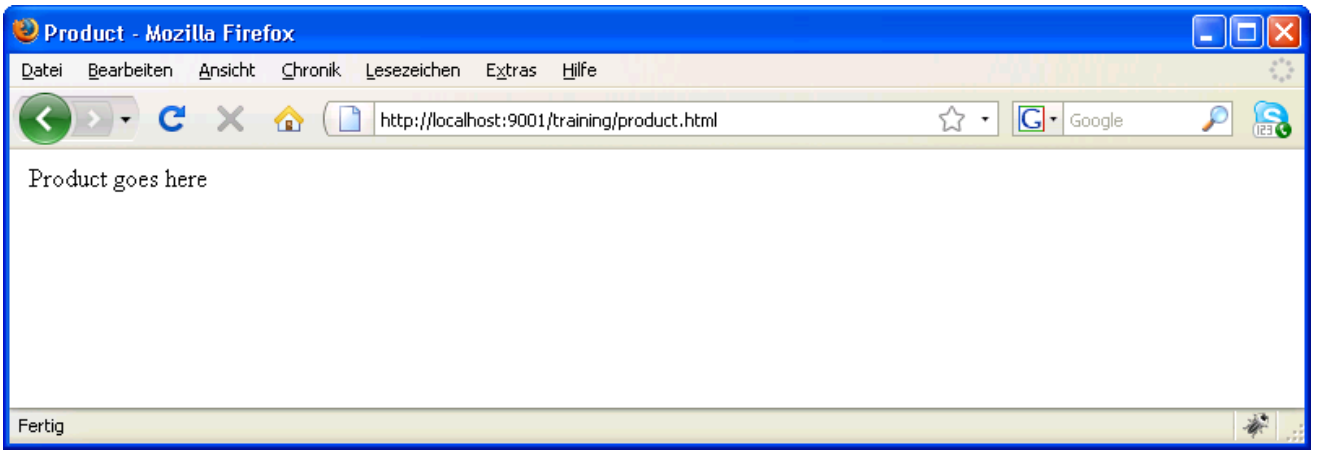
   b. To start the SAP Commerce Server:
      - On Windows systems call the `hybrisserver.bat` file.
      - On Unix systems call the `hybrisserver.sh` file, such as: `./hybrisserver.sh`

   Debug operation mode, requiring **develop** configuration template:

   a. Navigate to the *<${HYBRIS_BIN_DIR}>* `/platform` directory.

   b. To start the SAP Commerce Server:
      - On Windows systems run the `hybrisserver.bat` file with the debug parameter, such as `hybrisserver.bat debug`.
      - On Unix systems call the `hybrisserver.sh` file with the debug parameter, such as `./hybrisserver.sh debug`.

   For more information, see Configuration Templates.
7. If you have set up everything correctly you invoke the URL `http://localhost:9001/training/product.html`.
8. Check the results: You should see something like this:



Injecting the ProductService

To display a product with a certain **code**, passed through an URL such as `http://localhost:9001/training/product.html?code=HW2120-0341`, you need to use the respective service that can look up products by their code. To find out the name of a service, refer to Naming Conventions for Services. For product management, there is an instance of `de.hybris.platform.product.ProductService` with the ID `productService`.

1. To hold this service, add an instance variable and a setter method to the `ProductController`:

   `ProductController.java`

   ```
   private ProductService productService;

   public void setProductService(final ProductService productService)
   {
           this.productService = productService;
   }
   ```

2. To obtain the `ProductService`, use a dependency injection. For this, you have to modify the Spring bean definition of the controller:

   `web/webroot/WEB-INF/springmvc-servlet.xml`

   ```
   <bean name="/product.html" class="org.training.web.controllers.ProductController">
           <property name="productService" ref="productService"/>
   </bean>
   ```

3. The result is that a bean named `productService` is injected into the property of the same name.

For more information, see dependency injection in Spring Framework in SAP Commerce.

Looking Up the Product

In `ProductService`, there is a method called `getProductForCode(String code)`. The `code` parameter indicates the code of the product to look up. To pass a code, we will use a URL parameter, which we can retrieve from the `HttpRequest` object.

1. To extract the product code from the URL, add the following piece of code to the `handleRequest(...)` method of our `ProductController`:

   ```
   String code = request.getParameter("code");
   ProductModel product = null;
   if (code != null)
   {
           product = productService.getProductForCode(code);
   }
   ```

2. For the product to be available in the view, add it to the model map and add this model map to our `ModelAndView`:

   ```
   Map<String, Object> model = new HashMap<String, Object>();
   model.put("product", product);
   return new ModelAndView("product.jsp", model);
   ```

3. The entire `handleRequest(...)` method implementation now looks like this:

   `ProductController.java`

   ```
   public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception
   {
           String code = request.getParameter("code");
           ProductModel product = null;
           if (code != null)
   ```

```
                {
                        product = productService.getProductForCode(code);
                }
                Map<String, Object> model = new HashMap<String, Object>();
                model.put("product", product);
                return new ModelAndView("product.jsp", model);
        }
```

Displaying the Product

The frontend JSP page is pretty straightforward. It uses JSTL expressions (see also http://java.sun.com/products/jsp/jstl/) to access the model:

`web/webroot/product.jsp`

```
<html>
        <head>
                <title>Product</title>
        </head>
        <body>
                <h1>${product.name}</h1>
                ${product.description}
        </body>
</html>
```

First Attempt to Run

1. Rebuild SAP Commerce:
    - Open a command shell.
    - Navigate to the *<${HYBRIS_BIN_DIR}>* `/platform` directory.
    - Make sure that a compliant Apache Ant version is used:
        - On Windows systems, call the *<${HYBRIS_BIN_DIR}>* `/platform/setantenv.bat` file. Do not close the command shell after this call as the settings are transient and would get lost if the command shell is closed.
        - On Unix systems, call the *<${HYBRIS_BIN_DIR}>* `/platform/setantenv.sh` file, such as: `. ./setantenv.sh`
    - Call `ant clean all` to build SAP Commerce.
2. Restart SAP Commerce:

    Normal operation mode:

    a. Navigate to the *<${HYBRIS_BIN_DIR}>* `/platform` directory.

    b. To start the SAP Commerce Server:

    - On Windows systems call the `hybrisserver.bat` file.
    - On Unix systems call the `hybrisserver.sh` file, such as: `./hybrisserver.sh`

    Debug operation mode, requiring **develop** configuration template:

    a. Navigate to the *<${HYBRIS_BIN_DIR}>* `/platform` directory.

    b. To start the SAP Commerce Server:

    - On Windows systems run the `hybrisserver.bat` file with the debug parameter, such as `hybrisserver.bat debug`.
    - On Unix systems call the `hybrisserver.sh` file with the debug parameter, such as `./hybrisserver.sh debug`.

    For more information, see Configuration Templates.
3. Navigate to `http://localhost:9001/training/product.html?code=HW2120-0341`. This will result in an exception that the `session.catalogversions attribute` is not translatable.

Setting Up the Catalog Version

This exception is caused by the ProductService which uses the FlexibleSearch (FlexibleSearch) to look up the product. Because the default user is the anonymous user all search restrictions are firing, and to satisfy these restrictions we have to activate the hardware catalog online catalog version at the session scope, so we can then access its products:

1. Set the correct catalog version via the ProductController itself for reasons of simplicity. Note that this is a hack. The by-the-book way would be to either write a filter containing this logic or to define URL patterns for the catalog.

    To set the catalog version for the session the CatalogService provides a suitable method: `setSessionCatalogVersion`. So we only have to add the service to our controller and call this method. The overall ProductController code will look like this then:

ProductController.java

```
private CatalogService catalogService;
private ProductService productService;

public void setCatalogService(final CatalogService catalogService)
{
        this.catalogService = catalogService;
}
public void setProductService(final ProductService productService)
{
        this.productService = productService;
}

public ModelAndView handleRequest(final HttpServletRequest request, final HttpServletResponse response) throws Excep
{
        catalogService.setSessionCatalogVersion("hwcatalog", "Online");

        final String code = request.getParameter("code");
        ProductModel product = null;
        if (code != null)
        {
                product = productService.getProduct(code);
        }
        final Map<String, Object> model = new HashMap<String, Object>();
        model.put("product", product);
        return new ModelAndView("product.jsp", model);
}
```

2. Add the dependency of the `CatalogService` to the `springmvc-servlet.xml` file:

web/webroot/WEB-INF/springmvc-servlet.xml

```
<bean name="/product.html" class="org.training.web.controllers.ProductController">
        <property name="productService" ref="productService"/>
        <property name="catalogService" ref="catalogService"/>
</bean>
```

Second Attempt to Run

1. Rebuild SAP Commerce:
   - Open a command shell.
   - Navigate to the *<${HYBRIS_BIN_DIR}>* `/platform` directory.
   - Make sure that a compliant Apache Ant version is used:
     - On Windows systems, call the *<${HYBRIS_BIN_DIR}>* `/platform/setantenv.bat` file. Do not close the command shell after this call as the settings are transient and would get lost if the command shell is closed.
     - On Unix systems, call the *<${HYBRIS_BIN_DIR}>* `/platform/setantenv.sh` file, such as: `. ./setantenv.sh`
   - Call `ant clean all` to build SAP Commerce.
2. Restart SAP Commerce:

   Normal operation mode:

   a. Navigate to the *<${HYBRIS_BIN_DIR}>* `/platform` directory.

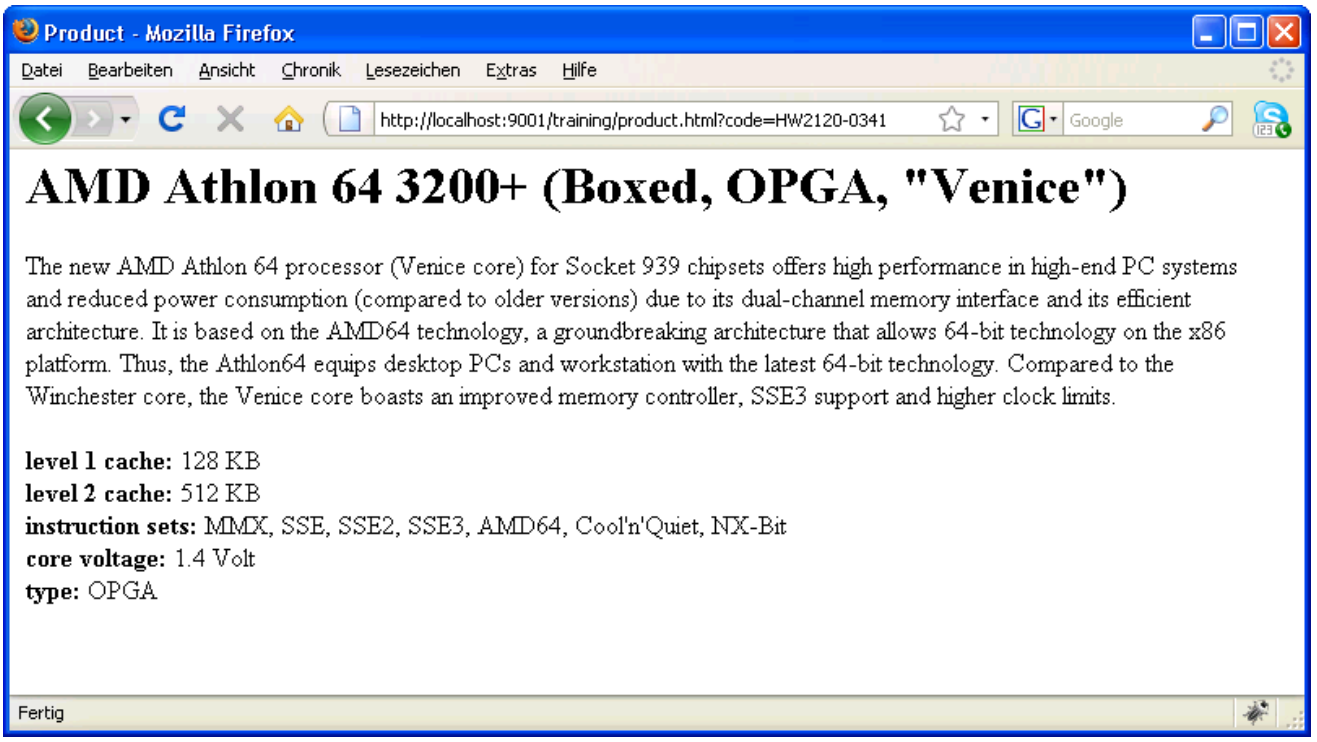
   b. To start the SAP Commerce Server:
      - On Windows systems call the `hybrisserver.bat` file.
      - On Unix systems call the `hybrisserver.sh` file, such as: `./hybrisserver.sh`

   Debug operation mode, requiring **develop** configuration template:

   a. Navigate to the *<${HYBRIS_BIN_DIR}>* `/platform` directory.

   b. To start the SAP Commerce Server:
      - On Windows systems run the `hybrisserver.bat` file with the debug parameter, such as `hybrisserver.bat debug`.
      - On Unix systems call the `hybrisserver.sh` file with the debug parameter, such as `./hybrisserver.sh debug`.

   For more information, see Configuration Templates.
3. Navigate to `http://localhost:9001/training/product.html?code=HW2120-0341`. This time, the result will look like this:



Other ServiceLayer Procedures

- Displaying a User: How to look up a user by using the ServiceLayer
- Adding a New Model: How to define a ServiceLayer model
- Extending a Service: How to extend an existing service
- Adding a New Service: How to define and implement a new, non-existing service. Requires Adding a New Model.

Adding a New Service

If you want to provide a form where users of web application can send contact requests, this is a simple request that contains `sender` and `message` fields. To accomplish this, you must develop a service that can store and read such a contact request.

Prerequisites

This example requires a prepared extension to be set up in a certain way. For details, see Preparing an Extension for ServiceLayer Examples. If you have completed other procedures of the ServiceLayer documentation, you typically have an extension available as needed.

Furthermore this tutorial requires the final extension state of Adding a New Model.

Defining a Model

See Adding a New Model.

Defining an Interface for the Service

The service needs a method to find `ContactRequest` items by sender. Creation and modification of `ContactRequest` items are done directly at the controller class. While technically it is not necessary to back a service with an interface definition, we recommend using interface-driven design. For more information, see Spring Framework in SAP Commerce.

The interface is placed at the `src` folder of the extension with the package `org.training` and looks like this:

`org.training.ContactRequestService`

```
package org.training;

public interface ContactRequestService
{
```

```
    ContactRequestModel getContactRequest(String sender);
}
```

Implementing the Service

1. Implement the service by writing a class implementing the defined interface:

   `org.training.impl.DefaultContactRequestService`

```
package org.training.impl;

public class DefaultContactRequestService implements ContactRequestService
{
   private FlexibleSearchService flexibleSearchService;

   public ContactRequestModel getContactRequest(final String sender)
   {
      final String queryString = "SELECT {PK} FROM {ContactRequest} WHERE {sender} = ?sender";
      final FlexibleSearchQuery query = new FlexibleSearchQuery(queryString);
      query.addQueryParameter("sender", sender);
      final SearchResult<ContactRequestModel> result = this.flexibleSearchService.search(query);
      final int resultCount = result.getTotalCount();
      if (resultCount == 0)
      {
         throw new UnknownIdentifierException(
            "ContactRequest with sender '" + sender + "' not found!"
         );
      }
      else if (resultCount > 1)
      {
         throw new AmbiguousIdentifierException(
            "ContactRequest with sender '" + sender + "' is not unique, " + resultCount
               + " requests found!"
         );
      }
      return result.getResult().get(0);
   }

   public void setFlexibleSearchService(final FlexibleSearchService flexibleSearchService)
   {
      this.flexibleSearchService = flexibleSearchService;
   }
}
```

2. You can now register this service in the Spring configuration to inject it at the controller later. Add the following lines to the `training-spring.xml` file:

   `training/resources/training-spring.xml`

```
<bean id="contactRequestService"
   class="org.training.impl.DefaultContactRequestService"  >
   <property name="flexibleSearchService" ref="flexibleSearchService"/>
</bean>
```

Creating a Frontend Controller and a View

Create a single view to display, create, and modify `ContactRequest` s. The view can be called with a sender parameter to display a single `ContactRequest` of a specific sender. Furthermore it has a form to modify the values of a `ContactRequest`. If no sender is specified as request parameter, the fields of the form is used to create a new `ContactRequest`.

1. Write a frontend controller `org.training.web.controllers.ContactRequestController.java`. It uses the new `ContactRequestService` to gather existing `ContactRequest` s and the `ModelService` to save or create a `ContactRequestModel`:

   `ContactRequestController`

```
public class ContactRequestController implements Controller
{
   private ContactRequestService contactRequestService;
   private ModelService modelService;

   @Override
   public ModelAndView handleRequest
      (final HttpServletRequest request, final HttpServletResponse response) throws Exception
   {
      final String sender = request.getParameter("sender");
      ContactRequestModel contactRequest = null;
      if (sender != null)
      {
         try
         {
            contactRequest = contactRequestService.getContactRequest(sender);
         }
```

```
            catch (final UnknownIdentifierException e)
            {
               // OK, nothing found
            }
         }

         if (request.getMethod().equalsIgnoreCase("POST"))
         {
            if (contactRequest == null)
            {
               contactRequest = new ContactRequestModel();
               modelService.attach(contactRequest);
            }
            final String newSender = request.getParameter("newSender");
            final String newMessage = request.getParameter("newMessage");
            if (newSender != null)
            {
               contactRequest.setSender(newSender);
            }
            if (newMessage != null)
            {
               contactRequest.setMessage(newMessage);
            }
            modelService.save(contactRequest);
         }

         final Map<String, Object> model = new HashMap<String, Object>();
         model.put("contactRequest", contactRequest);
         return new ModelAndView("contactRequest.jsp", model);
      }

      public void setContactRequestService(final ContactRequestService contactRequestService)
      {
         this.contactRequestService = contactRequestService;
      }

      public void setModelService(final ModelService modelService)
      {
         this.modelService = modelService;
      }
   }
```

2. Register the controller in the Spring configuration file, such as:

   `training/web/webroot/WEB-INF/springmvc-servlet.xml`

```
<bean name="/contactRequest.html"
   class="org.training.web.controllers.ContactRequestController">
   <property name="contactRequestService" ref="contactRequestService"/>
   <property name="modelService" ref="modelService"/>
</bean>
```

3. Write the view:

   `training/web/webroot/contactRequest.jsp`

```
<html>
<head><title>Contact Request</title></head>
<body>
   <form method="post">
       <label for="login">Sender: </label>
       <input type="text" name="newSender" value="${contactRequest.sender}"/>
       <br/>
       <label for="password">Message:</label>
       <textarea name="newMessage">${contactRequest.message}</textarea>
       <br/>
       <input type="submit" value="Send"/>
   </form>
</body>
</html>
```

(Optional) Implementing a Data Access Object (DAO)

The Data Access Object (DAO) pattern allows isolating persistence layer accesses from the business logic. You define an interface that contains all data access methods. Usually, DAOs will have finder methods that run searches on the persistence layer (such as While the service could directly access the persistence layer, it is better to put all storage related logic in a separate class. That way, you can better mock and test, but you can also switch the underlying storage implementation more easily.`findByCode(...)`) or methods to save, update, and remove objects.

The DAO is the bridge between models and items. SAP Commerce provides base classes that handle the conversion for you but you still have to use the FlexibleSearch to find your objects.

For more information, see:

- Mocking of Models
- FlexibleSearch

Generally, you have several options to implement a DAO:

- You can extend from `AbstractItemDao`While the service could directly access the persistence layer, it is better to put all storage related logic in a separate class. That, which provides many helper methods and already has injected the `ModelService` and `FlexibleSearchService` to facilitate the item retrieval.
- You can implement the `Dao` interface (`de.hybris.platform.servicelayer.internal.dao` package)

  The `Dao` interface is only a marker and does not declare any methods. By consequence, you will need to declare a sub-interface of `Dao` for your implementation class, such as:

```
public interface MyDao extends Dao
{
   public void myMethod()
}
```

For this example, you are going to define a sub-interface and extend from `AbstractItemDao`.

1. The `Dao` interface does not declare a method to find contact requests by sender, so you need to define a sub-interface:

   `org.training.daos.ContactRequestDao`

```
package org.training.daos;

public interface ContactRequestDao extends Dao
{
   public List<ContactRequestModel> findBySender(String sender);
}
```

2. For this DAO implementation, extend from `AbstractItemDao`, such as:

   `org.training.daos.impl.DefaultContactRequestDao`

```
package org.training.daos.impl;

public class DefaultContactRequestDao extends AbstractItemDao implements ContactRequestDao
{

   public List<ContactRequestModel> findBySender(final String sender)
   {
      final String queryString = String.format("SELECT {%s} FROM {%s} WHERE {%s} = ?sender",
            ContactRequest.PK, "ContactRequest",
            ContactRequest.SENDER);
      final FlexibleSearchQuery query = new FlexibleSearchQuery(queryString);
      query.addQueryParameter("sender", sender);
      final SearchResult<ContactRequestModel> result = flexibleSearchService.search(query);
      return result.getResult();
   }

}
```

3. Now you have to create a bean definition for this DAO:

   `training/resources/training-spring.xml`

```
<bean id="contactRequestDao"
   class="org.training.daos.impl.DefaultContactRequestDao"
   parent="abstractItemDao"  />
```

   This bean definition references `abstractItemDao` as parent. Thus, make sure you have the `flexibleSearchService` properly injected.

4. You can now adjust the service you already implemented to use the DAO instead of using the FlexibleSearch directly:

   `org.training.impl.DefaultContactRequestService`

```
package org.training.impl;

public class DefaultContactRequestService implements ContactRequestService
{

   private ContactRequestDao contactRequestDao;

   public ContactRequestModel getContactRequest(final String sender)
   {
      final List<ContactRequestModel> result = contactRequestDao.findBySender(sender);
      final int resultCount = result.size();
      if (resultCount == 0)
      {
         throw new UnknownIdentifierException(
            "ContactRequest with sender '" + sender + "' not found!"
```

```
            );
         }
         else if (resultCount > 1)
         {
            throw new AmbiguousIdentifierException(
               "ContactRequest with sender '" + sender + "' is not unique, "
                  + resultCount + " requests found!"
            );
         }
         return result.iterator().next();
      }

      public void setContactRequestDao(final ContactRequestDao contactRequestDao)
      {
         this.contactRequestDao = contactRequestDao;
      }
   }
```

5. Modify the bean definition of the service, such as:

   `training/resources/training-spring.xml`

```
<bean id="contactRequestService"
   class="org.training.impl.DefaultContactRequestService"  >
   <property name="contactRequestDao" ref="contactRequestDao"/>
</bean>
```

Other ServiceLayer Procedures

- Displaying a User: How to look up a user by using the ServiceLayer
- Displaying a Product: How to look up a product by using the ServiceLayer
- Extending a Service: How to extend an existing service

Extending a Service

SAP Commerce comes with a variety of built-in services. You can extend these existing SAP Commerce services to meet your specific business needs.

There are different ways of extending an existing service:

- You can **add** logic to an existing service. Your service has the original service's API plus your implementation.
- You can **change** (that is, override) an existing service's method. Your service has the original service's API with a different underlying implementation.
- You can **replace** an entire service. Your service has a different API than the original service.

Basic Procedure

Regardless of which way of extending a service you chose, perform the following steps.

Prerequisites

This tutorial requires a prepared extension to be set up in a certain way. For details see Preparing an Extension for ServiceLayer Examples. If you have completed other tutorials of the ServiceLayer documentation, you typically have an extension available as needed.

Procedure

1. Implement the actual service.
2. Reference the service by overriding the bean definition.
3. Restart the SAP Commerce Server.

   SAP Commerce Server Restarts Automatically After the Build

   Part of the build process of SAP Commerce is a check whether the SAP Commerce Server is running. If you rebuild SAP Commerce while the SAP Commerce Server is running, the Build Framework automatically restarts the SAP Commerce Server.

Create a Controller and a View

To see the result of calling your new logic, prepare a Spring Controller and a view. In this section, you will write a simple controller that listens for requests for `trainingProduct.html` and responds with a `trainingProduct.jsp` view.

Procedure

1. If it doesn't exist, create a `org/training/web/controllers` directory in the `training/web/src` directory.

   This contains the Java packages later on.

2. Create a new class in the `training/web/src/org/training/web/controllers` folder that implements the `org.springframework.web.servlet.mvc.Controller` interface:

```
package org.training.web.controllers;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class TrainingProductController implements Controller
{

        public ModelAndView handleRequest(final HttpServletRequest request,
                final HttpServletResponse response) throws Exception
        {
                return new ModelAndView("trainingProduct.jsp");
        }

}
```

3. Create a new file named `trainingProduct.jsp` in the `training/web/webroot` directory and add the following content:

```
<html>
        <head>
                <title>Product</title>
        </head>
        <body>
                Product goes here
        </body>
</html>
```

4. Reference the controller in the `springmvc-servlet.xml` file:

```
<bean
        name="/trainingProduct.html"
        class="org.training.web.controllers.TrainingProductController"/>
```

The Original Service

This example uses the `ProductService` shipped with the `platformservices` extension to show how to extend services.

The examples mainly use the `getProductForCode(String code)` method:

`ProductService.java`

```
package de.hybris.platform.product;

public interface ProductService
{
        ProductModel getProductForCode(String productCode);
        ...
}
```

The bean definition looks like this:

`platform/ext/platformservices/resources/product-spring.xml`

```
<alias alias="productService" name="defaultProductService"/>
<bean id="defaultProductService" class="de.hybris.platform.product.impl.DefaultProductService"
                parent="abstractBusinessService" >
                <property name="productDao" ref="productDao"/>
                <property name="unitDao" ref="unitDao"/>
                <property name="priceService" ref="priceService"/>
</bean>
```

Adding Logic to an Existing Service

Learn how to add business logic to an existing service.

Sometimes an existing service does precisely what you need to do, only some business logic is missing. Let's say, you wish to extend the `ProductService` definition with custom logic to look up a product by code and name instead of only by code. This section explains how to add business logic to an existing service demonstrated by adding a new method `getProductForCode(String code, String name)` to the `ProductService` in order to gather a product by code and name attribute.

→ Tip

Instead of adding logic to an existing service, consider creating a decorator for the service instead.

Extend the Interface

When you add a method, you have to extend the original service's interface. To do so, add a new interface `TrainingProductService` to the training's `src` folder:

`TrainingProductService.java`

```
package org.training;

public interface TrainingProductService extends ProductService
{
	ProductModel getProductForCode(String code, String name);
}
```

Extending the Implementation

After extending the interface, implement the `TrainingProductServiceImpl.java` class by extending the original implementation and adding the new method:

```
package org.training.impl;

public class TrainingProductServiceImpl extends DefaultProductService implements TrainingProductService
{
	ProductModel getProductForCodeAndName(String code, String name)
	{
	    final FlexibleSearchQuery query = new FlexibleSearchQuery("SELECT {pk} FROM {Product} WHERE "
	        + "{code}=?code AND {name} LIKE %?name%");
	        query.addQueryParameter(Product.CODE, code);
	        query.addQueryParameter(Product.NAME, name);
	        final SearchResult<ProductModel> result = this.flexibleSearchService.search(query);
	        final int resultCount = result.getTotalCount();
	        if (resultCount == 0)
	        {
	                throw new UnknownIdentifierException("Product not found!");
	        }
	        else if (resultCount > 1)
	        {
	                throw new AmbiguousIdentifierException("Product code and name is not unique!");
	        }
	        return result.getResult().get(0);
	};
}
```

Overriding the Bean Definition

Add a new bean definition with the ID `trainingProductService` to the training spring config file:

`training/resources/training-spring.xml`

```
<bean id="trainingProductService" class="org.training.impl.TrainingProductServiceImpl"
	parent="defaultProductService"/>
```

ⓘ Note

Choosing The Correct Configuration File

In extensions that use Spring MVC framework, there are three different spring configuration files. Make sure, you choose the right one to add the bean definition. For more information about different files, refer to Spring Framework in SAP Commerce.

Make sure to use `defaultProductService` as the parent bean definition. That way, your implementation inherits all dependencies from the original definition.

Now, you want to use this service instead of the original one. Without a clever approach, you would have to modify all Spring XML configuration files and replace all injections of `productService` with `trainingProductService`. Not only is this a tedious task, but you would also have to modify configuration files that are not part of your extension but part of the Platform. Although you can modify SAP Commerce configuration files, you would have to be careful during a SAP Commerce version upgrade: you would have to merge your modified configuration files into the configuration files coming with SAP Commerce. This would mean that you would have to do manual merging on every single SAP Commerce version update.

