SAP Customer Experience

# Commerce Services and Façades

SAP Commerce Cloud

THE BEST RUN SAP

# The Context

Commerce Services and Facades provides a **suite of APIs** that make up a unified multichannel storefront which can be used by multiple front-ends. The responsibility of a single Facade is to **integrate existing business services** from the full range of the SAP Commerce extensions and **expose a Data object** (DTO) response adjusted to meet the storefront requirements.

# Commerce Services

**Commerce Services**
Commerce Facades
Bean Generation
Conversion Process

# commerceservices

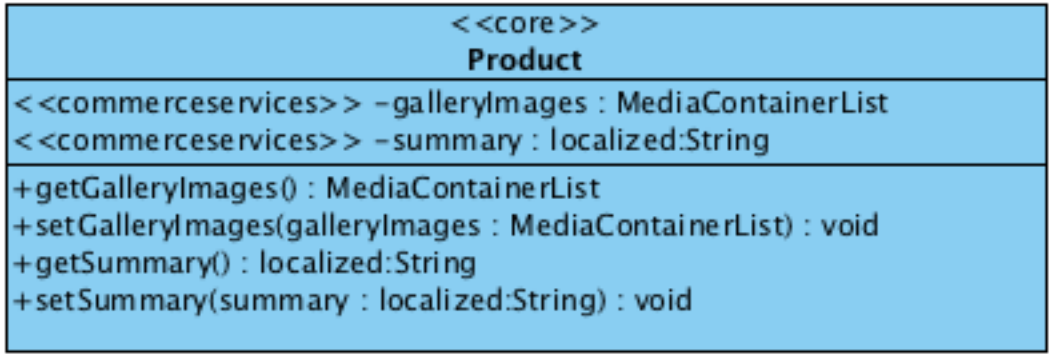
- Orchestrates platform and other extensions' services to provide complete **B2C use cases**
  - Example: The **commerceservices** extension provides the **CustomerAccountService**, which handles typical customer account management capabilities using the **userService**, **passwordEncoderService**, **baseStoreService**, and additional services from other extensions.

- Creates or extends more generic functionality from other extensions to **add more B2C features**
  - Example: The **commerceservices** extension extends the functionality of the **CartService** by creating the **CommerceCartService**, which adds promotions calculations and stock checks to the base functionality.

# Data model: Product

The **commerceservices** extension also extends the platform data model by injecting new attributes and into many existing Types, i.e. in Product type:

- summary
  - more concise product description (e.g. in search)
- galleryImages
  - storing multiple images each resized to a number of standard formats expected by the storefront



