Interceptors

- To intercept the behavior of life cycles of Models, there are various types of interceptors

- Each interceptor addresses a particular step of the life cycle

- When the life cycle of a model reaches a certain step, a corresponding interceptor is activated
- During the interception, we can perfom two operation

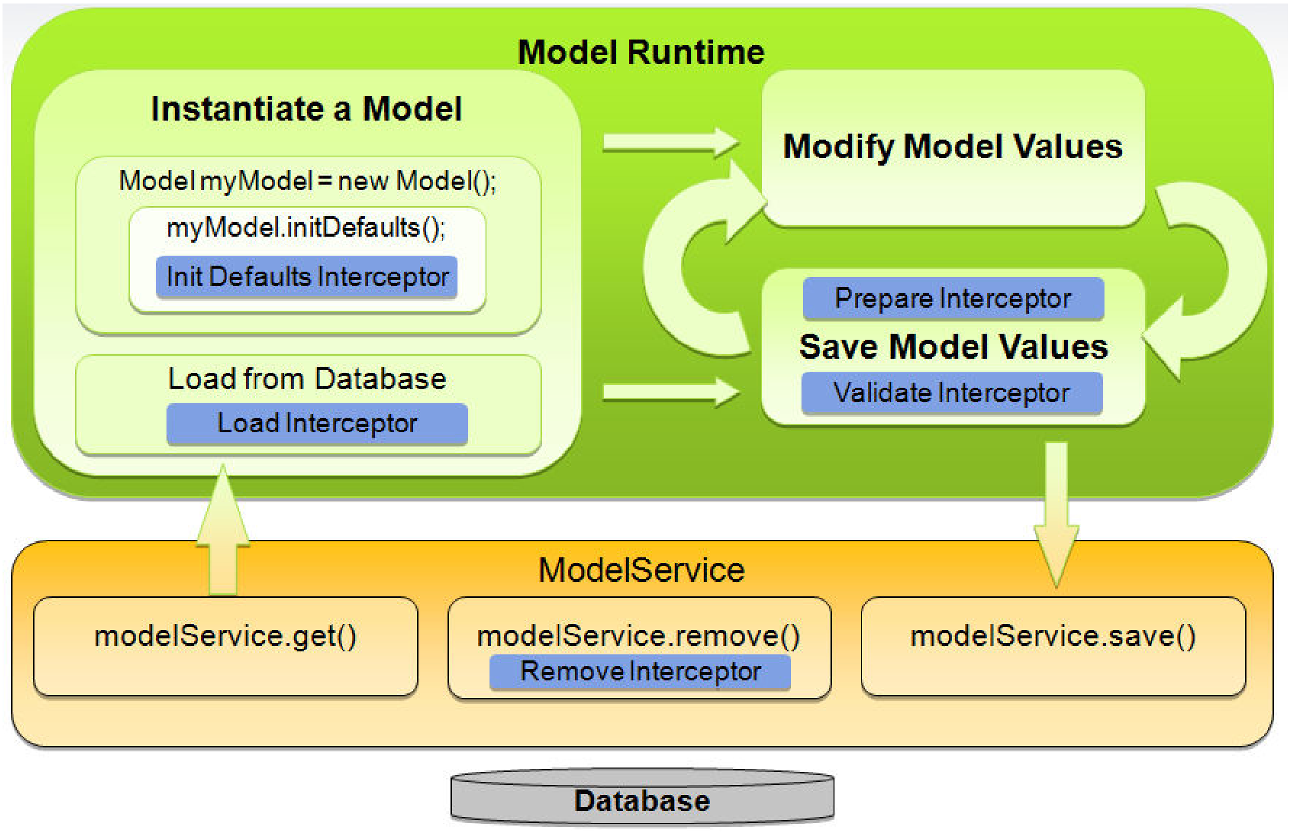
  1)Modify or perform operations on the model

  2)Raise an exception

Figure: Life cycle of a model. The blue boxes indicate where you can affect the model using Interceptors.

There are five types of interceptors,these ar listed bellow

- Load Interceptor
- Init Defaults Interceptor
- Prepare Interceptor



- Validate Interceptor
- Remove Interceptor

Load Interceptor:

- The Load Interceptor is called whenever a model is loaded from the database.

- You may want to use this interceptor if you want to change values of the model after load. An exception raised during execution prevents the model from being loaded.

Load Interceptor:

LoadInterceptor interface (de.hybris.platform.servicelayer.interceptor package):
public interface LoadInterceptor extends Interceptor
{
   void onLoad(Object model, InterceptorContext ctx) throws InterceptorException;
}

Init Defaults Interceptor:

- The Init Defaults Interceptor is called when a model is filled with its default values.

- This happens either when it is created via the modelService.create method or when the modelService.initDefaults method is called.

- You can use this interceptor to fill the model with additional default values, apart from the values defined in the `items.xml` file; see items.xml for details.

Init Defaults Interceptor

InitDefaultsInterceptor interface (de.hybris.platform.servicelayer.interceptor package):
public interface InitDefaultsInterceptor extends Interceptor
{
 void onInitDefaults(Object model, InterceptorContext ctx)throws InterceptorException;
}

Prepare Interceptor

- The Prepare Interceptor is called before a model is saved to the database before it is validated by Validate interceptors, see below.
- Use this to add values to the model or modify existing ones before they are saved. An exception raised during execution prevents the model from being saved.