To avoid this overhead, override the **alias** of the original bean definition. For more details on aliases, refer to Spring Framework in SAP Commerce, section *Beans*:

`platform/ext/platformservices/resources/product-spring.xml`

```
<alias alias="productService" name="defaultProductService"/>
```

To override the alias, add the following line to the extension-specific configuration file:

`training/resources/training-spring.xml`

```
<alias alias="productService" name="trainingProductService"/>
```

Because your configuration file is loaded after the original configuration file, Spring uses your alias instead of the original one. Thus, wherever a reference to `productService` turns up, Spring uses your `trainingProductService` instead.

Using the Extended Service Within the Controller

Now, you are going to display the product with a certain **code** and **name**, passed via a URL such as `http://localhost:9001/training/product.html?code=HW2120-0341&name=AMD`.

1. Implement the controller already created in the topic Create a Controller and a View above. You do it similarly as in Displaying a Product, but here you use the new `TrainingProductService` interface and, additionally, the parameter `name`:

   `TrainingProductController.java`

   ```
   private CatalogService catalogService;
   private TrainingProductService trainingProductService;

   public void setCatalogService(final CatalogService catalogService)
   {
           this.catalogService = catalogService;
   }

   public void setTrainingProductService(final TrainingProductService trainingProductService)
   {
           this.trainingProductService = trainingProductService;
   }

   public ModelAndView handleRequest(final HttpServletRequest request,
           final HttpServletResponse response) throws Exception
   {
           catalogService.setSessionCatalogVersion("hwcatalog", "Online");

           final String code = request.getParameter("code");
           final String name = request.getParameter("name");
           ProductModel product = null;
           if (code != null && name != null)
           {
                   product = trainingProductService.getProductForCode(code, name);
           }
           final Map<String, Object> model = new HashMap<String, Object>();
           model.put("product", product);
           return new ModelAndView("trainingProduct.jsp", model);
   }
   ```

2. Configure the service dependencies for our controller. For this, you have to modify the Spring bean definition of our controller:

   `web/webroot/WEB-INF/springmvc-servlet.xml`

   ```
   <bean name="/product.html" class="org.training.web.controllers.TrainingProductController">
           <property name="productService" ref="productService"/>
           <property name="catalogService" ref="catalogService"/>
   </bean>
   ```

   You do not have to inject the new `trainingProductService` bean because of the redefinition of the alias.

3. Adjust the view to display product information:

   `web/webroot/trainingProduct.jsp`

   ```
   <html>
           <head>
   ```

```
                <title>Product</title>
            </head>
            <body>
                <h1>${product.name}</h1>
                ${product.description}
            </body>
        </html>
```

Changing the Logic of an Existing Service

You may need to slightly change the way some services work, for example modifying the logic of a method. Changing logic of an existing service works much in the same way as adding logic. But instead of adding a method, you override an existing one.

Adding Logic

Let's say you wish to change the logic of the `getProductForCode(String code)` method in a way that the query uses a fixed value for the name attribute. To do so, you simply extend the `DefaultProductService` and add your logic. Use the already created service but remove implementation of the custom interface:

`TrainingProductService`

```
public class TrainingProductService extends DefaultProductService
{
        @Override
        public ProductModel getProductForCode(String code)
        {
                getProductForCodeAndName(code,"AMD");
        }
}
```

Here you changed the logic of the method by simply using our new method defined in the previous chapter.

Adjusting Controller

Now, you need to modify the controller that you have modified in the section above. You only need to change the product lookup.

Change the code from

```
product = trainingProductService.getProductForCode(code, name);
```

to:

```
product = trainingProductService.getProductForCode(code);
```

Replacing an Existing Service

Sometimes it can be preferable to completely replace the implementation of an existing service. Possible scenarios are creating mock-ups or decorating the original service. The effect is that you can totally change the service's API by new methods, different method signatures, and so on.

Procedure

1. Change the created `TrainingProductServiceImpl` class so that it does not extend the `DefaultProductService`, but implements the `ProductService` interface.

   By extending from `AbstractBusinessService`, you automatically get the used `FlexibleSearchService` injected. For that you have to adjust the spring configuration to extend the definition of the `abstractBusinessService` bean.

   ```
   public class MyDefaultProductServiceImpl extends AbstractBusinessService implements ProductService
   {
           public ProductModel getProductForCode(final String code)
           {
                   ...
           }

           public ProductModel getProductForCodeAndName(final String code, String name)
           {
                   ...
           }
   }
   ```

2. Override the bean definition.

   In difference to this, you do not declare the original `defaultProductService` as parent:

```
<bean id="myProductService" class="de.hybris.sample.services.impl.MyProductServiceImpl"
      parent="abstractBusinessService" />
```

Adding a New Model

Follow the steps to learn how to generate ServiceLayer models.

Prerequisites

In this example, you need to prepare an extension in a certain way. For details, see Preparing an Extension for ServiceLayer Examples. If you have completed other procedures of the ServiceLayer documentation, you should already have this extension prepared.

Context

ServiceLayer models are generated based on SAP Commerce item types. Therefore, to generate a new model, first define a new SAP Commerce item type.

To illustrate how to add a new model to SAP Commerce, we define the new item type: `ContactRequest`.

Procedure

1. Define the `ContactRequest`.

```
<itemtype generate="true"
   code="ContactRequest"
   jaloclass="org.training.jalo.ContactRequest"
   extends="GenericItem"
   autocreate="true">
        <deployment table="UserRights" typecode="32767"/>
        <attributes>
                <attribute qualifier="message" type="java.lang.String">
                        <description>Message</description>
                        <modifiers initial="true"/>
                        <persistence type="property"/>
                </attribute>
                <attribute qualifier="sender" type="java.lang.String">
                        <description>Sender</description>
                        <modifiers initial="true"/>
                        <persistence type="property"/>
                </attribute>
        </attributes>
</itemtype>
```

   Types in SAP Commerce are defined in the extension's `items.xml` file. It is highly recommended that you specify a deployment for a `GenericItem` subtype, otherwise the instances of that subtype are stored in the same table as instances of `GenericItem`.

2. Generate the model.

   - If you use Eclipse, the generation process is automatically started when you save the `items.xml` file (the default Eclipse project is shipped with an Eclipse build launcher) and will refresh the `gensrc` directory of your extension as well as the `bootstrap/gensrc` directory of the **platform**.
   - If you do not use Eclipse, trigger the generation process manually by building SAP Commerce:
     - Open a command shell.
     - Navigate to the *<HYBRIS_BIN_DIR>* `/platform` directory.
     - Make sure that a compliant Apache Ant version is used:
       - On Windows systems, call the *<HYBRIS_BIN_DIR>* `/platform/setantenv.bat` file. Do not close the command shell after this call as the settings are transient and would get lost if the command shell is closed.
       - On Unix systems, call the *<HYBRIS_BIN_DIR>* `/platform/setantenv.sh` file, such as: `. ./setantenv.sh`
     - Call `ant clean all` to build SAP Commerce.

3. Find the new model in the `bootstrap/gensrc` folder.

   The build framework generates a model class for you, `org.training.model.ContactRequestModel`.

The New Model

See an example of a ServiceLayer model to learn about its constituents.

As models reflect the attributes of their type definition, the generated model has member variables for **sender** and **message** (No. 1 in the code sample below). These are used by corresponding getter and setter methods (No. 4) and can be used very well for debugging purposes. Furthermore, there are constants (No. 2) holding the used attribute qualifiers. These are only for internal use such as marking an attribute as modified and so on. For instantiation of a model, there are overloaded constructors (No. 3): One without any parameters for generic creation (but do not forget to set mandatory attributes after calling) and one with all mandatory attributes:

Generated ContactRequestModel class

```
public class ContactRequestModel extends ItemModel
{
// 1
        private String _message;
        private String _sender;

// 2
        private static final String MESSAGE = "message";
        private static final String SENDER = "sender";

// 3
        public ContactRequestModel()
        {
                super();
        }

        @Deprecated
        public ContactRequestModel(final ItemModel _owner)
        {
                super();
                setOwner(_owner);
        }

// 4
        public String getMessage()
        {
          if( !isAttributeLoaded(MESSAGE))
          {
                this._message = getAttributeProvider() == null ? null : (String) getAttributeProvider().getAttribute(MESSA
                getValueHistory().loadOriginalValue(MESSAGE, this._message);
          }
          throwLoadingError(MESSAGE);
          return this._message;
        }

        public String getSender()
        {
          if( !isAttributeLoaded(SENDER))
          {
                this._sender = getAttributeProvider() == null ? null : (String) getAttributeProvider().getAttribute(SENDER
                getValueHistory().loadOriginalValue(SENDER, this._sender);
          }
          throwLoadingError(SENDER);
          return this._sender;
        }

// 5
        public void setMessage(final String value)
        {
                this._message = value;
                markDirty(MESSAGE);
        }

        public void setSender(final String value)
        {
                this._sender = value;
                markDirty(SENDER);
        }
}
```

Updating the System

You can now use the model class in your code. There is no need for additional registration in the Spring configuration. The model class is automatically registered when you launch the SAP Commerce Server.

Nevertheless, to use the Model at runtime you need to do a system update because saving of instances requires access to the persistence layer.

Other ServiceLayer Procedures

- Displaying a User
- Displaying a Product
- Extending a Service
- Adding a New Service: How to define and implement a new, nonexisting service (note: requires Adding a New Model)

Key Services Overview

The ServiceLayer offers a variety of key services. These are built in to SAP Commerce and comprise System, Infrastructure, and Platform Services.

Overview

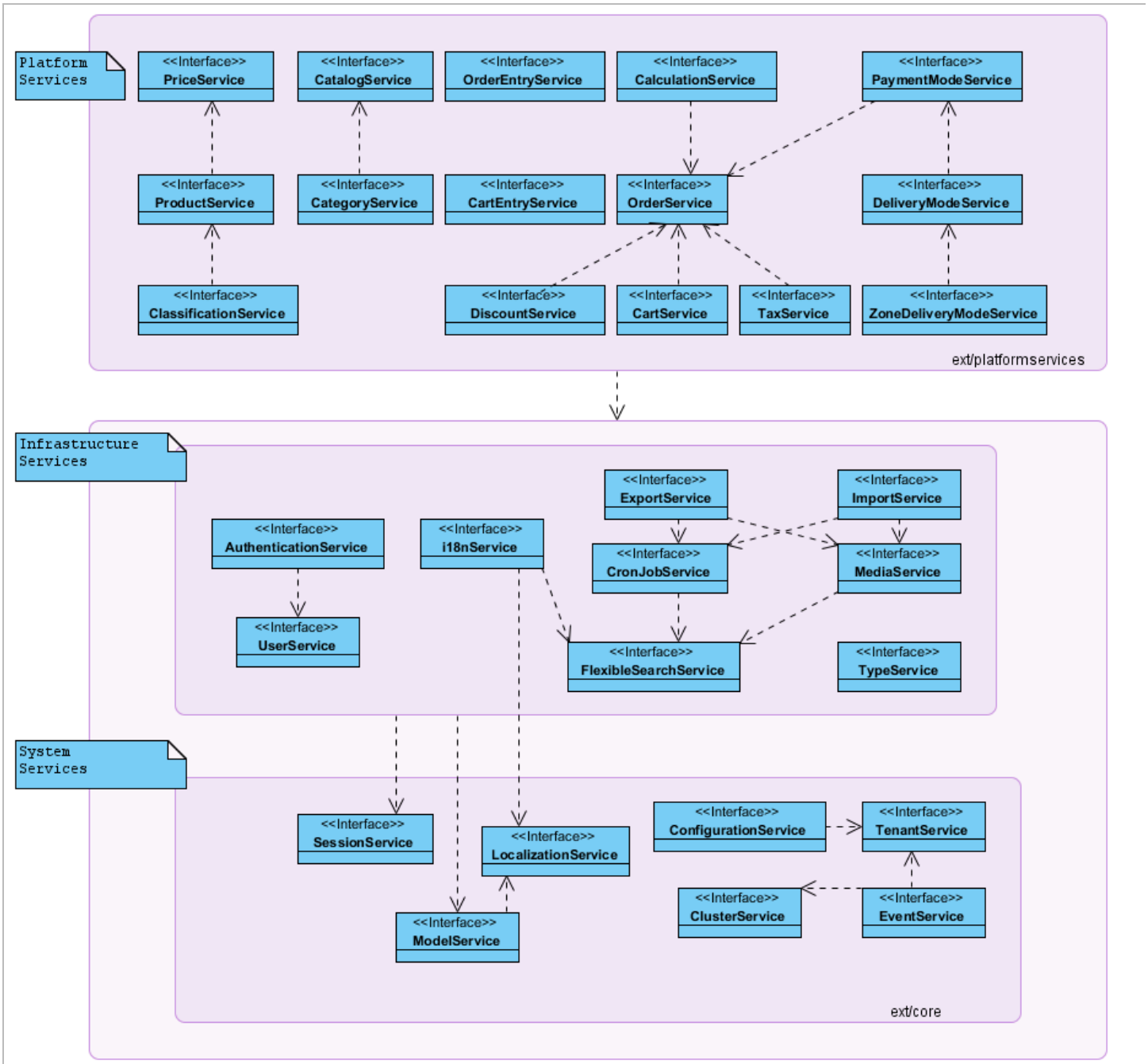


Figure: A list of available services and the relationships between them. A directed arrow means the target uses the source. For example, the **CategoryService** uses the **CatalogService**.

List of Services

System Services are essential services required for basic functionality. Infrastructure Services provide functionalities that are more advanced than System Services, but are still technical. These rely on System Services. Lastly, Platform Services offer business functionalities.

System Services	Infrastructure Services	Platform Services
SessionService	AuthenticationService	CatalogService
ModelService	UserService	KeywordService
LocalizationService	i18nService	CategoryService
ConfigurationService	ExportService	PriceService
TenantService	CronJobService	ProductService
EventService	FlexibleSearchService	UnitService
ClusterService	ImportService	VariantsService
	MediaService	ClassificationService
	TypeService	OrderService
		CartService
		CalculationService
		CartEntryService
		DeliveryModeService
		DiscountService
		OrderEntryService
		PaymentModeService
		TaxService
		ZoneDeliveryModeService

Related Information

platformservices Extension
ServiceLayer

Implementing Services

While you may be able to build your entire business application using SAP Commerce services alone, you will no doubt need to implement services yourself. Before you do this, there are certain princples and best practices you should make yourself familiar with.

Related Information

Naming Conventions for Services
Using Façades and DTOs - Best Practice
Implementation Principles for Services
ServiceLayer Security

Naming Conventions for Services

The pre-implemented services delivered with SAP Commerce use a regular naming convention. This naming convention enables you to easily find services that are related to a certain model, or belong to a certain extension.

Available Services

To gain an overview of the available service, you can have a look at the:

- the Key Services Overview DevNet-page
- Spring Configuration Files
- API doc: Service interfaces exposed by Spring are marked accordingly in the interface's JavaDoc:

de.hybris.platform.order.services

Interface CartService

All Known Implementing Classes:

DefaultCartService

```
public interface CartService
```

Spring Bean ID:

cartService

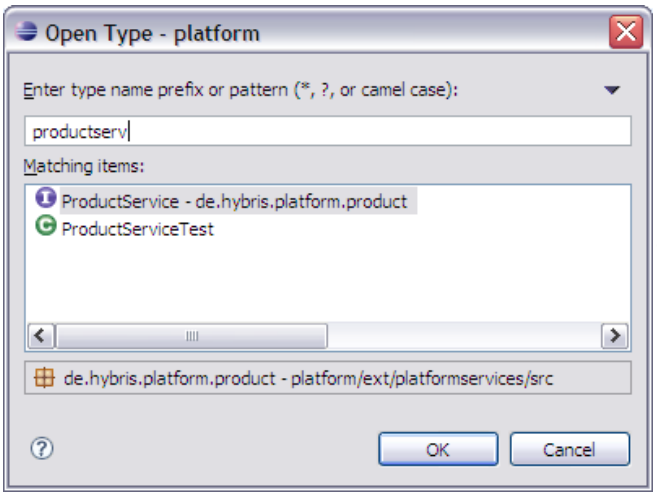
In addition, SAP Commerce also follows some simple conventions for finding services for models and for extensions.

For more information, see Spring Framework in SAP Commerce.

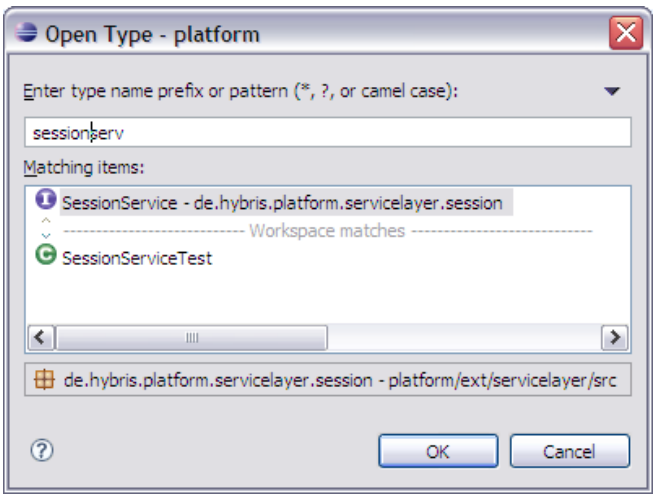
Finding a Service for a Model

If you have a class called `Xyz`, then your service to deal with this is called `XyzService`.

For example for `Product`, the service is called `ProductService`:



Or for `Session`, the service is called `SessionService`:



Finding a Service of an Extension

This table presents some examples of services, their location, and naming conventions.

Extension	Package Naming Convention	Interface	Implementation	Implementation Package Name
Core	de.hybris.platform.servicelayer.<service>	Session.java	DefaultSession.java	de.hybris.platform.servicelayer.session.impl
Platformservices	de.hybris.platform.<service>	CartService.java	DefaultCartService.java	de.hybris.platform.order.impl

Using Façades and DTOs - Best Practice

SAP Commerce offers recommended best practices for implementing facades and defining DTOs. Following these best practice models can reduce potential headaches for implementing your custom extensions.

A façade is a software design pattern that abstracts from an underlying implementation and offers an alternate, often reduced and less complex interface.

Data Transfer Objects (DTOs) are objects created to only contain values and have no business logic except for getter and setter methods. Using DTOs, you can "combine" SAP Commerce items - for example, this document adds price- and media-related data to a product object.

Define the DTO

Follow the steps to define a DTO.

Context

You want to display the following information of a product:

- Code
- Name
- Description
- Picture
- Price (currency and value)

Procedure

1. Create a class with name `org.training.data.ProductData`.

   It is a POJO with these instance variables:

   ```
   private String code;
   private String name;
   private String description;
   private String pictureUrl;
   private PriceData price;
   ```

2. Create getters and setters for those instance variables accordingly, such as:

   ```
   public void setCode(String code)
   {
      this.code = code;
   }

   public String getCode()
   {
      return code;
   }
   ```

3. Create a separate class `org.training.data.PriceData` for the price, with these instance variables:

   ```
   private String currency;
   private double value;
   ```

4. Create getters and setters again.

Using the Façade

Follow the steps to use the façade.

Context

To use the façade, you can use the SpringMVC controller created at the Product Tutorial.

Procedure

1. Change the controller from using the service to using the façade:

```
public class ProductController implements Controller
{

   private ProductFacade productFacade;

   public ModelAndView handleRequest(
      HttpServletRequest request, HttpServletResponse response) throws Exception
   {
      String code = request.getParameter("code");
      ProductData product = null;
      if (code != null)
      {
         product = productFacade.getProduct(code);
      }
      Map<String, Object> model = new HashMap<String, Object>();
      model.put("product", product);
      return new ModelAndView("product.jsp", model);
   }

   public void setProductFacade(final ProductFacade productFacade)
   {
      this.productFacade = productFacade;
   }

}
```

2. Alter the Spring bean definition of the controller to inject the façade:

```
<bean name="/product.html" class="org.training.web.controllers.ProductController">
   <property name="productFacade" ref="productFacade"/>
</bean>
```

You do not have to modify the view because the DTO's properties have the same name as the model's properties.

The Façade Interface

The ProductFacade interface provides a method that returns a Product DTO.

The façade interface looks like this:

```
package org.training.facades;

public interface ProductFacade
{
   public ProductData getProduct(String productCode);
}
```

For more information, see Define the DTO.

The Façade Implementation

Follow the steps to implement the ProductFacade interface.

Context

First, you have to find out the service we need. For details, see Naming Conventions for Services.

To get the actual product, we need an instance of `de.hybris.platform.product.ProductService` with the Spring ID `productService`. Because SAP Commerce keeps potential prices, `PriceInformation` items, separate from the products, you need an instance of `de.hybris.platform.product.services.PriceService` with the Spring ID `priceService`.

Procedure

1. Create the façade implementation with the instance variables for the services and their setters.

```
package org.training.facades.impl;

public class DefaultProductFacade implements ProductFacade
{
   private ProductService productService;
   private PriceService priceService;

   public ProductData getProduct(String productCode)
   {
      return null;
```

```
    }

    public void setProductService(ProductService productService)
    {
       this.productService = productService;
    }

    public void setPriceService(PriceService priceService)
    {
       this.priceService = priceService;
    }
}
```

2. Implement the `getProduct` method.

   Fetch product and price information and put everything in DTOs.

```
public ProductData getProduct(final String productCode)
   {
      ProductModel product = productService.getProductForCode(code);
      List<PriceInformation> prices = priceService.getPriceInformationsForProduct(product);
      ProductData productData = new ProductData();
      productData.setCode(product.getCode());
      productData.setDescription(product.getDescription());
      productData.setName(product.getName());
      if (product.getPicture() != null)
      {
         productData.setPictureUrl(product.getPicture().getUrl());
      }
      if (!prices.isEmpty())
      {
         PriceInformation price = prices.iterator().next();
         PriceData priceData = new PriceData();
         priceData.setCurrency(price.getPriceValue().getCurrencyIso());
         priceData.setValue(price.getPriceValue().getValue());
         productData.setPrice(priceData);
      }
      return productData;
   }
```

3. Define the façade as a Spring bean.

   Open the global application context file of your extension (`training-spring.xml`) and add this.

```
<bean id="productFacade"
   class="org.training.facades.impl.DefaultProductFacade"
   >

   <property name="productService" ref="productService"/>
   <property name="priceService" ref="priceService"/>
</bean>
```

Converters

Learn how to put the conversion logic into a separate class, the so-called converter.

Context

The majority of the logic of the `getProductCode()` method deals with conversion. At first, there is no problem with that but you may have other methods or classes that also have to convert `ProductModel` to `ProductData`. You could put the conversion logic in an extra method. But then there might be other facades that do the same conversion, too.

So, it is best to put the conversion logic in a separate class, a so-called converter. In fact, there is already a predefined interface, the `Converter`.

Procedure

1. Define two converters:

   a. One to convert `ProductModel` to `ProductData`.

```
public class ProductConverter implements Converter<ProductModel, ProductData>
{

   public ProductData convert(final ProductModel source) throws ConversionException
   {
      final ProductData data = new ProductData();
      return convert(source, data);
   }

   public ProductData convert(final ProductModel source, final ProductData prototype)
```

```
        {
            prototype.setCode(source.getCode());
            prototype.setDescription(source.getDescription());
            prototype.setName(source.getName());
            if (source.getPicture() != null)
            {
                prototype.setPictureUrl(source.getPicture().getUrl());
            }
            return prototype;
        }

    }
```

b. One to convert `PriceInformation` to `PriceData`.

```
    public class PriceInformationConverter implements Converter<PriceInformation, PriceData>
    {

        public PriceData convert(final PriceInformation source) throws ConversionException
        {
            final PriceData data = new PriceData();
            return convert(source, data);
        }

        public PriceData convert(final PriceInformation source, final PriceData prototype)
        {
            prototype.setCurrency(source.getPriceValue().getCurrencyIso());
            prototype.setValue(source.getPriceValue().getValue());
            return prototype;
        }
    }
```

2. Define the converters as Spring beans:

```
<bean id="productConverter" class="org.training.converters.ProductConverter"/>

<bean id="priceInformationConverter"
    class="org.training.converters.PriceInformationConverter"/>
```

3. Change the façade to use the converters:

```
public class DefaultProductFacade implements ProductFacade
{
    private ProductService productService;
    private PriceService priceService;
    private Converter<ProductModel, ProductData> productConverter;
    private Converter<PriceInformation, PriceData> priceInformationConverter;

    public ProductData getProduct(final String productCode)
    {
        final ProductModel product = productService.getProductForCode(code);
        final List<PriceInformation> prices = priceService.getPriceInformationsForProduct(product);
        final ProductData prodData = productConverter.convert(product);
        if (!prices.isEmpty())
        {
            final PriceInformation price = prices.iterator().next();
            final PriceData priceData = priceInformationConverter.convert(price);
            prodData.setPrice(priceData);
        }
        return prodData;
    }

    public void setProductService(final ProductService productService)
    {
        this.productService = productService;
    }

    public void setPriceService(final PriceService priceService)
    {
        this.priceService = priceService;
    }

    public void setProductConverter(final Converter<ProductModel, ProductData> converter)
    {
        this.productConverter = converter;
    }

    public void setPriceInformationConverter(
        final Converter<PriceInformation, PriceData> priceInformationConverter)
    {
        this.priceInformationConverter = priceInformationConverter;
    }
}
```

4. And finally, also alter the Spring bean definition of the `productFacade`

```
<bean id="productFacade"
    class="org.training.facades.impl.DefaultProductFacade"
    >
```

```
        <property name="productService" ref="productService"/>
        <property name="priceService" ref="priceService"/>
        <property name="productConverter" ref="productConverter"/>
        <property name="priceInformationConverter" ref="priceInformationConverter"/>
    </bean>
```

Implementation Principles for Services

Various aspects of the ServiceLayer architecture should be taken into account when implementing services. The more of these aspects you use, the more transparent and modular your service implementation becomes. However, you can implement any service to be as complex in architecture as your project requires and allows.

A minimum service implementation consists of:

- A service implementation for the business logic. Services must be implemented by you explicitly.
- Models for data management. Models are generated automatically by the SAP Commerce build framework. You do not have to create or implement models yourself.

A full-scale service implementation can contain all of the following architectural aspects:

- Model definition (required and automatically generated)
- Actual service implementation (required)
- DAOs (optional, but recommended)
- DTOs (optional, but recommended)
- Facades (optional, but recommended)
- Strategies (optional, but recommended)
- Converters (optional, but recommended)
- Interface definitions backing the service implementation (optional, but recommended)

Using Spring

The SAP Commerce ServiceLayer uses the traditional way to define Spring beans via XML configuration files but optionally there are other ways for bean definitions. This section sums up the alternatives and their pros and cons.

Annotations

Spring 2.5 and later versions allow defining Spring beans via Annotations.

Beans written by SAP Commerce do not define Spring beans using annotations, but use an explicit set of configuration files instead. That way, you can tell at once which beans are available for you to use and do not have to weed through all the Java classes.

Autowiring

Usually you define bean dependencies like this:

```
<property name="productService"     ref="productService"/>
<property name="priceService"       ref="priceService"/>
<property name="productConverter"   ref="productConverter"/>
```

But you can also use Spring autowiring capability. Autowiring is a way to automatically inject dependencies, for example by matching the name of the property with the name of the bean to be injected.

This feature can speed up development because you do not have to explicitly specify all of your dependencies. But at the same time it makes your components more fragile: it can be hard to track down injection errors (which may be caused by typos in a property of bean ID, for example).

If you define dependencies explicitly, the Spring container checks the dependencies on startup and throw an error if it can't resolve the dependencies. In addition, tools like Eclipse are able to notify you of errors during development.

Also, autowiring limits your flexibility. To have dependencies injected by name or type, you have to make sure that names and types always match and can't come up with other patterns to construct your beans. For services, you want to maintain maximum stability and flexibility. You may consider using autowiring for frontend components like controllers. There you should build on a solid foundation with a consistent bean naming scheme and can safely have the dependencies injected by name.

Beans written by SAP Commerce do not use autowiring to ensure that only explicitly referenced beans are injected.

Accessing Field Members

Field members of Abstract Services should be private (and therefore accessed with getters and setters).

```
public abstract class SomeAbstractService extends AbstractService{
  private ModelService modelService;  // Member fields should be private and accessed with getters, setters
```

Decorating a Service

Building a Decorator to an existing service can be a good idea if you want to establish a connection less dependent than extending the service. By creating a Decorator, you can virtually extend or override a service implementation without actually touching the original service. That way, you have fewer dependencies on the original service implementation and keep your code separated.

For more information, see How to Extend a Service.