# Commerce Facades

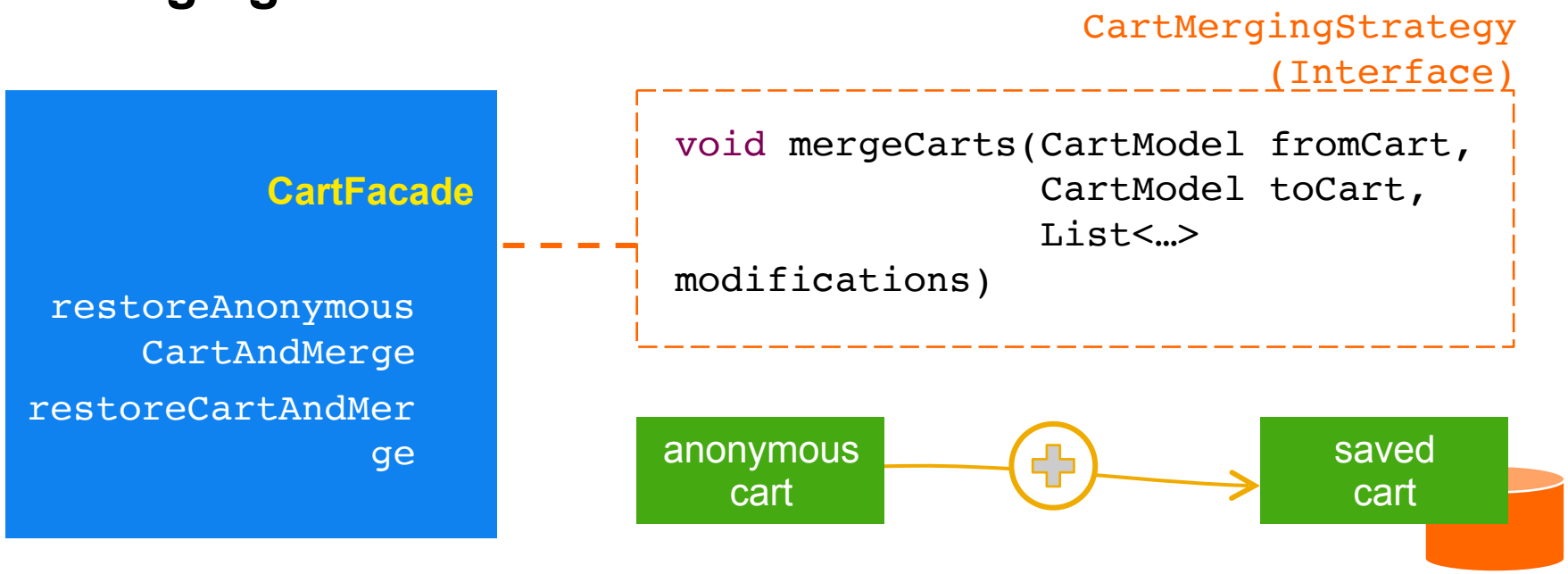


# commercefacades extension

Typical suite of Storefront Actions that make up a unified multichannel storefront API

- View product details
- Add a product to a cart
- Add a delivery address during checkout
- Post a review
- Search for products with a free text search

# Sample Cart-Merging Action




- Cart Merging aims to provide consistency across touch-points
- Customer's anonymous cart can be merged with their saved cart at login
  - possibly at start of checkout process
  - site doesn't lose items added to cart while customer browsed anonymously
- The CommerceCartMergingStrategy implementation is provided by the **commerceservices** extension
  - Used by the CartFacade of CommerceCartService (from **commercefacades** extension)

# Beans Generation for Façades



# The Use of JavaBean Instances as Data Objects

- Custom data objects (instances of JavaBean classes) carry data to the view
  - Populated with only the display-ready values that the target view needs
- Data objects are attached to a view by its controller
  - Default Spring MVC Views are JSPs that access data using JSTL and JSP EL
- To help the Spring MVC controller, we typically create a façade class with a method that obtains the Data Objects for the controller to send to the view
  - Typically, this method obtains its data from services that return ServiceLayer model objects (e.g. CarModel, CategoryModel)

E.g. `List<MovieDetailData> getMovieDetailViewData(Integer movieID)`

**Data Objects are also known as DTOs (Data Transfer Objects)**

# Auto-Generated JavaBean Classes – A Declarative Approach

- We can have JavaBean (and Enum) source code generated for us during ant builds
  - For each JavaBean class to be generated, a declaration must exist inside a **`resources/<extensionName>-beans.xml`** file
  - A JavaBean class declaration includes the fully-qualified class name, its superclass (optional), and the bean's "properties" (property names and Java types)
  - Each extension may contribute its own **`*-beans.xml`** file

`commercefacades-beans.xml`

```xml
<bean class="de.hybris.platform.commercefacades.order.data.DeliveryModeData">
   <property name="code" type="String"/>
   <property name="name" type="String"/>
   <property name="description" type="String"/>
   <property name="arriveByDate" type="java.util.Date"/>
   <property name="deliveryCost"
             type="de.hybris.platform.commercefacades.product.data.PriceData"/
>
</bean>
```

# What Gets Generated?

- Generate Java Beans from declarations within a **`*-beans.xml`** file

```xml
<bean class="org.training.data.MyPojo">
    <property name="id" type="String"/>
</bean>
```

```java
public class MyPojo implements java.io.Serializable
{
    private String id;
    public MyPojo() {} //no-argument constructor
    public String getId() {...}
    public void setId(String id) {}
}
```

