

ServiceLayer

The hybris ServiceLayer is an **API** to develop with and for the hybris Multichannel Suite. The main characteristics of the ServiceLayer are:

- It is based on a service-oriented architecture.
- It provides a clean **separation** of business logic and persistence logic.
- It provides a number of **services**, each with its well-defined responsibilities.
- It provides a **framework** to develop your own services and to extend existing ones.
- It is heavily based on the **Spring** Framework.
- It is based on **common patterns**, such as interface-oriented design and dependency injection.
- It is the layer in which partners should implement their business logic.
- It provides hooks into **model life-cycle events** for performing custom logic.
- It provides hooks into system event life-cycle events such as **init and update** process.
- It provides a framework for publishing and receiving **events**.

The service layer aims to be:

- Consistent
- Easily approachable
- Comprehensive
- Adaptable
- Extensible
- Flexible

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.

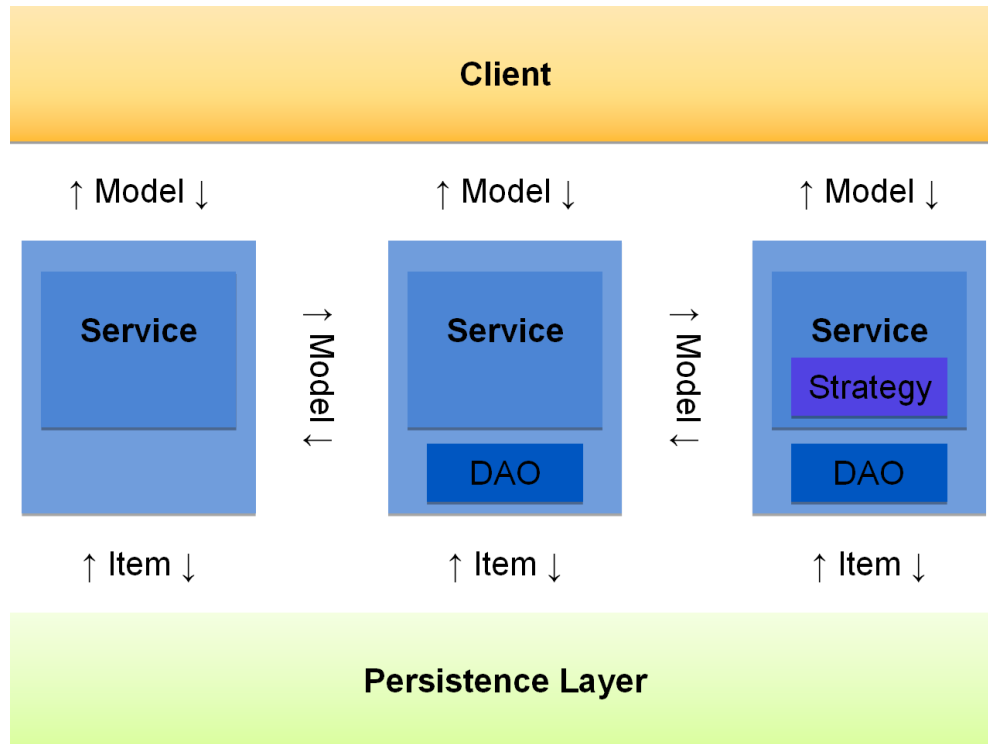


Figure: Overview of the integration of the hybris ServiceLayer. For a discussion of the components, please refer to ServiceLayer Architecture.

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.

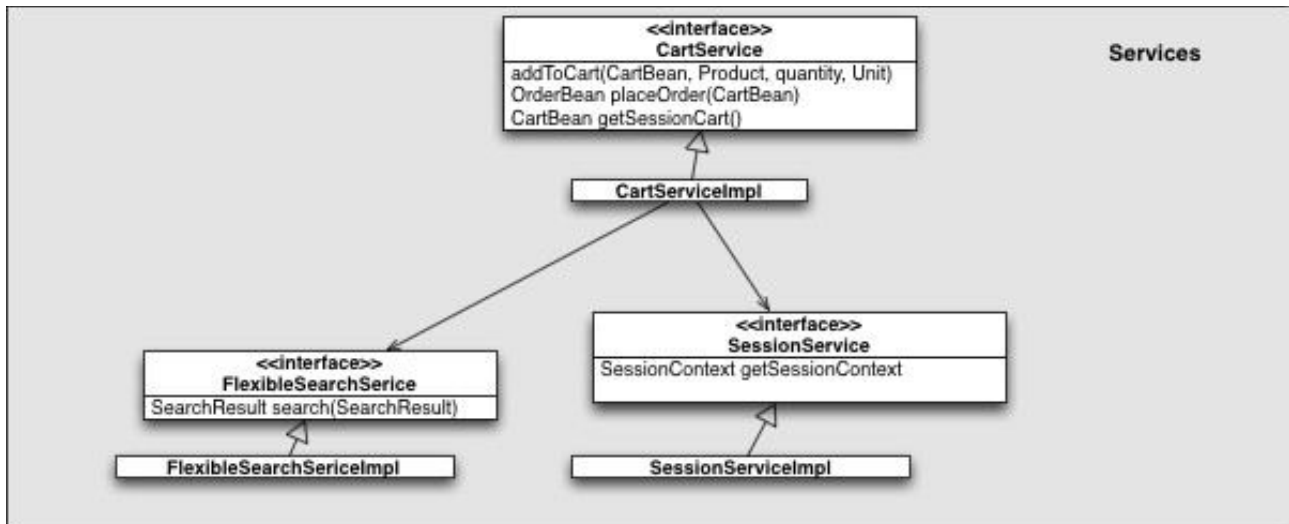


Figure: Sample of relations between services. Note the pattern of having an interface and an implementation per service.