An implementation could look like this:

```
public class MyDefaultProductServiceImpl extends AbstractBusinessService implements ProductService
{

        private ProductService originalProductService;

        public ProductModel getProductForCode(final String productCode) throws ProductNotFoundException
        {
                // Your code here
                return originalService.getProductForCode(productCode);
        }

        // Rest of the methods
}
```

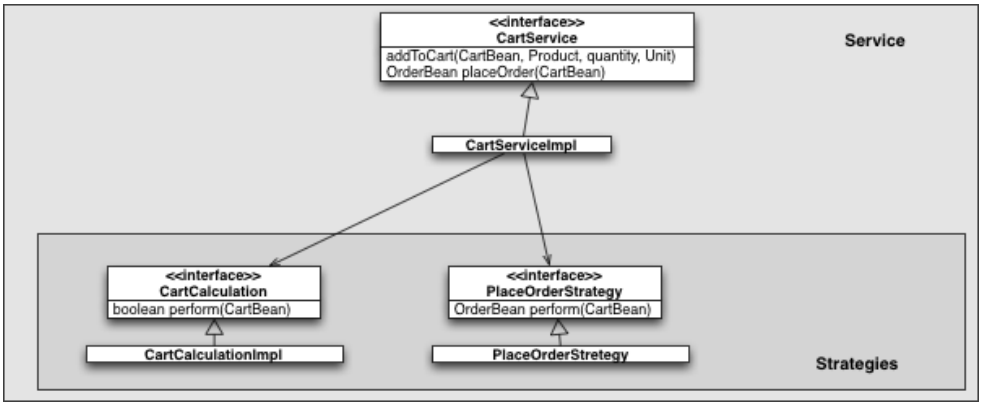
By delegating to the original service, you keep following the principle of Interface Driven Design. You do not have to make any assumptions on the details of the ServiceLayer implementation. While SAP Commerce guarantees the API to be stable, the underlying implementation may vary. The name of the implementation class or the way it works can change from release to release, which may result in breaking a dependent service implementation.

Strategies

One of the main goals of the ServiceLayer is to enable developers to easily extend existing functionality. While it would be possible to replace a whole service class implementation, this would not be a flexible approach. There are several drawbacks:

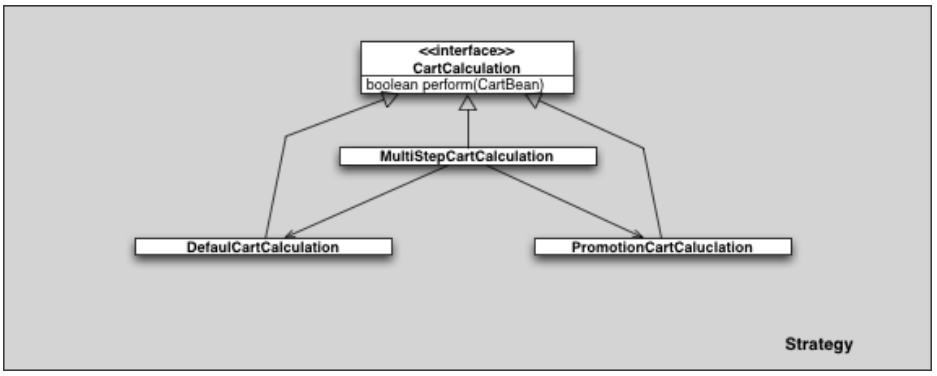
- You have to replace a whole class just to override one method.
- What do you do when several extensions want to modify the same logic?

A solution is to have the Business Services delegate to finer-grained components if necessary. This approach is known as Strategies. Such a Strategy can look like this:



These Strategies are unique to every use case. Each Strategy may use an individual interface (Interface-Driven-Design).

There can be numerous ways to adapt these Strategies. One way might be to aggregate several Strategies to a composite Strategy like this:



Implementing a Strategy

Strategies are optional. For simple use-cases or very project specific use-cases, simply implementing the services is sufficient.

For example, the `CartService`:

```
<alias alias="cartService" name="defaultCartService"/>
<bean id="defaultCartService"
                class="de.hybris.platform.order.services.impl.DefaultCartService"
                parent="abstractBusinessService"
                >
        <property name="addToCartStrategy" ref="addToCartStrategy"/>
        <property name="cartDao" ref="cartDao"/>
        <property name="orderCalculation" ref="orderCalculation"/>
        <property name="cartFactory" ref="cartFactory"/>
        <property name="sessionService" ref="sessionService"/>
</bean>
```

You see that this implementation depends on an `addToCartStrategy`, an `orderCalculation`, and a `cartFactory`. These strategies perform distinct tasks. So, perhaps it is a better idea to extend these instead of the service on top of them.

Using Facades and DTOs

There are several benefits to having facades and Data Transfer Objects (DTOs).

Decoupling

Using facades, you provide a stable interface to the frontend (which can be an MVC controller, web service client, or anything else). When the API of the ServiceLayer changes you can cope with these changes in one central location.

The same is true for DTOs. They are your own use-case-centric model. You can design them in the way you want, independently of the ServiceLayer data model - and even independently of the SAP Commerce data model.

So, with facades and DTOs you could even replace the underlying infrastructure, especially when writing mock-ups for tests.

Better Mocking and Testing

For each façade, you first define an interface. This interface you expose to the frontend. Now, you can change the underlying implementation at will.

You can easily provide a mock-up implemented in terms of stub methods. These methods don't call the ServiceLayer but only "fake" logic.

A `getProductForCode(...)` method could look like this:

```
public ProductData getProductForCode(final String productCode) throws ProductNotFoundException
{
        final ProductData data = new ProductData();
        data.setName("testName");
        data.setCode(productCode);
        return data;
}
```

This is a good way to parallelize development. Frontend developers can use the mock-up to develop their frontend. And façade developers can concentrate on implementing the actual use-cases without worrying to break frontend developing. Only later you would integrate the actual façade implementations.

This is also a good pattern for unit tests. When you only want to test frontend controllers without having to deal with potential façade bugs, you inject mock facades into your controllers before running the test.

Reusability Across Your Application

When you put your use case code in facades and not into controllers, there is another advantage. Frontend controllers are tied to the frontend technology. But what happens when you want to provide the same logic to web service clients or as batch process?

With facades, you can cleanly separate business logic from the frontend and reuse it wherever you want.

ServiceLayer Security

The ServiceLayer of SAP Commerce supports the Spring Security Framework. It includes a number of SAP Commerce-specific implementations you should also make yourself familiar with, in addition to the standard Spring security concept.

Spring Security Implementation in SAP Commerce

Spring Security Concept	SAP Commerce-Specific Implementations (API Doc)	Description
Spring `AuthenticationProvider` (Spring Documentation)	• `CoreAuthenticationProvider`	This is the central class that integrates the SAP Commerce user/group model with Spring Security. Spring Security uses an `AuthenticationProvider` to check if credentials are valid. The `AuthenticationProvider` then converts a SAP Commerce user object to an `Authentication` object.
Spring `AccessDecisionVoter` (Spring Documentation)	• `HybrisNotAnonymousVoter` • `HybrisNotInitializedVoter` (For SAP Commerce Administration Console only.)	Spring Security uses voters to determine if a user should be granted access or not. A voter can choose among voting for the access to be granted, denying, or abstaining voting. The SAP Commerce voter provides support for SAP Commerce specific rules, such as **HYBRIS_NOT_ANONYMOUS**.
Spring `PasswordCallback` (Spring Documentation)	• `HybrisPasswordCallback`	This is to protect web service calls for Apache CXF. See also http://cxf.apache.org/ .

Using Spring Security

To activate Spring Security mechanisms in an extension, edit the `${extension}-web-spring.xml` file, which is located in the `${extension}/web/webroot/WEB-INF` directory. Make two preparatory modifications to the file:

- Define the namespace itself:

  Add the line `xmlns:security="http://www.springframework.org/schema/security` to the namespace definition, for example:

  ```
  <beans xmlns="http://www.springframework.org/schema/beans"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:aop="http://www.springframework.org/schema/aop"
         xmlns:security="http://www.springframework.org/schema/security"
  ```

- Reference the schema location:

  Add the lines http://www.springframework.org/schema/security and http://www.springframework.org/schema/security/spring-security-3.0.xsd to the schema references, for example:

  ```
  xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop.xsd
     http://www.springframework.org/schema/security
     http://www.springframework.org/schema/security/spring-security.xsd">
  ```

Now you can configure Spring Security, for example:

`$extension-web-spring.xml`

```
<beans ...>

   <security:http>
        <session-fixation-protection="fixation" access-decision-manager-ref="accessDecisionManager">
```

```
        <security:intercept-url pattern="/checkout.action" access="HYBRIS_NOT_ANONYMOUS"/>
        <security:form-login
            login-page="/login.action"
            default-target-url="/browse.action"
            always-use-default-target="false"
            authentication-failure-url="/login.action?failed=true"/>
    </security:http>

    <security:global-method-security
          secured-annotations="enabled"
          access-decision-manager-ref="accessDecisionManager"/>

    <bean id="fixation" class="de.hybris.platform.servicelayer.security.spring.HybrisSessionFixationProtectionStrategy"/>
    <bean id="authenticationProvider" parent="hybrisAuthenticationProvider" >
          <security:custom-authentication-provider/>
    </bean>

    <bean id="accessDecisionManager" parent="hybrisAccessDecisionManager"  />

 </beans>
```

For learning about all aspects of Spring Security configuration, refer to the full Spring documentation at http://static.springsource.org/spring-security/site/docs/3.0.x/reference/ns-config.html .

The most basic variant is this:

```
<security:http>
        session-fixation-protection="fixation"
        access-decision-manager-ref="accessDecisionManager">
    ....
</security:http>
```

This is to protect web resources. It is important to turn on session fixation protection (see also http://static.springsource.org/spring-security/site/docs/3.0.x/reference/ns-config.html#ns-session-fixation) by using your own implementation of `SessionFixationProtectionStrategy`. Spring Security invalidates the current HTTP session after a user logs in. Spring Security then automatically opens a new HTTP session. This would result in the SAP Commerce session being closed if you use the standard implementation offered by the Spring framework.

Also you have to provide a reference to the `AccessDecisionManager` that is configured with the customized voters.

Within the basic web security tag, put:

```
<security:intercept-url pattern="/checkout.action" access="HYBRIS_NOT_ANONYMOUS"/>
```

This is to protect certain URLs. The URL can be a wild-carded pattern. You specify the access rules by means of strings that will later be passed to voters to check if access is granted or not. In this case, you can use a pre-implemented rule that access is granted for all non-anonymous users.

And to specify the login page, put:

```
<security:form-login
                login-page="/login.action"
                default-target-url="/browse.action"
                always-use-default-target="false"
                authentication-failure-url="/login.action?failed=true"/>
```

For further reference, see the Spring documentation at http://static.springsource.org/spring-security/site/docs/3.0.x/reference/ns-config.html#ns-form-and-basic . Nothing special for SAP Commerce is needed here.

If you want to protect method calls, you can either use AOP or annotations. In any case, it is important that you provide a reference to the `AccessDecisionManager`, like this:

```
<security:global-method-security
        secured-annotations="enabled"
        access-decision-manager-ref="accessDecisionManager"/>
```

Note that you also have to add the `AccessDecisionManager` and `AuthorizationProvider`:

```
<bean id="authenticationProvider" parent="hybrisAuthenticationProvider" >
        <security:custom-authentication-provider/>
</bean>

<bean id="accessDecisionManager" parent="hybrisAccessDecisionManager"  />
```

Both definitions inherit from abstract beans. Note that you have to explicitly register the `AuthenticationProvider`. Find details about registering in the Spring documentation at http://static.springsource.org/spring-security/site/docs/3.0.x/reference/appendix-namespace.html#d4e3397 .

Related Information

http://static.springframework.org/spring-security/site/index.html
http://www.springsource.org/
Spring Framework in SAP Commerce

Transitioning to the ServiceLayer

Previously, all persistence and business logic was written in the Jalo Layer. With the introduction of the Service Layer, all existing business logic in the Jalo Layer has moved to the Service Layer. As the Jalo Layer no longer contains business logic, the public API is significantly smaller.

The Jalo layer now consists primarily of the means to query flexible searches, and a generic way to save and remove data. This functionality is already provided in the Service Layer by adapter services like **FlexibleSearchService** and **ModelService**. In this case, any access to the Jalo Layer is no longer encouraged. The second goal is to eliminate all Jalo access in existing classes of the Service Layer.

Best Practices

This section describes how to identify and move the existing business logic to the ServiceLayer and how to replace the access to Jalo in the existing Service Layer code.

There is no simple and quick way to check if a custom extension is migrated to the Service Layer. To be sure that it is correctly and fully transited to use the Service Layer, it is necessary to review the whole extension code and configuration against each rule in this document, and apply all needed changes. As all APIs that get removed are marked as deprecated, a first step is to check your extension for deprecation warnings.

Manager Class

- Cockpit configuration in **createEssentialData** and **createProjectData**

  Instead of calling manual import for cockpit configuration, use the automated configuration import described in Importing Cockpit Configuration.

- ImpEx calls in **createEssentialData** and **createProjectData**

  Instead of calling manual import at **createEssentialData** and **createProjectData**, use the automatic import described in ImpEx Import for Essential and Project Data.

- Custom logic in **createEssentialData** and **createProjectData**

  Instead of putting logic for initialization or update process to **createEssentialData** and **createProjectData**, use a way allowing a proper injection of dependencies: Hooks for Initialization and Update Process.

- Create methods

  The method for creating new data instances in the ServiceLayer is different. You should use **modelService.create** to get a new Model instance. All attributes can be configured by proper setters before calling **modelService.save**. With that there is no need to use create methods, and create methods from a Manager is not replaced at all. For details see the Creating new instances of Business Objects section below.

Jalo Layer	Description	ServiceLayer
Manager.beforeItemCreation()	Allows you to hook into item creation from outside the actual Jalo class.	Write **PrepareInterceptor** instead.
Manager.checkBeforeItemRemoval()	Allows you to define checking of abort removal from an outside of the actual Jalo class.	Write **RemoveInterceptor** instead.
Manager.afterItemCreation()	Allows you to perform logic from outside of a Jalo class after a new item is created.	Listen for **AfterItemCreationEvent**.
Manager.notifyItemRemoval()	Allows you to intercept item removal from an outside the actual Jalo class.	Listen for **AfterItemRemovalEvent** instead.
Manager.notifyInitializationEnd()	Signals initialization is complete.	Listen for **AfterInitializationEndEvent** and **AfterInitializationStartEvent** instead.

→ Tip

AfterSaveEvent

You can use **AfterSaveEvent** to collect events created after each database operation and handle them according to your needs. For more details, read After Save Event.

Accessing Previous Values

The method **HMCManager.getSavedValue()** (see Item Attribute Modification History) has been removed, and has been replaced with **ServicelayerManager.getSavedValues()**.

Item Class

- Default value handling in **Item.createItem** should be adapted in **InitDefaultsInterceptor**.
- Logic in **Item.createItem** should be adapted in **PrepareInterceptor**.
- Logic in **Item.remove()** should be adapted in **RemoveInterceptor**.
- Consistency checks in **Item** setters should be adapted in **ValidationInterceptor**.
- Complex Jalo logic should be migrated to a service method, web layer should access this service instead.
- In **createItem** method there is an attribute mode **INITIAL** set on a given attribute, for example:

  ```
  allAttributes.setAttributeMode(QUALIFIER, AttributeMode.INITIAL);
  ```

  For the initial attribute mode logic in **Item.createItem** no proper solution is available as yet.

  For details on creating interceptors refer to Interceptors.
- **Item.checkRemovable()**: Allows to define checking of abort removal. Write **RemoveInterceptor** instead.

Job Class

For an example of how to rewrite a job implementation to use all benefits of the Service Layer, like dependency injection, see Migrating CronJobs.

Accessing Foreign Business Logic

In Jalo Layer the central point for business related logic of your extension was placed in a Manager class. It was accessible only by a static method, which does not allow you to mock or change the implementation of the dependency:

Jalo

```
ProductManager manager = ProductManager.getInstance();
```

In the ServiceLayer all business logic is covered by services configured using Spring. As a result, you can easily inject a dependency into your class and are free to mock it in your test cases:

ServiceLayer

```
@Autowired
private ProductService productService;
```

It is sometimes not easy to find an equivalence for Jalo methods in the ServiceLayer. Refer to Naming Conventions for Services to find some hints.

Type System Constants

The **`GeneratedXXXConstants`** files are deprecated. In general, the constants are replaced by appropriate constants part of the related Model classes. The following replacement of the constants contained in this files are available, grouped by inner class name:

- **Constants.TC** (Typecode)

  The **TC** class contains the constants for a code of each type defined in a related extension. The replacement can be found in the related Model with the fixed name **_TYPECODE**.

  Instead of

  Jalo

  ```
  CategoryConstants.TC.CATEGORY
  ```

  use

  ServiceLayer

  ```
  CategoryModel._TYPECODE
  ```

- **Constants.Attributes**

  The **Attributes** class contains for each attribute defined in a related extension a constant for the code of the attribute. The replacement can be found in the related Model class.

  Instead of

  Jalo

  ```
  CategoryConstants.Attributes.Media.SUPERCATEGORIES
  ```

  use

  ServiceLayer

  ```
  CategoryModel.SUPERCATEGORIES
  ```

- **Constants.Relations**

  The **Relations** class contains all codes to many-many relation types. The replacement can be found in the Model class of the relations source type Model.

  Instead of

  Jalo

  ```
  CategoryConstants.Relations.CATEGORYPRODUCTRELATION
  ```

  use

  ServiceLayer

  ```
  ProductModel._CATEGORYPRODUCTRELATION
  ```

- **Constants.EXTENSIONNAME**

  This constant holds the name of the related extension as defined in **extensioninfo.xml**. There is no direct replacement as there is no suitable place in an extension structure anymore. As these constants have a rare usage, it is recommended to copy them to the non-generated **`Constants`** file that gets created by the template mechanisms. The templates are already adjusted to put these constants once to the **Constants** class.

In general, check each FlexibleSearch to be refactored by using the new constants.

Instead of

Jalo

```
"SELECT {" + Product.PK + "} FROM {" + CoreConstants.TC.PRODUCT + "} WHERE {" + Product.CODE + "}=?code");
```

use

ServiceLayer

```
"SELECT {" + ProductModel.PK + "} FROM {" + ProductModel._TYPECODE + "} WHERE {" + ProductModel.CODE + "}=?code");
```

Constant values in non-generated constant Jalo classes can be used also after the migration to the Service Layer. Non-generated constants classes are not removed.

Session Handling

In Jalo the session handling is realized in **JaloSession** class, which can be accessed mostly only in a static way. Domain related logic, like for order, user, or language, can be also found here, which makes it really hard to be customized. The Service Layer introduced a central **SessionService** providing functionality for handling only sessions. All domain related logic is available in domain related services.

- Accessing general session attribute:

  Jalo

  ```
  JaloSession.getCurrentSession().setAttribute(..)
  ```

  ServiceLayer

  ```
  sessionService.setAttribute(..) or sessionService.getCurrentSession().setAttribute(..)
  ```

- Creating local session context:

  Jalo

  ```
  JaloSession.getCurrentSession().createLocalSessionContext();
  ```

  ServiceLayer

  ```
  sessionService.executeInLocalView(SessionExecutionBody);
  ```

- Getting the session user:

  Jalo

  ```
  JaloSession.getCurrentSession().setUser(..);
  ```

  ServiceLayer

  ```
  userService.setCurrentUser(..);
  ```

- Setting session catalog versions:

  Jalo

  ```
  CatalogManager.getInstance().setSessionCatalogVersions(..)
  ```

  ServiceLayer

  ```
  catalogVersionService.setSessionCatalogVersions(..)
  ```

- Setting session language:

  Jalo

  ```
  JaloSession.getCurrentSession().setLanguage(..)
  ```

  ServiceLayer

  ```
  i18nService.setCurrentLocale(..)
  ```

Jalo Layer	Description	ServiceLayer
JaloSessionListener.*	Allows you to listen to Jalo session events.	Listen for **AfterSessionCreationEvent**, **AfterSessionUserChangeEvent**, **BeforeSessionCloseEvent**
Extension.onFirstSessionCreation()	Allows you to hook into first session creation per tenant and virtual machine.	Not supported in the ServiceLayer.

There are no service layer equivalents for:

- Jalo Session deactivate - normally it is used by the **HybrisInitFilter** and the **SessionFilter,** which is one of the filters of the Platform filter chain.
- **JaloSession.getCurrentSession().getTenant()**. If it is used in a JSP page, the logic should be moved to Controller class, and **tenantService.getCurrentTenantId()** may be used to replace the logic.
- **JaloSession.getCurrentSession().getHttpSession()**.

For more information, see Platform Filters.

Transaction Handling

In the Jalo Layer, transaction handling was implemented using **Transaction** class where for example exceptions handling must be realized by yourself:

Jalo

```
Transaction.current().begin();
Transaction.current().commit();
```

Now you should use **TransactionTemplate** instead, which takes care of handling also exceptions and delegates internally to proper SAP Commerce classes:

ServiceLayer

```
transactionTemplate.execute(TransactionCallback)
```

A proper dependency injection is now possible. The **transactionTemplate** gets injected with a template definition like:

ServiceLayer

```
<bean id="myTransactionTemplate" class="org.springframework.transaction.support.TransactionTemplate" >
    <property name="transactionManager" ref="txManager"/>
</bean>
```

Configuration Values

To read configuration values that are stored in the **`project.properties`** or **`local.properties`** file, formerly the helper **Config** class was used.

Jalo

```
boolean sendMail = Config.getBoolean( "myextension.sendmail", false );
```

With the ServiceLayer, you access these configuration values using the **Configuration** object returned by the **ConfigurationService** (Spring bean id is **configurationService**).

ServiceLayer

```
Configuration config = configurationService.getConfiguration();
boolean sendMail = config.getBoolean( "myextension.sendmail", false );
```

Creating New Instances of Business Objects

There are different ways of creating new item instances using Jalo. The most common methods are:

- Using the corresponding Manager:

  Jalo

  ```
  ProductManager pm = ProductManager.getInstance();
  CatalogManager cm = CatalogManager.getInstance();

  CatalogVersion catalogVersion = ...;
  String code = "productCode";

  final Product p = pm.createProduct(code);
  cm.setCatalogVersion(p, catalogVersion);
  p.setDescription("some description");
  ```

- Using **ComposedType.newInstance(...)** method:

  Jalo

  ```
  ComposedType productType = TypeManager.getInstance().getComposedType(Product.class);
  CatalogVersion catalogVersion = null;
  Item.ItemAttributeMap values = new Item.ItemAttributeMap();
  values.put(Product.CODE, "productCode");
  values.put(CatalogConstants.Attributes.Product.CATALOGVERSION, catalogVersion);
  values.put(Product.DESCRIPTION, "some description");
  Product p = (Product) productType.newInstance(values);
  p.getCode();
  ```

  The data was directly persisted at each item related method call.

  Using the ServiceLayer, you create a new **ProductModel**, set the values as needed and then save the new **ProductModel** using **ModelService**. Here the data gets persisted only when **save** method is called.

  ServiceLayer

  ```
  CatalogVersionModel catalogVersion = ...;
  ProductModel p = new ProductModel(catalogVersion, "productCode");
  p.setDescription("some description");

  //model must explicitly be saved in order to persist it
  modelService.save(p);
  ```

  Using this method one must be aware that:

- No initial attribute values are set to Model when the new operator is used.
- **SaveAll** method does not save a Model that was created using **new** operator, since it is not attached to Mmodel context that is in transaction.
- To be able to use **SaveAll** method, you need to manually perform **modelService.attach(model)** method on a Model, in order to make it attached to Model context.

  There is another method, which is more preferred, using a factory method. Here is an example how to do it:

  ServiceLayer

```
CatalogVersionModel catalogVersion = ...;
ProductModel p = modelService.create(ProductModel.class)
p.setCatalogVersion(catalogVersion);
p.setCode("productCode")
p.setDescription("some description");

//model is saved using saveAll method, because thanks to this way of creating model, it is automatically attached to
modelService.saveAll();
```

Enumeration Handling

In Jalo Layer all enumeration handling was done by **EnumerationManager** and business logic was defined in **EnumerationValue** class:

Jalo

```
Product p = EnumerationValue value = p.getApprovalStatus();
EnumerationValue check = EnumerationManager.getEnumerationValue(
   "articleApprovalStatus","approved");
return value.getCode().equals(check.getCode());
```

In the ServiceLayer, you can directly use generated enum classes:

ServiceLayer

```
ProductModel p = return p.getApprovalStatus() == ArticleApprovalStatus.APPROVED;
```

Additionally you can use the Model classes for advanced operations like getting the **PK**:

ServiceLayer

```
PK pk = typeService.getEnumerationValue(ArticleApprovalStatus.class.getSimpleName(),
ArticleApprovalStatus.APPROVED.toString()).getPK();
```

Using the FlexibleSearch

In Jalo you used one of many search methods of **FlexibleSearch** class with many arguments and passing a map of search parameters:

Jalo

```
Map values = new HashMap();
values.put("code", code);
String query = "SELECT {" + Item.PK + "} FROM {" + getProductTypeCode() + "} WHERE {" + Product.CODE + "}= ?code";
SearchResult res = FlexibleSearch.getInstance().search(query, values, Collections.singletonList(Product.class),
false, // no, dont fail on unknown fields
false, // yes, need total
0, -1 // range
);
return res.getResult();
```

Now you can use the search method of **FlexibleSearchService**, besides two other convenient signatures, having a single DTO class as argument:

ServiceLayer

```
FlexibleSearchQuery query = new FlexibleSearchQuery("SELECT {" + ProductModel.PK + "}
FROM {" + ProductModel._ITEMTYPE + "} WHERE {" + ProductModel.CODE + "}= ?code");
query.addQueryParameter("code", code);
SearchResult<ProductModel> result = flexibleSearchService.search(query);
```

Internationalization And Localization

Session handling related to the internationalization was mainly provided by **JaloSession** class, completely based on Jalo item classes. Furthermore there were utility methods in **Utilities** class for getting preconfigured formats like **NumberFormat**. Now you may find all internationalization related functionality in:

- **I18NService**, based on Java library classes like **Locale**
- **CommonI18NService**, based on SAP Commerce Models like **Language**

Moreover, **FormatFactory** is available for getting preconfigured formats. See Internationalization and Localization for more information.

Localized text defined in locales files of type system localization was accessible by **de.hybris.platform.util.localization.Localization** utility class. This localization mechanism, as well as custom resource bundles, should be now accessed by **L10NService**.

Accessing ApplicationContext

When you write code based on the ServiceLayer you should never be in a situation that you want to directly access system **ApplicationContext**. You should always try to get your object instance managed by Spring framework so that all dependencies can be injected. There are situations in front-end frameworks where the current component instance is not managed by Spring, except of Spring MVC where it is always possible, like in cockpit framework. Here you should use Spring **Utils** class to access the **ApplicationContext**.

When migrating to the ServiceLayer, it may happen that you need to access the ServiceLayer from Jalo because of half-migrated logic, which is a bad cross-layer access. In this situation, you can access the **ApplicationContext** using **Registry.getApplicationContext()**.

Persistent Key Generation

Use the new **KeyGenerator** interface and its **PersistentKeyGenerator** implementation instead of **NumberSeriesManager**.

This allows users to customize the order generation process using values specified in the **`local.properties`** file. There is a number of available customization options. To use them, define the following set of properties in **`local.properties`**.

You can specify:

- How many digits are in the order code
- Start order code
- Whether the code is alphanumeric, numeric, or uuid
- Template to be used in order code generation

local.properties

```
keygen.order.code.name=order_code
keygen.order.code.digits=8
keygen.order.code.start=00000000
keygen.order.code.type=numeric
keygen.order.code.template=$
```

The bean with properties is now in **`core-spring.xml`**

```
<bean id="orderCodeGenerator" class="de.hybris.platform.servicelayer.keygenerator.impl.PersistentKeyGenerator">
<property name="key" value="${keygen.order.code.name}"/>
<property name="digits" value="${keygen.order.code.digits}"/>
<property name="start" value="${keygen.order.code.start}"/>
<property name="type" value="${keygen.order.code.type}"/>
<property name="template" value="${keygen.order.code.template}"/>
```

You can also use the symbol @, which stands for the system PK:

local.properties

```
keygen.order.code.template=ACC-@-$-DE
```

This generates order codes that contain a unique Platform number (return value of **MasterTenant.getClusterIslandPK()**), for example **123456**. The whole order code in this case looks like this:

```
ACC-123456-00000001-DE
```

Exception Handling

Do not extend or throw any Jalo exceptions, use **SystemException** or **BusinessException** from the ServiceLayer as base.

