Commerce

Generated on: 2024-12-03 11:09:38 GMT+0000

SAP Commerce | 2205

Public

Original content: https://help.sap.com/docs/SAP_COMMERCE/9d346683b0084da2938be8a285c0c27a?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.

Extend Commerce Services

Commerce Services can be extended by creating an AddOn with one additional `Controller` class and using appropriate methods. However, in most cases some more complex changes are required to build an AddOn which adds the functionality to Commerce Services.

This document describes how to successfully extend Commerce Services.

Note

For details on how to create an AddOn for OCC Web Services refer to: Creating an AddOn for OCC Web Services.

Extending process is described based on the `occaddon`. It is a sample AddOn that extends Commerce Services. The `occaddon` adds new properties to the `Customer` item type:

- `nickname`
- `workOfficeAddress`

A REST call allows you to set these properties. In the `occaddon,` there is also an example of how to override an existing request mapping.

Extending Data Objects

You can extend the data model and data transfer object for Commerce Services using an AddOn.

Extending the Data Model

Procedure

1. In the `extensionname-items.xml` file, replace the `extensionname` part in the file with the name of the actual extension.

   The sample AddOn extends the `CustomerModel` class with two properties:

   - `nickname`
   - `workOfficeAddress`

   Note

   For more details, see items.xml.

2. Verify that you see the correct results in the `*-items.xml` file.

   The `*-items.xml` file looks like the following sample.

   `occaddon-items.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<items  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        xsi:noNamespaceSchemaLocation="items.xsd">
        <itemtypes>
                <typegroup name="Customer">
                        <itemtype code="Customer" autocreate="false" generate="false">
                                <description>Extending Customer type with additional attributes.</description>
                                <attributes>
                                        <attribute autocreate="true" qualifier="nickname" type="java.lang.String">
                                                <modifiers read="true" write="true" optional="true" />
                                                <persistence type="property" />
                                                <description>Customer nickname</description>
                                        </attribute>
                            <attribute autocreate="true" qualifier="workOfficeAddress" type="Address" isSelectionOf="addresses">
                           <modifiers read="true" write="true" search="false" optional="true" />
                               <persistence type="property" qualifier="WorkOfficeAddress"/>
                        </attribute>
                                </attributes>
                        </itemtype>
                </typegroup>
        </itemtypes>
 </items>
```

Extending the Data Transfer Object

Procedure

1. In the `extensionname-beans.xml` file, replace the *<extensionname>* part with the name of the current extension.

   The sample AddOn extends the `CustomerData` with two properties:

- `nickname`
- `workOfficeAddress`

> **Note**
>
> For details on generating custom Java Beans, see Generating Beans and Enums.

2. Verify that you see the correct results in the `*-beans.xml` file.

`occaddon-beans.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="beans.xsd">
        <!-- Additional nickname and workofficeAddress property for CustomerData -->
        <bean class="de.hybris.platform.commercefacades.user.data.CustomerData" extends="de.hybris.platform.commercefacades.user
                <property name="nickname" type="String" />
                <property name="workOfficeAddress" type="de.hybris.platform.commercefacades.user.data.AddressData"/>
        </bean>
</beans>
```

Populating Data

Context

The `Converter` objects in the *<commerce>-** extensions are configured in Spring only and have no separate concrete implementation classes. During the conversion process, they use a list of `Populator` objects. It is possible to add a `Populator` to an existing `Converter` without having to redeclare the `Converter`. Use the `modifyPopulatorList` method.

This procedure provides the steps to populate new attributes.

Procedure

1. Define the Populator Class in `OccaddonCustomerPopulator.java`.

```
public class OccaddonCustomerPopulator implements Populator<CustomerModel, CustomerData>
{
    private Converter<AddressModel, AddressData> addressConverter;
    @Override
    public void populate(final CustomerModel source, final CustomerData target) throws ConversionException
    {
        Assert.notNull(source, "Parameter source cannot be null.");
        Assert.notNull(target, "Parameter target cannot be null.");
        target.setNickname(source.getNickname());
        if (source.getWorkOfficeAddress() != null)
        {
            target.setWorkOfficeAddress(getAddressConverter().convert(source.getWorkOfficeAddress()));
        }
    }
    protected Converter<AddressModel, AddressData> getAddressConverter()
    {
        return addressConverter;
    }
    @Required
    public void setAddressConverter(final Converter<AddressModel, AddressData> addressConverter)
    {
        this.addressConverter = addressConverter;
    }
    }
```

