How to Implement Populators and Converters

OMS has been made more extensible by enhancing the converters and populators. It is now possible to support multiple populators for each converter. The populators are also easily interchangeable. These improvements allow for better alignment of converters and populators between OMS and the platform.

About this Document

This document describes the new implementation of converters and populators in OMS.

Audience: Developers, consultants, partners

Related concept: Order Management Services (Core+)

Validity: 5.1.0 and higher

Based on hybris version: 5.1.0

Rate the Document:

Conversion API

The **com.hybris.commons.conversion.Converter** is an interface for creating a target object based on a source object:

Converter.java

```
public interface Converter<S, T>

{

    T convert(S source) throws ConversionException;

}
```

The **com.hybris.commons.conversion.Populator** is an interface for updating an existing target object based on a given source object:

Populator.java

```
public interface Populator<S, T>

{

    void populate(S source, T target) throws ConversionException,
IllegalArgumentException;
```

```
    void populateFinals(S source, T target) throws ConversionException,
IllegalArgumentException;

}
```

The **com.hybris.commons.conversion.ConversionException** is used to signal problems with data conversion. This exception may be thrown when calling **populateFinals** if a final field is being updated more than once.

Additionally there is a utility class, **com.hybris.oms.facade.conversion.util.Converters**, that provides a method for converting a list of source objects into a list of target objects:

Converters.java

```
<S, T> List<T> convertAll(final Collection<? extends S> sourceList, final Converter<S,
T> converter)
```

This method converts every element from the source list using a given converter. It also takes care of null checks.

This class is defined as a Spring bean with the **converters** ID, which is located in the **`commons-conversion-spring.xml`** file.

```
<bean id="converters" class="com.hybris.commons.conversion.util.Converters" />
```

Implementing Converters and Populators

The following sections provide details if you need to implement a new converter and populator for a new type.

Creating the Converter Directly

You may choose to write a converter directly; this means that you will write the conversion logic directly in the converter and that you may not make use of populators or the extensibility provided by the **AbstractPopulatingConverter**.

Note

This approach has only been used for reverse converters for **Value Types** since these are immutable and their attributes must be set in the constructor.

Here is an example of using a direct converter for converting an address value type:

```
/**
 * Converts {@link Address} DTO into {@link AddressVT} value type instance.
 */
```

```
public class AddressReverseConverter implements Converter<Address, AddressVT>

{

    @Override

    public AddressVT convert(final Address source) throws ConversionException

    {

        if (source == null)

        {

            return null;

        }

        else

        {

            return new AddressVT(source.getAddressLine1(), source.getAddressLine2(),
source.getCityName(),

                    source.getCountrySubentity(), source.getPostalZone(),
source.getLatitudeValue(), source.getLongitudeValue(),

                    source.getCountryIso3166Alpha2Code(), source.getCountryName(),
source.getName(), source.getPhoneNumber());

        }

    }

}
```

Creating the Converter by Extending the AbstractPopulatingConverter

The AbstractPopulatingConverter

The abstract populating converter implements both the **Converter** and the **Populator** interface and acts as a converter which will also perform the population. It contains a single populator and demands that all child implementations implement a **createTarget** method to create an actual instance of the target.

```
public abstract class AbstractPopulatingConverter<S, T> implements Converter<S, T>,
Populator<S, T>

{

    private Populator<S, T> populator;

    /**

     * Override this method to create the instance of target type.

     *

     * @return the new target instance

     */
```

```
    protected abstract T createTarget();

    ...

}
```

The **convert** method of this class will perform the following tasks:

1. Create a new instance of the target class.
2. Populate all final fields.
3. Populate all remaining fields.
4. Return the new converted and populated instance of the target class.

```
@Override

public T convert(final S source) throws ConversionException

{

    if (source == null)

    {

        return null;

    }

    final T target = this.createTarget();



    this.populateFinals(source, target);



    this.populate(source, target);

    return target;

}
```

This class is defined as a Spring bean with the **abstractPopulatingConverter** ID which is located in the **`commons-conversion-spring.xml`** file.

```
<bean id="abstractPopulatingConverter"
class="com.hybris.oms.facade.conversion.impl.AbstractPopulatingConverter"
abstract="true">

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

</bean>
```

Creating the Populator

The first thing you need is a populator (or reverse populator) that implements the **Populator** interface above. You can choose to either extend the existing

AbstractPopulator or simply implement the **Populator** interface.

Note

If the target in your conversion does not have any final fields to populate, then it is suggested to extend the **AbstractPopulator**.

Implementing the Populator Interface

Here is an example of a class that implements the Populator interface directly. Note that the **populateFinals** method has been overridden to set just the final field for this type.