Writing Tests

One of the major advantages of using the ServiceLayer is an easy way of testing. For details about how to write tests based on the ServiceLayer refer to Testing with JUnit.

If you have existing tests based on Jalo, you need to migrate them, because the Jalo will not be available and the tests will not compile anymore. The migration rules are mostly the same as described in chapters above. You should replace usage of Manager classes by appropriate services where the services can be easily autowired to member variables.

Related Information

ImpEx Import for Essential and Project Data
Hooks for Initialization and Update Process
Importing Cockpit Configuration
Interceptors
Migrating CronJobs
Internationalization and Localization

After Save Event

After each database operation (whether caused by a committed transaction or not), an event is created. This event contains information about the item that was edited and the type of a database operation performed. You can collect these events and handle them according to your needs.

The event created after transaction or single database operation is a `AfterSaveEvent` POJO that holds the following information:

- PK of a modified item.
- Type of a database operation that can be CREATE, UPDATE, or REMOVE.

Within one transaction, an event is not created after each operation but after successfully committed transaction. If there is no transaction, then an event is created after every database operation.

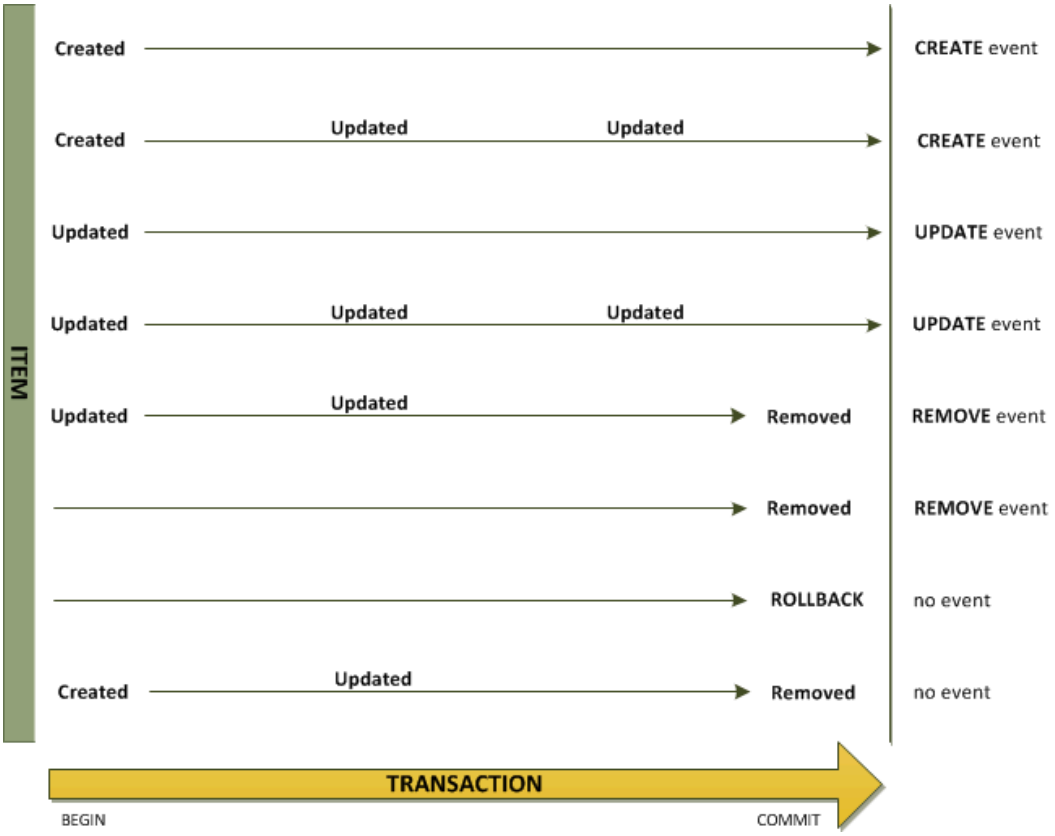


Figure: List of events that are created or not, depending on type of operations performed on the item during the transaction.

You can collect these events and handle them according to your needs. Implement your own listener using the `AfterSaveListener` interface. It should collect events of specific type and perform custom business logic.

Related Information

Transactions

Modes of Notifying Listeners About Events

A listener can be notified about newly created events in two modes: synchronous and asynchronous.

- Synchronous: In this mode event is notified to a listener immediately after it is created.
- Asynchronous: In this mode information about new event is put in a queue first. Then depending on the configuration it is also notified to a listener.

The asynchronous mode is enabled by default with some default configuration. Modes are set globally for all listeners.

Enable Synchronous Mode

To switch to synchronous mode put in your `local.properties` file the following property set to `false`.

`local.properties`

```
core.aftersave.async=false
```

Configure Asynchronous Mode

You can also configure the behavior of the asynchronous mode by, for example, setting the size of a queue for incoming events. The default configuration is set in the `advanced.properties` file. You can change it by copying specific property to your `local.properties` and setting your own value.

Creating Custom Listener

All listeners are notified with the same collection of events. Create a listener to collect and handle created events.

You can have as many listeners as you want. For example, you can create a Solr index listener that gets notified after a product is updated. It gets information on PK of the product that was updated. Then your listener can determine that a price of the product is reduced from 109 to 99. After collecting these data, your listener can reindex Solr and set that product in the category **0–100**.

Create Listener Definition

Your custom listener must implement the `AfterSaveListener` interface with its `afterSave` method. There is no default implementation available. It is up to you to decide how to deal with created events. Below example collects UPDATE events of all products:

`MyAfterSaveListener.java`

```
public class MyAfterSaveListener implements AfterSaveListener
{
    private ModelService modelService;

    @Override
    public void afterSave(final Collection<AfterSaveEvent> events)
        {
                for (final AfterSaveEvent event : events)
                {
                        final int type = event.getType();
                        if (AfterSaveEvent.UPDATE == type)
                        {
                                final PK pk = event.getPk();
                                //The product deployment code is "1"
                                if (1 == pk.getTypeCode())
                                {
                                        final ProductModel product = modelService.get(pk);
                                        //Put your business code here
                                }
                        }
                }
        }

    @Required
    public void setModelService(final ModelService modelService)
    {
        this.modelService = modelService;
    }
}
```

Declare a Bean for the Listener

After implementing the custom listener, declare a bean in the Spring configuration file like in the following example:

extension-spring.xml

```
<bean id="myAfterSaveListener" class="de.hybris.platform.MyAfterSaveListener" >
        <property name="modelService" ref="modelService" />
</bean>
```

Startup Tuning

SAP Commerce comes with strategies that you can use to optimize the Platform startup time.

SAP Commerce is a complex enterprise application designed to handle multiple use-cases. Due to this complexity, the Platform startup time may become a considerable factor in project development. Long startup times negatively impact developer productivity. They are also problematic in the cloud environment where nodes are expected to become operational quickly.

Platform Startup Overview

SAP Commerce consists of many web applications (for example Administration Console, Backoffice, and storefront) that are deployed to a pre-packaged Tomcat server (SAP Commerce Server). These web applications have a common application context that contains the Platform business logic.

`Registry` is the central class of this shared context and has to be started before web applications begin to operate. In a typical startup sequence:

1. The Tomcat server process is started via *hybrisserver.sh*.
2. Tomcat begins to load the deployed web applications.
3. `HybrisContextLoaderListeners` of the web applications are invoked.
4. The first web application starts `Registry` while others are blocked until this process is finished.
5. `Registry`:
    a. Reads all required configurations files.
    b. Loads the type system metadata from the database.
    c. Creates the Spring application context.
6. The web applications can proceed to initialize the Spring web context.
7. SAP Commerce is up and running.

Some of these operations can become time consuming.

Platform Startup Tuning

There are three main areas where it is possible to reduce the time needed for execution:

- changing the strategy for loading the type system information; for more information, see Type System Information Loading Strategies
- adjusting the Tomcat configuration; for more information, see Configuring TLD and Pluggability Scans and Enabling Multithreaded WebApp Loading
- configuring the rules to reduce the classpath traversal by class loaders; for more information, see Ignoring Nonexistent Classes on Classloader and Classpath Analysis Tool

Type System Information Loading Strategies

You can tune up the Platform startup time by using the batch strategy for loading the type system information. Disabling type migration is another option of tuning the startup time.

Loading Type System Information

While loading the type system information, SAP Commerce reads the information describing the type system from the database. There are two strategies to choose from:

- batch type system loading
- legacy type system loading

The batch loading strategy is a default and improved implementation that is faster and doesn't use the low-level EJB persistence layer that reduced performance. It is enabled by default:

```
# Enable batch typesystem loading strategy
typeinfomap.loading.batch.mode=true
```

To revert to the legacy approach, set `typeinfomap.loading.batch.mode` to `false`:

```
# Enable legacy typesystem loading strategy
typeinfomap.loading.batch.mode=false
```

The following table compares both the batch and the legacy strategies:

Batch mode	Legacy mode
Direct SQL	EJB Layer
Few db queries, fast	Multiple db queries, slow
Default	Nondefault

Disabling Type Migration

The SAP Commerce Registry can perform migration of core types during each Platform start. This operation is no longer needed and is disabled by default because it has an impact on the startup time.

If you want to enable migration due to a legacy reason, set the `should.migrate.core.types` parameter to `true`:

```
# enable core type migration
should.migrate.core.types=true
```

Configuring TLD and Pluggability Scans

SAP Commerce Platform allows you to change and adjust the TLD and pluggable feature scanning behaviors. You can do it by switching off the default behaviors and scanning only your allowlisted jars.

During the server startup, there is much jar scanning, especially scanning for Tag Library Descriptor files (TLD), and for pluggable features. Hundreds of jars may be scanned, for example TLD files, even if there is nothing TLD-related inside. You can change the default TLD and pluggability scan behavior. To let every extension specify which jars to scan, we added the `JarScanner` element into the generated Tomcat context.

The default for every web application looks as following:

```
<Context path="/test" docBase="/Users/i303816/Desktop/workspaces/git/platform_united/bin/platform/ext/testweb/web/webroot"
   <Manager pathname="" />
   <Loader platformHome="/Users/i303816/Desktop/workspaces/git/platform_united/bin/platform" className="de.hybris.tomcat.H
   <JarScanner >
      <JarScanFilter defaultTldScan="true" defaultpluggabilityScan="true" />
   </JarScanner>
</Context>
```

It is the default jar scanning mechanism for Tomcat that you can change. For more information about the Jar Scanner, see https://tomcat.apache.org/tomcat-8.0-doc/config/jar-scan-filter.html .

Changing the TLD Scanning Behavior

If you want to change the default TLD scanning behavior perform these two steps for every web extension:

- disable the default scanning mechanism
- specify your allowlist so that only the listed jars are scanned

To disable the default TLD scan, set the `<extName>.tomcat.tld.default.scan.enabled` property to `false`, for example:

```
hac.tomcat.tld.default.scan.enabled=false
```

Setting the property to `false` makes Tomcat scan only those jars that you specified on your allowlist.

To configure a allowlist for a given extension, use the `<extName>.tomcat.tld.scan` property, with the jars you want to scan as the property value, for example:

```
hac.tomcat.tld.scan=spring*.jar
```

The generated Tomcat context element for the `hac` extension looks as follows:

```
<!-- 'hac' extension's context for tenant 'master' -->
<Context path="" docBase="/Users/i303816/Desktop/workspaces/git/platform_united/bin/platform/ext/hac/web/webroot" >
   <Manager pathname="" />
   <Loader platformHome="/Users/i303816/Desktop/workspaces/git/platform_united/bin/platform" className="de.hybris.tomcat.H
   <JarScanner >
      <JarScanFilter defaultTldScan="false" defaultpluggabilityScan="false" tldScan="${tomcat.util.scan.StandardJarScanFil
   </JarScanner>
</Context>
```

Note that you don't override the standard jars to be scanned set by Tomcat in `tomcat.util.scan.StandardJarScanFilter`. If you want to force it, do it in `catalina.properties` directly.

If for some reason you want to switch to the previous scanning behavior, set the `<extName>.tomcat.tld.scan.enabled` property to `true`:

```
<extName>.tomcat.tld.default.scan.enabled=true  //or leave it empty, 'true' is default
```

Changing the Pluggability Scanning Behavior

Scanning for pluggable features was introduced with Servlet 3.0. To change the scanning behavior, use the same rules as for the TLD scanning.

To change the default behavior, perform these two steps for every web extension:

- disable the default scanning mechanism
- specify your allowlist so that only the listed jars are scanned

To disable the default pluggability scan, set the `<extName>.tomcat.pluggability.default.scan.enabled` property to `false`, for example:

```
hac.tomcat.pluggability.default.scan.enabled=false
```

Setting the property to `false` makes Tomcat scan only those jars that you specified on your allowlist.

To configure a allowlist for a given extension, use the `<extName>.tomcat.pluggability.scan` property, with the jars you want to scan as the property value, for example:

```
hac.tomcat.pluggability.scan=spring*.jar,libwrapper*
```

The generated Tomcat context element for the `hac` extension looks as follows:

```
<!-- 'hac' extension's context for tenant 'master' -->
<Context path="" docBase="/Users/i303816/Desktop/workspaces/git/platform_united/bin/platform/ext/hac/web/webroot" >
   <Manager pathname="" />
   <Loader platformHome="/Users/i303816/Desktop/workspaces/git/platform_united/bin/platform" className="de.hybris.tomcat.H
   <JarScanner >
      <JarScanFilter defaultTldScan="false" defaultpluggabilityScan="false" tldScan="${tomcat.util.scan.StandardJarScanFil
   </JarScanner>
</Context>
```

As is the case with the TLD scan - you don't override the standard jars to be scanned set by Tomcat in `tomcat.util.scan.StandardJarScanFilter`. If you want to force it, do it in `catalina.properties` directly.

If for some reason you want to switch to the previous scanning behavior, set the `<extName>.tomcat.pluggability.scan.enabled` property to `true`:

```
<extName>.tomcat.pluggability.default.scan.enabled=true //or leave it empty, 'true' is default
```

Performance Gain

The performance gain strongly depends on the number of webapps you configure in your setup. With limiting the TLD scanning for the internal Platform extensions only, there isn't much gain, about 1-2 seconds.

It looks much better on a b2c scenario containing additional 29 webapps - we gained about 25% on the server startup (300 vs 400 seconds). The results were repeatable.

The scanning behavior configurations were as follows:

```
productcockpit.tomcat.tld.scan=jstl-impl*.jar
productcockpit.tomcat.tld.default.scan.enabled=false

productcockpit.tomcat.pluggability.scan=jstl-impl*.jar
productcockpit.tomcat.pluggability.default.scan.enabled=false

//similar cofiguration for the other webapps
```

Enabling Multithreaded WebApp Loading

You can adjust the Tomcat configuration by using the multithreaded webapp loading. This can help you reduce the Platform startup time.

The multithreaded webapp loading is enabled by default by the `tomcat.startStopThreads` parameter. This parameter sets:

- the number of threads this engine uses to start child Host elements in parallel
- the number of threads that the Host uses to start the child Context elements in parallel; The same thread pool is used to deploy new Contexts if automatic deployment is used.

The special value of `0` (used as default) results in the value of `Runtime.getRuntime().availableProcessors()` being used. Negative values result in `Runtime.getRuntime().availableProcessors()` + `value` being used unless this is less than 1 in which case 1 thread will be used. For more information, see https://tomcat.apache.org/tomcat-8.0-doc/config/host.html .

Ignoring Nonexistent Classes on Classloader

We added a general way for a class loader to ignore loading nonexistent classes.

During an application startup, the Spring framework attempts to load a class with the `BeanInfo` suffix for each Bean (`java.beans.Introspector`). In most application cases, the `BeanInfo` classes don't exist. In addition, the internal mechanism in `java.beans.Instrospector` always tries to load a class with the `Customizer` suffix in the `findCustomizerClass` method. Since there is no control over the `Introspector`, we added a general way in our class loader to ignore the loading of classes that don't exist.

General Information

When `loadClass` from `URLClassLoader` is called, you can skip executing this method for some classes by enabling one or more of the following rules:

- `de.hybris.bootstrap.loader.rule.internal.CacheClassNotFoundRule`: Ignores loading the class when `ClassNotFoundException` is thrown for the first time for this class
- `de.hybris.bootstrap.loader.rule.internal.IgnoreIntrospectorFilterClassRule`: Ignores loading all of the classes with the `Customizer` and `BeanInfo` suffixes, searched from `java.beans.Introspector`
- `de.hybris.bootstrap.loader.rule.internal.ClassWhiteListRule`: includes classes indicated in the rule parameters:
  - class scope - by indicating a name of a particular class to be included
  - package scope - by configuring to include all classes from a package by indicating the package name with the * char at the end. For example, `de.hybris.customizers.*` .

    You can specify a comma-separated list of combination class scope and package scope in one configuration.

i Note

Rules are executed in a configured order. When some rules ignore or include a class, then the execution of the remaining rules is skipped for this class.

Configuration

To configure rules, add the following properties to the `env.properties` file:

```
YURLCLASSLOADER.IGNORE.[NUMBER_OF_RULE_CLASS].CLASS=fully qualified name of rule class
YURLCLASSLOADER.IGNORE.[NUMBER_OF_RULE_PARAMS].PARAMS=parameters for rule if required
```

i Note

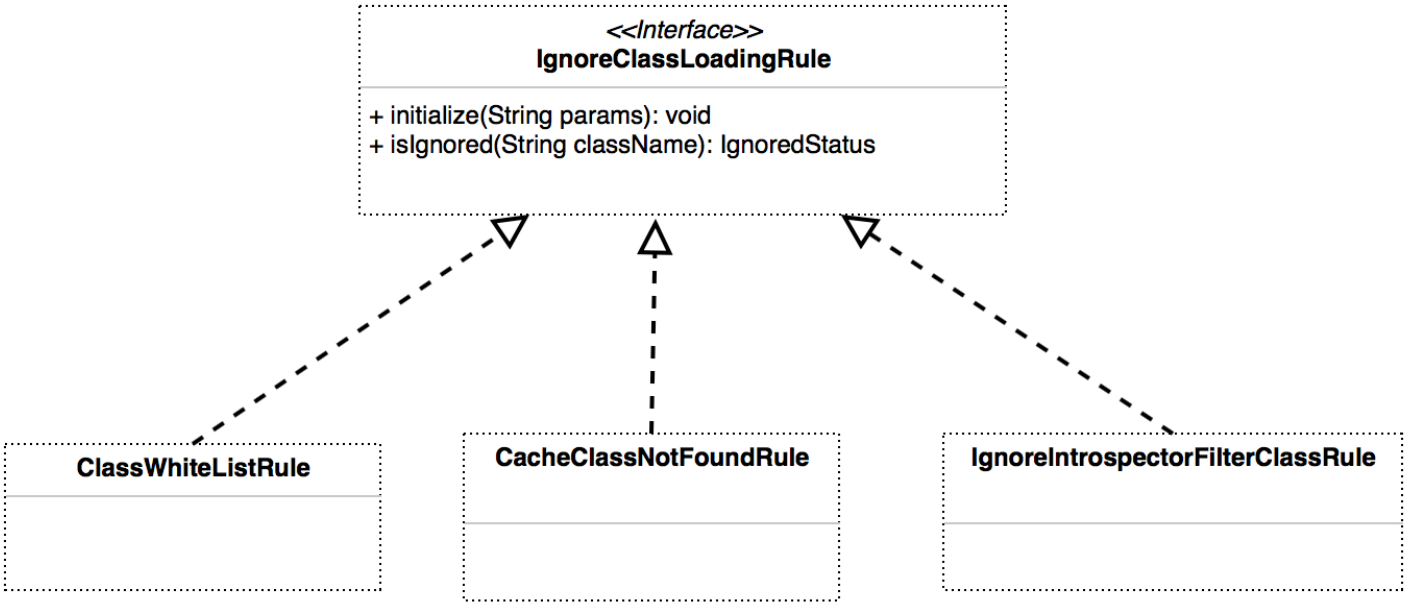
`NUMBER_OF_RULE_CLASS` must match `NUMBER_OF_RULE_PARAMS` for a configured rule. The numbers determine an ascending rule execution order.

Configuration example:

```
#YURLCLASSLOADER.IGNORE.RULE.0.CLASS=de.hybris.bootstrap.loader.rule.internal.ClassWhiteListRule
#YURLCLASSLOADER.IGNORE.RULE.0.PARAMS=de.hybirs.customizers.*,de.hybris.SimpleCustomizer
YURLCLASSLOADER.IGNORE.RULE.1.CLASS=de.hybris.bootstrap.loader.rule.internal.CacheClassNotFoundRule
YURLCLASSLOADER.IGNORE.RULE.2.CLASS=de.hybris.bootstrap.loader.rule.internal.IgnoreIntrospectorFilterClassRule
```

Customization

You can create your own rule by implementing the `IgnoreClassLoadingRule` interface:



The implemented `isIgnored(String className)` method should return one of the following statuses:

- `IGNORED`: `ClassNotFoundException` is thrown and the remaining part of the `loadClass` method is skipped
- `WHITELISTED`: The remaining rules are skipped and the `loadClass` method from Classloader is executed
- `UNDEFINED`: The algorithm moves to the next rule

Classpath Analysis Tool

Platform provides a classpath analysis tool for developers to inspect `ClassLoader` behavior and get metrics related to classes and resources. Developers can use it to evaluate what impact the `ClassLoader` rules have on `ClassLoader` performance.

Metrics

To enable the `ClassLoader` metrics, set the Java `hybris.log.classloading` property to `true`. You can do it by adding the property to `tomcat.javaoptions` or `tomcat.debugjavaoptions`:

```
tomcat.javaoptions=-Dhybris.log.classloading=true
tomcat.debugjavaoptions=-Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,address=8000,suspend=y -Dhybris.log.class
```

Run *ant server* to update the Tomcat configuration files.

When the classpath analysis is enabled, `PlatformInPlaceClassLoader` records attempts to load classes and resources from a classpath. It records:

- successful attempts to find and load class on the classpath (once since loaded classes are kept)
- unsuccessful attempts to find class on the classpath (it can happen many times because classes that weren't found are not cached)
- classes ignored by rules

You can query metrics by calling one of the `ClassLoaderMetricRegistry` methods:

- `findMetrics`: returns a list of classes/resources and the number of recorded classloader events; the query can be parametrized with a criteria object to select required subset of data
- `findLoadedByClasspathLocation`: returns a summary of the number of classes/resources loaded from each location

Sample Usage

The classpath analysis tool is a low-level tool aimed at developers and can be accessed programmatically, for example via the Scripting Console. We provide sample scripts below that demonstrate the usage of the API.

Number of classes/resources loaded from each jar/location:

```
import de.hybris.bootstrap.loader.PlatformInPlaceClassLoader;

classpathSummary = PlatformInPlaceClassLoader.getClassloaderMetrics().findLoadedByClasspathLocation()
for (entry in classpathSummary)
{
    println entry.location + ' - ' + entry.usage
}
```

Classes / Resources not found on classpath:

```
import de.hybris.bootstrap.loader.PlatformInPlaceClassLoader;
import de.hybris.bootstrap.loader.metrics.MetricQueryCriteria;
import de.hybris.bootstrap.loader.metrics.EventType;

entries = PlatformInPlaceClassLoader.getClassloaderMetrics().findMetrics(MetricQueryCriteria.query().forEventType(EventTyp

for (entry in entries)
{
    println entry.name + ' ' + entry.eventTypeCount
}
```

Classes / Resources ignored by rules:

```
import de.hybris.bootstrap.loader.PlatformInPlaceClassLoader;
import de.hybris.bootstrap.loader.metrics.MetricQueryCriteria;
import de.hybris.bootstrap.loader.metrics.EventType;

entries = PlatformInPlaceClassLoader.getClassloaderMetrics().findMetrics(MetricQueryCriteria.query().forEventType(EventTyp

for (entry in entries)
{
    println entry.name + ' ' + entry.eventTypeCount
}
```

Other Options

There are other options that you may find significant when tuning the Platform startup.

Disabling the JUNIT Tenant

For faster startup in a production environment, we recommend disabling the junit tenant completely. Set the `installed.tenants` flag to an empty value to have no subordinate tenants. Optionally, you could only add custom partner tenants if there is a need for that.

In a development environment, you may leave the default configuration as it is - `installed.tenants=junit,foo,t1,t2`. It allows you to execute all SAP Commerce tests, also from the SAP Commerce Administration Console test servlet. However, you can always set it up using the configuration recommended for a production environment if you don't need the junit tenant.

For details, see section Creating New Tenants in Multitenancy.

Migrating Core Types

If you want to migrate core types during system startup, set the `should.migrate.core.types` to `true`. This property is by default set to false, `should.migrate.core.types=false`

Suspending and Resuming SAP Commerce

Platform allows you to suspend a running SAP Commerce system, and resume it.

You can use this function to disconnect the database from Platform for maintenance. After you reconnect the database, you can resume SAP Commerce. You can also suspend SAP Commerce to eventually shut it down gracefully.

To suspend the system, collect all operations running in the system, register, and classify each of them in one of the two categories:

- non-suspendable operations, which have to finish before the system can get suspended
- suspendable operations, which can live or must live while the system is being suspended

By default, all tasks, cronjobs, as well as http threads are non-suspendable. For example, if a cronjob starts before you manage to send a request to suspend the system, the system cannot switch to the suspended state until the cron jobs finishes. For all other types of operations, you have to decide how you want to classify them, and register them as such by providing respective information.

After you send a request to suspend the system and there are still some non-suspendable operations running, the system switches to a special **waiting state** in which the system waits for the operations to finish. In the waiting state, non-suspendable operations can borrow the connections to the database on condition the operations were started before you managed to send the request to suspend the system. When the system is in the **suspended state**, all database connections are blocked.

Once you send a request to suspend the system, new tasks and cronjobs won't start until the system is resumed, and up and running again.

While the system is waiting to be suspended, or is already suspended, all http requests are blocked. There is only one exception – requests related to the very suspend operation and the resume operation are allowed. For reference, see Methods for Suspending and Resuming SAP Commerce.

To suspend SAP Commerce, collect all operations running in the system, register, and classify each of them as either non-suspendable or suspendable operations.

Methods for Suspending and Resuming SAP Commerce
You can resume and suspend SAP Commerce from Administration Console, from a REST interface, or by using a command line script.

Methods for Suspending and Resuming SAP Commerce

You can resume and suspend SAP Commerce from Administration Console, from a REST interface, or by using a command line script.

Suspending and Resuming from Administration Console

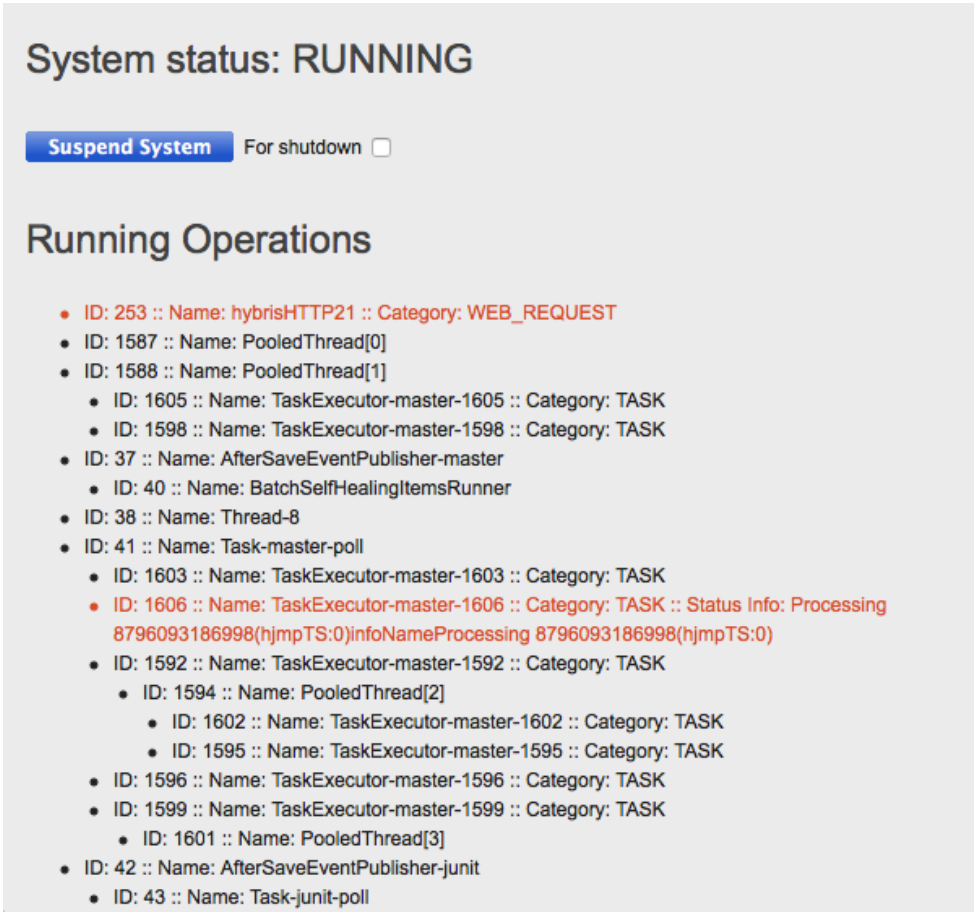
⚠ Caution

To be able to run the procedures, log into Administration Console as the **administrator**, or use the `hac_monitoring_suspendresume` role.