# *Why* a Declarative Approach?

- A single JavaBean class definition can be split-up across multiple extensions
  - All partial declarations having the same class name are merged (from all extensions participating in the build) and generate a single JavaBean class
  - This way, an extension could be made optional
  - New, custom extensions can expand existing JavaBean definitions
- Java Enum classes (with singleton member values) can be defined similarly
- Generated classes are placed in **`hybris/bin/platform/bootstrap/gensrc`**

**Does this sound familiar?**

**It should! items.xml and beans.xml share the same paradigm**

# How Bean Definitions Get Merged

extension1-beans.xml

```
<bean
class="org.training.data.MyPojo">
    <property name="id"
type="String"/>
</bean>
```

extension2-beans.xml

```
<bean
class="org.training.data.MyPojo">
    <property name="timeStamp"
            type="java.util.Date"/>
</bean>
```

```
public class MyPojo implements java.io.Serializable
{
    private String id;
    private java.util.Date timeStamp;
    public MyPojo() {}
    public String getId() {...}
    public java.util.Date getTimeStamp() {...}
    public void setId(String id) {...}
    public void setTimeStamp(java.util.Date timeStamp)
{...}
}
```

Triggered by: `ant all`

Generated in platform/bootstrap/gensrc

# Conversion Process



# Converters and Populators

Implementation of `Converter<SOURCE, TARGET>`

- Transforms an object of type `SOURCE` into an object of type `TARGET`
- Primary callback method is: `TARGET convert( SOURCE )`
  1. Instantiates a new, empty instance of `TARGET` (typically a DTO)
  2. Delegates the population to a Populator - passing in `SOURCE` and `TARGET` (see below)
  3. Afterwards return the populated `TARGET` instance

Implementation of `Populator<SOURCE, TARGET>`

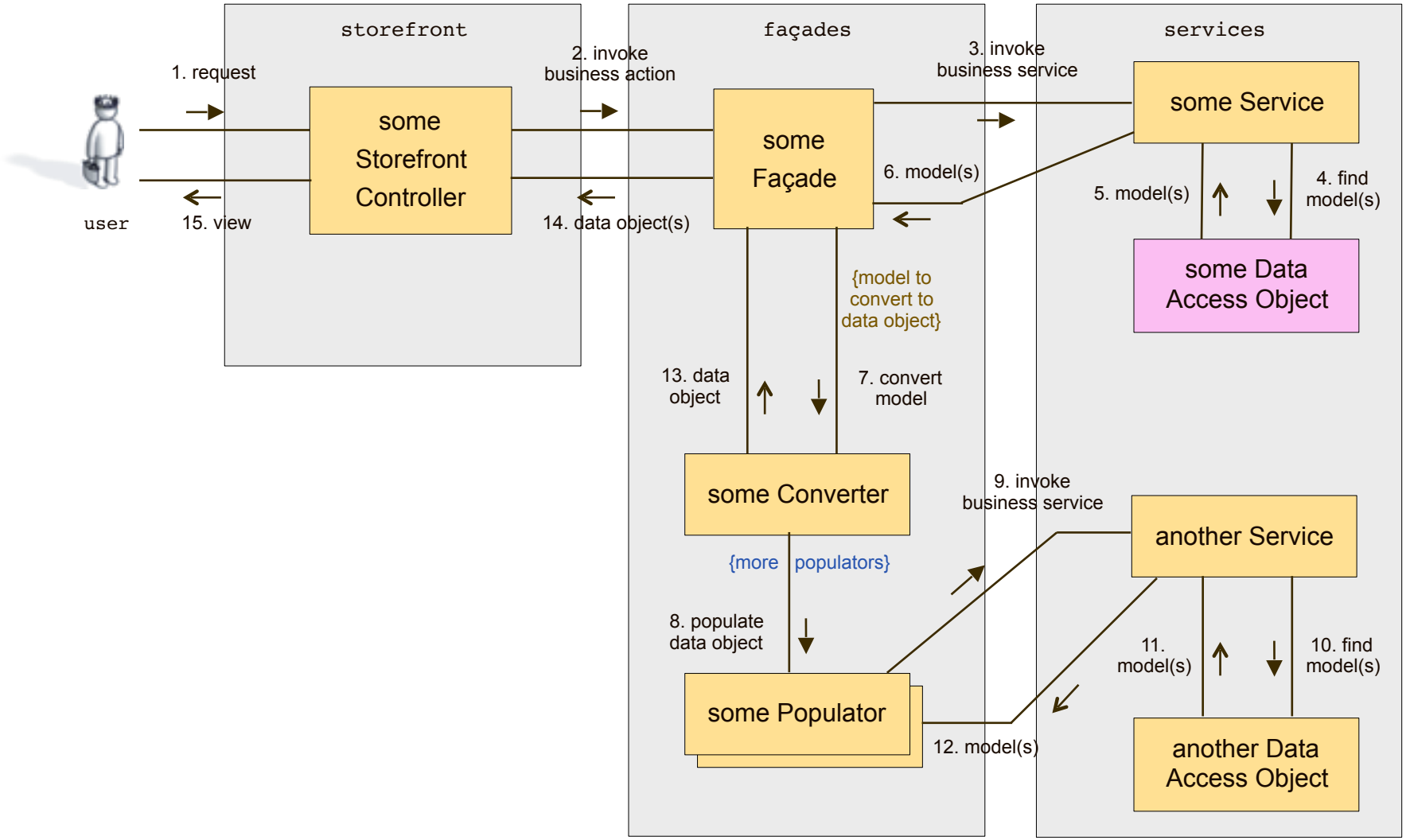
- Sets values in `TARGET` instance based on values in `SOURCE` instance
- Primary callback method is: `void populate( SOURCE, TARGET )`
  - Uses values from `SOURCE` instance to populate values of `TARGET` instance