BinReversePopulator.java

```
public class BinReversePopulator implements Populator<Bin, BinData>
{
    private PersistenceManager persistenceManager;
    @Override
    public void populateFinals(final Bin source, final BinData target) throws ConversionE
    {
        target.setBinCode(source.getBinCode().toLowerCase());
    }
    @Override
    public void populate(final Bin source, final BinData target) throws ConversionExcepti
    {
        target.setStockroomLocation(this.persistenceManager.getByIndex(StockroomLocatio
                source.getLocationId()));
        target.setDescription(source.getDescription());
        target.setPriority(source.getPriority());
    }
    @Required
    public void setPersistenceManager(final PersistenceManager persistenceManager)
    {
        this.persistenceManager = persistenceManager;
    }
}
```

Note

All reverse populators (DTO to ManagedObject) where the ManagedObject has final fields that should implement this interface directly.

Extending the Abstract Populator

If your target type does not require the conversion of final fields, then simply override the following **AbstractPopulator** to avoid having to redundantly override the **populateFinals** method.

AbstractPopulator.java

```
/**

 * Basic {@link Populator} implementation that assumes that there are no final fields
to populate.

 */

public abstract class AbstractPopulator<S, T> implements Populator<S, T>

{

    @Override

    public void populateFinals(final S source, final T target) throws
ConversionException, IllegalArgumentException

    {


    }

}
```

Note

All forward populators (ManagedObject to DTO) should extend this abstract class.

The CompositePopulator

If you require more than one populator to perform your population for a given type, then you should use the **CompositePopulator** bean to group your populator beans.

The **populateFinals** method will iterate over all populators and call their **populateFinals** method.

```
@Override

public void populateFinals(final S source, final T target)

{
```

```
    for (final Populator<S, T> populator : getPopulators())
    {
        populator.populateFinals(source, target);
    }
}
```

The **populate** method will iterate over all populators and call their **populate** method.

```
@Override
public void populate(final S source, final T target)
{
    for (final Populator<S, T> populator : getPopulators())
    {
        populator.populate(source, target);
    }
}
```

Define All XML Beans

All of the **magic** behind the converters lies in the **`oms-facade-conversion-spring.xml`** Spring XML file.

Define the Populator

For a Regular Populator

```
<bean id="binPopulator"
class="com.hybris.oms.facade.conversion.impl.inventory.BinPopulator" />
```

For a Reverse Populator

```
<bean id="binReversePopulator"
class="com.hybris.oms.facade.conversion.impl.inventory.BinReversePopulator">

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

</bean>
```

Note

If your reverse populator requires a search to retrieve another ManagedObject by its unique index, then you must remember to inject the **PersistenceManager** dependency. You may also need to inject a service to properly populate your ManagedObject.

For a Composite Populator

Most types to be converted are entities that may contain schemaless attributes and this requires the **PropertyAwarePopulator**. To combine these populators, define your final populator as follows:

```
<alias name="propertyAwareBinPopulator" alias="binPopulator" />

<bean id="propertyAwareBinPopulator" parent="compositePopulator">

    <property name="populators">

        <list>

            <ref bean="defaultBinPopulator" />

            <ref bean="propertyAwarePopulator" />

        </list>

    </property>

</bean>
```

Define a Prototype Bean

For a Regular Converter

You will have to define a prototype scoped bean representing an instance of your DTO.

```
<bean id="binDTO" class="com.hybris.oms.domain.inventory.Bin" scope="prototype" />
```

For a Reverse Converter

You will have to define a prototype scoped bean representing an instance of your ManagedObject.

```
<bean id="binMO" parent="abstractMO">

    <property name="arguments">

        <list>

            <value
type="java.lang.Class">com.hybris.oms.service.managedobjects.inventory.BinData</value>

        </list>

    </property>

</bean>
```

The following **abstractMO** abstract bean is provided simply as a means to avoid XML duplication.

```
<bean id="abstractMO"
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean"
abstract="true">

    <property name="singleton" value="false" />

    <property name="targetObject" ref="persistenceManager"/>

    <property name="targetMethod" value="create"/>

</bean>
```

Define the Converter

When you define your converter via this method, you will not actually create a concrete **Converter** class, instead you will opt for extending the **abstractPopulatingConverter** bean and override the **createTarget** method as well as the list of populators.

For a Regular Converter

```
<bean id="defaultBinConverter" parent="abstractPopulatingConverter">

    <lookup-method name="createTarget" bean="binDTO" />

    <property name="populator" ref="binPopulator" />

</bean>
```

For a Reverse Converter

```
<bean id="defaultBinReverseConverter" parent="abstractPopulatingConverter">

    <lookup-method name="createTarget" bean="binMO" />

    <property name="populator" ref="binReversePopulator" />

</bean>
```