Figure: Sample of relations between services. Note the pattern of having an interface and an implementation per service.

The hybris Commerce Suite 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, hybris recommends implementing services in terms of interfaces.

Strategies

A service may delegate parts of its tasks to smaller micro-services, called strategies. The service then serves as a kind of facade 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.

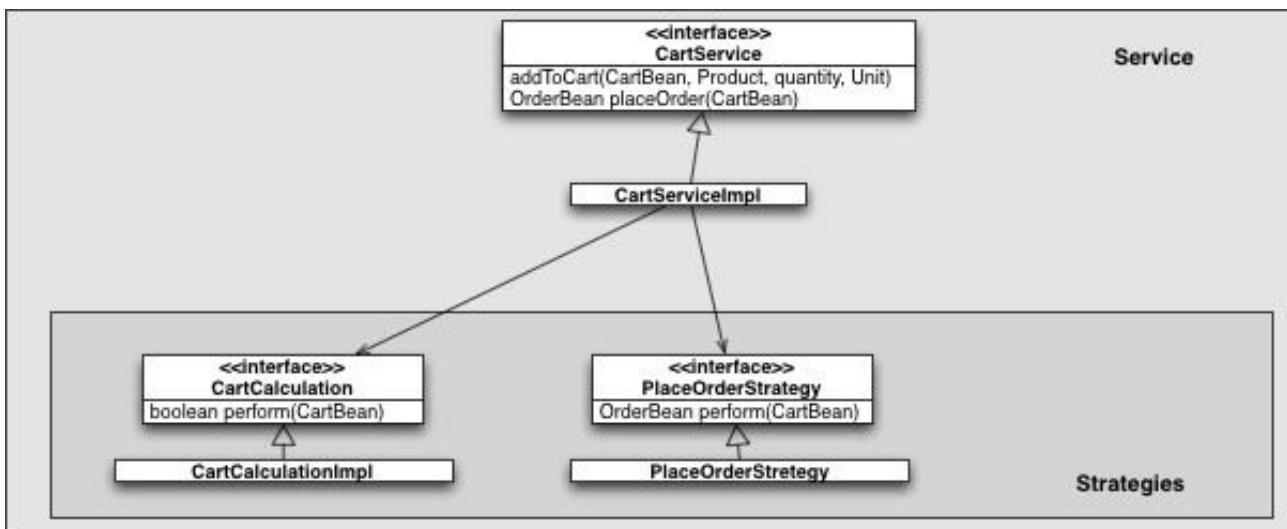


Figure: Sample of a service (consisting of an interface definition and the related implementation) relying on strategies. Note how the strategies also follow the pattern of interface definition and related implementation.

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.

In the hybris Commerce Suite, DAOs use the hybris Type System for persistence. This means that hybris Commerce Suite DAOs do not implement any individual logic and simply call the underlying persistence layer.

Models

Models are a new way to represent hybris 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 the hybris Commerce Suite; 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.

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.

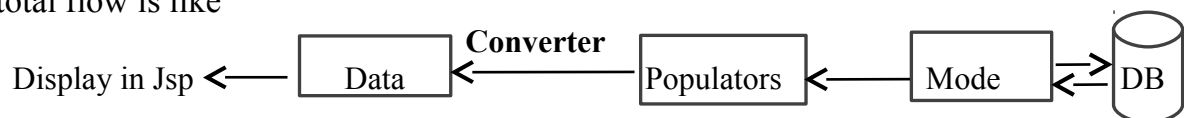
Modelservice contain three methods:

- 1.modelservice.Load()
- 2.modelservice.create()
- 3..modelservice.save()
- 4.modelservice.remove()

Populator:

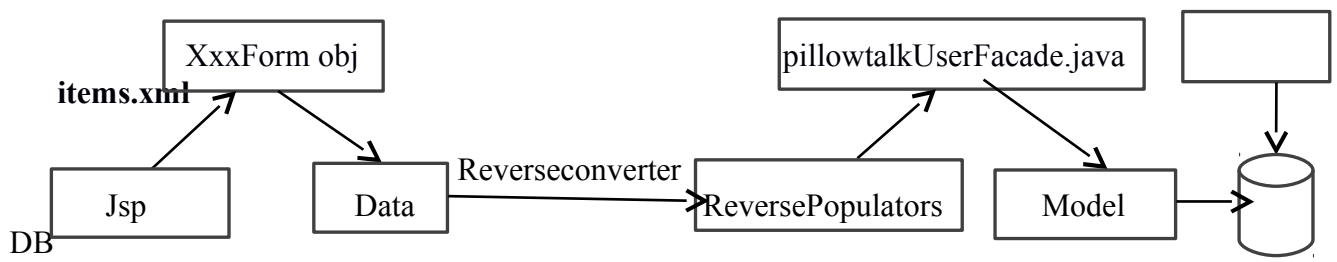
- Populator is used to convert Model object into Data object so that it can be used in the front end by capturing the related field value from Data object.
- Populators are always used inside any Converter, so main role of converting model to data will be done by Converter by using the Populators inside it.
- Default Populator and converters are defined in the **commercefacades-spring.xml**

The total flow is like



ReversePopulator:

- ReversePopulator is used to convert Data object into Model object so that it can be stored in the (DB)back end by capturing the related field value from model object.
- ReversePopulators are always used inside any Reverseconverter, so main role of converting data to model will be done by Reverseconverters by using the ReversePopulators inside it.
- The reverse populators and converters are declared in the **commercefacades-spring.xml**
- We can use default ones or we can create custom reverse populator and reverse converters.



