Items.xml

- Core.items.xml
- **Items.xml**: it defines the type of an extension into xml format
- **Items**.xml file contains the type definition in hybris

Structure of items.xml

<items xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceschemaLocation="items.xsd">

1. Atomictypes
2. Collectiontypes+
3. Enumtypes
4. Maptypes
5. Relations
6. Itemtypes

<items>
< atomictypes>
< /atomictypes>

<collectiontypes>
< /collectiontypes>

<enumtypes>

</enumtypes>

<maptypes>
</maptypes>

<relations>
</relations>

<itemtypes>
</itemtypes>

</items>

Note: we need to follow the structure of itemstypes otherwise build will file

--- if object is catalog un aware than we extend GenricItem.

--- in Hyris Order and cart both extend AbstractOrder

-- orderEntryModel and cartEntryModel both extend AbstractOrderEntryModel

--- Language,country and currency extend C2LItem type

---- SAP recommends us to create Subtypes as it keeps SAP Core types untouched.

Sample items.xml

<items ..... >
<atomictypes>

<automictype class="java.lang.object" autocreate="true" generate="false" />
<atomictype class="java.lang.Number" extends="java.lang.Object" autocreate="true" generate="false" />
</atomictypes>

<collectiontypes>

<collection code ="samleCollection" elementtype="Item" autocreate="true" generate="false" />
</collectiontypes>

<enumtypes>
<enumtype code = "TestEnum"

<value code = "abc"/>

</enumtypes>
<enumtype code="QuoteState" autocreate="true" generate="true" dynamic="true">
<value code="CREATED"/>
<value code="DRAFT"/>
<value code="SUBMITTED"/>
<value code="OFFER"/>
<value code="ORDERED"/>
<value code="CANCELLED"/>
<value code="EXPIRED"/>
</enumtype>

<maptypes>

<maptype code="myFirstMap" argumenttype="language" returntype="java.math.BigInteger" autocreate ="true" generate="true"

</maptypes>

here argumnettype is key and return type is value

<relations>

<relation code="User2Orders" generate="true" localized="false" autocreate="true">
<sourceElement type="User" cardinality="one" qualifier="user">
<modifiers read="true" write="true" search="true" optional="false"/>
</sourceElement>
<targetElement type="Order" cardinality="many" qualifier="orders">
<modifiers read="true" write="true" search="true" optional="true" partof="true"/>
</targetElemewnt>
</relation>

</relations>

a. autocreate=true means , item will be created in DB during initialization.
It hints hybris to create a new database entry for this type at initialization/update process
If we set it to false,build will fail.
We should set it to true for the first definition of item type

b. Qualifier : Qualifier of attribute which will be generated at type configured for opposite relation end.If navigable is not set to false , the qualifier is mandatory.Default is empty

c. carinality : the cardinality of this relation end. Choose 'one' for 'one' part of 1:n relation or 'many' when part of a n:m relation. A 1:1 relation is not supported.Default is many
d. qualifier :

<itemtypes>
<itemtype code="AttributeDescriptor"
extends="Descriptor"
jaloclass="de.hybris.platform.jalo.type.AttributeDescriptor"
deployment="de.hybris.platform.persistence.type.AttributeDescriptor"
autocreate="true"
generate="false">
<attributes>
<attribute qualifier="databaseColumn" type="java.lang.String" autocreate="true">
<persistence type="cmp" qualifier="columnNameInternal"/>
<modifiers read="true" write="true" search="false" optional="true" removable="true"/>
</attribute>

</attributes>

</itemtype>

<itemtypes>

</items>

how db treats

In one to many , we have one attribute at source side which will have list of all pks (comma separate target pk)

in many to one - we have attribute called source pk at target end.in source side we have getter method which will query db for all target tables and see which target table has source pk = current one

many to many - relation between teacher and subject . teacher can teach multiple subjects
+
comman table - 2 attr. sourCe pk and target pk
link table - linking between source and target
when you delete this link , original entries don't get deleted ony link get deleted.

many to many -

<deployment table="MyContactInfo" typecode="12000"/>
This process of defining the table for the item type in items.xml is called deployment.
Table name and typecode should be unique globally.

it increase [performance. query the data directly from thsi table.

