Services

Exercise

SAP Customer Experience

TABLE OF CONTENTS

GOAL

This exercise covers the SAP Commerce Cloud service layer. You will learn how to create a service containing business logic pertaining to the Customer Account and Consent Management.

INSTRUCTIONS

Preparation

Begin by running the **setup** Ant target to create a new extension with **trainingservices** as its name:

P1 If you haven't done so in the current *Terminal* or *cmd* window, set Ant home by navigating to the `MYPATH/workspace/hybris/bin/platform` directory and executing:

`. ./setantenv.sh` (on MacOS or Linux) or `setantenv.bat` (on Windows).

P2 Then navigate to `MYPATH/workspace/TrainingLabTools/exercise_Services` and execute: `ant -f trainingservices_tasks.xml setup`

P3 Now, in order to see the new extension, you need to add it to the Eclipse Workspace. You can import it manually, or you can go to the *SAP* menu and select *Synchronize Projects with yPlatform*.

Note, if you see a small error sign on the trainingservices extension in eclipse, it's actually not so relevant. But if you insist on having it gone, you can select the eclipse menu bar item: SAP | *Synchronize Projects with yPlatform* again.

P4 By the way, if you open:

`trainingservices/src/my/commerce/trainingservices/service/impl/DefaultCustomizedCustomerAccountService.java`

you might notice some import errors.

To fix this, navigate to `MYPATH/workspace/hybris/bin/platform` and execute: `ant all`

After the successful build, please refresh the "platform" project (not the trainingservices project) in eclipse to reload the model classed needed in the DefaultCustomizedCustomerAccountService.java.

You don't have to wait until the build is successful, you can proceed with the exercise.

Your tasks are outlined in the **trainingservices** extension java files, `CustomizedCustomerAccountService`, `DefaultCustomizedCustomerAccountService`, and configuration file `trainingservices-spring.xml`, located here:

`MYPATH/workspace/hybris/bin/custom/trainingservices/src/my/commerce/trainingservices/service/CustomizedCustomerAccountService.java`

`MYPATH/workspace/hybris/bin/custom/trainingservices/src/my/commerce/trainingservices/service/impl/DefaultCustomizedCustomerAccountService.java`

`MYPATH/workspace/hybris/bin/custom/trainingservices/resources/trainingservices-spring.xml`

Java Build Path problems in Eclipse? ⚠

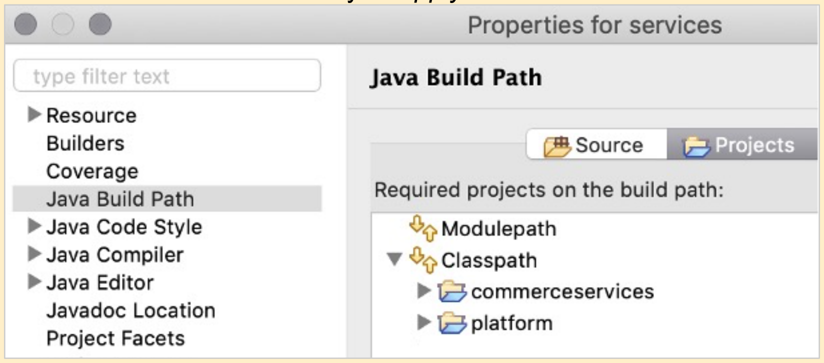
Remember that if you want to build code in Eclipse you may need to manually setup your Java Build Path to allow your new project to find all relevant class files.

But you can also use the SAP commerce cloud plugin to automatically setup the Build Path by clicking on *Synchronize Projects with yPlatform*.

To manually setup the Build Path, right click your project, select *Build Path* >> *Configure Build Path*… and add projects:

commerceservices and **platform** by clicking *Add…*

It should look like this before you *Apply and Close*:



Required projects on the build path:

```
Modulepath
Classpath
  commerceservices
  platform
```

Step 1 • Configure a Spring bean for our customized CustomerAccountService

Examine the interface *my.commerce.trainingservices.service.***CustomizedCustomerAccountService** and its implementing class, *my.commerce.trainingservices.service.impl.***DefaultCustomizedCustomerAccountService**.

Just as the **CustomizedCustomerAccountService** interface extends the *de.hybris.platform.commerceservices.customer.***CustomerAccountService** interface, we want our new implementation class to reuse, that is to say, inherit, an existing bean's configuration (use the bean **defaultCustomerAccountService**, defined in commerceservices-spring.xml).