Task -1:

Aim: To Create an Attribute(State) in Register Form and the value has to be Stored in Database Using **Reverse Populator** and **Reverse Converters**

CREATE AN ACCOUNT

For a fast checkout, easy access to previous orders, and the ability to create an address book and store settings. Register below.

Title *
Please select

First Name *

Last Name *

Email Address *

State * → create new attribute(state) in registration page

Mobile Number *

Password *
Minimum length is 6 characters

Confirm Password *

REGISTER

Step 1:

- Create an attribute in jsp (to display in front end i.e UI)
- Go to **register.tag**
- **Note :**In hybris every jsp page is divided into some selectors & every selector having different tag files,i.e, in hybris the real changes occurs on tag files.

- **register.tag** file which is nothing but jsp tags and the code:

register.tag

```
<formElement:formInputBox idKey="register.state" labelKey="register.state" path="state"
inputCSS="text" mandatory="true"/>
```

Step 2:

- create a label in **base_en.properties** file.
- **base_en.properties** file is used to display the **labels** in jsp/tags
- Now search **base_en.properties** file (ctrl+shift+R ,i.e., resources <project_name>storefront\web\webroot\WEBINF\messages**base_en.properties**), which is nothing but base or basic properties file.
- Here search for register related data and add the respective code,i.e,**register.state=State**.
- After adding in properties file perform command **ant clean all**.

Step 3:

- Now create State(attribute) in Data Class to capture the value from jsp
<project_name>**facade-beans.xml**

<project_name>**facade-beans.xml**

```
<bean class="de.hybris.platform.commercefacades.user.data.RegisterData">
    <property name="state" type="String"/>
</bean>
```

- Now perform **ant build**, then property will be created in related Data class **Eg:**

RegisterData.java,

```
public class RegisterData {
    private String state;

    public void setState(final String state)
    {
        this.state = state;
    }
    public String getState()
    {
        return state;
    }
}
```

Step 4:

- We need to set/bind form related data to Data object.
- So, go to **LoginPageController** and search for **/register(post)** method then open the method implementation for **processRegisterUserRequest()** here form object is passed.

- There you can find creation of RegisterData object, now set values from RegistrationForm object into the RegisterData class object, i.e,

```
data.setState(form.getState())
```

Note: First create State variable in RegisterForm (if variable is not available)

go to the path

acceleratorstorefrontcommons/commonweb/src/de/hybris/platform/acceleratorstorefrontcommons/forms/RegisterForm.java and create variable

private String **state**; in RegisterForm

- then perform **ant build**. (we should do ant build whenever we are changing any of the .java, .xml file)

Step 5:

- Create a new column in the existing table in Data base using **item.xml**
- create object model part like mentioned below in **<project_name>CoreItem.xml**

<project_name>core-items.xml

```
<itemtype code="Customer"
  extends="User"
  jaloclass="de.hybris.platform.jalo.user.Customer"
  autocreate="false"
  generate="true">
  <attributes>
    <!-- auto ID which is generated by NumberSeries -->
    <attribute autocreate="true" qualifier="state" type="java.lang.String">
      <modifiers read="true" write="true" search="true" optional="true"/>
      <persistence type="property"/>
    </attribute>
  </attributes>
</itemtype>
```

- Perform **ant build** or **ant all**.
- Then new attribute will be created in its related model class, after refreshing the platform folder in workspace we can see in the related Model class.
- In our example attribute will be created in **CustomerModel.java** like,

CustomerModel.java

```
public class CustomerModel {
    private String state;
    private xxxxxxxx
    .....
}
```

- Now to make changes in Database, start the server and go to hac i.e. Hybris admin console... **url: localhost:9001/**
- Then **go to platform-> update**: here select first option i.e. update running system and deselect remaining options and click on update. It will create the corresponding attribute in the database.
- After that you can see the attribute inside in HMC for any customer.

Step 6:

- Now we need to populate related property value into the model class (to save in data base)

Data class -----> Model class

- To convert from Data Class to Model class, we should not write code in any default extensions or classes given by Hybris suite.
- We need to create our own classes (in our custom extensions) for **Populators** or **Facades** and they must extend the default classes.
- For this in Hybris we have Populators and it can be done in two methods:

Method 1:

Using Default Reverse Populator & Reverse Converters

Step 1:

- Now to convert Data object to Model object we have to create our own Facade class **<project_name>CustomerFacade** and it must extends **DefaultCustomerFacade.java** and in that override the `register()` method
- Create a new FacadeClass

Eg:

ShoppersstopCustomerFacade extends DefaultCustomerFacade

- Override the register method in that i.e,

ShoppersstopCustomerFacade.java
--

```
@Override
public void register(final RegisterData registerData) throws DuplicateUidException
{
    copy the entire code from default register method i.e, from the DefaultCustomerFacade

    add code :
    newCustomer.setState(registerData.getState());
}
```

Step 2:

- Configure the **<project_name>CustomerFacade** bean in **<project_name>facades-spring.xml**

Eg:

shoppersstopfacades-spring.xml

```
<alias name="ssCustomerFacade" alias="customerFacade" />
<bean id="ssCustomerFacade" parent="defaultCustomerFacade"
      class="com.techouts.shoppersstop.facades.ShoppersstopCustomerFacade">
</bean>
```

Step 3:

- Now perform **ant build** , start the server and Go to site and Register
- Now your entered State in the RegistrationForm will directly Store in the database.
- To check whether the value is Stored or not by the following procedure

Go to Hmc

url: localhost:9001/hmc/hybirs

open User ----->Customer search using registered mail id

open the customer see the value of State

Method 2: Using Our Created Reverse Populator&Reverse Converters

Step 1:

- Create class **<project_name>ReversePopulator.java** and it should Implement **Populator**, then it becomes Populator. Eg:

```
public class ShoppersStopCustomReversePopulator implements Populator<RegisterData, CustomerModel>
```

- Now override the **populate()** method i.e,

```
<project_name>ReversePopulator.java
```

```
public class ShoppersStopCustomReversePopulator implements Populator<RegisterData, CustomerModel>
{
    @Override
    public void populate(final RegisterData source, final CustomerModel target) throws ConversionException
    {
        target.setMobileNumber(source.getMobileNumber());
    }
}
```

Step 2:

- Configure the <project_name>**ReversePopulator** bean in <project_name>**facades-spring.xml**

```
shoppersstopfacades-spring.xml
```

```
<bean id="ssReversePopulator"
class="com.techouts.shoppersstop.facades.populators.ShoppersStopCustomReversePopulator" />
```

- Now configure for customReversepopulator and default reversePopulator in the Reverseconverter using the defaultReverseConverter in the <project_name>**facades-spring.xml**

```
shoppersstopfacades-spring.xml
```

```
<alias name="defaultCustomerReverseConverter" alias="customerReverseConverter"/>
    <bean id="defaultCustomerReverseConverter" parent="abstractPopulatingConverter">
        <property name="targetClass"
value="de.hybris.platform.core.model.user.CustomerModel"/>
        <property name="populators">
            <list>
                <ref bean="customerReversePopulator"/>
                <ref bean="ssReversePopulator"/>
            </list>
        </property>
    </bean>
```

Step 3:

- Go to **ShoppersstopCustomerFacade.java** , in that create an Object for **ShoppersStopCustomReversePopulator** i.e,

```
@Autowired
```

```
ShoppersStopCustomReversePopulator ssReversePopulator;
```

- In that, from the register method call the populate method i.e,

ShoppersstopCustomerFacade.java

```
@Override
public void register(final RegisterData registerData) throws DuplicateUidException
{
    copy the entire code from default register method i.e, from the DefaultCustomerFacade

    add code :
    ssReversePopulator.populate(registerData, newCustomer);
}
```

Step 4:

- Now perform **ant build** , start the server and Go to site and Register
- Now your entered State in the RegistrationForm will directly Store in the database.
- To check whether the value is Stored or not by the following procedure

Go to Hmc

url: localhost:9001/hmc/hybirs

open User ----->Customer search using registered mail id

open the customer see the value of State

Task 2:-

Aim: Display(State) in profile page i.e, value has to be Retrived form the databse using **Populator & Converter**

MY ACCOUNT**PROFILE****Profile**

Address Book

Payment Details

Order History

Title: Mr.

First Name: kumar

Last Name: ajay

New Email Address: b2@gmail.com

State : telangana

create state attribute and it should
Display the attribute value in profile page

Change your password

Update personal details

Update your email

Step 1:

- Create an attribute in jsp (to display in front end i.e UI)
- Go to **accountprofilepage.jsp** and add the code,

accountprofilepage.jsp

```
<tr>
    <td><spring:theme code="profile.state" text="State"/>: </td>
    <td>${fn:escapeXml(customerData.state)} </td>
</tr>
```

Step 2:

- create a label in **base_en.properties** file.
- **base_en.properties** file is used to display the **labels** in jsp/tags
- Now search **base_en.properties** file (ctrl+shift+R ,i.e., resources <project_name>storefront\web\webroot\WEBINF\messages**base_en.properties**), which is nothing but base or basic properties file.
- go to **base_en.properties** file to add the code,
i.e. profile.state=State

Step 3:

- Now create State(attribute) in Customer Data Class object to capture the value from model class
- Now add code in <project_name>**facades-beans.xml** in CustomerData i.e. for Data object,

```
<project_name>facades-beans.xml
<bean class="de.hybris.platform.commercefacades.user.data.CustomerData"
    extends="de.hybris.platform.commercefacades.user.data.PrincipalData">
    <property name="state" type="String" />
</bean>
```

- Again perform **ant build**, then property will be created in related Data class i.e, open **CustomerData** class you can see State logic is added. **Eg:**

CustomerData.java,

```
public class CustomerData {
private String state;
    public void setState(final String state)
    {
        this.state = state;
    }
    public String getState()
    {
        return state;
    }
}
```

Step 4:

- Now we need to populate related model class property value into the Data class(CustomerData)

Model class -----> Data class

Method 1:

Using Default Populator & Converters

- Now go to **CustomerPopulator** which implements Populator and add the code in populate() method, i.e. after setUid(source,target), because after this only target nothing but Data object is available.

Eg:

CustomerPopulator.java

```
target.setState(source.getState());
```

- Now perform **ant build** and start the server.
- Go to site and see the field in **Profile** Page

Note:

We should neither add nor do any modifications in Default classes provided by Hybris ie., don't write any code in **CustomerPopulator**. Instead of that follow another method

Method 2: Using Our Created populator and Converter

Step 1:

- Create class **<project_name>Populator.java** and it should extend the **CustomerPopulator**, Eg:


```
public class ShoppersStopCustomPopulator extends CustomerPopulator
```

- Now override the **populate()** method i.e,

```
<project_name>ReversePopulator.java
```

```
public class ShoppersStopCustomPopulator extends CustomerPopulator
{
    @Override
    public void populate(final CustomerModel source, final CustomerData target)
    {
        target.setState(source.getState());
    }
}
```

Step 2:

- Configure the <project_name>Populator bean in <project_name>facades-spring.xml

```
shoppersstopfacades-spring.xml
```

```
<bean id="ssPopulator"
class="com.techouts.shoppersstop.facades.populators.ShoppersStopCustomPopulator"
parent="customerPopulator" />
```