To suspend a running system, log into Administration Console and navigate to **Monitoring** > **Suspend** page.

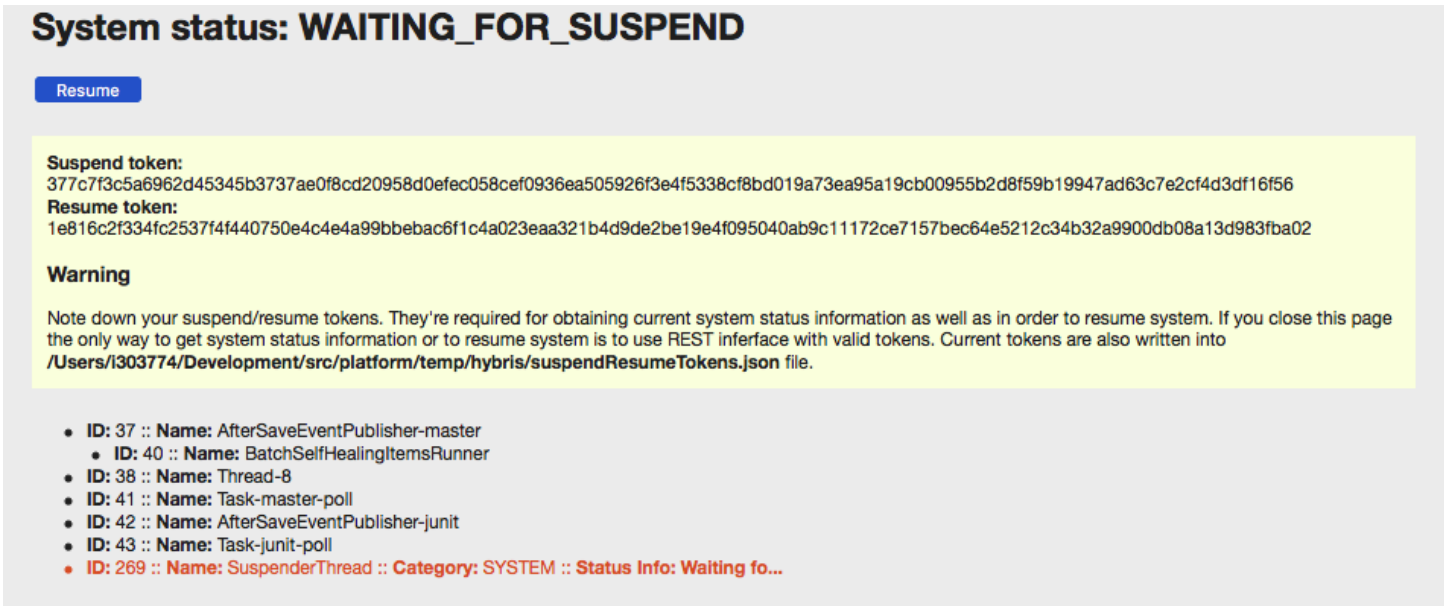
The **Suspend** page shows all running operations such as tasks, cronjobs, and any other custom operations that may be working in the background. The operations are listed in a tree view. They are categorized and marked in colors. Operations listed in red are not suspendable.

Next to the **Suspend System** button, you can find the **For shutdown** checkbox. By selecting it, you instruct the system to **gracefully shut down**, which is possible only after all running background operations, such as cronjobs, are completed.



Suspending

Click the Suspend System button to suspend the system. As a result, a request is sent to suspend the system. You are also redirected to a page informing you that the system is in the WAITING_FOR_SUSPEND state. At this point the system is waiting for the background operations to finish.



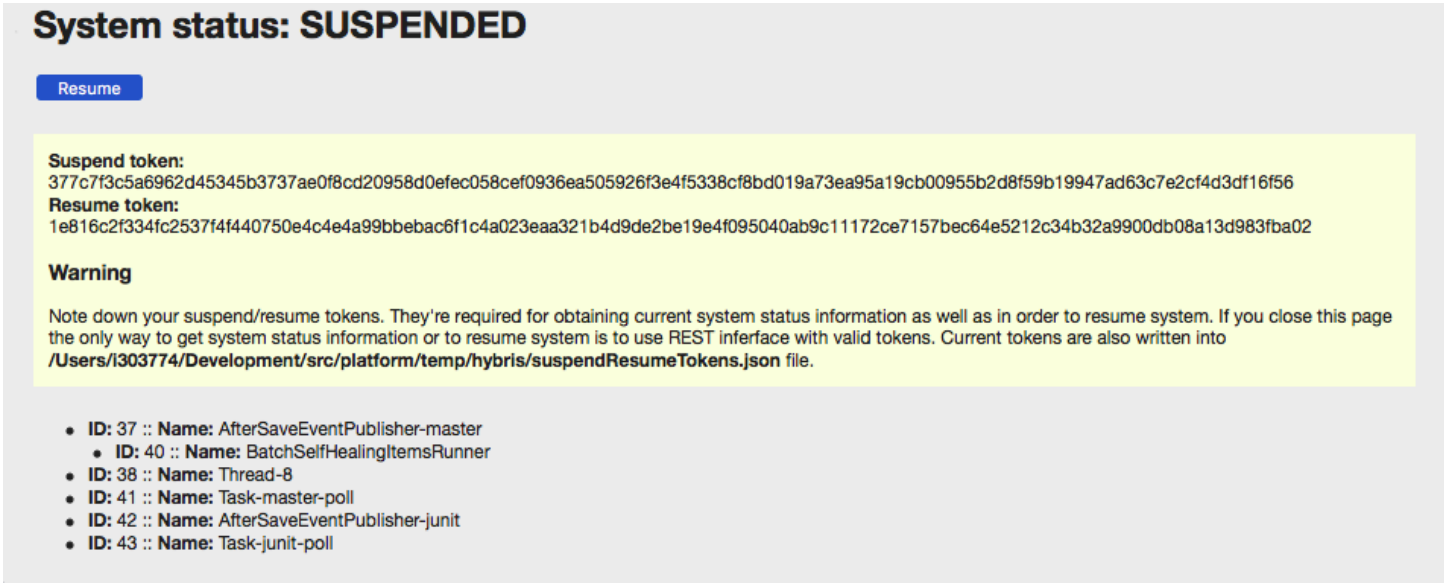
In the yellow background on the page, you can see the suspend and resume tokens, and a warning.

⚠ Caution

If you lose the tokens, for example as a result of accidentally closing the tab, or your browser crashing, you **will not** be able to resume the system. We recommend that you record them to prevent such situations.

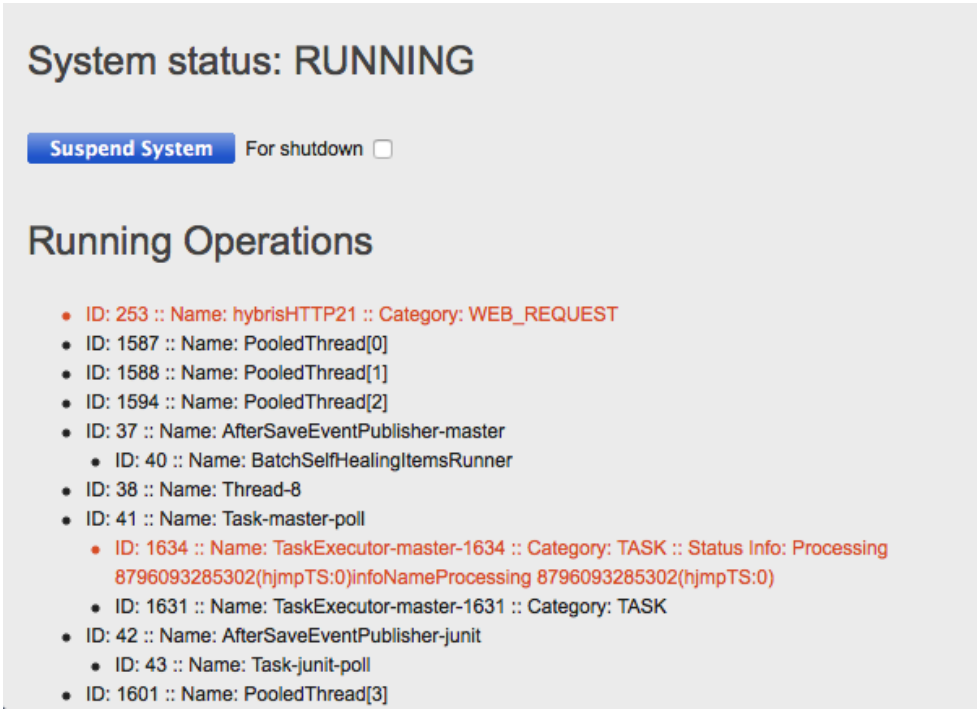
The tokens, however, are stored in the `suspendResumeTokens.json` file located in *<HYBRIS_HOME_DIR>*`/platform/temp/hybris` directory. You can get them from there as the final option.

After all operations have been finished, the system can be set to the SUSPENDED state.



Resuming

To resume a suspended system, click the Resume button. As a result the system should be up and operational again.



Suspending and Resuming from a REST Interface

You can suspend and resume SAP Commerce from any REST client, or by using a console program called curl.

Get familiar with the available REST points required to perform the Suspend and Resume actions.

Description	URL	Method	Auth	Json Body	Json Respon
Returns `suspendToken` for single usage	`/monitoring/suspendresume/suspendtoken`	`GET`	Basic		`{ "suspendT }`
Performs the Suspend action	`/monitoring/suspendresume/suspend`	`POST`	Basic	`{ "suspendToken": "suspendTokenString", "forShutdown": "false|true" }`	`{ "resu "susp "resu "erro`

Description	URL	Method	Auth	Json Body	Json Respon
					"syst "forS "runn }

Resumes the system	/monitoring/suspendresume/resume	POST	Basic	{
				"resumeToken": "resumeToken"
				}
					{
					"syst
					"runn
					}

Shows the current system status after Suspend has been requested	/monitoring/suspendresume/status	POST	NONE - Only valid resumeToken	{
				"resumeToken": "resumeToken"
				}
					{
					"resu
					"susp
					"resu
					"erro
					"syst
					"forS
					"runn
					}

Shows the current system status before Suspend is requested	/monitoring/suspendresume/status	POST	Basic		{
					"syst
					"runn
					}

Using curl to Suspend and Resume

Follow the procedure to learn how to suspend and resume SAP Commerce using curl.

Context

The procedure consists of two main steps. The first step explains how to suspend the system. The second step explains how to resume it.

Adapt each of the provided examples to your installation, including your own admin user password and URL.

Procedure

1. Suspend the system.
    a. Run the command to receive the `suspendToken`. Replace the password with the password you set for the admin user.

    ```
    $ curl -i --user admin:yourPassword --header "Accept: application/json" --header "Content-Type: application/jsc
    ```

    b. Write the token from the response you received into the `suspend.json` file.

    ```
    {
    "suspendToken":"4893e925ccb8fcb259e4331d25e318900e35e1439ed028b9be0af1e850b26a0e48b5f92b8e92ff2382b146d728e94b8
    "forShutdown":false
    }
    ```

    If you change the value of the `forShutdown` option to `true`, the system will gracefully shut down.

    c. Perform the suspend action using the `suspend.json` file.

    ```
    curl -i --user admin:yourPassword --header "Accept: application/json" --header "Content-Type: application/json"
    ```

    The request responds with information such as `resumeToken`, `systemStatus`, `resumeResource` path, and `runningOperations`:

    ```
    {"resumeResource":"/monitoring/suspendresume/resumestatus","suspendToken":"4893e925ccb8fcb259e4331d25e318900e35
    ```

    You can now check the current system status as many times as you need by using this endpoint:

    ```
    curl -i --user admin:yourPassword --header  "Accept: application/json" --header "Content-Type: application/jsor
    ```

    d. Write the token from the response into the `resume.json` file.

```
{
"resumeToken": "575ae2c2e996737263aaece5a8b6a8cc088d279bc567010479f37e725ea801534d5679fd187b7c42681fc5307eba411
}
```

2. Resume the system using the `resume.json` file.

```
curl -i --user admin:yourPassword --header "Accept: application/json" --header "Content-Type: application/json" -X P
```

The request should respond with a similar message:

```
{"systemStatus":"RUNNING","runningOperations":[{"threadId":257,"childThreads":[],"category":"WEB_REQUEST","threadNam
```

Results

You have successfully suspended and resumed SAP Commerce.

Platform Shutdown Script

You can invoke Platform graceful shutdown from the command line on a Linux system by using the `shutdown.sh` script.

To shut down Platform, execute `shutdown.sh` as shown:

```
$HYBRIS_HOME_DIR/hybris/bin/platform/shutdown.sh
```

The `shutdown.sh` script initiates the graceful shutdown procedure through a rest call. It waits until all threads get suspended except the one handling the request. As a result, all long-running operations are stopped, and Platform shuts down when the script finishes execution.

The following example shows this behavior. There are 3 threads:

- T1 is idle
- T2 handles a long-running user request
- T3 runs an impex job

When you invoke shutdown:

1. The script makes a rest call to a locally running SAP Commerce.
2. T1 handles the shutdown request and initiates the graceful shutdown procedure (creates `SuspenderThread`).
3. T1 waits for suspension of all other threads: T2, and T3.
4. T1 returns a response.
5. `SuspenderThread` sees that all threads (T1, T2, T3) are suspended and allows Platform to shut down.
6. Platform shuts down.

Testing

SAP Commerce and its components use several various frameworks and abstraction layers. Depending on the abstraction level and the framework in use, you have to use various methods of testing. For example, you need different tools to test a web application, and different tools to test its underlying business code.

Ensuring All Spock Test Classes Are Compiled

To ensure that all Spock tests are run correctly, use the following property:

- `<extname>.backoffice.compile.groovy=true` - ensures that all Spock tests from the `backoffice/testsrc` directory are compiled. The default value is `true`.

Creating Builds with Forked groovyc Ant Task

By default, the `groovyc` ant task compiles builds in a forked mode for all operating systems. To disable forking, use the following property with the value `false`:

```
groovyc.fork=false
```

Change the value of the property to `true` if you want to restore the default behavior.

To create builds with forked `groovyc` for selected extensions, use the following property:

```
<extname>.groovyc.fork=
```

Use the values `true` or `false` to enable or disable forking for selected extensions. The *<extname>*`.groovyc.fork=` property overrides global settings.

For more information on `groovyc` ant task, see The <groovyc> Ant Task .

Testing with JUnit

You can use the JUnit framework to test your code with Unit and Integration tests. SAP Commerce provides the libraries of the current JUnit version, as well as specific SAP Commerce classes for easier testing. It is also shipped with several components to make it easy to execute tests, such as a special test servlet, which is a part of the SAP Commerce Administration Console or designed Ant targets.

Before writing your first JUnit test, it is recommended to read the JUnit cookbook which explains the basic structure of a test class. To test SAP Commerce, you have to use special classes. You can distinguish two types of tests:

- **Unit** test: It does not require access to the database and you do not need the running SAP Commerce environment to execute them. For more details, read the Writing Unit Tests section.
- **Integration** test: It is also called functional tests and needs access to SAP Commerce, which has to be started in order to execute them. For more details, read the Writing Integration Tests section.

Your test classes should have the **`Test`** suffix in the name, and be stored in the following locations:

- ***extensionname*** / **`testsrc`**: Here you can test classes from the ***extensionname*** / **`src`** directory.
- ***extensionname*** / **`web`** / **`testsrc`**: Here you can test classes from the ***extensionname*** / **`web`** / **`src`** directory.

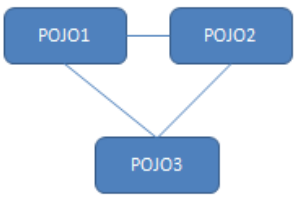
Writing and Executing Tests

Familiarize yourself with the preferred practices for writing both the unit, and integration tests. Learn how to use JUnit tenant to execute tests.

Writing Unit Tests

When developing new software, a key programming practice is to write small, loosely coupled classes, with each one concentrating on one area of responsibility. Such classes are commonly referred to as Plain Old Java Objects, or POJOs. The `Servicelayer` follows this coding principle, and therefore consists of many POJOs. Each POJO implements an interface, and is wired to other POJOs using Spring's Dependency Injection. This approach enables you to easily isolate a class for testing by replacing dependent POJOs with mocks as described below.

Unit tests each focus on single POJOs. Consider the three POJOs as seen in the example below - POJO1, POJO2, and POJO3:



The Unit tests for **POJO1** should test and demonstrate the behavior of **POJO1** for expected and unexpected input. The Unit tests should not reference other dependent POJOs, such as **POJO2** and **POJO3**. These POJOs should be mocked, which means they should be simulated with correct behavior. By focusing the tests on POJO1, the tests are able to:

- Avoid false positives, for example, when a method in **POJO1** has an error, but an error in **POJO2** compensated for it and thus hid the error.
- Avoid false negatives, for example, when **POJO1** is working correctly, but an error in **POJO2** causes the test to fail.
- Run quickly, as **POJO2** or **POJO3** might have long-running methods, or even worse, require a SAP Commerce environment boot-up in which to run.

The main focus of Unit tests is to test the programmed code, and the content of the java class. The tests should check:

- Each public method of the class with valid parameters
- The same methods with illegal parameters, like NULL or negative values or values where exceptions are expected
- If possible the test should cover all possible values accepted in the method body. For example if the method contains an if-then-else statement then if clause should be tested with a true and a false value, to make sure it covers the then clause and the else clause.

Other conventions to keep in mind include:

- The Unit test class should be in the same package as **POJO1**, but is located in the test folder, typically **`/testsrc`**
- The class name should be called **`POJO1Test`**
- Each method should focus on testing one use-case, for example **`testFindUserWithNull()`**, **`testFindUserWithIncorrectCase()`**
- The test code, variable names, and method names are intuitively named in order for commenting not to be necessary.

Mockito should be used to help to isolate a POJO for Unit testing. Mockito is a test framework that enables you to mock entire interfaces, entire classes, and parts of classes. In the example below, you can see how Mockito enables you to modify the behavior of a class without touching the class itself.

```
package de.hybris.mockito;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

import java.util.LinkedList;

import org.junit.Test;

public class MockedLinkedList
{
	@Test
	public void testMockito()
	{
		// In this example mock the behaviour of LinkedList.
		// First step is to get a reference to a mocked LinkedList, by calling Mockito's mock method.
		final LinkedList mockedList = mock(LinkedList.class);

		// Tell mockito that if at a later stage mockedList.get(0) is called,
		// the real implementation should NOT called,
		// but instead the value "first" should be returned
		when(mockedList.get(0)).thenReturn("first");

		// Tell mockito that if at a later stage mockedList.get(1) is called,
		// the real implementation should NOT called,
		// but instead a RuntimeException should be thrown/
		when(mockedList.get(1)).thenThrow(new RuntimeException());

		// Now if you call mockedList.get(0), you will get the value "first"
		assertEquals("The call to LinkedList.get(0) is mocked and returns 'first'", "first", mockedList.get(0));
		System.out.println(mockedList.get(0));

		try
		{
			mockedList.get(1);
			fail("The call to LinkedList.get(1) is mocked and should have thrown a RuntimeException");
		}
		catch (final RuntimeException rte)
		{
			// The call to LinkedList.get(1) is mocked and threw a RuntimeException
		}

		// Call the method mockedList.get(999), which has not been stubbed,
		// and therefore calls the LinkedList's "real" get(999)
		assertEquals("The call to LinkedList.get(999) is not mocked so calls the genuine method", null, mockedList
	}
}
```

Stubbing

To help you focus on testing a single POJO, you can replace any POJO on which it depends with stub implementation. A stub object is used to replace more heavy-weight objects on which an object under test usually depends. You could, for example, mock a method in **POJO2** or **POJO3** and simply return hard-wired responses. This enables you to speed up tests due to removing a dependency on slow POJOs, and to avoid the necessity to connect to the database. For example, if you fake a DAO finder method in **POJO2**. Stubbing is state verification, because it verifies if the state of the POJO is as expected.

Using Mockito you can stub out entire interfaces, entire classes, and parts of classes. The following code snippet presents different examples from Mockito:

```
// Two methods of the LinkedList class are mocked
// If you call any other method of LinkedList, it will call the original method, as stubs are provided only for only get(0
LinkedList mockedList = mock(LinkedList.class);

// Provide a stub, such that when mockedList.get(0) is called, the real implementation is NOT called,
// but instead directly returns the value "first"
when(mockedList.get(0)).thenReturn("first");

// Provide a stub, such that when mockedList.get(1) is called, the real implementation is NOT called,
// but instead directly throws RuntimeException
when(mockedList.get(1)).thenThrow(new RuntimeException());

// Call mockedList.get(0), which will invoke the stub and return "first"
System.out.println(mockedList.get(0));

// Call mockedList.get(1), which will invoke the stub and throw the RuntimeException
```

```
System.out.println(mockedList.get(1));

// Call the method mockedList.get(999), which has not been stubbed,
// and therefore calls the LinkedList's "real" get(999)
System.out.println(mockedList.get(999));
```

BDD Way of Stubbing

Behavior-driven development (BDD) focuses on obtaining a clear understanding of desired software behavior through discussion with stakeholders. It extends Test-driven development (TDD) by writing test cases in a natural language that non-programmers can read. Behavior-driven developers use their native language in combination with the ubiquitous language of domain driven design to describe the purpose and benefit of their code. Mockito provides a nice interface to write BDD-oriented tests with simple **given/when/then** scenarios. Below is sample test written in this manner. Notice the self-explanatory name of the test method with the prepending word, **should**.

```
@Test
public void shouldCreateSimpleDateFormatObjectOfGivenFormatForCurrentLocale()
{
        // given
        final String format = "yyyy-mm-dd";
        given(i18nService.getCurrentLocale()).willReturn(new Locale("en"));

        // when
        final DateFormat result = factory.createDateTimeFormat(format);

        // then
        assertThat(result).isNotNull();
        assertThat(result).isInstanceOf(SimpleDateFormat.class);
        verify(i18nService, times(1)).getCurrentLocale();
}
```

The idea of writing BDD tests is to write small test cases that cover one behavior.

Mocking

Mocking helps to verify a conversation. For example, if the sequence of methods calls between **POJO1** and its dependent POJOs is as expected. It is usually done hand-in-hand with stubbing.

Mocking is a behavioral verification because it verifies if the behavior of the POJO is as expected. To use Mockito framework to perform the verification, follow the basic steps below:

1. Create the **POJO2** mock and then wire this mock instead of the original **POJO2** in to **POJO1**.
2. Call method A from **POJO1**, which you expect to call methods B and C from **POJO2**.
3. Verify that **POJO1** did call methods B and C from **POJO2**.

An example of verifying that calls to a mocked object are made:

```
import static org.mockito.Mockito.*;

// Create a mock of List interface
List mockedList = mock(List.class);
mockedList.add("one");
mockedList.clear();

// Verify that add and clear methods from List are called
verify(mockedList).add("one");
verify(mockedList).clear();
```

Mockito makes it easy to combine mocking and stubbing, which also enables you to perform state and behavioral tests on your services. See, for example, **`DefaultCompetitionServiceTest.java`** from the **cuppy** extension.

DefaultCompetitionServiceTest

```
package de.hybris.platform.cuppy.services;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

import de.hybris.platform.cuppy.daos.CompetitionDao;
import de.hybris.platform.cuppy.model.CompetitionModel;
import de.hybris.platform.cuppy.services.impl.DefaultCompetitionService;
import de.hybris.platform.servicelayer.model.ModelService;

import java.util.Arrays;
import java.util.List;

import org.junit.Before;
```

```
import org.junit.Test;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

public class DefaultCompetitionServiceTest
{
        // The service under test..
        private DefaultCompetitionService competitionService;

        // Mock out the services/daos on which competition service depends using Mockito's mock annotation
        @Mock
        private PlayerService playerService;
        @Mock
        private ModelService modelService;
        @Mock
        private CompetitionDao competitionDao;

        @Before
        public void setUp()
        {
                // So that the @Mock annotations is recognised, you need the following line..
                MockitoAnnotations.initMocks(this);

                // Create the service and wire in the services/daos
                competitionService = new DefaultCompetitionService();
                competitionService.setPlayerService(playerService);
                competitionService.setModelService(modelService);
                competitionService.setCompetitionDao(competitionDao);
        }

        @Test
        public void testGetAllCompetitions()
        {
                // The purpose of this test is to check that..
                // CompetitionService's method getAllCompetitions makes a call to CompetitionDao's method findCompetitions
                // The value that  CompetitionService's method getAllCompetitions
                final CompetitionModel comp1 = new CompetitionModel();
                comp1.setCode("comp1");

                final CompetitionModel comp2 = new CompetitionModel();
                comp2.setCode("comp2");

                // Create a stub of competitionDao's method findCompetitions.  Should it be called, return a hard-wired re
                // This helps you focus this test on CompetitionService, and not also on CompetationDAO, and also means yo
                // access to the persistence layer to test CompetitionService.

                when(competitionDao.findCompetitions()).thenReturn(Arrays.asList(comp1, comp2));  // <--- Example of Stubb

                // Now call the service's method (which you expect to call findCompetitions)
                final List<CompetitionModel> result = competitionService.getAllCompetitions();
                // If it did call findCompeitions, then our hard-coded response will have been returned to the service.
                // and you can confirm that competiationService did what you expect with this hard-wired response
                assertEquals(2, result.size());
                assertEquals(comp1, result.get(0));
                assertEquals(comp2, result.get(1));

                // Finally confirm that the call to findCompetitions was made
                verify(competitionDao).findCompetitions(); // <--- Example of Behaviour Verification using Mocking
        }
}
```

Setting up Mocks Using Mockito

Mockito provides annotations that are helpful in making test classes simpler. For instance, if you have to test some service that is dependent on two other services and you want to test it in isolation, then you have to mock them. Instead of using setters on a tested service to provide mocked services, you can use **`@InjectMocks`** annotation like this:

```
@UnitTest
public class DefaultFormatFactoryTest
{
        @InjectMocks
        private final DefaultFormatFactory factory = new DefaultFormatFactory();
        @Mock
        private I18NService i18nService;
        @Mock
        private CommonI18NService commonI18NService;

        @Before
        public void setUp()
        {
                MockitoAnnotations.initMocks(this);
        }
}
```

Notice that the **`factory`** instance variable is initialized directly instead of doing it in the **`setUp()`** method. This speeds up whole test. The **`@InjectMock`** annotation injects all required mocks into the **`DefaultFormatFactory`** object in one call of **`MockitoAnnotations.initMocks(this)`**, so there is no

need to use **`set`** methods in **`setUp()`**.

Code Coverage

Using stubbing and mocking gives the possibility to thoroughly test your POJOs. To know how complete the testing is, you need statistics of the code coverage. An easy way to determine this in the SpringSource Tool Suite using the **EclEmma** plugin:

1. Download and install **EclEmma**.
2. Right-click a test and run it by selecting **Coverage As → JUnit Test**.
3. Select the class you are testing from the **Coverage** view, and spot the green covered code, and the red uncovered code.

The goal should be to maximize your test coverage, and this is much easier to do by Unit testing with mocks and stubs than by Integration tests only.

Writing Integration Tests

SAP Commerce provides the possibility to use a preconfigured, JUnit tenant where you can execute your JUnit tests isolated from other tenants. You can test your business functionality using real item instances.

Basic Mechanism

The easiest way to add specific functionality to your test class is to extend the **`ServicelayerTest`** class. In this way, you gain access to methods that create some very basic test items that can be used for testing. By extending the **`ServicelayerTest`** class, you may create instances of all basic SAP Commerce Models like language, currency, unit, and country using the **`createCoreData`** method. You may also import test data stored in a CSV file using the **`importCsv`** method, or import test data using the **`importStream()`** method. For more details, see the API documentation.