Doing this will conveniently let our new service bean *inherit* the existing bean's configuration as a starting point. Mainly, we want to avoid having to manually copy over every injected dependency at the risk of inconsistency.

And because we're extending the behavior of the OOTB customer account service, we want to define an *alias* that overrides the **customerAccountService** alias (also defined in commerceservices-spring.xml) to now link to our newly defined **defaultCustomizedCustomerAccountService** bean.

1.1 Inside `trainingservices-spring.xml`, uncomment the alias **customerAccountService** and bean **defaultCustomizedCustomerAccountService** definition and make the appropriate changes to the definition of the **defaultCustomizedCustomerAccountService** bean so that it inherits the **defaultCustomerAccountService** bean's configuration. (Remember: use the **parent** attribute to specify inheritance of bean configurations in Spring.)

Then add or inject a dependency to the **commerceConsentService** bean as a property. The injected **commerceConsentService** service will be necessary for the exercise implementation steps.

Hint: You need to define a *parent* attribute in this bean and set a property reference in `trainingservices-spring.xml`. If you have trouble understanding how to do it, take a peek at the solution within `MYPATH/workspace/TrainingLabTools/exercise_Services/solution/trainingservices-spring.xml`

✓

To save time and reduce confusion about what we have or haven't already done for you, here's the standard checklist (by now you've probably seen this pattern numerous times):

If it doesn't already exist in `trainingservices-spring.xml`, define a **defaultCustomizedCustomerAccountService** bean so that you can…

- Inject (within this bean definition) a reference to the **commerceConsentService** bean using a new bean property (named accordingly).
- Enable this bean injection by modifying the **defaultCustomizedCustomerAccountService** bean implementation class as follows:
  - Add an appropriately named and typed *private member variable* to hold the injected reference to the class.
  - Create a getter/setter method pair to implicitly define a bean property (according to naming conventions), and to provide controlled access to the new private member variable.

    Eclipse can actually generate this getter/setter method pair for you: Right-click within your class' Java editor, then follow *Source→Generate Getters and Setters…*

Note: Even if this seems complicated at first, nothing in this checklist is special – all of it just follows the standard Spring DI (dependency-injection) pattern, and it will very quickly become routine.

Step 2 • Complete the implementation of business logic

You are now ready to code the service implementation class. Complete the implementation of the partially written method *getAllActiveConsents()* in **DefaultCustomizedCustomerAccountService**. Please examine the method's signature to better understand the incoming method parameters and return type. The main purpose of the method is to return the active consents for a given Customer and **BaseSite**.

What is a **BaseSite**?

A **BaseSite** is assigned to one or more stores, known as base stores, which define the product catalogs available on the website. You can treat BaseStore as a physical shop, and a **BaseSite** as a reference to the website serving as that shop's front end. We will provide more details later, in the WCMS lesson.

If you notice some import errors in **DefaultCustomizedCustomerAccountService**, this is already mentioned in the preparation step P4, please fix it accordingly.

2.1 Use your injected **CommerceConsentService** to retrieve a complete list of all **ConsentTemplates** defined for the given BaseSite.

What is a **ConsentTemplate**?

A **ConsentTemplate** defines the consent content displayed to a customer in the storefront, e.g., whether the customer agrees to receive marketing emails or not. A storefront can have multiple **ConsentTemplates** defined for different purposes.

If a customer agrees (gives consent), a **ConsentModel** object is created and attached to that customer; it records the status of the consent (a customer can later withdraw their consent).

Next, for each **ConsentTemplate**, use the same service again to obtain the active **ConsentModel** (and verify whether its status is "active") for the provided Customer. Use a loop for this task and collect all active-status **ConsentModel** objects in the already provided *activeConsents* List variable.

CLARIFICATION: The second **commerceConsentService** method you will call is named **getActiveConsent()**, which is unfortunately very misleading. It gives the (incorrect) impression that it returns a **ConsentModel** object whose **isActive()** status is always TRUE. Instead, the method returns the ***most recent*** (i.e., ***active*** in the sense that it is the most *current*) **ConsentModel** object for the supplied parameters. You must then call the returned ConsentModel's **isActive()** method to check its status to determine whether or not to add it to the *activeConsents* List variable.