- Now configure for custom Populator(*ssPopulator*) and default Populator(*customerPopulator*) in the <project_name>facades-spring.xml

```
shoppersstopfacades-spring.xml
```

```
<alias name="defaultCustomerConverter" alias="customerConverter" />
<bean id="defaultCustomerConverter" parent="abstractPopulatingConverter">
    <property name="targetClass"
        value="de.hybris.platform.commercefacades.user.data.CustomerData" />
    <property name="populators">
        <list>
            <ref bean="customerPopulator" />
            <ref bean="ssPopulator" />
        </list>
    </property>
</bean>
```

Note : comment the added code in **CustomerPopulator**

Step 3:

- Now perform **ant build** , start the server.
- Go to site and see the field in **Profile** Page

Flexible Search Query

This is a very flexible search to get data from D.B . here we no need to know the table field name...only by name of class or object we can able to retrieve the data from database..

Here we are writting the flexible query and internally it is converting to sql query to retrieve data from database.

It is used only to search, not to delete, not to update ,not to insert...i.e, only for retriving from database..

it search for item type & attribute name

here we have two thing:

1.FlexibleSearchQuery-->which is responsible for writing query.

2.FlexibleSearchService-->which is responsible to search....in which two methods mostly we used **getModelByExample**(for getting a single entry of object) and **getModelsByExample**(for getting list of entries or objects from db).
apart from that we have **ModelService** class which is responsible for **create**, **remove** ,**save** the data in the Data base.

Task1:-Email validation whethere the email is alredy exist in the database or not using ajax and getModelsByExample

from register.tag

we taken the value *register.email* (here just change register.email to registe_email because ajax not working id contain .)

Register.tag
<pre><formElement:formInputBox idKey="register_email" labelKey="register.email" path="email" inputCSS="text" mandatory="true"/></pre>

then we created customeremail.js file in place like:

D:\Hybris5.4\hybris\bin\custom\yepme\yepmestorefront\web\webroot_ui\desktop\common\js\customeremail.js

customeremail.js
<pre>\$("#register_email").blur(function(){ var emailjquery=\$("#register_email").val();</pre>

```

alert("email id is:"+emailjquery);

$.ajax({
    url:'login/check',
    data:{'emailId':emailjquery},
    success: function(result){
        document.getElementById("res").innerHTML=result;
    },
    error:function(result){
        alert("sorry.....email is not proccessed");
    }
});

});

```

then we place this code in **js.tag** file

js.tag

```

<script type="text/javascript" src="${commonResourcePath}/js/customeremail.js"></script>

```

Create Controller, custom facade , custom service and custom dao layer we have to add these code;

Note:-

If u create controller in own package we need to configure in Springmvc .xml file otherwise u create the controller in already configured package

CustomerController:-----

```

/**
 *
 */
package com.techouts.pillowtalk.storefront.controllers.pages;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

import com.techouts.pillowtalk.CustomFacade.PillowtalkCustomFacadeImp;

```

```

/**
 * @author jagadish
 */
@Controller
@RequestMapping(value = "/login")
public class PillowtalkCustomController
{
    @Autowired
    private PillowtalkCustomFacadeImp facade;

    @ResponseBody
    @RequestMapping(value = "/check")
    public String checkemail(@RequestParam("emailId") final String email)
    {
        final String status = facade.isEmailPresent(email);
        return status;
    }
}

```

PillowtalkCustomFacade:-----

```

/**
 *
 */
package com.techouts.pillowtalk.CustomFacade;

/**
 * @author jagadish
 */
public interface PillowtalkCustomFacade
{
    public String isEmailPresent(final String mail);
}

```

PillowtalkCustomFacadeImp:-----

```

/**
 *
 */
package com.techouts.pillowtalk.CustomFacade;

import org.springframework.beans.factory.annotation.Autowired;

import com.techouts.pillowtalk.CutomService.PillowtalkCustomServiceImp;

/**
 * @author jagadish
 *
 */
public class PillowtalkCustomFacadeImp implements PillowtalkCustomFacade
{
    @Autowired
    private PillowtalkCustomServiceImp service;

    /**
     * (non-Javadoc)
     * @see
     com.techouts.pillowtalk.CustomFacade.PillowtalkCustomFacade#isEmailPresent(java.lang.
     String)
     */
    @Override
    public String isEmailPresent(final String email)
    {
        // YTODO Auto-generated method stub

        final int status = service.getEmail(email);
        if (status == 1)
        {
            return "email is not valid";
        }
        else
        {
            return null;
        }
    }
}

```

PillowtalkCustomService:-----

```

/**
 *
 */

```

```
package com.techouts.pillowtalk.CutomService;
```

```
/**
```

```
 * @author jagadish
```

```
 *
```

```
 */
```

```
public interface PillowtalkCustomService
```

```
{
```

```
    public int getEmail(final String mail);
```

```
}
```

```
PillowtalkCustomServiceImp:-----
```

```
/**
```

```
 *
```

```
 */
```

```
package com.techouts.pillowtalk.CutomService;
```

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
import com.techouts.pillowtalk.CustomDao.PillowtalkCustomDaoImp;
```

```
/**
```

```
 * @author jagadish
```

```
 *
```

```
 */
```

```
public class PillowtalkCustomServiceImp implements PillowtalkCustomService
```