If you want to use other services in a test, you can inject them using the **`@Resource`** annotation. Use the **`@Before`** annotation to execute code or import data before each test method. Do not use the **`@Before`** annotation more than once in your test. All methods with this annotation are executed before each test method, but it is not clear in which order. For example, the Sun JDK executes them in 99% of all cases from top-to-down, whereas the IBM JDK executes them in an alphabetical order. To execute code or clean up data after each method, use the **`@After`** annotation. Do not use the **`@BeforeClass`** annotation as it is executed in a static way and is out of the test run scope. With this, a tenant is not set and changes are not reverted.

```
public class CreateUserTest extends ServicelayerTest
{
  @Resource
  private ModelService modelService;

  @Resource
  private UserService userService;

  @Test
  public void testCreateUser()
  {
    //create a user
    final EmployeeModel user = new EmployeeModel();
    user.setUid("testUser");
    modelService.save(user);

    //test creation
    assertNotNull(userService.getUserForUID("testUser"));
  }
```

Pay attention to the static import of the **`Assert`** class of the JUnit framework, which makes all assert methods available in your test class. The listed import of the SAP Commerce **`de.hybris.platform.testframework.Assert`** class provides additional assert methods written by SAP Commerce. Extending a test class is not always suitable, for example, when already extending another class. In this case, you can easily add the functionality of the SAP Commerce test class using annotations, as done by the **`HybrisJUnit4Test`** too.

```
import static org.junit.Assert.*;
import static de.hybris.platform.testframework.Assert.*;

import de.hybris.platform.testframework.*;
import de.hybris.platform.jalo.ConsistencyCheckException;
import de.hybris.platform.servicelayer.i18n.commonI18NService;

import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(HybrisJUnit4ClassRunner.class)
@RunListeners({ ItemCreationListener.class, LogRunListener.class, PlatformRunListener.class })
public class MyTestClass
{
        @Test
        public void countryTest()
        {
                //because this class does not extend the ServicelayerTest class,
                //Spring wiring does not work; you must get the service
```

```
                //over the registry application context
                CommonI18NService commonI18NService = Registry.getApplicationContext().getBean(CommonI18NService.class);

                CountryModel myCountry = commonI18NService.getCountry("en");
                assertNotNull(myCountry);
        }
}
```

All the additional logic is implemented using JUnit **RunListener** classes. Unfortunately, these listeners cannot be registered easily to the test process. Instead, you can use the **HybrisJUnit4ClassRunner** class, which has to be registered using the JUnit **@RunWith** annotation. This runner registers all listeners specified using the **@RunListeners** annotation. When using the **ItemCreationListener** class, all created items are removed after a test is finished. This kind of proprietary transaction system has several flaws. All creations of items are written persistent to the database and then removed, so you have no isolation in concurrent processing. Furthermore, only a creation of an item is registered, not a modification of an item. If you modify a standard meta item in your test, for example, if you change the write flag of an **AttributeDescriptor**, it will not be reverted at the end of the test. For more details about all available listeners and their usage, see the API documentation.

ℹ Note

When using the annotation style, ***jaloSession*** and ***defaultLanguage*** variables, as well as the **getOrCreateLanguage** method provided by the **ServicelayerTest** class are not available.

Preferred Mechanism

The preferred way of writing tests is to extend the **ServicelayerTransactionalTest** class. This class adds transaction-based isolation logic to your tests by registering the **TransactionRunListener** listener. Before the **@Before** method of your test is called, a new transaction is started. After the **@After** method of your test is called, a transaction rollback is performed. This assures complete isolation of your test. The **ServicelayerTransactionalTest** class extends the **ServicelayerTest** class, and therefore also provides the variables and methods of this class.

The following example demonstrates how to test a modification of the **admin** user. As the **admin** user already exists before the test starts, the modifications are not reverted when using the direct **ServicelayerTest** class, as only newly created items are tracked. By using the **ServicelayerTransactionalTest** class, all changes are reverted. Be aware that for testing logic using transactions with this test class can cause problems.

```
public class ModifyAdminTest extends ServicelayerTransactionalTest
{
        @Resource
        private ModelService modelService;

        @Resource
        private UserService userService;

        @Test
        public void modifyAdminTest()
        {
                UserModel admin = userService.getAdminUser();
                admin.setName("test");
                modelService.save(admin);
                assertEquals("test",admin.getName());
        }
}
```

You also can add the transaction logic using annotations in the same way as described in Basic Mechanism section:

```
@RunWith(HybrisJUnit4ClassRunner.class)
@RunListeners({TransactionRunListener.class, ItemCreationListener.class, LogRunListener.class, PlatformRunListener.class }
@Transactional
public class MyTestClass
{
      // ...
}
```

The additional **TransactionRunListener** listener is registered. This listener enables the transaction support if the annotation **@Transaction** is found. The **@Transactional** annotation can be specified on a class or method level, so you can enable the transaction support for separate methods. This is also the reason for registering the **ItemCreationListener**. If no transactional support is enabled, because the **@Transactional** annotation is missing or the underlying database does not support transactions, the proprietary mechanism is used.

Be aware that if you want to test transaction functionality, for example, assuring the correctness of a rollback, you cannot use the transactional support.

Annotations

The table below presents available class annotations for inclusion in Java test files. They are used as labels to help generate Bamboo scripts that automatically select and run tests of particular types.

Annotation	Usage
`@IntegrationTest`	Tests that are to be executed inside an application server environment, like Apache Tomcat or WebSphere.
`@DemoTest`	Typically well or self-documented Integration tests for inclusion in SAP Commerce Wiki and available to partners.
`@UnitTest`	Tests that do not require a SAP Commerce environment in which to run, and are therefore fast. Unless your test needs the SAP Commerce environment, for example, because of the persistence layer, then you should consider writing a unit test instead with mocked-out dependencies.
`@PerformanceTest`	A performance test can be configured to run under different circumstances, for example: • 6000 times in a row, • with 1 mio items, • under heavy load It is designed for helping to examine performance characteristics.
`@ManualTest`	Some tests require external software, for example the CyberSource tests. These tests should not be executed during automated testing.

Testing Exceptions

Follow the rules below when writing tests:

1. Do not use **`@Test(expected)`**. This feature can potentially make a few tests cleaner, but is really error-prone:

   ```
   @Test(expected=Exception.class)
   public void testSomething() {
     //line 1: throws NullPointerException
     //lines 2-30: some test code
   }
   ```

   If due to some bug in the setup, the test method breaks in the first line, most of the test code is not even executed. The expected exception hides this issue and the test is green, so it is completely useless.

2. If you violate the rule above, then at least the exception should be very specific. For example:

   ```
   @Test(expected=BreadOutOfStockException.class)
   public void shouldBuyBread() {}
   ```

   Even if you use a specific exception, the test does not document exactly where the exception is thrown. Sometimes it may lead to subtle bugs. Therefore, if the test method has multiple lines, do not violate the #1 rule. Instead, you can simply write:

   ```
   @Test
   public void shouldBuyBread() throws Exception {
     //given
     me.goToBakery(bakery);
     bakery.setBreadAmount(0);

     try {
       //when
       me.buyBread();
       //then
       fail("BreadOutOfStockException to be thrown");
     }
     catch (BreadOutOfStockException e) {
        //ok here, we expect this exception here
     }
     catch (Exception e) {
       fail(" caught unexpected exception: "+e);
     }
   }
   ```

Executing Tests

Use JUnit tenant to execute tests. When you update to the newer version of SAP Commerce, you also should configure new extensions for the JUnit tenant. Follow the steps below to make sure that all needed extensions are enabled, and to initialize JUnit tenant:

1. Open the SAP Commerce Administration Console.

2. Go to the **Platform** tab and select **Tenants** option.
3. The **Tenants Overview** page displays a list of existing tenants. Click the **View** button next to the **JUnit** tenant.
4. On the **Edit Tenant** page, check if all extensions are enabled.
5. Now initialize the **JUnit** tenant. Click the **Initialize** button.
6. The **JUnit** tenant activates and **Initialization** page displays.
7. Click the **Initialize** button.

Furthermore, you can initialize SAP Commerce by calling **`ant yunitinit`** from the command prompt or calling the **`Initialization.initializeTestSystem()`** within a standalone class. All three variants initialize the test system with a minimum data containing no essential data and no project data of installed extensions. If you want to set up essential or project data of a specific extension in your **`@Before`** method, you can execute the following code:

```
// creating essential data of MyExtension
MyExtensionManager.createEssentialData(Collections.EMPTY_MAP, null);
// creating project data of MyExtension
MyExtensionManager.createProjectData(Collections.EMPTY_MAP, null);
```

For creating essential data of the **core** extension, you should execute:

```
new CoreBasicDataCreator().createEssentialData(Collections.EMPTY_MAP, null);
```

You are not able to use **ImpEx** on the initialized test system because of missing encodings. You can install them by creating essential data of the core extension or by installing the encodings directly:

```
new CoreBasicDataCreator().createSupportedEncodings();
```

There are three different ways of executing tests:

- Using Eclipse
- Using Ant
- Using the Test Servlet

ℹ Note

If all tests are completed, then a report is generated and the message is displayed, stating **build successful** (even if some tests fail). To see the details of the outcomes of the tests, find the report in `platform-module/log/junit`.

Eclipse

To execute a test using the Eclipse, follow below instructions:

1. Right-click your test class.
2. From the context menu, select **Run As,** and click **JUnit Test**

You may get:

- Warnings like below:

```
WARN  [Utilities] cannot load extension storefoundation because:
java.lang.ClassNotFoundException: ystorefoundationpackage.jalo.YStoreFoundationManager
WARN  [Utilities] If you receive this error while starting from eclipse,
you need to add the project that
WARN  [Utilities] contains the storefoundation extension to the java build path of your project from where you have
WARN  [Utilities] started this class. (project->right click->properties->build path->tab Projects.


WARN  (junit) [ComposedTypeEJBImpl] missing jalo class
'de.hybris.platform.xprint.jalo.PurgeGettersJob' for item type 'XPurgeGettersJob' - trying supertype class instead
```

- Exceptions like below:

```
java.lang.ClassNotFoundException:
de.hybris.platform.webfoundation.jalo.workflow.orderprocess.OrderProcessingBatchJob
```

If this is the case, you probably do not have all configured extensions of your **`localextensions.xml`** file registered at the classpath of the Eclipse JUnit test configuration. To fix it, perform the following steps:

1. Open the **Run** menu click **Run Configurations** to open **Run Configurations** dialog box.

2. In the **Run Configurations** dialog box, select your test configuration.
3. Navigate to the **Classpath** tab.
4. Select the **`platform`** folder.
5. Click the **Add Projects** button. In the **Project Selection** dialog box, select all extensions you have configured in the **`localextensions.xml`** file.

If your Spring configuration does not get bootstrapped at the startup of the Platform, the Spring file is probably not a part of test **classpath**. This can either be caused by a missing project at the classpath, or by not including your resources folder in the Eclipse classpath at all. Assure that your **`.classpath`** file contains the following line:

```
<classpathentry exported="true" kind="lib" path="resources"/>
```

Ant

All ant targets can be executed in the **`platform`** folder or in the specific extension directory. If you execute a target inside an extension directory, the extension filter is added automatically for the current extension.

Ant Targets

List of available ant targets:

Ant Call	Description
`ant alltests`	Executes all tests found by **`testClassesUtil.getAllTestClasses()`**
`ant unittests`	Executes all tests found by **`testClassesUtil.getAllUnitClasses()`**
`ant demotests`	Executes all tests found by **`testClassesUtil.getAllDemoClasses()`**
`ant integrationtests`	Executes all tests found by **`testClassesUtil.getAllIntegrationClasses()`**
`ant performancetests`	Executes all tests found by **`testClassesUtil.getAllPerformanceClasses()`**
`ant manualtests`	Executes all tests found by **`testClassesUtil.getAllManualClasses()`**

Commandline Modifiers

If you want to specify an ant target a bit more, you can pass one or more of these properties to the ant call:

Modifier	Example	Description
`testclasses.extensions`	`ant unittests -Dtestclasses.extensions=basecommerce,core`	executes all tests from given extensions
`testclasses.annotations`	`ant alltests -Dtestclasses.annotations=demotests,unittests,manualtests`	executes tests with given annotations
`testclasses.packages`	`ant unittests -Dtestclasses.packages=de.hybris.platform.payment.*`	executes tests from given package
`testclasses.split`	`ant alltests -Dtestclasses.split=1-3`	Divides the number of tests by the second number, and executes the part indicated by the first number. For example, a `split=1-5` executes the first 20% of the tests.
`-Dtestclasses.suppress.junit.tenant`	`ant unittests -Dtestclasses.suppress.junit.tenant=true`	switches off starting up of Platform for every test

Like all other ant properties, these properties are set once and cannot be modified anymore. If you call **`ant demotests -Dtestclasses.annotations=unittests`**, all **UnitTests** are executed.

Including and Excluding Tests

For more information, see `-Dtestclasses.packages` in Customizing Test Execution.

WebServices Tests

Use the ant target **-allwebtests** to run all tests inside the **web/testsrc** folder.

Ant Callbacks

If you want to provide custom logic before or after test execution, or initialization, you can write a buildcallback in **`buildcallbacks.xml`** of your extension:

Callback Name	When it is Performed
`(extension_name)_before_yunit`	Performed right before test execution
`(extension_name)_after_yunit`	Performed right after test execution, if tests succeed
`(extension_name)_before_yunitinit`	Performed right before the test framework is initialized
`(extension_name)_after_yunitinit`	Performed right after the test framework is initialized

An example below creates essential data of core extension right after the initialization of the test system. It uses the **`yrun`** macro that starts the Platform and executes the given code as BeanShell command. The first line of the code sets the tenant of test system, the second call creates the essential data of core, and last statement shuts down the Platform.

```
<macrodef name="(extension_name)_after_yunitinit">
  <sequential>
   <yrun>
    de.hybris.platform.core.Registry.setCurrentTenantByID("junit");
    new de.hybris.platform.jalo.CoreBasicDataCreator().createEssentialData(null,null);
    de.hybris.platform.util.RedeployUtilities.shutdown();
   </yrun>
  </sequential>
 </macrodef>
```

Test Servlet

> **Note**
>
> Do not forget to initialize the test systems by using the **Initialize** button.

For information on SAP Commerce JUnitEE TestRunner, see The SAP Commerce Testweb Front End.

Customizing Test Execution

There is a set of filters available you can use with Java to create your own servlet for test execution, and to design custom Ant targets. With this, you can customize the execution of tests tailored for your project needs.

Overview

Customization of the tests execution is possible because of the `TestClassesUtil`, that scans the SAP Commerce environment for test classes and is located in the SAP Commerce bootstrap. It does not need a running SAP Commerce instance and is therefore usable to get different collections of JUnit test classes before and while runtime.

To get an instance, follow below examples:

- Using Java:

```
final PlatformConfig platformConfig = ConfigUtil.getPlatformConfig(this.getClass());
TestClassesUtil testClassesUtil = new TestClassesUtil(platformConfig);
```

- Using Ant Targets:

```
<taskdef name="testclasses" classname="de.hybris.ant.taskdefs.TestClassesTask">
        <classpath>
                <fileset dir="${platformhome}/bootstrap/bin">
                        <include name="*.jar" />
                </fileset>
        </classpath>
</taskdef>
<testclasses property="testlist" />
```

Afterwards, the given property can be used as a list property.

Test Class Scan

The `TestClassesUtil` does not hold a result or a list. It searches for the test classes on the classpath every time you invoke a method. This lets you change the behavior at runtime, for example by setting new filters. The result can be filtered by annotations, package, or extension. For more details, see the API documentation.

> **Note**
> As a fallback, all Tests that do not have one of the SAP Commerce annotations are treated as `@IntegrationTests`.

Filters

Filters can be used in many various ways. It all depends how you want to customize the execution of your tests.

If you want to filter the result in a specific way, you can use one of the following filters:

Filter Name	Description	Usage Format
Annotation	This filter accepts only test classes that are annotated with the given annotation.	The expected format is a single annotation class or a collection of class files. Supported annotations are `DemoTest`, `IntegrationTest`, `PerformanceTest`, `ManualTest`, and `UnitTest`.
Package	This filter accepts only test classes for the given packages.	Regular expressions like de.hybris.*.cockpit.* are allowed. The expected format is a single string or a collection of strings.
Negative Package	This filter accepts only test classes that do not match these packages. The positive list is evaluated before the negative list.	Regular expressions like de.hybris.*.cockpit.* are allowed. The expected format is a single string or a collection of strings.
Extension	This filter allows only extensions to be scanned that are in the given list. This filter is automatically applied if you call an `ant` target inside the extension directory.	The expected format is a single string or a collection of strings.
Split	This filter splits the final number of tests by the given string, which can be used for automatic testing purposes. **Note** Different classloaders load classes in different order. For getting the same result for all classloaders, the test classes are sorted alphabetically before they are split.	The given object has to be a String with the format a-b. **b** is the total number of parts you want the tests to be split into. **a** is the number of the current part you want to get. These are the possible string values: • 1-4 - tests from 1 to 250 • 2-4 - tests from 251 to 500 • 3-4 - tests from 501 to 750 • 4-4 - tests from 751 to 1000

Using Filters with Java

You can use below examples of how to use specific filter in your own test servlet.

Filter Name	Example

Annotation	• Search for all test classes that are annotated as `UnitTests`:
	`testClassesUtil.addFilter(FILTER.FILTER_BY_ANNOTATION, UnitTest.class);`
	`Collection<Classes> testClasses = testClassesUtil.getFilteredTestClasses();`
	• Search for all test classes that are annotated as `UnitTest` and `IntegrationTest`:
	`Set<Class> annotations = new HashSet<Class>();`
	`annotations.add(IntegrationTest.class);`
	`annotations.add(UnitTest.class);`
	`testClassesUtil.addFilter(FILTER.FILTER_BY_ANNOTATION, annotations);`
	`Collection<Classes> testClasses = testClassesUtil.getFilteredTestClasses();`

Package	• Search for all test classes that are in the standard SAP Commerce package:

Filter Name	Example

	`testClassesUtil.addFilter(FILTER.FILTER_BY_PACKAGE, "de.hybris.platform.*");`
	`Collection<Classes> testClasses = testClassesUtil.getFilteredTestClasses();`
	
	• Search for all test classes that are from SAP Commerce and in a cockpit package or from google:
	
	`Set<String> packages = new HashSet<String>();`
	`packages.add("de.hybris.*.cockpit.*");`
	`packages.add("com.google.*");`
	`testClassesUtil.addFilter(FILTER.FILTER_BY_PACKAGE, packages);`
	`Collection<Classes> testClasses = testClassesUtil.getFilteredTestClasses();`

Negative Package	Get all classes that are not from SAP Commerce:
	
	`testClassesUtil.addFilter(FILTER.FILTER_BY_PACKAGE_NEGATIVE, "de.hybris.*");`
	`Collection<Classes> testClasses = testClassesUtil.getFilteredTestClasses();`

Extension	Execute all `payment`, `basecommerce`, and `core` extensions tests:
	
	`Set<String> extensions = new HashSet<String>();`
	`extensions.add("payment");`
	`extensions.add("basecommerce");`
	`extensions.add("core");`
	`testClassesUtil.addFilter(FILTER.FILTER_BY_EXTENSION, extensions);`
	`Collection<Classes> testClasses = testClassesUtil.getFilteredTestClasses();`

Split	You have one thousand tests and you want to split them into four pieces:
	
	`testClassesUtil.addFilter(FILTER.SPLIT_FILTER, "1-4");`
	`Collection<Classes> testClasses = testClassesUtil.getFilteredTestClasses();`

Using Filters with Ant Target

You can use below examples to design your own Ant targets to filter and execute specific test. You can do it in the `antmacros.xml` file.

Filter Name	Ant Example	Ant Command Line

Annotation	• Search for all test classes that are annotated as `UnitTests`:
	
	`<testclasses property="testlist" annotations="unittests" />`
	
	• Search for all test classes that are annotated as `UnitTest` and `IntegrationTest`:
	
	`<testclasses property="testlist" annotations="unittests,integrationtests" />`
		`ant unittests -Dtestclasses.`
		
		`ant unittests -Dtestclasses.`

Package	• Search for all test classes that are in the standard SAP Commerce package:
	
	`<testclasses property="testlist" packages="de.hybris.platform.*" />`
	
	• Search for all test classes that are from SAP Commerce and in a cockpit package or from google:
	
	`<testclasses property="testlist" packages="de.hybris.*.cockpit.*,com.google.*" />`
		`ant unittests -Dtestclasses.`
		
		`ant unittests -Dtestclasses.`

Negative Package	Get all classes that are not from SAP Commerce: `<testclasses property="testlist" packages_negative="de.hybris.platform.*" />`	`ant unittests -Dtestclasses.`
Extension	Execute all `payment`, `basecommerce`, and `core` extensions tests: `<testclasses property="testlist" extensions="payment,basecommerce,core" />`	`ant unittests -Dtestclasses.`
Split	You have one thousand tests and you want to split them into four pieces:	`ant unittests -Dtestclasses.`

Filter Name	Ant Example	Ant Command Line
	`<testclasses property="testlist" split="1-4" />`	

The SAP Commerce Testweb Front End

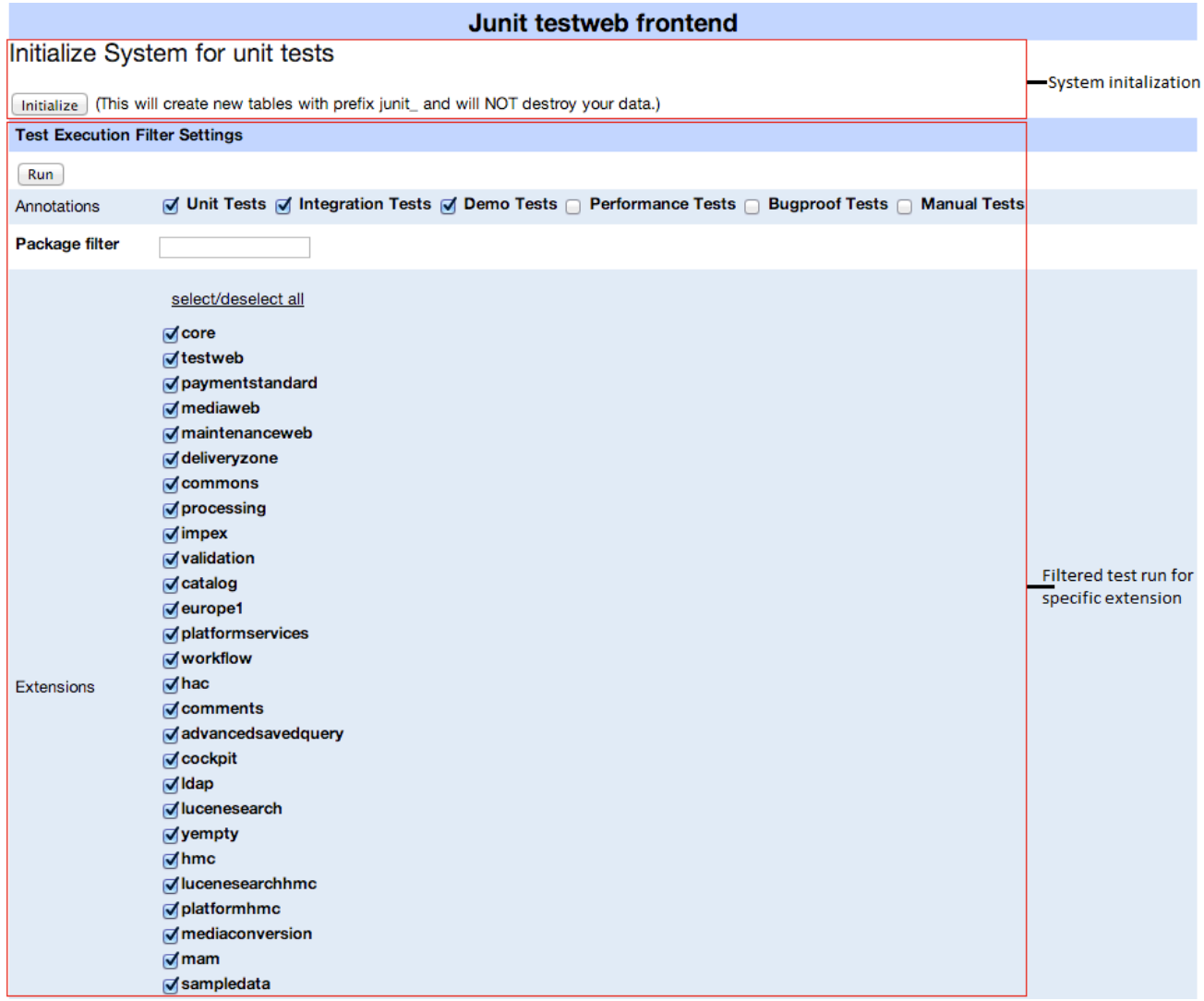
With the Testweb front end, you can execute one or more tests or test suites. This can be done on a running system, especially when there is no ant environment from which to execute such tests.

Access the Testweb front end using the direct address `http://<server-address>/test`. For example, `http://localhost:9001/test`.

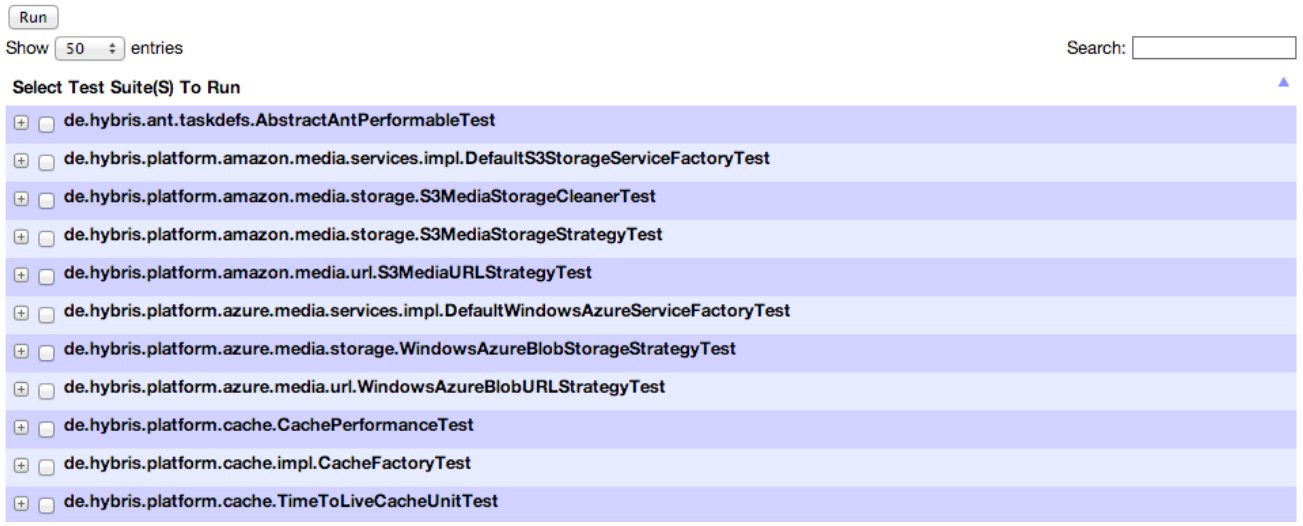
Main Areas

The Testweb front end consists of the following main areas:

- System initialization
- Filtered test run for specific extension
- Test run by test suite name



Testweb Front End: Test Servlet System Initialization and Filtered Test Run

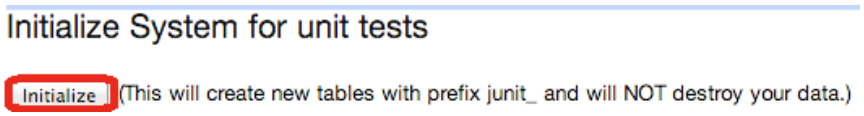


Testweb Front End: Test Run by Test Suite Name

System Initialization in Testweb Front End

To run any test, first you need to initialize the system to prepare it for all future test runs.

You can initialize the system from the test web frontend by clicking the Initialize button. You will see the information Test system initialized correctly when the initialization finishes.



Initialization Button

ℹ Note

Initialize the master tenant before initializing the JUnit tenant.