2. Add the Populator to the Converter in `occaddon-spring.xml`.

```
...
        <alias name="defaultOccaddonCustomerPopulator" alias="occaddonCustomerPopulator"/>
        <bean id="defaultOccaddonCustomerPopulator" class="de.hybris.platform.occaddon.customer.converters.populator.OccaddonCus
                <property name="addressConverter" ref="addressConverter"/>
        </bean>

        <bean parent="modifyPopulatorList">
                <property name="list" ref="customerConverter"/>
                <property name="add" ref="occaddonCustomerPopulator"/>
        </bean>
...
```

Next Steps

The same actions must be performed to the reverse `Converter`.

> **Note**
>
> For more information on Converters and Populators, refer to Converters and Populators.

Localizing Attributes

Procedure

Add `<key>=<localized string>` entries in the following type system localization file: `occaddon/resources/localization/occaddon-locales_en.properties`.

`occaddon-locales_en.properties`

```
type.Customer.nickname.name=Nickname
type.Customer.nickname.description=Nickname description
type.Customer.workOfficeAddress.name=Work office address
type.Customer.workOfficeAddress.description=Work office address
```

Extending the Data Transfer Object (DTO) for v2

Context

For **v2** of Commerce Services, there is an additional DTO layer. It was created to improve the stability and configurability of the response data.

To use the new DTO and configure responses, perform the following steps:

ℹ Note

For more information, refer to WsDTO Concept.

Procedure

1. Extend the DTO in `occaddon-beans.xml`.

```
...
    <!-- Additional nickname and workofficeAddress property for UserWsDTO -->
    <bean class="de.hybris.platform.commercewebservicescommons.dto.user.UserWsDTO"
          extends="de.hybris.platform.commercewebservicescommons.dto.user.PrincipalWsDTO">
        <property name="nickname" type="String" />
        <property name="workOfficeAddress" type="de.hybris.platform.commercewebservicescommons.dto.user.AddressWsDTO"/>
    </bean>
    ...
```

2. Add new fields to the predefined field level configurations in `occaddon-web-spring.xml`.

   Remember that you need to do this in the *<addonname>*`-web-spring.xml` file localized in the resource directory, which is added to the Commerce Services context.

```
...
        <bean parent="fieldSetLevelMapping">
        <property name="dtoClass"
            value="de.hybris.platform.commercewebservicescommons.dto.user.UserWsDTO" />
        <property name="levelMapping">
            <map>
                <entry key="BASIC" value="nickname" />
                <entry key="DEFAULT" value="nickname" />
                <entry key="FULL" value="nickname,workOfficeAddress(FULL)" />
            </map>
        </property>
    </bean>
    ...
```

   Additional Resources

   - For more information on the web spring context refer to Extend Commerce Services.

3. Populate data between commerce data and web services DTO. Populating data from the commerce layer to web services DTO is done with the help of `Orika` - a popular Java Bean mapper framework. Fields with the same names are populated automatically by methods from version v2 with the help of the DataMapper object.

   UsersController.java

```
...
    public UserWsDTO getUser(@RequestParam(defaultValue = "BASIC") final String fields)
    {
        final CustomerData customerData = customerFacade.getCurrentCustomer();
        final UserWsDTO dto = dataMapper.map(customerData, UserWsDTO.class, fields);
        return dto;
    }
    ...
```

   You can also use the DataMapper in the AddOn controllers. The default implementation of DataMapper is defined in the `commercewebservicescommons` extension. To use it in an AddOn, define the dependency for this extension and add the proper resource in the controller:

   ...Controller.java

```
...
    @Resource(name = "dataMapper")
```

```
        protected DataMapper dataMapper;
        ...
```

Extending the REST API

You can extend the REST API for Commerce Services using an AddOn.

Defining a Controller

Context

To expose new calls, define a `Controller` class with the appropriate methods.

Procedure

Create the `Controller` in the `/acceleratoraddon/web/src/de/hybris/platform/acceleratorwebservicesaddon/controllers` directory.