```
{
```

```
    @Autowired
```

```
    private PillowtalkCustomDaoImp dao;
```

```
    /*
```

```
     * (non-Javadoc)
```

```
     *
```

```
     * @see
```

```
com.techouts.yepme.customservice.YepmeCustomServiceInterface#getEmail()
```

```
     */
```

```
    @Override
```

```
    public int getEmail(final String email)
```

```
    {
```

```
        // YTODO Auto-generated method stub
```

```
        System.out.println("in service implementation class");
```

```
        return dao.getTheEmail(email);
```

```
    }
```

```
}
```

PillowtalkCustomDao:-----

```
/**
 *
 */
package com.techouts.pillowtalk.CustomDao;

/**
 * @author jagadish
 *
 */
public interface PillowtalkCustomDao
{
    public int getTheEmail(final String mail);
}
```

PillowtalkCustomDaoImp:----

```
/**
 *
 */
package com.techouts.pillowtalk.CustomDao;

import de.hybris.platform.core.model.user.CustomerModel;
import de.hybris.platform.servicelayer.search.FlexibleSearchService;

import java.util.List;

import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;

/**
 * @author jagadish
 *
 */
public class PillowtalkCustomDaoImp implements PillowtalkCustomDao
```

```

{

    private static final Logger LOG = Logger.getLogger(PillowtalkCustomDaoImp.class);
    @Autowired
    FlexibleSearchService flexibleSearhService;

    /*
     * (non-Javadoc)
     *
     * @see
    com.techouts.pillowtalk.CustomDao.PillowtalkCustomDao#gettheEmail(java.lang.String)
     */
    @Override
    public int getTheEmail(final String mail)
    {
        // YTODO Auto-generated method stub

        LOG.info("in dao impl using getModelsByExample() method");
        final CustomerModel user = new CustomerModel();
        user.setOriginalUid(mail);
        final List<CustomerModel> result =
flexibleSearhService.getModelsByExample(user);
        try
        {
            final CustomerModel email = result.get(0);
            LOG.info("emial is:" + email.getOriginalUid());
            return 1;

        }
        catch (final Exception e)
        {
            LOG.info("good choose for email id");
            return 0;

        }
    }
}

```

Register the Following Class in facdes-Spring.xml like this

<Yourproject>facdes-Spring.xml

<bean id="dao" class="com.techouts.pillowtalk.CustomDao.PillowtalkCustomaoImp"/>


<bean id="service"
class="com.techouts.pillowtalk.CutomService.PillowtalkCustomServiceImp"/>

<bean id="facade"
class="com.techouts.pillowtalk.CustomFacade.PillowtalkCustomFacadeImp"/>

ModelService

Aim : ModelService.save();

Take one default Address Form in Profile Page and save it in database using ModelService.save()



Welcome kumar | [My Account](#) | Call us: +1 302 295 5067 | [Find a Store](#) | [Sign Out](#) | your shopping cart **1** \$99.85

Brands | Digital Cameras | Film Cameras | Hand Held Camcorders | Power Supplies | Flash Memory | Camera Accessories & Supplies

HOME > MY ACCOUNT > PROFILE

MY ACCOUNT

PROFILE

Profile

Address Book

Payment Details

Order History

Title: Mr.

First Name: kumar

Last Name: ajay

New Email Address: b2@gmail.com

State : telangana

Change your password

Update personal details

Update your email

Default address

create this button, by clicking this it should display the default address form using ajax

Address:

Accelerator
About Commerce Accelerator
FAQ

Hybris
About hybris
Contact Us

Follow Us
Agile Commerce Blog
Linked In

Facebook
Twitter

© 2015 hybris software

Steps to use modelservice.save();

Step 1:

- Create a new column in the existing table in Data base using **item.xml**
- Go to **core-items.xml** and Copy customer item type and paste in **extensionName-item.xml**
- Now create ' **defaultAddress** ' attribute in ' **customer** ' item type.
- Perform **ant build**(it creates variable in a CustomerModel.java).

shoppersstopcore-items.xml

```
<itemtype code="Customer" extends="User"
          jaloclass="de.hybris.platform.jalo.user.Customer"
autocreate="false"
          generate="true">
  <attributes>
    <!-- auto ID which is generated by NumberSeries -->
    <attribute autocreate="true" qualifier="defaultAddress"
              type="java.lang.String">
      <modifiers read="true" write="true" search="true"
                optional="true" />
      <persistence type="property" />
    </attribute>
  </attributes>
</itemtype>
```

Step 2:

- Go to /shoppersstopstorefront/web/webroot/WEB-INF/tags/desktop/address/
- create a new Tag file i.e.
- write one form with the extension of tag(ex: defaultAddress.tag)

defaultAddress.tag

```
<form action="addDefaultAddress" id="newadd">
<input type="text" name="defaultAddress">
<input type="submit" value="submit">
</form>
```

Step 3:

- Now go to **accountprofilePage.jsp** and Embed created tag file in that.
- Give the path of tag file in **accountprofilePage.jsp**

```
<%@ taglib prefix="add" tagdir="/WEB-INF/tags/desktop/address" %>
```

- Now in **accountprofilePage.jsp** add one tag. Eg:

accountprofilePage.jsp

```
<a class="button" id="addNewAdress" ><spring:theme text="Default Address"/></a>
<add:defaultAddress></add:defaultAddress>
```

Step 4:

- Go to /shoppersstopstorefront/web/webroot/_ui/desktop/common/js/
- create a .js file **addNewAddress.js**
- write jquery for the created form in js folder to hide form

addNewAddress.js

```
$("#newadd").hide();
$("#addNewAddress").click(function(){
    $("#newadd").toggle();
});
```

- The above code will hide our form.
- If we click **Default Address** button, then it shows the text box.
- Now configure the created .js file in **js.tag** file. Then only our js will work.