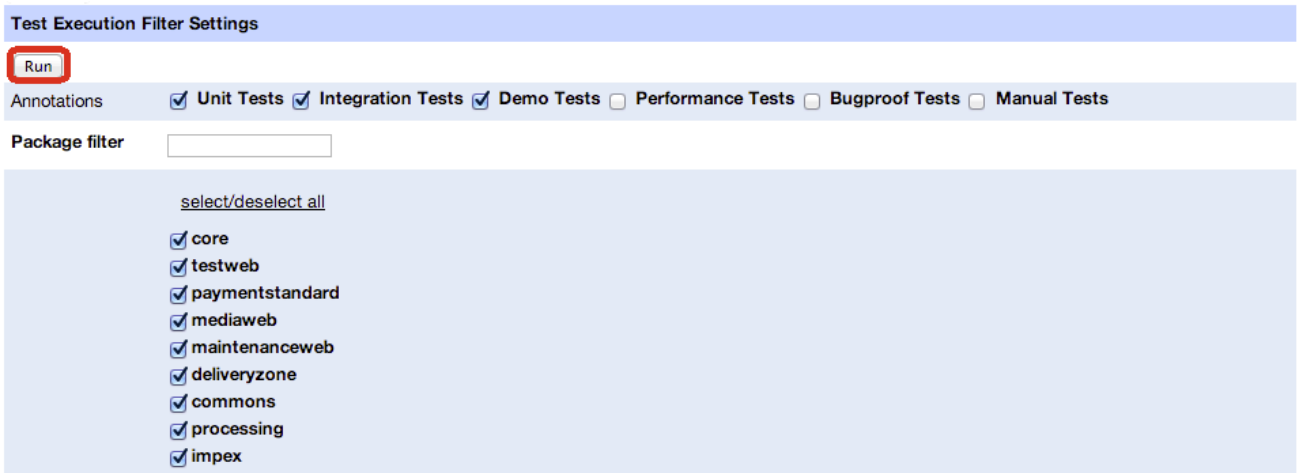
Filtered Test Run for Specific Extension

There are several filters that you can use when running your tests.

In this area, you can run tests selected using one or all of the following filters:

- Annotations: You can filter by annotations, so that only those tests are executed that have that specific annotation. If you do not select any annotations at all, then only tests without any special annotation are executed.
- Package Filter: You can filter by package name by specifying pattern that all package paths of the tests should match. This filter is not case-sensitive and accepts wildcard characters. For example, *de.hybris.*impex*. It also supports filtering down to the actual test class if you prefer that approach: *.LocalizationTest. If you do not use the Package Filter, all tests are executed that have not been filtered out by the Annotations and Extensions filter.
- Extensions: With this filter, you can select extensions of which the tests are executed. If you do not use the Extensions filter, no tests will be run.

After choosing the appropriate filter, click the Run button.



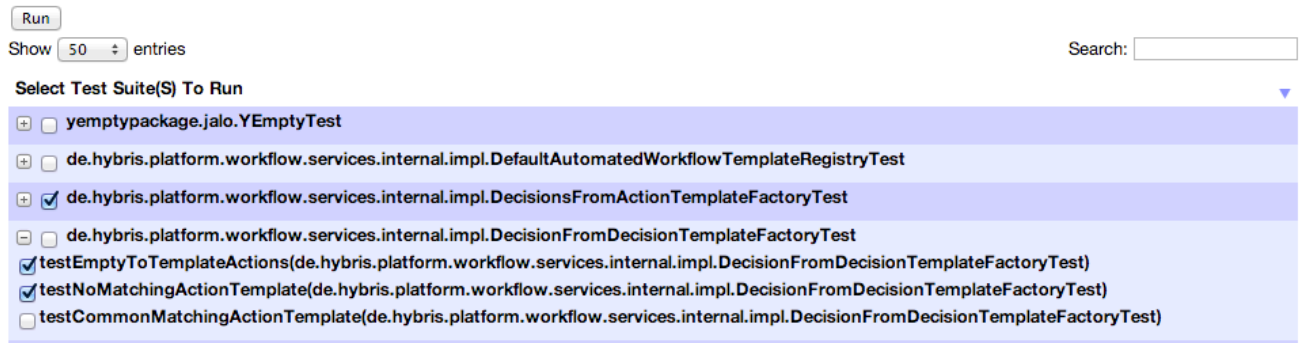
Run Button

Test Run by Test Suite Name

You can run tests by selecting the test suite from the list. The test suite is the name of the test class with its full package path. It is called a suite as it may contain more than one test method.

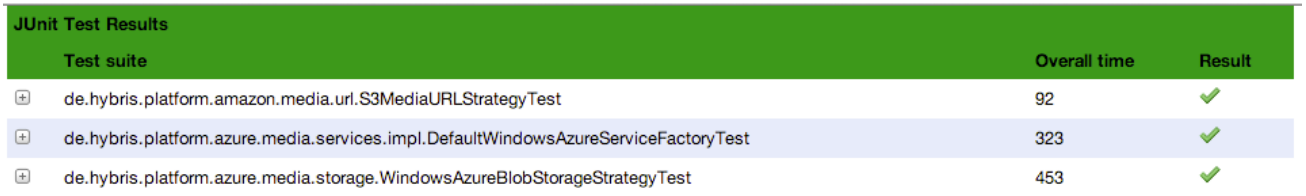
Selecting Test Suites to Be Execute

You can select individual test suites or test cases by selecting checkboxes:



Selecting Individual test Suites or test Cases

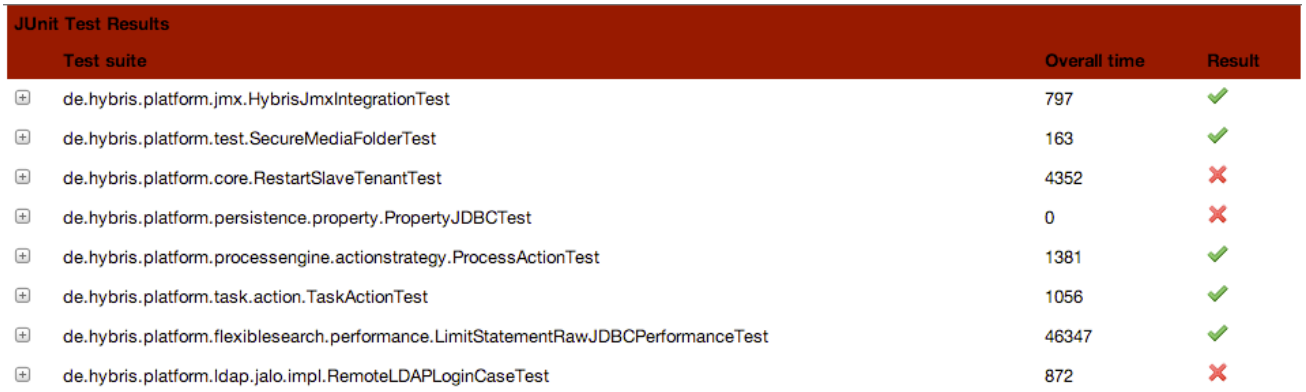
Results are displayed as a simple table.



JUnit Test Results		
Test suite	Overall time	Result
de.hybris.platform.amazon.media.url.S3MediaURLStrategyTest	92	✓
de.hybris.platform.azure.media.services.impl.DefaultWindowsAzureServiceFactoryTest	323	✓
de.hybris.platform.azure.media.storage.WindowsAzureBlobStorageStrategyTest	453	✓

Initialization Button

If results contain errors, they are marked with a red cross. Click an individual test case cross icon to get failure details:



JUnit Test Results		
Test suite	Overall time	Result
de.hybris.platform.jmx.HybrisJmxIntegrationTest	797	✓
de.hybris.platform.test.SecureMediaFolderTest	163	✓
de.hybris.platform.core.RestartSlaveTenantTest	4352	✗
de.hybris.platform.persistence.property.PropertyJDBCTest	0	✗
de.hybris.platform.processengine.actionstrategy.ProcessActionTest	1381	✓
de.hybris.platform.task.action.TaskActionTest	1056	✓
de.hybris.platform.flexiblesearch.performance.LimitStatementRawJDBCPerformanceTest	46347	✓
de.hybris.platform.ldap.jalo.impl.RemoteLDAPLoginCaseTest	872	✗

© hybris AG, 2013

Failure Details

Test Case Details

REST API for Calling Specific Test Groups

It is possible to call special URLs to execute tests by their annotations or all test in current extensions set.

URL	tests group
`/test/run/all`	all tests
`/test/run/unit`	@UnitTest annotated tests
`/test/run/integration`	@IntegrationTest annotated tests
`/test/run/manual`	@ManualTest annotated tests
`/test/run/performance`	@PerformanceTest annotated tests
`/test/run/demo`	@DemoTest annotated tests

To each of the above URLs, you can always pass the `extensions` parameter in which you can specify comma-separated extension names to narrow down the search results. Sample URL calls can look as follows:

```
// run all tests
http://localhost:9001/test/run/all

// run all tests but from extensions core and ldap
http://localhost:9001/test/run/all?extensions=core,ldap
```

Additionally, there is another URL to execute specific test suites by their fully qualified names by using parameter `names` as follows:

`http://localhost:9001/test/run/suites?`
`names=de.hybris.platform.catalog.jalo.classification.ClassificationTest,de.hybris.platform.catalog.jalo.CatalogTest`

A sample call to URL above executes tests for two test suites: ClassificationTest and CatalogTest.

The result is generated in the XML form. Below you can find an example of a generated xml file:

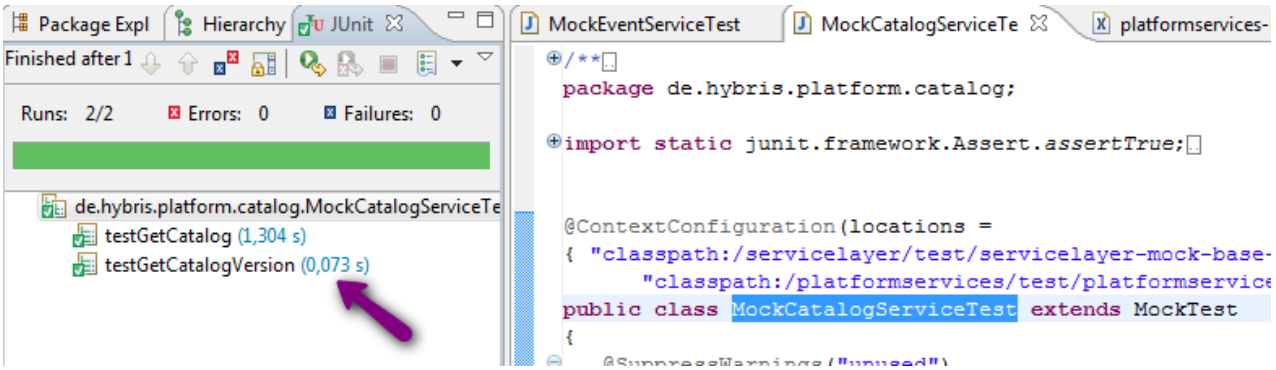
```
<?xml version="1.0" encoding="UTF-8" ?>
<testsuites>
  <testsuite name="HybrisJmxIntegrationTest" package="de.hybris.platform.jmx" tests="1" time="778" failures="0" errors="0"
    <testcase name="testJMXAPI" time="778" ></testcase>
  </testsuite>
  <testsuite name="SecureMediaFolderTest" package="de.hybris.platform.test" tests="1" time="160" failures="0" errors="0" >
    <testcase name="testSecureFolder" time="160" ></testcase>
  </testsuite>
  <testsuite name="RestartSlaveTenantTest" package="de.hybris.platform.core" tests="5" time="4308" failures="1" errors="1"
    <testcase name="testCanQueryFlexibleSearchAfterRestart" time="0" >
      <error message="" type="java.lang.NullPointerException">
<![CDATA[java.lang.NullPointerException
        at de.hybris.platform.jalo.Item.getCacheBoundItem(Item.java:3762)
        at de.hybris.platform.jalo.type.TypeManager.getComposedType(TypeManager.java:400)
        at de.hybris.platform.servicelayer.interceptor.impl.DefaultInterceptorRegistry.getAssignableTypes(DefaultIntercept
        .....
]]>
     </error>
   </testcase>
 </testuite>
</testsuites>
```

Mocking of Models

The SAP Commerce `Servicelayer` uses Models. Models can be mocked easily to be used in unit tests that can run completely without starting the SAP Commerce stack. This allows you to benefit from a very fast turnaround for tests of your business logic. The SAP Commerce software package contains test classes (MockTest) for your convenience.

Test Case Example

Example of a test case with mocked-up models:



- The test file:

  MockCatalogServiceTest.java

```
@ContextConfiguration(locations =
  { "classpath:/servicelayer/test/servicelayer-mock-base-test.xml",
    "classpath:/platformservices/test/platformservices-mock-catalog-test.xml" })

public class MockCatalogServiceTest extends MockTest
{
  @Resource(name = "catalogService")
  private CatalogService catalogService;

  @Test
  public void testGetCatalog()
  {
    final Collection<CatalogModel> catalogs = this.catalogService.getAllCatalogs();
    assertTrue("There must be two catalogs in the system!",
      (catalogs != null && catalogs.size() == 2));
  }
  ...
}
```

- Registering the DAO for the mock:

  platformservices-mock-catalog-test.xml

```
<bean id="defaultCatalogService"
  class="de.hybris.platform.catalog.impl.DefaultCatalogService"
  parent="abstractBusinessService" >
  <property name="catalogDao" ref="catalogDao"/>
  <property name="catalogVersionDao" ref="catalogVersionDao"/>
</bean>

<bean id="catalogDao"
      class="de.hybris.platform.catalog.CatalogMockDao"
       />
<bean id="catalogVersionDao"
      class="de.hybris.platform.catalog.CatalogVersionMockDao"
       />

<bean id="catalogService" class="de.hybris.platform.catalog.impl.DefaultCatalogService" .../>
  <property name="catalogDao" ref="mockCatalogDao"/>
</bean>

<bean id="catalogDao" class="de.hybris.platform.catalog.CatalogMockDao" ../>
```

- The DAO for the mocked test:

  CatalogMockDao

```
package de.hybris.platform.catalog;

import de.hybris.platform.catalog.impl.CatalogDao;
import de.hybris.platform.catalog.model.CatalogModel;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.List;

/**
 *
 */
public class CatalogMockDao implements CatalogDao
{

  public static final String DOESNOTEXIST = "doesnotexist";
  public static final String ONECATALOG = "onecatalog";
  public static final String TWOCATALOGS = "twocatalogs";

  @Override
```

```
        public Collection<CatalogModel> findCatalogs(final String id)
        {
          if (id == null)
          {
            final List<CatalogModel> allCatalogs = new ArrayList<CatalogModel>();

            final CatalogModel test0 = new CatalogModel("one");
            allCatalogs.add(test0);
            final CatalogModel test1 = new CatalogModel("two");
            allCatalogs.add(test1);

            return allCatalogs;
          }
          if (id.equals(DOESNOTEXIST))
          {
            return Collections.EMPTY_LIST;
          }
          else if (id.equals(ONECATALOG))
          {
            final List<CatalogModel> twoCatalogs = new ArrayList<CatalogModel>();

            final CatalogModel test0 = new CatalogModel(ONECATALOG);
            twoCatalogs.add(test0);

            return twoCatalogs;
          }
          else if (id.equals(TWOCATALOGS))
          {
            final List<CatalogModel> twoCatalogs = new ArrayList<CatalogModel>();

            final CatalogModel test0 = new CatalogModel(TWOCATALOGS);
            twoCatalogs.add(test0);
            twoCatalogs.add(test0);

            return twoCatalogs;
          }
          throw new UnsupportedOperationException("");
        }

      }
```

Related Information

Testing in SAP Commerce

TestThreadsHolder

Sometimes you may need to test some functionality running on many threads at once, to see if concurrent execution won't break anything.The **TestThreadsHolder** is a class that can help you with that.

To do that, create a new instance with a number of threads and a **Runnable** with a code that you want to execute. Additionally, you can set a flag that makes all threads aware of the tenant.

```
final TestThreadsHolder<Runnable> holderA1 = new TestThreadsHolder<>(10, someRunnable, true);
```

The class offers many support methods that help with concurrent testing.

To start all threads at once, use the **startAll()** method, and stop them using the **stopAll()** or **waitForAll(final long timeout, final TimeUnit unit)** methods.

When using **stopAll()**, please make sure that the tested logic is handling thread interrupts correctly.

To get all the errors that happened during execution, use **getErrors()** (it returns a map with a thread number key and the Throwable value).

The **getAlive()** method returns the current number of threads that didn't complete their testing logic. The **getStartToFinishMillis()** method returns the number of milliseconds that passed since all threads started until the time when all of them finished.

Testing with Spock

Spock is an optional framework for testing your code with Unit and Integration tests. SAP Commerce provides the libraries of the current groovy and spock versions, as well as specific SAP Commerce classes for easier testing.

ℹ Note

Before writing your first Spock test, it is recommended to read the Spock cookbook that explains the basic structure of the test class. For details, see http://spockframework.org/spock/docs/1.1/index.html .

To test SAP Commerce, you have to use special classes. You can distinguish two types of tests:

- **Unit test**: It doesn't require access to the database and you don't need the running SAP Commerce environment to execute it. For more details, see Writing Unit Tests.
- **Integration test**: It is also called functional, and needs access to SAP Commerce, which has to be started to execute the test. For more details, read Writing Integration Tests.

Your test classes should have the `Test` suffix in the name, and be stored in the following locations:

- `extensionname / testsrc`: Here you can test classes from the `extensionname / src` directory.
- `extensionname / web / testsrc`: Here you can test classes from the `extensionname / web / src` directory.

Writing Unit Tests

Follow the guidelines for writing unit tests.

When developing new software, a key programming practice is to write small, loosely coupled classes, with each one concentrating on one area of responsibility. Such classes are commonly referred to as Plain Old Java Objects, or POJOs. The `ServiceLayer` follows this coding principle, and therefore consists of many POJOs. Each POJO implements an interface, and is wired to other POJOs using Spring's Dependency Injection. This approach enables you to easily isolate a class for testing by replacing dependent POJOs with mocks as described below.

Unit tests each focus on single POJOs. Consider the three POJOs as seen in the example below - `POJO1`, `POJO2` and `POJO3`:

The Unit tests for `POJO1` should test and demonstrate the behavior of `POJO1` for expected and unexpected input. The Unit tests should not reference other dependent `POJOs`, such as `POJO2` and `POJO3`. These POJOs should be mocked, which means they should be simulated with correct behavior. By focusing the tests on `POJO1`, the tests are able to:

- Avoid false positives, for example, when a method in `POJO1` has an error, but an error in `POJO2` compensated for it and thus hid the error.
- Avoid false negatives, for example, when `POJO1` is working correctly, but an error in `POJO2` causes the test to fail.
- Run quickly, as `POJO2` or `POJO3` might have long-running methods, or even worse, require a SAP Commerce environment boot-up in which to run.

The main focus of Unit tests is to test the programmed code, and the content of the java class. The tests should check:

- Each public method of the class with valid parameters.
- The same methods with illegal parameters, like NULL or negative values or values where exceptions are expected.
- If possible the test should cover all possible values accepted in the method body. For example if the method contains an if-then-else statement then if clause should be tested with a true and a false value, to make sure it covers the then clause and the else clause.

Other conventions to keep in mind include:

- The Unit test class should be in the same package as `POJO1`, but is located in the test folder, typically `/testsrc`.
- The class name should be called `POJO1Test`.
- Each method should focus on testing one use-case, for example `"test find user with null"()`, `"test find user with incorrect case"()`.
- The test code, variable names, and method names are intuitively named in order for commenting not to be necessary.

Spock includes a mocking mechanism and it should be used to help isolate a POJO for Unit testing.

```
package de.hybris.platform

import org.junit.Test
import spock.lang.Specification


class MockedLinkedList extends Specification {
```

```
    @Test
    def "test mocking"()
    {
        setup:
        // In this example mock the behaviour of LinkedList.
        // First step is to get a reference to a mocked LinkedList, by calling Mock(Class).
        def mockedList = Mock(LinkedList)
            {
                // Tell the mock that if at a later stage mockedList.get(0) is called,
                // the real implementation should NOT called,
                // but instead the value "first" should be returned
                get(0) >> "first"

                // Tell mock that if at a later stage mockedList.get(1) is called,
                // the real implementation should NOT called,
                // but instead a RuntimeException should be thrown
                get(1) >> {throw new RuntimeException()}
            }

        // Now if you call mockedList.get(0), you will get the value "first"
        when:
        def first = mockedList.get(0)
        then:
        first == "first"

        //If you call mockedList.get(1), exception will be thrown
        when:
        mockedList.get(1)
        then:
        thrown(RuntimeException)

        // Call the method mockedList.get(999), which has not been recorded,
        // and therefore mock's default value is returned
        when:
        def defaultValue = mockedList.get(999)
        then:
        defaultValue == null
    }
}
```

Stubbing

To help you focus on testing a single POJO, you can replace any POJO on which it depends with stub implementation.

A stub object is used to replace more heavy-weight objects on which an object under test usually depends. You could, for example, mock a method in`POJO2` or `POJO3` and simply return hard-wired responses. This enables you to speed up tests due to removing a dependency on slow `POJOs`, and to avoid the necessity to connect to the database. For example, if you fake a DAO finder method in `POJO2`. Stubbing is state verification, because it verifies if the state of the POJO is as expected.

Using `Stub()`, you can stub out entire interfaces, entire classes, and parts of classes. The following code snippet presents different examples:

```
setup:
// In this example mock the behaviour of LinkedList.
// First step is to get a reference to a mocked LinkedList, by calling Mock(Class).
def mockedList = Spy(LinkedList)
    {
        // Tell the mock that if at a later stage mockedList.get(0) is called,
        // the real implementation should NOT called,
        // but instead the value "first" should be returned
        get(0) >> "first"

        // Tell mock that if at a later stage mockedList.get(1) is called,
        // the real implementation should NOT called,
        // but instead a RuntimeException should be thrown
        get(1) >> {throw new RuntimeException()}
```

```
    }

// Now if you call mockedList.get(0), you will get the value "first"
when:
def first = mockedList.get(0)
then:
first == "first"

//If you call mockedList.get(1), exception will be thrown
when:
mockedList.get(1)
then:
thrown(RuntimeException)

// Call the method mockedList.get(999), which has not been recorded,
// and therefore mock's default value is returned
when:
def defaultValue = mockedList.get(999)
then:
defaultValue == null
```

BDD Way of Stubbing

Behavior-driven development (BDD) focuses on obtaining a clear understanding of desired software behaviour through discussion with stakeholders. It extends Test-driven development (TDD) by writing test cases in a natural language that non-programmers can read. Behavior-driven developers use their native language in combination with the ubiquitous language of domain driven design to describe the purpose and benefit of their code. Spock requires usage of following blocks:

- setup
- given
- then
- expect
- cleanup
- where

Below is a sample test written in this manner. Notice the self-explanatory name of the test method with the prepending word, `should`.

```
@Test
def "should create simple date format object of given format for current locale"()
{
    setup:
    def format = "yyyy-mm-dd"
    def i18nService = Mock(I18NService)
            {
                getCurrentLocale() >> new Locale("en")
            }


    when:
    final DateFormat result = factory.createDateTimeFormat(format)

    then:
    reuslt != null
    SimpleDateFormat.isCase(result)
    1 * i18nService.getCurrentLocale()
}
```

The idea of writing BDD tests is to write small test cases that cover one behavior.

Mocking

Mocking helps to verify a conversation. For example, whether the sequence of methods calls between `POJO1` and its dependent POJOs is as expected. It is usually done hand-in-hand with stubbing.

Context

Mocking is a behavioral verification because it verifies whether the behavior of the POJO is as expected. To use Mock to perform the verification, follow the basic steps below:

Procedure

1. Create the `POJO2` mock and then wire this mock instead of the original `POJO2` in to `POJO1`.
2. Call method `A` from `POJO1`, which you expect to call methods `B` and `C` from `POJO2`.
3. Verify that `POJO1` did call methods `B` and `C` from `POJO2`.

   An example of verifying that calls to a mocked object are made:

```
@Test
def "verify mock"()
{
    setup:
    // Create a mock of List interface
    def mockedList = Mock(List)

    when:
    mockedList.add("one");
    mockedList.clear();

    then:
    // Verify that add and clear methods from List are called
    1 * mockedList.add("one")
    1 * mockedList.clear()
}
```

   Spock makes it easy to combine mocking and stubbing, which also enables you to perform state and behavioral tests on your services.

Parameterized Tests

Spock provides full support for parameterized tests.

The test data is provided by the `where:` block. Do not forget to put parameters into the test method. See example:

```
@Test
@Unroll
def "test square"(int parameter, int expected)
{
    when:
    def result = parameter * parameter

    then:
    result == expected

    where:
    parameter | expected
     1        |   1
    -2        |   4
     3        |   9
     95       |   9025
}
```

`@Unroll` annotation is not required, but useful. Without this annotation, spock treat all invocations as one test. With this annotation, spock separates foregoing test into four test suites.

Writing Integration Tests

SAP Commerce provides the possibility to use a preconfigured JUnit tenant where you can execute your tests isolated from other tenants. You can test your business functionality using real item instances.

Basic Mechanism

The easiest way to add specific functionality to your test class is to extend the `ServicelayerSpockSpecification` class. In this way, you gain access to methods that create some very basic test items that can be used for testing. By extending the `ServicelayerSpockSpecification` class, you may create instances of all basic SAP Commerce Models like language, currency, unit, and country using the `createCoreData` method. You may also import test data stored in a CSV file using the `importCsv` method, or import test data using the `importStream()` method. For more details, see the API documentation.

If you want to use other services in a test, you can inject them using the `@Resource` annotation. Use the `setup` block to execute code or import data before each test method. Do not use the setup block more than once in your test. All methods with this annotation are executed before each test method, but it is not clear in which order. To execute code or clean up data after each method, use the `cleanup` block. Do not use the `setupSpec` block as it is executed in a static way and is out of the test run scope. With this, a tenant is not set and changes will not be reverted.

```
class CreateUserTest extends ServicelayerSpockSpecification {
    @Resource
    private ModelService modelService;

    @Resource
    private UserService userService;

    @Test
    def "test create user"() {
        //create a user
        when:
        def user = new EmployeeModel();
        user.setUid("testUser");
        modelService.save(user);

        then:
        userService.getUserForUID("testUser") != null
    }
}
```

Preferred Mechanism

The preferred way of writing tests is to extend the `ServicelayerTransactionalSpockSpecification` class. This class adds transaction-based isolation logic to your tests by registering the `TransactionRunListener` listener. Before the `setup()` method of your test is called, a new transaction is started. After the `cleanup()` method of your test is called, a transaction rollback is performed. This assures complete isolation of your test. The `ServicelayerTransactionalSpockSpecification` class extends the `ServicelayerSpockSpecification` class, and therefore also provides the variables and methods of this class.

The following example demonstrates how to test a modification of the admin user. As the admin user already exists before the test starts, the modification will not be reverted when using the direct `ServicelayerTest` class, as only newly created items are tracked. By using the `ServicelayerTransactionalSpockSpecification` class, all changes will be reverted. Be aware that for testing logic using transactions with this test class can cause problems.

```
class ModifyAdminTest extends ServicelayerTransactionalSpockSpecification {
    @Resource
    private ModelService modelService;

    @Resource
    private UserService userService;

    @Test
    def "modify admin test"()
    {
        when:
        UserModel admin = userService.getAdminUser()
        admin.setName("test")
        modelService.save(admin)

        then:
        admin.getName() == "test"
    }
}
```

Running SAP Commerce Data Supplier

System registration for all products distributed by SAP is a demand of the SAP internal standard L216 Guidelines for Product Standard Operations and Support V10 (former IT/SAM standard). Data collected in SLD is used by different applications, for instance in the SAP Solution Manager. Here you will find the installation procedure for the SAP Commerce data supplier.

Installing Data Supplier

The `hybrisdatasupplier` extension is delivered with the SAP Commerce Platform.

1. Add the extension to `localextensions.xml`
2. Edit `HYBRIS_CONFIG_DIR/localextensions.xml` and add the `hybrisdatasupplier` and `hybrisdatasupplierbackoffice` extensions (see Installation Based on Specified Extensions for details). Both are located in `hybris\bin\modules\hybris-utilities` folder.
3. Install `SAPHostAgent`

   → Tip

   SAP host agent is used for the communication with the SLD system. The following steps show how it could be done but, for a real use case, please make sure that you do it according to your SAP system deployment.

   a. Download the `SapHostAgent` from the SAP page:

   - Go to **Support Packages & Patches** at http://support.sap.com/swdc and look for **SAP HOST AGENT**.

   b. Install the `SapHostAgent` according to the instructions inside the package.

   ⚠ Caution

   When the host agent is installed on a UNIX operating system, an environment variable *<LD_LIBRARY_PATH>* has to be set for successful communication with the SLD system. Not doing this leads to unexpected behavior which is difficult to troubleshoot or debug. The *<LD_LIBRARY_PATH>* should point to the correct path in the host agent directory tree, that is, "`/usr/sap/hostctrl/exe/`".

   c. After the installation process finishes, run the following command from the `SAPHostAgent` installation folder (on Windows machine it may be `C:\Program Files\sap\hostctrl\exe`). You will be prompted to give the SLD connection information.