`ExtendedCustomersController.java`

```
/**
 * Controller which extends Customer Resources
 */
@Controller("sampleExtendedCustomerController")
@RequestMapping(value = "/{baseSiteId}/customers")
public class ExtendedCustomersController
{
...
        @Secured("ROLE_CUSTOMERGROUP")
        @RequestMapping(value = "/current/nickname", method = RequestMethod.GET)
        @ResponseBody
        public String getCustomerNickname()
        {
                final String name = getCustomerFacade().getCurrentCustomer().getNickname();
                return name;
        }

        @Secured("ROLE_CUSTOMERGROUP")
        @RequestMapping(value = "/current/nickname", method = RequestMethod.PUT)
        @ResponseBody
        public CustomerData setCustomerNickname(@RequestParam final String nickname) throws DuplicateUidException
        {
                final CustomerData customer = customerFacade.getCurrentCustomer();
                customer.setNickname(nickname);
                customerFacade.updateFullProfile(customer);
                return customerFacade.getCurrentCustomer();
        }
...
}
```

ℹ Note

When you create a `controller` in the AddOn, you should not use the class defined in `ycommercewebservices`. The package name for such a class changes once the `extgen` process is completed.

Creating the Web Spring Context

Context

Since the sample controller is annotated as `@Controller`, Spring must be told where it should look for it.

Procedure

In the resource directory, update the *<addonname>*`-web-spring.xml` file.

The file extends Commerce Services context. This file **is not** the standard web context file from the `web\webroot\WEB-INF` directory.

Note

If your AddOn was generated from the `yoccaddon` template, component scan configuration for `<your_addon_package>.controllers` is set.

`resource/occaddon/web/spring/occaddon-web-spring.xml`

```
<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:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
           http://www.springframework.org/schema/context
                   http://www.springframework.org/schema/context/spring-context-3.1.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.1.xsd">

        <context:component-scan base-package="de.hybris.platform.occaddon.controllers"/>
</beans>
```

Adding the Web Context to Commerce Services

Context

Extend Commerce Services context with the context from the newly created AddOn. You can accomplish this by using the `additionalWebSpringConfigs` mechanism. The `project.properties` file must contain the following line where `Spring Config Classpath` points to any classpath resource from the `<addonExtension>:`

```
<targetExtension>.additionalWebSpringConfigs.<addonExtension>=[Spring Config Classpath]
```

Procedure

Add the following line to the `project.properties.template` file.

`project.properties.template`

```
ycommercewebservices.additionalWebSpringConfigs.<addOnName>=classpath:/occaddon/web/spring/<addOnName>-web-spring.xml
```

This file is a template for the `project.properties` file, which is generated during the installation process.

Note

If your AddOn was generated from `yoccaddon` template, `project.properties.template` should already have the proper content.

Verify the New Endpoints Defined in the Controller

Note

If you use the sample code in the Extending Data Objects, you need to rebuild and initialize/update the Commerce System to apply the new defined attributes in the `Customer` type and new beans. For the details, see Installing and Upgrading SAP Commerce.

Request Flow:

Method	URL	Header	Body Parameter
POST	https://localhost9002/authorizationserver/oauth/token	Content-Type: application/x-www-form-urlencoded	`client_id=$CLIENT_ID$&client_secret=$CLIENT_SECRET$&grant_type=passwor`
PUT	https:/localhost9002/occ/v2/electronics/users/8716fdb8-6cea-4e28-8396-1258e819f758/nickname • 8716fdb8-6cea-4e28-8396-1258e819f758 is userId.	Content-Type: application/x-www-form-urlencoded	nickname=myNickName

Method	URL	Header	Body Parameter
GET	https://localhost9002/occ/v2/electronics/users/8716fdb8-6cea-4e28-8396-1258e819f758/nickname • 8716fdb8-6cea-4e28-8396-1258e819f758 is userId.		

Overriding the REST API

Context

In an **AddOn** for Commerce Services, you can override the existing calls. The only difference between extending and overriding the REST API is using the `@RequestMappingOverride` annotation.

Procedure

Use the `@RequestMappingOverride` annotation to set priorities for methods with identical `@RequestMapping`.

`RequestMappingOverride`

```
@Target(
{ ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface RequestMappingOverride
{
    /**
     * Name for property, which stores priority value
     */
    String priorityProperty() default "";
}
```

The method with the **highest** priority is used.

⚠ Caution

`@RequestMappingOverride` should be used to override an identical @RequestMapping annotation.