Prepare Interceptor

PrepareInterceptor interface (de.hybris.platform.servicelayer.interceptor package):
public interface PrepareInterceptor extends Interceptor
{
void onPrepare(Object model, InterceptorContext ctx) throws InterceptorException;
}

Validate Interceptor

- The Validate Interceptor is called before a model is saved to the database after is been prepared by the Prepare interceptors, above.

- You can use Validate Interceptors to validate values of the model and raise an InterceptorException if any values are not valid.

Validate Interceptor

```
       public interface ValidateInterceptor extends Interceptor
  {
    void onValidate(Object model, InterceptorContext ctx)  throws InterceptorException;
 }
```

Remove Interceptor

- The Remove Interceptor is called before a model is removed from the database. You can use this interceptor, for example:
- To remove models that are related to the model but are not in the model context.
- To prevent the removal of the model by raising an InterceptorException.

Remove Interceptor

```
       public interface RemoveInterceptor extends Interceptor
 {
   void onRemove(Object model, InterceptorContext ctx) throws InterceptorException;
}
```

Register an Interceptor:

After implementing an interceptor you register it as a Spring bean.

1. To register an interceptor, add it to the Spring application context XML:

myextension-spring.xml

```
<bean id="myValidateInterceptor" class="mypackage.MyValidateInterceptor"
     autowire="byName"/>
```

The id is used in the following bean.

Add a de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping to the XML file:

myextension-spring.xml

```
<bean id="MyValidateInterceptorMapping"
     class="de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping">
   <property name="interceptor" ref="myValidateInterceptor"/>
   <property name="typeCode" value="MyType"/>
   <property name="replacedInterceptors" ref="uniqueCatalogItemValidator"/>
   <!-- The order property is only effective with 4.1.1 and later -->
   <property name="order" value="5000"/>
</bean>
```

Task1: - Creating a RemoveInterceptor that Stores Deleted Users in a Separate Table.

1) Define an item that stores the data of each deleted user:

lycamobilecore-items.xml

```
<itemtype code="UserAuditEntry" generate="true" autocreate="true">
        <deployment table="UserAuditEntries" typecode="8998" />
        <attributes>
                <attribute qualifier="uid" type="java.lang.String">
                        <persistence type="property" />
                </attribute>
                <attribute qualifier="name" type="java.lang.String">
                        <persistence type="property" />
                </attribute>
                <attribute qualifier="displayName" type="java.lang.String">
                        <persistence type="property" />
                </attribute>
                <attribute qualifier="changeTimestamp" type="java.util.Date">
                        <persistence type="property" />
                </attribute>
        </attributes>
</itemtype>
```

2) Create an interceptor that creates an instance of the above item each time a user is deleted:

AuditingUserRemoveInterceptor.java

```
public class AuditingUserRemoveInterceptor implements
RemoveInterceptor
{
@Override
public void onRemove(final Object o, final InterceptorContext ctx)
throws InterceptorException
{
if (o instanceof UserModel)
{
final UserModel user = (UserModel) o;
final UserAuditEntryModel auditEntryModel =
ctx.getModelService().create(UserAuditEntryModel.class);
auditEntryModel.setChangeTimestamp(new Date());
auditEntryModel.setDisplayName(user.getDisplayName());
auditEntryModel.setName(user.getName());
auditEntryModel.setUid(user.getUid());
ctx.registerElementFor(auditEntryModel,
PersistenceOperation.SAVE);
}
}
}
```

3) Register the interceptor in the spring context using InterceptorMapping.

myextension-spring.xml

```
<bean id="myRemovalInterceptor"
class="com.lycamobile.AuditingUserRemoveInterceptor"
autowire="byName"/>
```

The id is used in the following bean
Add a de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping to the XML file:

myextension-spring.xml

```
<bean id="MyValidateInterceptorMapping"
class="de.hybris.platform.servicelayer.interceptor.impl.Intercepto
rMapping">
<property name="interceptor" ref="myRemovalInterceptor"/>
<property name="typeCode" value="customer"/>
<property name="replacedInterceptors"
ref="uniqueCatalogItemValidator"/>

<!-- The order property is only effective with 4.1.1 and later -->
<property name="order" value="5000"/>
</bean>
```

Note:-Now go to Hmc select one customer and right click remove then interceptor is called and removed details are stored in given item type for this see in hac with Flexibule Search Query search you are given item type then you get the values

Task2:Validating UserAuditEntryModels

Optionally, we can consider a more contrived scenario where we want to validate UserAuditEntryModels and only allow those where the username is not empty.

To achieve this goal, let's define the following ValidateInterceptor:

AuditEntryValidateInterceptor.java

```
public class AuditEntryValidateInterceptor implements ValidateInterceptor
{
   @Override
   public void onValidate(final Object o, final InterceptorContext ctx) throws InterceptorException
   {
      if (o instanceof UserAuditEntryModel)
      {
         final UserAuditEntryModel auditEntry = (UserAuditEntryModel) o;
         if (StringUtils.isEmpty(auditEntry.getName()))
         {
            throw new InterceptorException("User audit entries cannot have empty username");
         }
      }
   }
}
```

When this interceptor is registered, all UserAuditEntries created by the **AuditingUserRemoveInterceptor** are validated with the above interceptor. If the validation fails, all changes are rolled back (the user is not removed and **UserAuditEntry** is not created).