Typecode value should be greater than 10000 and less than 32767.

Typecode is used during the PK generation mechanism and hence it should be unique

Typecode values between 0 & 10000 are reserved by Hybris

typecode specifies the unique number which will be used internally by Hybris to reference the type.
Its value should be between 0 & 32767 and should be unique.
If we use the typecode which is already exists then build will fail.
typecode values between 0 and 10000 are reserved for hybris internal use, typecode values greater than 10000 can be used.

generate=true at the item type level - It hints hybris to generate a new jalo class for this type during build time. If we set it to false,then jalo class will not be generated however model class will always be generated. We should set it to true for the first definition of item type.

autocreate is true, which lets the hybris Commerce Suite to create a new database entry for this type at initialization/update process. Setting the autocreate modifier to false causes a build failure. The first definition of a type has to enable this flag.

==========

Modifiers for attributes
read = true means attribute is readable and corresponding getter method will be generated for the attribute
Default value is true.

write=true means attribute is writable and corresponding setter method will be generated for the attribute
Default value is true.

search=true means attribute can be searchable by a FlexibleSearch
Default value is true.

Optional=true means attribute is not mandatory
Default value is true.

To make any attribute as mandatory, set optional as false

==========

in attributes in item type :
persistent type = "property" - this attribute is going to be stored ito db table

persistent type = "dynamic" - it is going to be stored into memory.once hybris server is down , all memory will be washed out

persistent type="jalo"
Jalo attributes have non-persistent values, and are defined in a <persistence type="jalo"> tag in the items. xml file. ... Unlike persistent attributes, the values of jalo-only attributes are held in memory and not written to the SAP Commerce database. The values of jalo-only attributes exist only during runtime.

Every object in hybris saves in Generic item table by default

Dynamic attrubtes : Dynamic attributes enables you to add attributes to the Models , and to create custom logic behind them , without touching SAP commerce model class itself.

Dynamic attributes have non persistent values
We write an handler where we write our custom logic while declaring a dynamic attribute.

<attributes>

<attribute type="int" qualifiier="humanReadableDays">
<persistence type="dynamic" attributeHandler="myHandlerId" />
<modifiers read="true" write="false" />

</attributes>

==========

==========

========

There are 3 ways of defining an item type in Hybris.

We need to decide one of the ways based on the requirement.

What are those 3 ways?

1) Define the new item type without extending any existing item type

2) Define the new item type by extending it with existing item type

3) Define the existing item type again with new attributes

1) Define new item type without extending any existing item type

In this case, we are supposed to define the new item type in our items.xml file without extending any of the existing item type.

Requirement:
Store third party integration system credentials like username and password in DB and access them while making a third party call.

As per the requirement,we need to define one new item type which can store username and password.

So create a new item type with attributes as below

```
<itemtype code="IntegrationSystemCredentials" autocreate="true" generate="true">
        <deployment table="IntegrationSystemCredentials" typecode="11000" />
        <attributes>
          <attribute qualifier="code" type="java.lang.String">
            <modifiers optional="false" unique="true"/>
            <persistence type="property" />
          </attribute>
          <attribute qualifier="username" type="java.lang.String">
            <modifiers unique="false"/>
            <persistence type="property" />
          </attribute>
          <attribute qualifier="password" type="java.lang.String" >
            <modifiers unique="false" encrypted="true"/>
            <persistence type="property" />
          </attribute>
        </attributes>
      </itemtype>
```

We have defined a new item type called IntegrationSystemCredentials which has 3 attributes as explained below

Code: It will be unique identifier for each record
Username: username of the third party system
Password: password of the third party system

autocreate=true at the item type level
It hints hybris to create a new database entry for this type at initialization/update process
If we set it to false,build will fail.
We should set it to true for the first definition of item type.

generate=true at the item type level
It hints hybris to generate a new jalo class for this type during build time.
If we set it to false,then jalo class will not be generated however model class will always be generated.
We should set it to true for the first definition of item type.