```
<script type="text/javascript" src="{commonResourcePath}/js/addNewAddress.js"></script>
```

Step 5:

- We need to write a method in Controller to handle the request and it can be done in two ways

Method 1:

- By writing the method in **AccountPageController.java** and call the **modelService.save()**
- Go to **AccountPageController.java**, create controller method in that

AccountPageController.java

```
@RequestMapping(value = "/addDefaultAddress",method = RequestMethod.GET)
public void addDefaultAddress(@RequestParam("defaultAddress")final String defaultAddress){

    String uid =customerFacade.getCurrentCustomerUid();
    final CustomerModel model =new CustomerModel();
    model.setOriginalUid(uid);
    final CustomerModel cmodel= flexiblesearchservice.getModelByExample(model);
    cmodel.setDefaultAddress(defaultAddress);
    modelService.save(cmodel);

}
```

Note: Here we have to declare FlexibleSearchService,ModelService classes in the **AccountPageController** class.

```
@Resource
private FlexibleSearchService flexibleSearchService;

@Resource
private ModelService modelService;
```

Method 2:

- Create our own Facade Interface and write implementation class for it and extend it to DefaultCustomerFacade
- we have to follow the pattern like Controller-->Facade->Service--->Dao
- Now write the controller method in **AccountPageController.java** and call the Facade Layer.

Step 1:

- Create defaultAddress(attribute) in Data Class to capture the value from jsp
<project_name>facade-beans.xml

```
<project_name>facade-beans.xml  
  
<bean class="de.hybris.platform.commercefacades.user.data.CustomerData">  
    <property name="defaultAddress" type="String"/>  
</bean>
```

- Now perform **ant build**, then property will be created in related Data class **Eg:**

```
CustomerData.java  
  
public class CustomerData {  
  
    private String defaultAddress;  
  
    public void setDefaultAddress(final String defaultAddress)  
    {  
        this.defaultAddress = defaultAddress;  
    }  
  
    public String getDefaultAddress()  
    {  
        return defaultAddress;  
    }  
}
```

Step 2:

- We have to create our own Facade Interface
 - Go to /shoppersstopfacades/src/com/techouts/shoppersstop/facades/
create a package in that i.e., **customfacades**
 - Now create an Interface in that i.e., ShoppersstopCustomerAddressFacade
- Eg:

```
interface ShoppersstopCustomerAddressFacade
```

- Write a method **addDefaultAddress()** with **CustomerData** as a parameter.

```
ShoppersstopCustomerAddressFacade.java
```

```
public interface ShoppersstopCustomerAddressFacade
{
    public void addDefaultAddress(final CustomerData customerData);
}
```

Step 3:

- Now write a class which implements **ShoppersstopCustomerAddressFacade** and the Facade class must extend to **DefaultCustomerFacade**
- Go to the path /shoppersstopfacades/src/com/techouts/shoppersstop/facades/customfacades create a package in that i.e., **impl**
- create the implementation class in that package i.e.,

create **DefaultShoppersstopCustomerAddressFacade.java**

Eg:

```
DefaultShoppersstopCustomerAddressFacade extends DefaultCustomerFacade implements
ShoppersstopCustomerAddressFacade
```

- override method **addDefaultAddress()** and pass the **CustomerData** to it

DefaultShoppersstopCustomerAddressFacade.java

```
@Autowired
FlexibleSearchService flexibleSearchService;

@Override
public void addDefaultAddress(final CustomerData customerData)
{
    final String uid = getCurrentCustomerId();
    final CustomerModel customerModel = new CustomerModel();
    customerModel.setOriginalUid(uid);

    final CustomerModel customerModel2 =
flexibleSearchService.getModelByExample(customerModel);

    //to populate from Data class to Model Class
    customerModel2.setDefaultAddress(customerData.getDefaultAddress());

    getModelService().save(customerModel2);
}
```

Note: Here we have to declare FlexibleSearchService object,