Type conversion is typically broken down into a sequence of population steps

- Converter is assigned to one or more Populators and are called sequentially

Reverse Converters can be defined to convert from DTO to Model, but this is rarely needed

# Conceptual Interaction Diagram



# A Typical Populator

XyzTypeBasicPopulator.java

```
public class XyzTypeBasicPopulator implements Populator<XyzTypeModel extends ProductModel,
XyzTypeData>
{
    @Override
    public void populate(final XyzTypeModel source, final XyzTypeData target)
     throws ConversionException
    {
        target.setDescription( source.getDescription() );   // E.g. String to String
        // populator can leverage services to get source price (double)
        double priceValue =
           getPriceService().getWebPriceForProduct(source).getPriceValue().getValue()
        DecimalFormat currencyFmt = DecimalFormat.getCurrencyInstance( getLocale() );
        String displayPrice = currencyFmt.format( priceValue );
        target.setPrice( displayPrice );   // target price: String
        // . . . etc.
    }
    // property getters, setters, and private attributes not shown
}
```

myfacadesextension-spring.xml

```
<bean name="defaultXyzPopulator"
      class="org.training.facades.populators.XyzTypeBasicPopulator">
    <property name="priceService" ref="mySimplePriceService" />
</bean>
```

# Use case 1: Defining a New Converter

- The `platformservices` extension provides a base `abstractPopulatingConverter` bean
  - Allows you to define a new converter bean without having to write a Java class
  - Allows for easy reuse of populators (as beans backed by custom Java classes)

myfacadesextension-spring.xml

```
<alias alias="carConverter"
       name="defaultCarConverter" />

<bean  id="defaultCarConverter"
       parent="abstractPopulatingConverter">
    <property name="targetClass"
              value="my.project...data.CarData"/>
    <property name="populators">
        <list>
            <ref bean="carBasicPopulator"/>
            <ref bean="carFeaturesPopulator"/>
        </list>
    </property>
</bean>
```

Populators injected into converter bean here:

# Use case 2: Add Populator to Existing Converter

- How can type conversion be hooked-into without rewriting the basic code or existing converters?
  - Use a `modifyPopulatorList` to modify existing populator lists
    - defined in `commerceservices-spring.xml`
    - available operations: **add** and **remove**
    - Processed by BeanPostProcessor

Name of converter bean defined in another, pre-existing extension:

myfacadesextension-spring.xml

```
<bean parent="modifyPopulatorList">
    <property name="list" ref="productConverter"/>
    <property name="add"  ref="fooProductPopulator"/>
</bean>
```

# Use case 3: Converters for Extended Types

- New attributes of extended types can be transferred to view using pre-existing Controllers

```
consider:    <itemtype code="FooProduct" extends="Product" ...
                <attribute qualifier="bar" type="java.lang.String" ...
```

- Solution 1
  - Define a new converter bean whose "parent" is base type's converter bean instead of the usual, `parent="abstractPopulatingConverter"`
  - Spring's `<list merge="true">` can *merge* new populators with 'inherited' ones, if desired

```
<bean id="fooProductConverter" parent="productConverter" >
      <property name="populators" >
          <list merge="true" >
              <ref
bean="fooProductAdditionalAttribsPopulator" />
          </list>
```

Inherits populator list; merges two lists instead of overriding

  - Façade must decide which converter to use per `SOURCE` instance, based on its type

- Solution 2
  - Merge new subtype's attributes/properties into base type's DTO (i.e., `ProductData`)
  - Keep existing converter, but add additional populator using a `modifyPopulatorList`
  - New populator must check `SOURCE` item type; accesses new attributes only if appropriate

# The Façade Class

- The façade class needs to be written – it typically looks like this:

```
public class DefaultCarFacade implements CarFacade
{
   private CarService carService;
   private CarConverter carConverter;

   public CarData getCarOfTheYear(final int year)
   {
      CarModel car = carService.getFeaturedCar(year);
      CarData carData = carConverter.convert(car);
      return carData;
   }

   // getters & setters (for carConverter and carService injections) not shown
   //    these will be injected in <extensionname>-spring.xml

}
```

Converter injected here (using the corresponding setter) by *myfacadesextension*-spring.xml

# Associating the Converter With the Façade

- Declare the façade as a Spring bean
- Inject the converter bean, along with all the other service beans the façade will need

myfacadesextension-spring.xml

```
<alias alias="carFacade"
       name="defaultCarFacade" />
<bean id="defaultCarFacade"
      class="my..commercefacades.car.impl.DefaultCarFacade">
    <property name="carService" ref="carService"/>
    <property name="customerReviewService" ref="customerReviewService"/>
    <property name="userService" ref="userService"/>
    <property name="modelService" ref="modelService"/>

    <property name="carConverter" ref="carConverter"/>

</bean>
```

Converter injection within myfacadesextension-spring.xml

# Using the Façade

- Within the controller class:

```
@Controller
@RequestMapping(value = "/**/car")
public class CarPageController extends AbstractPageController
{
    @Resource(name = "carFacade")
    private CarFacade carFacade;

    //...


    @RequestMapping(value = YEAR_PATH_VARIABLE_PATTERN, method = RequestMethod.GET)
    public String showCarDetail(@PathVariable("year") final String encodedProductCode, final
Model model,
            final HttpServletRequest request, final HttpServletResponse response)
    {
        final CarData carData = carFacade.getCarOfTheYear( year );
        //...
    }
}
```

Façade injected into controller here

Façade being used here


The Façade layer is responsible for **converting models** to **data transfer objects**

Concrete conversion is implemented by a **converter** and its associated **populators**

Different ways exist to add new attributes to the type conversion:

- Create a new converter
- Reuse an existing converter and add new populator using a modifyPopulatorList
- Extend a parent converter and merge new populators with inherited ones

The commerceservices and commercefacades extensions contain major functionality to support B2C features. They also provide a good example to demonstrate the relationship among converters, populators, façades, services, models, and data transfer objects.

# Exercise

## Facades

# Thank you.