Deployment table and typecode
Deployment specifies the table name where all the instances of the item type are stored.

typecode specifies the unique number which will be used internally by Hybris to reference the type.
Its value should be between 0 & 32767 and should be unique.
If we use the typecode which is already exists then build will fail.
typecode values between 0 and 10000 are reserved for hybris internal use, typecode values greater than 10000 can be used.

Attributes of an item type
We can define as many attributes as we want for an item type.
Each attribute has to specify the data type,modifiers and persistent type.
Type can be either primitive type or can be reference to any existing type.

Modifiers for attributes
read = true means attribute is readable and corresponding getter method will be generated for the attribute
Default value is true.

write=true means attribute is writable and corresponding setter method will be generated for the attribute
Default value is true.

search=true means attribute can be searchable by a FlexibleSearch
Default value is true.

Optional=true means attribute is not mandatory
Default value is true.
To make any attribute as mandatory, set optional as false

persistent type for attributes
Persistent type can be either property or dynamic.
If it is set as property then value will be stored in DB and it's called persistent attribute.
If it is set as dynamic then value will not be stored in DB and it's called dynamic attribute.

2) Define new item type but extend it with existing item type

In some cases we need to define new item type but we have to extend the existing item type.

Requirement:
Add a new functionality such that any product in our system should hold the list of genders and that product is used only for those genders list.
If product "A" is designed for only Male, the Product A gender list will have only Male element in it.
If product "B" is designed for both Male and Female then product "B" gender list will have 2 elements for Male and Female.

As per the requirement, we have to make sure that Product item type will have list of genders as a new element.

So let's define one new item type called "ApparelProduct" which will extend existing item type called Product and add the new property as gender list.

<itemtype code="ApparelProduct" extends="Product"

```
        autocreate="true" generate="true" jaloclass="org.training.core.jalo.ApparelProduct">
        <description>Base apparel product extension that contains additional attributes.</description>
        <attributes>
          <attribute qualifier="genders" type="GenderList">
            <description>List of genders that the ApparelProduct is designed for</description>
            <modifiers />
            <persistence type="property" />
          </attribute>
        </attributes>
      </itemtype>
```
We can see that autocreate and generate set as true as it's a new item type.

3) Define the existing item type again with new attributes

In some cases we need to add attributes to the existing item type without defining a new item type.
In this case we just need to define the item type with existing item type code and add new attributes.

Requirement:
Add a new flag to the Address type to identify it as permanent address or not.

We need to add a new Boolean flag for Address type.

let's define the Address type with existing item code as below

```
<itemtype code="Address" autocreate="false" generate="false">
        <attributes>
<attribute qualifier="permanentAddress" type="java.lang.Boolean">
            <description>PermanentAddress</description>
            <defaultvalue>Boolean.FALSE</defaultvalue>
            <modifiers read="true" write="true" search="true" optional="false"/>
            <persistence type="property"/>
          </attribute>
        </attributes>
```
We can see that item type with code as Address is defined already in core-items.xml file and we have defined it again with same code.
So its not a new item type and hence autocreate and generate is set as false.

It means we are adding the new attribute directly in the existing item type without creating a new item type.

So newly added attribute will be generated in existing Address model class.
Because of which, we have to give autocreate and generate as false.

Deployment and Typecodes in items.xml
We know that in Hybris items can be persisted in database.
It means they will be stored in the tables.

So we need to specify the table name while defining the item type so that the values of the item type will be persisted in that table.

This process of defining the table for the item type in items.xml is called deployment.

Each instances of an item type is stored as one row in the table.

In Deployment tag we specify table name and typecode as below.

<deployment table="MyContactInfo" typecode="12000"/>
Table name and typecode should be unique globally.

Typecode value should be greater than 10000 and less than 32767.

Typecode is used during the PK generation mechanism and hence it should be unique

Typecode values between 0 & 10000 are reserved by Hybris

Deployment inheritance from a type to its subtypes
If we don't specify the deployment for any of our item type then deployment specified in the closest type in the hierarchy will be considered.

If there is no hierarchy which means if the item type is not extending any item type then by default GenericItem will be considered as the parent of the item type.
And hence the deployment of GenericItem will be used by the item type.