It will however **not** work correctly in a situation where the original request mapping supports **two HTTP methods** and you will try to override only one of them:

- request mapping for original method = @RequestMapping(value = "/current/addresses/default/{id}", method = { RequestMethod.PUT, RequestMethod.POST })

- request mapping for method, which should override original = @RequestMapping(value = "/current/addresses/default/{id}", method = RequestMethod.PUT)

Even example below is considered as different mapping :

- request mapping for original method = @RequestMapping(value = "/{productCode}", method = RequestMethod.GET)

- request mapping for method, which should override original = @RequestMapping(value = "/{productId}", method = RequestMethod.GET)

In such cases methods will not be overridden and error "Ambiguous handler methods..." will appear during request (not during platform start up)

The `Priority value` is read from the properties file (`project.properties`, `local.properties` files) based on:

- `priorityProperty` given in the annotation or
- property name **requestMappingOverride.<className>.<methodName>.priority**

Example:

`requestMappingOverride.de.hybris.platform.occaddon.controllers.ExtendedCustomersController.updateDefaultAddress.priority`

If there is no property defined in the properties file, the priority value is set to zero.

Example:

- `@RequestMappingOverride(priorityProperty="occaddon.updateDefaultAddress.priority")` - here the priority value is read from the **occaddon.updateDefaultAddress.priority** property.
- `@RequestMappingOverride` - here the priority value is read from `requestMappingOverride.<className>.<methodName>.priority` property (e.g. `requestMappingOverride.de.hybris.platform.occaddon.controllers.ExtendedCustomersController.updateDefaultAddress.priority`).

ℹ Note

The `priority value` is read from the properties file to resolve a situation when more than one method overrides the original call. In this case, you can select the preferred method by setting the highest priority value for it in the `local.properties` file.

Overriding the Request Mapping Example

`ExtendedCustomerController.java`

```
/**
 * Controller which extends Customer Resources
 */
@Controller("sampleExtendedCustomerController")
@RequestMapping(value = "/{baseSiteId}/customers")
public class ExtendedCustomersController
{
...
    /**
     * This is example of overriding existing request mapping. Annotation {@link RequestMappingOverride} allows override
     * existing request mapping, defined by {@link RequestMapping} annotation.
     */
    @Secured("ROLE_CUSTOMERGROUP")
    @RequestMapping(value = "/current/addresses/default/{id}", method = RequestMethod.PUT)
    @RequestMappingOverride
    @ResponseBody
    public String updateDefaultAddress(@PathVariable final String id) throws DuplicateUidException
    {
        final AddressData address = userFacade.getAddressForCode(id);
        userFacade.setDefaultAddress(address);
        return "Address was updated successfully by method from sampleExtendedCustomerController";
    }
...
}
```

Assigning New Calls to a Specific API Version

Context

There are two versions available for Commerce Services (**v1** and **v2**). Both versions are available simultaneously. As a result, you might need to specify the version to call from the AddOn. To do this, you can use the `ApiVersion` annotation.

```
/**
 * Annotation can be used for controllers. It allows restrict visibility of methods annotated with
 * {@code @RequestMapping} only to selected version of commerce web services (e.g. v1 or v2).
 */
@Target(
{ ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface ApiVersion
{
    /**
    * Returns API version for which methods from controller should be registered (e.g. v1).
    */
    String value();
}
```

This annotation can be used at the `controller` level.

```
@Controller("sampleExtendedCustomerController")
@RequestMapping(value = "/{baseSiteId}/customers")
@ApiVersion("v1")
public class ExtendedCustomersController
{
...
}
```

ℹ Note

If there is no `@ApiVersion` annotation assigned to the controller in the AddOn, methods from this controller are added for all available versions.

For example - method annotated by `@RequestMapping(value = "/{baseSiteId}/customers/current/nickname")` are available from `URI` `/rest/v1/{baseSiteId}/customers/current/nickname` and `/rest/v2/{baseSiteId}/customers/current/nickname`.

Extending Server-Side Caching

Server-side caching is available for AddOns. In order to add your own cache to the existing configuration, use the `wsCacheManagerList` bean to store all cache managers (implementations of `org.springframework.cache.CacheManager` interface) that are in use.

Procedure

Use the **List Merge Directive** to extend the list from the AddOn.