   ```
   sldreg.exe -configure config_file_name.cfg -usekeyfile
   ```

Configuring Data Supplier

Specify the following properties in your `local.properties` file. Make sure you change the default values to your valid values.

`local.properties`

```
#FQDN for the database host. If empty it will be taken from the connection URL
datasupplier.database.fqdn=

#Command which runs sldreg
datasupplier.sldreg.exe.cmd=C:/Program Files/SAP/hostctrl/exe/sldreg.exe
#On linux it may be following
#datasupplier.sldreg.exe.cmd=/usr/sap/hostctrl/exe/sldreg

#Path to the sldreg configuration file.  The file which should be specified in the datasupplier.sldreg.config.path is gene
datasupplier.sldreg.config.path=C:/sldreg/sldtest.cfg
#On linux it may be following
#datasupplier.sldreg.config.path=/usr/sap/hostctrl/exe/sldtest.cfg

# Tomcat SHUTDOWN port - used by SLD to generate a unique id for this hybris installation - change if necessary
tomcat.internalserver.port=9009

#The ExtSIDName property in SLD. System's LONG SID to be used in SAP Solution Manager
#Up to 8-character identifier of the Apache Tomcat system.
#Note that this value can be set before first SLDDS run
#You can not change the SID for an existing  Tomcat system.
#However, you can change the SID with the the System Maintenance UI in the SAP Solution Manager
#The value should match "^[A-Z\d]{3}[A-Z\d_]{0,5}$". That's default value please set it properly
com.sap.sup.admin.sldsupplier.SYSTEM_ID = 0A0HY_00
```

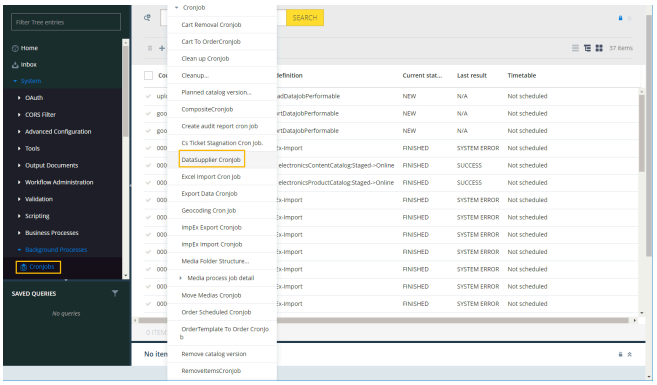
Updating the SAP Commerce Platform

1. Open the command line (windows cmd) in *<HYBRIS_BIN_DIR>*`/bin/`
   - run: `setantenv.bat`
   - run: `ant all`

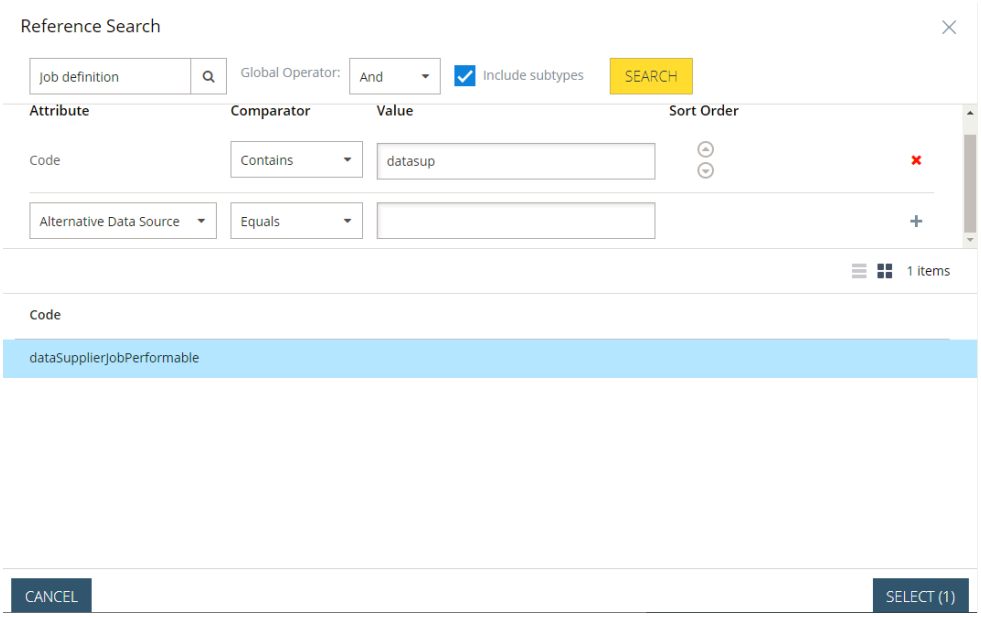
2. Go to SAP Commerce Administration Console and update Platform. If you want to have dataSupplierCronJob created, enable hybrisdatasupplier's projectdata on the update page, (if you do it, you can skip creating DataSupplier Cronjob in Backoffice, see next paragraph).

Creating and Running DataSupplier Cronjob

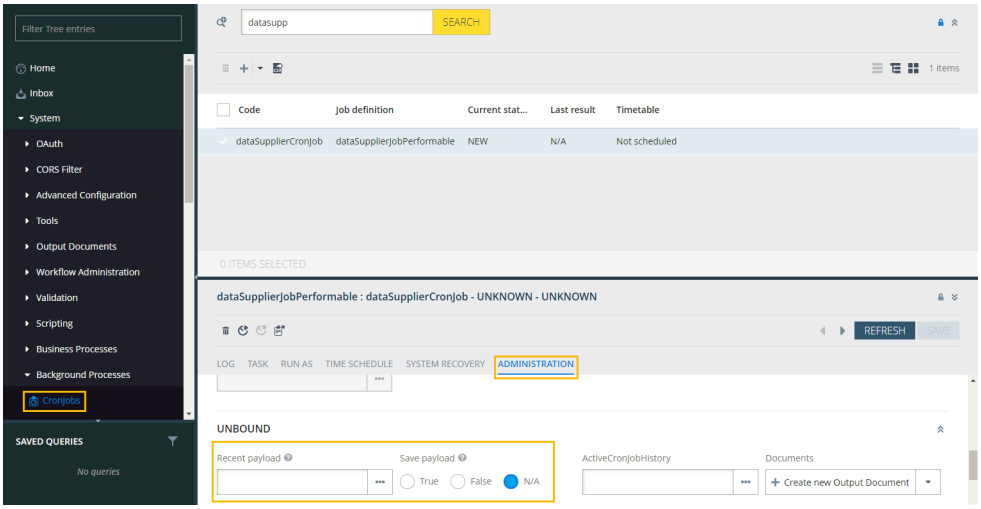
1. Create **DataSupplier CronJob** in Backoffice. If you have ticked projectdata during the Platform update, you can skip this point and go straight to point **b**.



Select dataSupplierJobPerformable as the job for the cronjob:



In the Administration tab, you can configure the system to save the generated payload. By default it is disabled; if you want to use it, set Save payload flag to Yes. Save the changes before running the cronjob.



2. Open dataSupplierCronJob (either imported from project data or the one which you created in the previous step) in Backoffice.
3. You can either run the cronJob manually by pressing Start CronJob now or you can define a trigger that runs it periodically. To define the trigger, go to Time Schedule in opened cronJob.
4. After the manual run, reload the cronJob, then you can check the generated xml with payload.
5. Look at the last result field in the cronjob. You will see that the state is either **Successful** or **Error:**

- Successful - data has been sent to SLD or SLD integration is disabled. Examine the logs for more information.
- Error - if there was a problem while generating or sending the payload to SLD.

Transactions

SAP Commerce allows you to use the transaction system of the underlying database through a simple-to-use API. You do not have to resort to looking up a transaction object from a JNDI tree. Simply employ begin, commit, and rollback statements in your code.

A transaction is a series of statements on a database that belong together in a certain context. Transactions are executed either in full or not at all, which keeps the database in a consistent state at any time.

When using the SAP Commerce API, every transaction needs explicit begin, commit, and rollback statements called from within the Java code. You can also use the declarative approach by using Spring Transactions. See the chapter below for more information.

Quick Start

Note

If you do not want to use the SAP Commerce API but use Spring Transactions instead, scroll down to the Spring chapter. But although you do not need to fully understand the SAP Commerce Transactional API it would be a good idea to go through all chapters to get an in-depth view.

Every thread in SAP Commerce has a **Transaction** object, which offers transaction functionality. To get this **Transaction** object, use the static **current()** method, such as:

```
Transaction tx = Transaction.current();
```

Getting the Transaction object does not trigger a transaction begin. You need to explicitly trigger transactions via the *begin()*, *commit()*, and *rollback()* methods.

The most basic implementation of a transaction in SAP Commerce looks as follows:

```
tx.begin();
boolean success = false;
try
{
   // do business logic
   doSomeBusinessLogic();
   success = true;
}
finally
{
   if( success )
      tx.commit();
   else
      tx.rollback();
}
```

However, it is recommended to avoid using explicit transaction operations whenever possible and use TransactionBody interface instead. By using TransactionBody one avoids all sort of problems that might arise when one uses given snippet (for example, forgetting to switch success flag to true or wrongly initializing it to true.)

Note

Always use **finally** blocks. Using the **finally** operator here for rollback is an important aspect. During transaction execution, there may be issues which do not result in an **Exception**, but in an **Error**. Errors are not intercepted by a typically used **catch(Exception)** statement. If such an error arose during a transaction, the **Transaction.current().commit();** statement would not be executed. The (incomplete) transaction would remain open, which blocks resources and impacts performance.

The **finally** statement, however, will be executed if **Errors** arise, and can roll back the transaction. Then the database is in a consistent state again, and the connection to the database closes.

Be aware that since there is only one single Transaction object for every single thread, using several instances of the Transaction object does not result in different actual Transaction objects. For example, in the following code snippet, **tx1** and **tx2** are the same instance of the Transaction object.

Do not use the following method:

```
Transaction tx1 = Transaction.current();
Transaction tx2 = Transaction.current();  //RETURNS THE SAME OBJECT AS TX1
```

```
tx1.begin();
tx2.begin(); //CALLS BEGIN ON AN ALREADY RUNNING TX
```

As there is only one Transaction object per thread, the two following code snippets are effectively the same:

- ```
  Transaction tx = Transaction.current();
  tx.begin();
  ...
  tx.commit();
  ```

- ```
  Transaction.current().begin();
  ..
  Transaction.current().commit();
  ```

Whether you use the first or the second style of code does not matter.

Although both conventions produce the same results, introducing local variable to refer to current transaction usually leads to better coding style, because this makes transactional statements more explicit and "local" - the scope of the variable determines the scope of possible transactional operations. This is important when transactional code becomes more complicated.

Using the SAP Commerce Transaction API

Nested Transactions

A side effect of the fact that there is only one Transaction object per thread is that it does not matter whether a submethod call opens a transaction of its own (that is, a nested transaction). The nested transaction does not start an individual transaction, but is executed in the context of the transaction of the calling method. Even if the underlying database system does not actually support nested transactions, SAP Commerce still accepts nested transactions.

```
void doSomethingBig()
  {
     ..begin();
     doSomethingInTX();
     ..commit();         //commit executed because not nested
  }

  void doSomethingInTX()
  {
      ...begin();        //begins a nested TX
                         //will NOT throw error because tx is already running
      do()
      ...
      commit();         //commit ignored because inside nested TX
  }
```

Using TransactionBody

SAP Commerce also offers an encapsulation for the Transaction object **begin()**, **commit()**, and **rollback()** methods. By instantiating a **TransactionBody** object and overriding the **TransactionBody execute()** method, you can encapsulate the entire transaction process, as in the following code snippet:

```
Transaction.current().execute( new TransactionBody()
   {
      public Object execute()
      {
         doSomething();
         return something;
      }
   }
);
```

Being wrapped in the **execute()** method of the **TransactionBody** object, the **doSomething()** method is executed in a transaction.

Delayed Store

To optimize transaction speed and reduce database load, SAP Commerce offers a delayed-store mechanism. By default, SAP Commerce executes every statement as soon as it comes up. If "Delayed-store" is activated all database modification statements done by a transaction are kept in memory until the transaction is actually flushed and then executed at once. This allows optimizing (and possibly reducing) the set of database modification statements.

The disadvantage is that since all of the database modification statements are executed at once, no data changes take place until the transaction is actually flushed. This may, for example, cause FlexibleSearch statements to return obsolete values because the database still returns old and stale content.

To flush transaction in delayed-store mode, call the **Transaction** object **flushDelayedStore()** method, such as:

```
tx.flushDelayedStore();
```

There are two ways of setting whether or not to use delayed-store. First, there is the **transaction.delayedstore** property in the `project.properties` file that sets the global default value for delayed-store transactions. In addition, you can explicitly enable or disable delayed-store transactions via the **Transaction** object `enableDelayedStore(Boolean)` method. The value of the `delayed-store transactions(Boolean)` method is valid until the transaction is committed (or rolled back, depending on the outcome), and is reset to the global default afterwards.

For both the property and the `enableDelayedStore(Boolean)` method, **true** enables delayed-store transactions, while **false** disables delayed-store transactions.

```
Transaction.current().begin();
Transaction.current().enableDelayedStore(false);
...
```

Transaction Isolation

READ_COMMITTED is the transaction isolation level used by transactions in SAP Commerce. Oracle databases use READ_COMMITTED by default. The Microsoft SQL Server 2005 uses READ_COMMITTED by default. For additional information, see Caching.

Intra Transaction Cache

It is possible to cache data manually for each transaction. Use the `Transaction.set/getContextEntry(Object, Object)` for storing data only for one transaction. Note that this data is reset to null whenever commit() or rollback() is called.

```
Transaction.current().setContextEntry("mykey","myvalue");
```

Transactions and JUnit Tests

If executing JUnit-Tests, it might be useful to roll back all changes at the end of the test to retain a fresh system and remove all data that was created during the test. For more information on how to write JUnit-Tests with SAP Commerce, see Testing with JUnit.

```
public class MyTest extends HybrisJUnit4TransactionalTest
{
   @Test
   public void hereIsATest()
   {
       createProduct();
       checkIfThereIsOneProductInTheSystem();
   }
}
```

There is no need to remove the product because all data is removed automatically at the end of each test.

Using Transactions via Spring

The `HybrisTransactionManager` implements the Spring `PlatformTransactionManager` registered at the core application context delegating all transaction calls to the SAP Commerce transaction framework. With that you are able to use the Spring Transaction Framework by means of using the **@Transaction** annotation, AOP-style transaction declaration (using tx schema) or by using the TransactionTemplate.

The transaction template mechanism can be used by:

1. Inject or get the configured transaction manager
2. Instantiate a new template
3. Call the template execute method by defining a new body

```
// 1.
PlatformTransactionManager manager=(PlatformTransactionManager) Registry.getApplicationContext().getBean("txManager");
// 2.
TransactionTemplate template = new TransactionTemplate(manager);
// 3.
template.execute(new TransactionCallbackWithoutResult()
{
  @Override
  protected void doInTransactionWithoutResult(final TransactionStatus status)
  {
    // do something transactional
  }
});
```

The AOP-style declaration can be made similar to the following xml snippet at your application-context.xml. Assuming that you have a FooService and you want to declare all getter as read-only transactional methods and all others as read and write ones, then declare it like that:

```
<bean id="fooService" class="x.y.service.DefaultFooService"/>
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
<aop:config>
  <aop:pointcut id="fooServiceOperation" expression="execution(* x.y.service.FooService.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
</aop:config>
```

You have to define the tx and aop scheme at XML header.

Frequently Asked Questions (FAQ)

Does SAP Commerce use the UserTransaction Interface?

No.

Can we participate in global transactions of multiple systems?

No.

Resilience Against Connection Errors

The following property determines how many times transactions retry to connect with the database. If the connection isn't created after the number of attempts specified by this property, the retry mechanism is stopped.

```
transaction.connection.retryOnBind.count=1
```

Related Information

Clustered Environment

Workflow and Collaboration

The workflow and collaboration tools rely on the `workflow` and `comments` extensions.

Modeling a Workflow Using the API
To create the representation of workflows, you can use the SAP Commerce system API.

Processing an Action Using the API
The SAP Commerce `workflow` extension allows modeling in-house business processes for companies.

Setting Backoffice Access Rights
The SAP Commerce `workflow` extension relies on the Backoffice Framework for user interaction.

Related Information

comments Extension
workflow Extension

Modeling a Workflow Using the API

To create the representation of workflows, you can use the SAP Commerce system API.

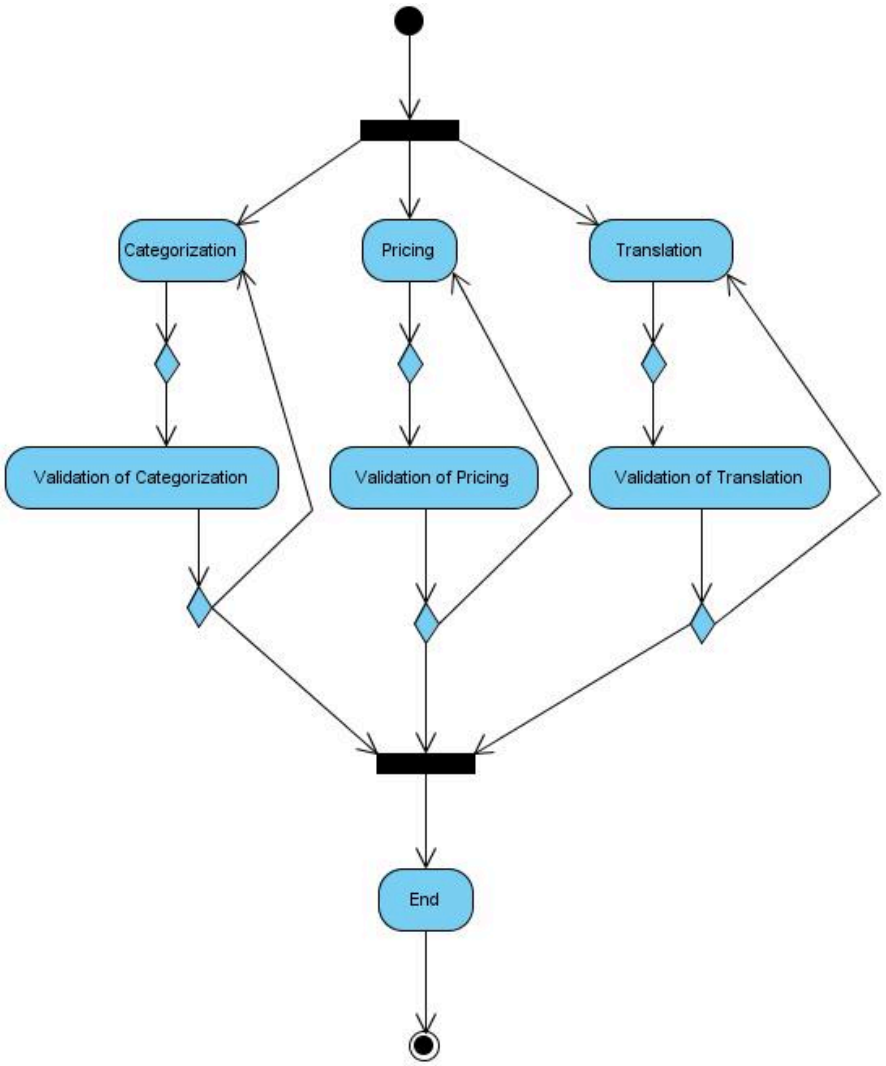
The procedure of modeling a workflow assumes you know the workflow you want to model. To make modeling workflows easier, SAP Commerce recommends creating a sketch of the workflows first.

Once the workflow sketches are done, it's quite easy to represent the workflows in the SAP Commerce `workflow` extension.

Modeling a Workflow

To create a workflow you have to model it using a workflow template. The best practice would be to draw the workflow you want to model before. Modeling workflows directly in the `workflow` extension without setting up a sketch beforehand is not an easy task.

The workflow modeled in this example looks like this:



This workflow consists of three sub-processes each of which consists of two actions:

- **Categorization** and **Validation of Categorization**
- **Pricing** and **Validation of Pricing**
- **Translation** and **Validation of Translation**

When the first action of each sub-process is completed (**Pricing**, for example), the upcoming action (**Validation of Pricing**, in that case) will be a validation of the previous action's result. If the result is not approved, the first action will be re-activated. If the result is approved, the workflow will be able to proceed to the next phase. If and only if all three sub-processes have passed their respective validation action, the workflow is completed.

Modeling a Workflow via API

1. A workflow template can then be used for creating as many workflow instances as needed.

```
final WorkflowTemplateModel workflowTemplate = modelService.create(WorkflowTemplateModel.class);
                            workflowTemplate.setOwner(userService.getUserForUID("admin"));
                            workflowTemplate.setCode("SampleWorkflow");
                            workflowTemplate.setName("New Product", Locale.ENGLISH);
                            workflowTemplate.setName("Neues Produkt", Locale.GERMAN);
                            workflowTemplate.setDescription("This is an examplary workflow template which would be s
                            workflowTemplate.setDescription("Dies ist eine exemplarische Vorlage für einen Workflow,
                            modelService.save(workflowTemplate);
```

2. Next, the action templates should be added to the workflow template. The code for creating an action template looks like this:

```
final WorkflowActionTemplateModel workflowActionTemplate = modelService.create(WorkflowActionTemplateModel.class);
                            workflowActionTemplate.setOwner(userService.getUserForUID("admin"));
                            workflowActionTemplate.setCode("SampleAction1");
```

```
workflowActionTemplate.setPrincipalAssigned(userService.getUserForUID("productmanager_wf
workflowActionTemplate.setWorkflow(workflowTemplate);
workflowActionTemplate.setSendEmail(Boolean.TRUE);
workflowActionTemplate.setEmailAddress("productmanagergroup@hybris.de");
workflowActionTemplate.setName("Validation of Categorization", Locale.ENGLISH);
workflowActionTemplate.setName("Validierung der Kategorisierung", Locale.GERMAN);
workflowActionTemplate.setDescription("Please assign the attached product to the categor
workflowActionTemplate.setDescription("Bitte ordnen Sie das angehängte Produkt in den Ka
workflowActionTemplate.setActionType(WorkflowActionType.NORMAL);
modelService.save(workflowActionTemplate);
```

3. Each action template needs decisions that refer to the next action template:

```
// create a decision
final WorkflowDecisionTemplateModel decision1 = modelService.create(WorkflowDecisionTemp
decision1.setCode("SampleDecision1");
decision1.setActionTemplate(workflowActionTemplate);
decision1.setName("finished validating the categorization", Locale.ENGLISH);
decision1.setName("Validierung der Kategorisierung beendet", Locale.GERMAN);
modelService.save(decision1);
decision1.setToTemplateActions(Collections.singletonList(workflowActionTemplate2));

//set AND-Connection
workflowProcessingService.setAndConnectionBetweenActionAndDecision(decision1, workflowAc
```

4. After modeling the workflow you can create a real instance. Assumed a product with the code **sample_product** was added to the platform , then you need to create such a workflow using the following code:

```
final WorkflowModel workflow = workflowService.createWorkflow(workflowTemplate, userService.getUserForUID("productma

final ProductModel product = ...

final WorkflowItemAttachmentModel attachment = modelService.create(WorkflowItemAttachmen
attachment.setItem(product);
attachment.setWorkflow(workflow);
attachment.setCode("SampleProductAtt");
workflow.setAttachments(Collections.singletonList(attachment));

workflowProcessingService.startWorkflow(workflow);
```

The first code line creates a workflow with all preconfigured actions using the template. All other lines except the last one will create an attachment holding the new product and setting it to the workflow.

By executing the last line, the workflow processing will be triggered. That means all actions of action type **start** will be activated. Now the workflow as well as the start actions are active and wait for user interaction. The triggering has to be done explicitly when using the API.

Modeling a Workflow via ImpEx

A workflow template can also be created via ImpEx.

```
INSERT_UPDATE WorkflowTemplate;code[unique=true];name[lang=de];name[lang=en];owner(uid);description[lang=de];description[l
                    ;SampleWorkflow;Produkt Workflow;Product Workflow;admin;Dies ist eine exemplarische Vorlage für einen
```

The field **owner** is a mandatory field when creating a workflow template. It is also recommended to set a **code** because when creating an action template for a workflow template you have refer to the workflow template in some way.

This is how an action template is created:

```
INSERT_UPDATE WorkflowActionTemplate;code[unique=true];name[lang=de];name[lang=en];description[lang=de];description[lang=e
                    ;SampleAction1;Kategorisierung;Categorization;Bitte ordnen Sie das angehängte Produkt in den Kategorie
```

This ImpEx statement has the following mandatory fields:

- `actionType`
- `principalAssigned`
- `sendEmail`
- `workflow`

As with the workflow template it is recommended to set the **code** for the action template because it is needed for creating decision templates.

A decision template is created like this:

```
INSERT_UPDATE WorkflowDecisionTemplate;code[unique=true];name[lang=de];name[lang=en];actionTemplate(code);description[lang
                    ;SampleDecision1;Kategorisierung beendet;finished Categorization;SampleAction1;The categorization of t
```

This ImpEx statement has the following mandatory fields:

- `actionTemplate`

Also, it is recommended to set the **code** for the decision template because it is needed for creating the link between the decision templates and its successive action templates.

Finally you can set the link from the decision to the next action:

```
insert_update WorkflowActionTemplateLinkTemplateRelation;source(code)[unique=true];target(code)[unique=true];andConnectior
                    ;SampleDecision1;SampleAction1;false;WorkflowActionTemplateLinkTemplateRelation
```

This ImpEx statement has the following mandatory fields:

- `source`
- `target`
- `qualifier`

Processing an Action Using the API

The SAP Commerce `workflow` extension allows modeling in-house business processes for companies.

Business processes in which customers are involved are only covered where an in-house employee has to do with customer input. For example, the SAP Commerce `workflow` extension allows modeling the following business processes:

- Handle customer complaints
- Create and put live a website paragraph
- Rewrite and review a product description

Processing an Action via API

If the workflow is started, you have to know which actions are active and have to be processed. Either you know that there is a workflow where an action has to be performed (caused by mail notification, for example) or you just do a search for all actions you are assigned to and where the status is active (as used for the inbox of Backoffice):

```
final Map params = new HashMap();
params.put("status", WorkflowActionStatus.IN_PROGRESS);
params.put("user", userService.getUserForUID("productmanager_wf1"));

final String query = "SELECT tbl.action FROM ("
                              + "{{SELECT {actions:PK} action FROM {WORKFLOWACTION as actions} WHERE {actions:status}=?s
                              + " UNION ALL "
                              + "{{SELECT {actions:PK} action FROM {WORKFLOWACTION as actions}, {PrincipalGroupRelation
                              + "{actions:principalAssigned}={rel:target} AND {rel:source} = ?user}}" + ") tbl";

final List<WorkflowActionModel> actions = flexibleSearchService.search(query, params).getResult();
```

So if you have an active action, you can access all attributes as usual and complete the action finally through the following code:

```
workflowProcessingService.decideAction(action1, decision1);
```

This line sets the selected decision for that action as completed and the second one starts the activation of the selected decision.

Setting Backoffice Access Rights

The SAP Commerce `workflow` extension relies on the Backoffice Framework for user interaction.

Setting access rights using Backoffice

The Backoffice Framework allows defining specific access rights for individual users. For more information, refer to Permissions in The Backoffice Framework.

Setting access rights using impex

The following code sample shows how to set Backoffice access rights related to the SAP Commerce `workflow` extension:

```
$START_USERRIGHTS
Type;UID;MemberOfGroups;Password;Target;read;change;create;remove;change_perm;;
UserGroup;productmanagergroup;employeegroup;;;;;;;;;
;;;;WorkflowTemplate;+;;;;;;
;;;;Workflow;+;;;;;;
;;;;AbstractWorkflowAction;+;;;;;;
;;;;WorkflowActionTemplate;+;;;;;;

;;;;WorkflowAction;;;;;-;;
;;;;WorkflowAction.name;+;;;;;;
;;;;WorkflowAction.attachments;+;;;;;;

;;;;WorkflowItemAttachment;+;;;;;;
;;;;WorkflowActionStatus;+;;;;;;
;;;;WorkflowActionComment ;;;+;;;;
;;;;WorkflowDecision ;+;;;;;;
;;;;WorkflowActionLinkRelation ;+;;;;;;

;;;;InboxView;+;;;;;;
$END_USERRIGHTS
```

Related Information

ImpEx

Users in Platform

Permissions in The Backoffice Framework