So all the instances of this item type will be stored in the GenericItem table

We can also extend GenericItem for our item type however it will be extended by default if item type is not specified with any parent type.

It's same as Object class for all the java classes.

Example:

\\
<itemtype code="PaymentInfo"
extends="GenericItem"
jaloclass="de.hybris.platform.jalo.order.payment.PaymentInfo"
autocreate="true"
generate="true">
is same as

<itemtype code="PaymentInfo" jaloclass="de.hybris.platform.jalo.order.payment.PaymentInfo"
autocreate="true"
generate="true">

When we should define deployment mandatorily?

We should define deployment for an item type in the following scenarios

1) Defining new item type by extending GenericItem
Example:

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

```
        <deployment table="Units" typecode="10"/>
```
2)Defining new item type by extending existing item type for which there is no deployment
Example:

```
<itemtype code="AbstractOrder"
              extends="GenericItem"
              jaloclass="de.hybris.platform.jalo.order.AbstractOrder"
              autocreate="true"
              generate="true"
              abstract="true">
<itemtype code="Cart"
              extends="AbstractOrder"
              jaloclass="de.hybris.platform.jalo.order.Cart"
              autocreate="true"
              generate="true">
         <deployment table="Carts" typecode="43"/>
```
deployment is not defined for AbstractOrder as its abstract type,Cart is extending AbstractOrder , so deployment must be defined for Cart.

Note:
1) If we don't specify deployment for the above scenarios then build will fail.
If we want to pass the build and let items to be stored in GenericItem table then define below property in the local.properties file

build.development.mode=false
This is not advisable because storing many item types in GenericItem table will decrease the performance and possibility of data truncation due to columns limit in the table.

2)Deployment table should not be defined for any Item type if there is already a deployment defined for its super type otherwise it will decrease the performance as it has to perform multiple joins while retrieving.

Example:
```
<itemtype code="Product"
              extends="GenericItem"
              jaloclass="de.hybris.platform.jalo.product.Product"
              autocreate="true"
              generate="true">
         <deployment table="Products" typecode="1"/>

<itemtype code="ApparelProduct" extends="Product"
             autocreate="true" generate="true" jaloclass="org.training.core.jalo.ApparelProduct">
             <description>Base apparel product extension that contains additional attributes.</description>
             <attributes>
                <attribute qualifier="genders" type="GenderList">
                   <description>List of genders that the ApparelProduct is designed for</description>
                   <modifiers />
                   <persistence type="property" />
                </attribute>
             </attributes>
         </itemtype>
```
In this case Its not advisable to specify the deployment for ApparelProduct as it is extending Product and Product has the deployment defined already.

Defining collection in items.xml
Collection basically contains the elements of element type.
element type means the type of elements we are adding into the collection.

Example:
Collection of String will contain the elements of String element type.
Collection of Integer will contain the elements of Integer type.
Collection of Address will contain the elements of Address type.

Note:
Element type can be a simple atomic type or it can be any other complex type.

We define collection of string as below

<collectiontype code="StringCollection" elementtype="java.lang.String" autocreate="true" generate="false"/>
We have not specified the type,so it takes Collection of String by default.

We define List of string as below

<collectiontype code="StringCollection" elementtype="java.lang.String" autocreate="true" generate="false" type="list"/>
We have specified type=list so It will be List of String.

We define Set of string as below

<collectiontype code="StringCollection" elementtype="java.lang.String" autocreate="true" generate="false" type="set"/>
In all the above definitions, elementtype is String which is an atomic type

We define Collection of Address as below

<collectiontype code="AddressCollection" elementtype="Address" autocreate="true" generate="false"/>
Here elementtype is Address which is a complex type and also another item type.

We define List of Language as below

<collectiontype code="LanguageList" elementtype="Language" autocreate="true" generate="false" type="list"/>
Here elementtype is Language which is a again another item type

How we use these collections with item type?
Once we have created the collection, we need to inject them to the required item type.