`occaddon-web-spring.xml`

```
<!-- Cache manager for occaddon -->

        <alias name="defaultOccAddonCacheManager" alias="occAddonCacheManager"/>
        <bean id="defaultOccAddonCacheManager" class="org.springframework.cache.ehcache.EhCacheCacheManager">
                <property name="cacheManager" ref="occAddonEhcache"/>
        </bean>

        <alias name="defaultOccAddonEhcache" alias="occAddonEhcache"/>
        <bean id="defaultOccAddonEhcache" class="de.hybris.platform.commercewebservicescommons.cache.TenantAwareEhCacheManagerFactoryB
                <property name="cacheNamePrefix" value="occAddonCache_"/>
                <property name="configLocation" value="/WEB-INF/cache/addons/occaddon/ehcache.xml"/>
        </bean>

        <bean depends-on="wsCacheManagerList" parent="listMergeDirective">
                <property name="add" ref="occAddonCacheManager" />
        </bean>
```

This example is available in the `occaddon`.

ℹ Note

Remember that all beans described here must be placed in the `ADDON_NAME/resources/ADDON_NAME/web/spring/ADDON_NAME-web-spring.xml` file.

In the example, the Ehcache library is used by default. You can replace the library with any other cache library that is supported by the Spring Framework (i.e. Guava, GemFire, JSR-107).

The last thing you have to do (when using default configuration generated by `yoccaddon` template) is modify the `ehcache.xml` file located in the `ADDON_NAME/acceleratoraddon/web/webroot/WEB-INF/cache` directory and customize it to your needs.

Related Information

Caching

Extending Message Bundles

You can use an AddOn to define a message bundle that is available in Commerce Services.

Procedure

Define the message bundle in the `/acceleratoraddon/web/webroot/WEB-INF/messages` directory.

`ExtendedUsersController.java`

```
@Controller("sampleExtendedUserController")
@RequestMapping(value = "/{baseSiteId}/users")
@ApiVersion("v2")
public class ExtendedUsersController
{
...
    @Resource(name = "messageSource")
    protected MessageSource messageSource;
...

    @ResponseStatus(value = HttpStatus.PAYMENT_REQUIRED)
    @ResponseBody
    @ExceptionHandler({ WebserviceValidationException.class })
    public ErrorListWsDTO handleWebserviceValidationException(final WebserviceValidationException ex)
    {
```

```
        final ErrorListWsDTO errorListDto = handleErrorInternal(ex);
        return errorListDto;
    }
    protected ErrorListWsDTO handleErrorInternal(final WebserviceValidationException ex)
    {
        final ErrorListWsDTO errorListDto = new ErrorListWsDTO();
        final Locale locale = i18nService.getLocaleForLanguage(i18nService.getCurrentLanguage());
        final Errors errors = (Errors) ex.getValidationObject();
        final List<ErrorWsDTO> errorsList = errors.getAllErrors().stream().map(eo -> mapError(eo, locale))
                .collect(Collectors.toList());
        errorListDto.setErrors(errorsList);
        return errorListDto;
    }
    protected ErrorWsDTO mapError(final ObjectError error, final Locale locale)
    {
        final ErrorWsDTO result = new ErrorWsDTO();
        final String message = messageSource.getMessage(error.getCode(), error.getArguments(), locale);
        result.setMessage(message);
        result.setReason("Validation failed");
        result.setType("Error");
        return result;
    }
}
```

All files from this location are copied during the build phase into the `/web/webroot/WEB-INF/messages/addons/addonName/` directory of Commerce Services and automatically loaded by the `MessageSource` implementation.

The message bundle defined in the AddOn is visible in class from Commerce Services, which uses the `messageSource` bean.

A common `messageSource` bean is used by AddOn controllers. The example uses `messageSource` for exception handling.

`AddonAwareMessageSource` class is a custom implementation of `MessageSource` interface delivered by Commerce Services. It is configured to scan target directory `/WEB-INF/messages/addons/` for any `xml` and `properties` files that can be used as message bundles. Scanning and indexing files is done once during bean initialization.

`AddonAwareMessageSource` provides the following extra properties for customization:

- baseAddonDir - usually `/WEB-INF/messages/addons/`, it scans for all files in this directory and its subdirectories.
- fileFilter - filter for files that should be loaded. By default, the filter loads all `xml` and `properties` files.
- dirFilter - filter for subdirectories. By default, all subdirectories are scanned.

Related Information

Generating Beans and Enums