Hint: If you have trouble completing the method, have a quick look at the solution here:
`MYPATH/workspace/TrainingLabTools/exercise_Services/solution/DefaultCustomizedCustomerAccountService`

Step 3 • Override the inherited closeAccount() method

3.1 Override the *closeAccount* method inherited from **DefaultCustomerAccountService** in **DefaultCustomizedCustomerAccountService** class. Your method will withdraw all the active consents that the customer had given before closing the account (a mocked business requirement). You can leverage the method implemented in step 2.

3.2 To find out the current **BaseSite**, you can use the method *getBaseSiteService().getCurrentBaseSite()* inherited from **DefaultCustomerAccountService**. Then, iterate over all the active **ConsentModel** objects and use the **CommerceConsentService** to withdraw consent from each one.

3.3 When done, make sure that you also call the overridden (super) method to close the account for this customer (which returns the modified **CustomerModel**).

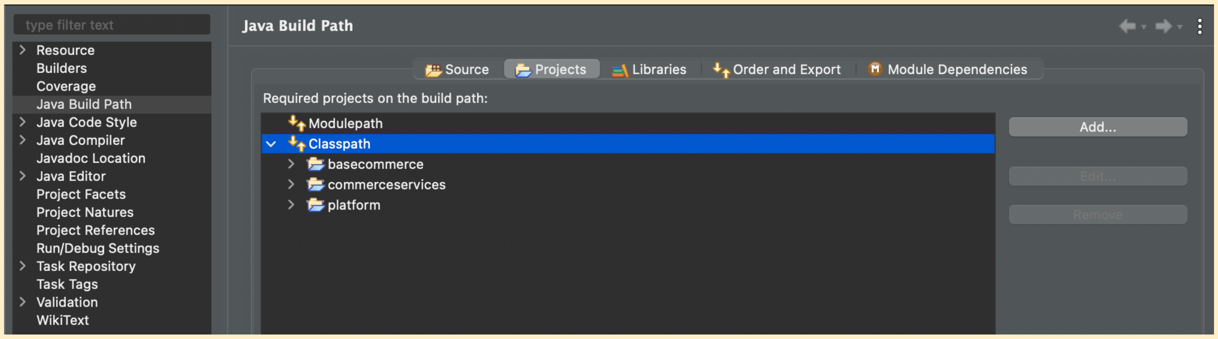
Java Classpath problems?

To use the BaseSiteService in Eclipse, you will need to add the basecommerce extension to the Classpath of the trainingservices extension. (Remember that Eclipse thinks of extensions as *projects*.)

Remember that Eclipse may require you to set up your Java Build Path manually so that your new project will find all relevant class files.

Right click your trainingservices project, select *Build Path* >> *Configure Build Path*… and select *Projects* tab, there you select *Add*… and add **basecommerce** to your Classpath.

It should look like this before you *Apply and Close*:



Required projects on the build path:

```
Modulepath
Classpath
  basecommerce
  commerceservices
  platform
```

Hint: If you have difficulty completing the implementation, please have a quick look at the solution here: `MYPATH/workspace/TrainingLabTools/exercise_Services/solution/DefaultCustomizedCustomerAccountService`

Verify

V1 Recompilation is necessary for your code changes to be used by the SAP Commerce Cloud platform environment. To compile, stop the SAP Commerce Cloud Platform, run an `ant all`, and once again start the SAP Commerce server.

V2 To verify your solution, go to **HAC** and run the script `verifyServicesExercise`. Please note that the verification script does check the business logic, it will test not only the existence of the **required** methods but also the implementation.

Solution

If you don't wish to complete this exercise manually, you can install the solution provided:

S1 Navigate to the *Terminal* or *cmd* window where the server is running, and if it is running, stop it by entering `CTRL-C`.

S2 Navigate to `MYPATH/workspace/TrainingLabTools/exercise_Services` and execute:

```
ant -f trainingservices_tasks.xml solution
```

RECAP

In this exercise, you learned how to extend an existing service with new functionality, and how to override the standard functionality with your own implementation. We also used two standard commerce services:

- **DefaultCustomerAccountService**, which provides customer self-service functionality, and
- **DefaultCommerceConsentService**, which articulates consent operations for a customer.

www.sap.com

SAP Customer Experience