We can inject list of language and list of String to our CustomUser type as below

```
<itemtype code="CustomUser" autocreate="true" generate="false">
        <deployment table="CustomUser" typecode="11001" />
        <attributes>
          <attribute qualifier="userId" type="java.lang.String">
            <modifiers optional="false" unique="true"/>
            <persistence type="property" />
          </attribute>
          <attribute qualifier="languages" type="LanguageList">
```

```
            <modifiers unique="false" read="false"/>
            <persistence type="property" />
          </attribute>
          <attribute qualifier="nickNames" type="StringCollection">
            <modifiers unique="false" read="false"/>
            <persistence type="property" />
          </attribute>
        </attributes>
      </itemtype>
```

We can see that 2nd attribute has a qualifier called “languages”
And its type is “LanguageList” which is basically the code we defined in the collection of Language using below tag

```
<collectiontype code="LanguageList" elementtype="Language" autocreate="true" generate="false" type="list"/>
```

We can see that 3rd attribute has a qualifier called “nickNames”
And its type is “StringCollection” which is basically the code we defined in the collection of String using below tag

```
<collectiontype code="StringCollection" elementtype="java.lang.String" autocreate="true" generate="false"/>
```

The code in the collectiontype tag should become the type for an attribute in the attribute definition.

After adding this change in items.xml file, do the build

CustomUserModel.java file will be generated
Check this file for the attribute called “languages” and “nickNames”

It should be generated as below

```
private List<LanguageModel> _languages;
private Collection<String> _nickNames;
```

So we have defined List of langauge and a collection of String for CustomUserModel.

While defining collection,
make autocreate as true so that collection type gets created in the backend.
make generate as false as we don’t need to generate Jalo class for collection type.

==========
-==========

Enum Type

Enum types are used to define the set of constant values.

Example:
Set of Color like Red,Blue and Set of status like Created,InProgress,Submitted etc.

Enum for Gender to specify Male or Female

```
<enumtype code="Gender" autocreate="true" generate="true">
        <value code="MALE"/>
        <value code="FEMALE"/>
</enumtype>
```

Enum for Phone type to specify its Office or Home or Mobile phone.

```
<enumtype code="PhoneContactInfoType" generate="true" autocreate="true" dynamic="true">
        <description>Phone type</description>
        <value code="MOBILE"/>
        <value code="WORK"/>
        <value code="HOME"/>
    </enumtype>
```

All the Enum types are stored in one table.

It will be inside the deployment of EnumerationValue item type.

We need Enum type and Enum class to be generated for the Enum type,hence make both autocreate and generate as true.

Dynamic in Enum
Dynamic in Enum is completely different from Dynamic attributes.

If an Enum type is dynamic then values can be added at the run time.
We can make Enum type as Dynamic by specifying dynamic=true in the Enum type definition.

If an Enum type is non-dynamic (by default, dynamic=”false”) we are not allowed to add new values at runtime.

If we add any non-dynamic Enum type without values,build will fail as it does not have any effect.

So if you want to add new values at run time we have to make dynamic=”true” for an Enum type.

We can change the flag anytime but enforces a system update.

If dynamic=”false” the servicelayer generates real java enums (having a fixed set of values).

If dynamic=”true” it generates hybris enums which can be used without fixed values(means we can add run time values).

Map Type
We know that Map is a collection of Key value pair.

Key is also called argument and value is also called return value.

We can define Map as below

```
<maptype code="addressMap"
            argumenttype="java.lang.String"
            returntype="Address"
            autocreate="true"
            generate="false"/>
```

we can reference the code defined above as a type to any attribute in the item type definition.

Example:

```
<itemtype code="CustomUser" autocreate="true" generate="false">
            <deployment table="CustomUser" typecode="11001" />
            <attributes>
               <attribute qualifier="userId" type="java.lang.String">
                  <modifiers optional="false" unique="true"/>
                  <persistence type="property" />
               </attribute>
               <attribute qualifier="languages" type="LanguageList">
                  <modifiers unique="false" read="false"/>
                  <persistence type="property" />
               </attribute>
               <attribute qualifier="nickNames" type="StringCollection">
                  <modifiers unique="false" read="false"/>
                  <persistence type="property" />
               </attribute>
               <attribute qualifier="userAddressMap" type="addressMap">
                  <modifiers unique="false" read="false"/>
                  <persistence type="property" />
               </attribute>

            </attributes>
         </itemtype>
```

Now we have defined the Map and injected it into userAddressMap attribute of CustomUser

Do the Build using ant all

Check CustomUserModel.java to have below entry

private Map<String,AddressModel> _userAddressMap;
So we have created a Map of String and Address inside CustomUser item type.