```
@Autowired
FlexibleSearchService flexibleSearchService;
```

Step 4:

- Now Configure the **Default<project_name>CustomerAddressFacade.java** object in **<project_name>facades-spring.xml**

Eg:

shoppersstopfacades-spring.xml
<pre><!-- creates an object for DefaultShoppersstopCustomerAddressFacade class --> <bean id="shoppersstopCustomerAddressFacade" class="com.techouts.shoppersstop.facades.customfacades.impl.DefaultShoppersstopCustomerAddressFacade" parent="defaultCustomerFacade"/></pre>

Step 5:

- Go to AccountPageController.java ,create controller method in that

AccountPageController.java
<pre>@RequestMapping(value = "/addDefaultAddress",method = RequestMethod.GET) public void addDefaultAddress(@RequestParam("defaultAddress")final String defaultAddress){ final CustomerData customerData = customerFacade.getCurrentCustomer(); customerData.setDefaultAddress(defaultAddress); shoppersstopCustomerAddressFacade.addDefaultAddress(customerData); }</pre>

Note: Here we have to declare ShoppersstopCustomerFacadeDefaultAddress object,

AccountPageController.java
<pre>@Autowired ShoppersstopCustomerAddressFacade shoppersstopCustomerAddressFacade;</pre>

Step 6:

- Now perform **ant build** , start the server and Go to site and add Default Address
- Now your entered Address will directly Store in the database.
- To check whether the value is Stored or not by the following procedure

Go to Hmc

url: localhost:9001/hmc/hybirs

open User ----->Customer search using registered mail id

open the customer see the value of DefaultAddress

here we have to declare FlexibleSearchService,ModelService classes in the

AccountPageController class.

@Resource
private FlexibleSearchService flexibleSearchService

@Resource
private ModelService modelService

Actually here we have to follow the pattern like controller-->facade->service--->dao.but every thing I wrote in controller only.

You should follow this pattern like task1 metioned

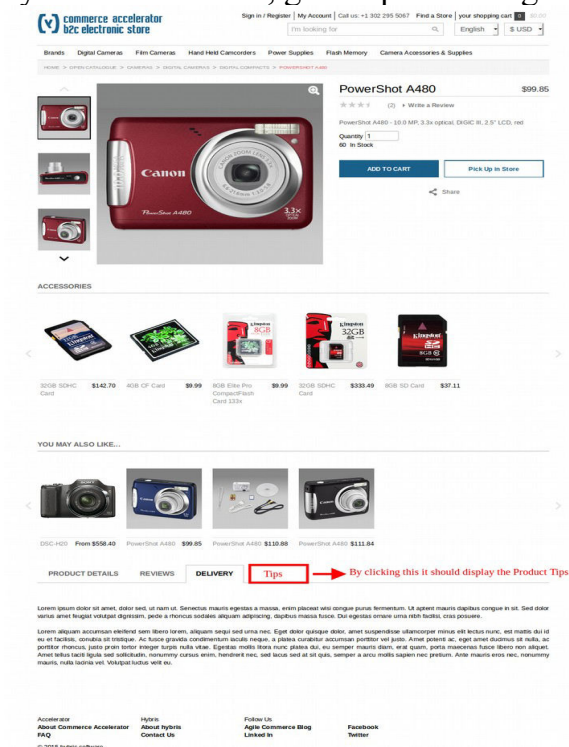
Task:3 Adding Tab(Tips) To The Product Page

Aim: Create a **Tab(Tips)** in product page

Step1:

- To display tab in front end, go to productPageTab tag file ,

i.e.



<yourprojectname>storefront/web/webroot/WEB-INF/ tags/desktop/product/
productPageTabs.tag and add below code in it.

ProductPageTabs.tag

```
<div class="tabHead">
```

```

        <spring:theme code="product.tips" />
    </div>

    <div class="tabBody">
        <product:productTipsTab product="{product}" />
    </div>

```

- Create **productTipsTab.tag** and add below code:

ProductTipsTab.tag

```

<%@ tag body-content="empty" trimDirectiveWhitespaces="true" %>
<%@ attribute name="product" required="true"
type="de.hybris.platform.commercefacades.product.data.ProductData" %>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="template" tagdir="/WEB-INF/tags/desktop/template" %>
<%@ taglib prefix="product" tagdir="/WEB-INF/tags/desktop/product" %>
<%@ taglib prefix="ycommerce" uri="http://hybris.com/tld/ycommercetags" %>

    <div class="productTips">
        ${product.tips}
    </div>

```

Step2:

- To display label on jsp , go to **base_en.properties** ,find product related data in it and add below code in it

base_en.properties

```
product.tips =Tips
```

Step3:

- Now to create the tips attribute in the ProductData class, add below code in **<yourprojectname>facades-beans.xml** ,

shoppersstopfacades-spring.xml

```

<bean class="de.hybris.platform.commercefacades.product.data.
        ProductData">
    <property name="tips" type="String" />
</bean>

```


Step4:

- Now create tips field in the productModel
- Go to `<yourprojectname>core-items.xml` ,add the below code:

shoppersstopcore-items.xml

```
<typegroup name="shoppersstop">

<itemtype code="Product" extends="GenericItem"
          jaloclass="de.hybris.platform.jalo.product.Product"
          autocreate="false" generate="true">

  <attributes>
    <attribute autocreate="true" qualifier="Tips"
              type="localized:java.lang.String" generate="true">
      <persistence type="property" />
      <modifiers read="true" write="true" search="true"
                initial="true" optional="true" unique="true" />
    </attribute>
  </attributes>
</itemtype>
</typegroup>
```

Step5:

- Now we need to populate related model class(**productModel**) property value into the Data class(**productData**)

Model class -----> Data class

- Now Create custom populator i.e., `<project_name>Populator.java` and it should extend the **CustomerPopulator**, Eg:

```
public class ShoppersStopProductPopulator implements Populator<ProductModel,ProductData>
```

- Now override the **populate()** method i.e,

ShoppersStopProductPopulator.java

```
public class ShoppersStopProductPopulator implements Populator<ProductModel,ProductData>
{
    @Override
    public void populate(final ProductModel source, final
                        ProductData target) throws ConversionException
    {
        if (source.getTips() != null)
        {
            target.setTips(source.getTips());
        }
    }
}
```

```
}  
}
```

Step 6:

Now Configure the `<project_name>Populator` bean in `<project_name>facades-spring.xml`

shoppersstopfacades-spring.xml

```
<bean id="ssProductPopulator"  
  
class="com.techouts.shoppersstop.facades.ShoppersstopProductPopulator">  
</bean>
```

- Now configure for custom Populator(`ssProductPopulator`) and default Populator(`productPopulator`) in the `<project_name>facades-spring.xml`

shoppersstopfacades-spring.xml

```
<alias name="defaultProductConverter" alias="productConverter" />  
  <bean id="defaultProductConverter" parent="abstractPopulatingConverter">  
    <property name="targetClass"  
  
value="de.hybris.platform.commercefacades.product.data.ProductData" />  
    <property name="populators">  
      <list>  
        <ref bean="productPopulator" />  
        <ref bean="ssProductPopulator" />  
      </list>  
    </property>  
  </bean>  
</beans>
```

- Now Perform **ant build** and start the server
- go to product page and check whether tips tab has come or not.
- Go to hmc --> catalogue --->product-->

search for any product and go to administration add data to **Tips** and save

- Now to refresh the site and check the data in **Tips tab**