Like collection, we don't need jalo class to be generated for Map type hence set generate as false.
Set autocreate as true so that item type for Map will be generated.

Defining Relation in items.xml in Hybris
Relation types help us to maintain the m:n relation between 2 tables.

Let's consider 2 possible cases in relation

Case 1:
One to Many/Many to One
One user can have any number of contacts

```
<relation code="User2ContactInfos" generate="true" localized="false" autocreate="true">
         <sourceElement type="User" cardinality="one" qualifier="user">
            <modifiers read="true" write="true" search="true" optional="false"/>
         </sourceElement>
```

```
        <targetElement type="AbstractContactInfo" cardinality="many" qualifier="contactInfos"
ordered="true">
            <modifiers read="true" write="true" search="true" optional="true" partof="true"/>
        </targetElement>
    </relation>
```

We can see below entries in model class after build

UserModel.java

```
private Collection<AbstractContactInfoModel> _contactInfos;
```

AbstractContactInfoModel.java

```
private UserModel _user;
```

Since its a One To Many relation, we can observe that Collection is generated in UserModel class and just single User entity is generated in AbstractContactInfoModel class

Case 2:
Many to many
Consider the relation between Product and categories
One product can belong to many categories
And one category can contain many products.
It is m:n relation.

We can define the relation for such case as below

```
<relation code="CategoryProductRelation" autocreate="true" generate="true" localized="false">
        <deployment table="Cat2ProdRel" typecode="143"/>
        <sourceElement qualifier="supercategories" type="Category" cardinality="many"
ordered="false">
            <description>Super Categories</description>
            <modifiers read="true" write="true" search="true" optional="true"/>
        </sourceElement>
        <targetElement qualifier="products" type="Product" cardinality="many" collectiontype="list"
ordered="true">
            <description>Products</description>
            <modifiers read="true" write="true" search="true" optional="true"/>
        </targetElement>
    </relation>
```

We have defined the relation with code “CategoryProductRelation”
Speicifed the deployment table called “Cat2ProdRel”

We have 2 important things here,sourceElement and targetElement.
We can see that cardinality is mentioned as many in source and many in target, so it’s a many to many relation.

We can see qualifier as superCategories in the source which is of type Category.
We can see qualifier as products in the target which is of type Product.
It means Inside a Category it creates a variable called products and inside a Product it creates a variable called superCategories

Generate will have no effect for relation.

Autocreate true makes the relation type to be created.

We have specified collection type as List for target.

After the build, we can see below entries in java class

CategoryModel.java

private List<ProductModel> _products;

ProductModel.java

private Collection<CategoryModel> _supercategories;

How Relation works in backend?

M:N relation

One table will be created for Products
One table will be created for Categories
One extra table will be created for LinkItem(elements on both sides of the relation are linked together via instances of a LinkItem table)

Each LinkItem instance stores the PKs of the related items for each row.

So every row of product with associated category will have one row in the LinkItem table with PK of Catrgory and associated PK of Product.

LinkItem instances are used internally by hybris.
We just need to call getters and setters at the API level to set and get the values.

Dynamic Attribute

As we all know that any attribute we define in item type will have a tag called persistent type.

persistent type=”property”
In this case,
Corresponding column will be created in the database and hence the values will be stored in the DB. So it’s called persistent attribute.

persistent type=”dynamic”
In this case,
There will be no column created in the database and hence values will not be stored in the database.
So it’s called Non persistent or dynamic attribute.

For every dynamic attribute we define, we need to mention the attribute handler otherwise Bean Id will be generated automatically and we have to use the same bean id while defining Spring bean in XML.

Attribute handler is implemented using Spring.

So we need to mention the spring bean id for the attribute handler.

Then we need to define the class for that spring bean id which provides the custom logic for the dynamic attribute.

Note:
It is possible that one item type can have any number of dynamic attributes.

Requirement:
Identify how old the customer is in our site so that we can give some discount to those customers who are 3 years old.

How to achieve it?
We need to get the customer's registration date which we store as creation_time in DB and we need to get current time from Java Date API.

We need to do current_time – creation_time to know how old the customer is.

We can add a new attribute called "customer_site_age" as dynamic attribute.

Let's see the steps for the same.

Step 1:
Define a new attribute in the items.xml for Customer item type and make it as dynamic attribute.

```
<itemtype code="Customer" autocreate="false" generate="false">
            <description>Extending Customer type with additional attributes.</description>
    <attributes>
        <attribute autocreate="true" qualifier="customer_site_age" type="java.lang.Integer">
                    <modifiers read=”true” write="true"/>
                    <persistence type="dynamic"/>
                <description>Dynamic attribute for customer site age</description>
        </attribute>
    </attributes>
</itemtype>
```

Here we have defined the persistent type as dynamic.

Step 2:
Define the Attributehandler which should be the bean id of the class which implements DynamicAttributeHandler.

```
<itemtype code="Customer" autocreate="false" generate="false">
              <description>Extending Customer type with additional attributes.</description>
    <attributes>
        <attribute autocreate="true" qualifier="customer_site_age" type="java.lang.Integer">
                <modifiers write="false"/>
                <persistence type="dynamic" attributeHandler="customerSiteAge"/>
                <description>Dynamic attribute for customer site age</description>
        </attribute>
    </attributes>
</itemtype>
```

We have added attributeHandler=”CustomerSiteAge” in the persistent tag.

Step 3:
Define the class with custom logic

We can define the new class either by extending AbstractDynamicAttributeHandler or by implementing DynamicAttributeHandler

Define it in the trainingcore(your custom core) extension in org.training.core.attributes package

```
public class CustomerSiteAgeHandler extends
AbstractDynamicAttributeHandler<Integer, CustomerModel>
{
  @Override
     public Integer get(final CustomerModel model)
     {
          int customerSiteAge = 0;
          try
          {
               final Date customerRegisteredDate = model.getCreationtime();
               final Calendar cal = Calendar.getInstance();
               cal.setTime(customerRegisteredDate);
               final int registeredYear = cal.get(Calendar.YEAR);
               final int currentYear = Calendar.getInstance().get(Calendar.YEAR);
               customerSiteAge = currentYear - registeredYear;
          }
          catch (final Exception e)
          {
               e.printStackTrace();
          }
          return customerSiteAge;
     }
}
```

Step 4:
Associate the attribute handler specified in items.xml with spring bean id in trainingcore-spring.xml(your_custom_core-spring.xml)

```
<bean id="customerSiteAge"
class="org.training.core.attributes.CustomerSiteAgeHandler"/>
```

Note:
customerSiteAge mentioned as spring bean id above should be same as attributeHandler mentioned in items.xml.

Step 5:
Build and then update the system either using HAC or using ant command

Now whenever we access customer Model, we can also access the customerSiteAgeattribute from customer model,
we will get the result of our custom logic in that attribute.

We can display it in any UI page by setting its value in appropriate controller and model attribute.

Advantages of Dynamic attribute
Data will not be saved in DB as its dynamic

The custom logic is written once and used all the time wherever that attribute is required.

When we should go for Dynamic attribute?

We should choose dynamic attribute whenever we want to get some derived data based on existing values.

So instead of saving one more column, we can make it as dynamic and compute its value based on the current values.

Dynamic in Enum?
Dynamic in enum is completely different from Dynamic attributes.

If an Enumtype is non-dynamic (by default, dynamic=”false”) we are not allowed to add new values at runtime.

If we add any non-dynamic enumtype without values,build will fail as it does not have any effect.

So if you want to add new values at runtime we have to make dynamic=”true” for an enum.

We can change the flag anytime but enforces a system update.

If dynamic=”false” the servicelayer generates real java enums (having a fixed set of values).

If dynamic=”true” it generates hybris enums which can be used without fixed values(means we can add run time values).

==========