# **Development Process**

## **Spring** **MVC**

### **Spring MVC - Form Tags and Data Binding**

#### Text Fields

|  |
| --- |
| 1. Create Form object class  2. Create Controller class  3. Create html form  use form modelattribute to bind the form to the model object  use <form:input>  to bind form inputs to model object fields, use path attribute  4. Create form processing code in Controller handler methods  in handler method that binds object to form, add object to model  in handler methods that retrieves the form w/ model attribute, use @modelAttribute  5. Create confirmation page |

#### Drop-down Lists

|  |
| --- |
| Same as text fields, but in html form:  use <form:select> with <form:option>  if options are dynamic, use <form:select> with <form:options items=${key/value map for options}> |

#### Radio buttons

|  |
| --- |
| Same as text fields, but in html form:  use <form:radiobutton>  if options are dynamic, use <form:radiobuttons items=${key/value map for options}> |

#### Checkboxes

|  |
| --- |
| Same as text fields, but in html form:  use <form:checkbox>  if options are dynamic, use <form:chehckboxes items=${key/value map for options}> |

#### Form Validation - Applying Built-in Validation Rules

|  |
| --- |
| 1. Add validation rule to Form object class  2. Display error messages on html form  3. Perform validation in the Controller class  4. Update confirmation page |

#### Validation - Applying Custom Validation Rules

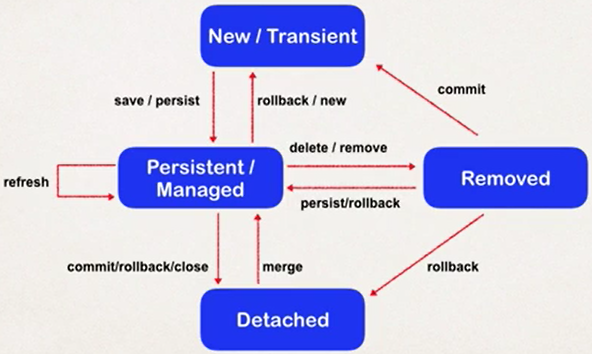
|  |
| --- |
| 1. Create custom validation rule    1. Create annotation    2. Create constraint validator   2. Add validation rule to Form object class  3. Display error messages on html form  4. Perform validation in the Controller class  5. Update confirmation page |

## **Hibernate**

### **Entity Lifecycle**

|  |  |
| --- | --- |
| **Operations** | **Description** |
| Detach | If entity is detached, it is not associated with a Hibernate session |
| Merge | If instance is detached from session, the merge will reattach to session. |
| Persist | Transitions the new instance to managed state. Next flush/commit will save in db. |
| Remove | Transitions managed entity to be removed. Next flush/commit will delete from db. |
| Refresh | Reload/sync object with data from db. Prevents stale data. |

### **Entity Lifecycle – session method calls**



### Note: Entity must be in session to be persistent.

|  |  |
| --- | --- |
| **State** | **How to get to** |
| Transient | Instansiate an entity class |
| Persistent/managed | From Transient:  Begin transaction  Session.save(<transient object>)  From Detached: (must be existing in db)  Session.get(<detached object class>,< detached object id>)  From Persistent/managed:  Session.refresh(<persistent object>) |
| Detached | From Persistent:  Transaction.commit(); |

|  |  |
| --- | --- |
| **Relationship** | **Description** |
| Uni-directional | Parent entity has field/property for detail entity.  (parent record can traverse to detail record but not vice-versa) |
| Bi-directional | Both parent and detail entities have field/properties of each other. |

### **Hibernate Development Process w/o Spring integration**

|  |
| --- |
| 1. Add hibernate config file (hibernate.cfg.xml) 2. Annotate Java class    1. Map class to database table (@Entity, @Table)    2. Map fields to database columns (@Id, @Column) 3. Develop Java code to perform database operations 4. Create session factory from configuration (create only once for entire uptime of app) 5. For each database operation, get a session from session factory    * Get transaction from session & begin transaction    * Perform db crud operations using session    * End transaction 6. On app termination, close session factory (unless used as a managed bean) |

### **Hibernate Advanced Mappings**

#### Notes

* cascading operations will not work using hql, use session
* hql update/delete is faster than retrieving a persisted entity & updating/deleting via session
* For detail record deletion to work, detail record must be disassociated from parent record (set fk to null) before deleting detail record in session
* If using fine-grained cascading (not CascadeType.ALL), if saving a parent entity with its detail entity set, use session.persist(<parent>) instead of session.save(<parent>) to automatically save the detail records in db.
* If parent entity's detail property has @OneToMany with lazy loading fetch type, the detail will not be included when retrieving the parent record. To retrieve lazy-loaded detail, use Hibernate.initialize(parent.getDetail()) while in a session. Note that doing parent.getDetail(); alone will not retain the details outside the session.
* Best practice: prefer lazy loading (load detail records on demand) over eager loading, in general
* When using join fetch, take note that NoResultException will be raised if joining table has no matching records

#### One-to-One, uni-directional

|  |
| --- |
| 1. Create db tables 2. Detail table has pk 3. parent table has pk, fk to detail table (owning side)  * Disable delete & update cascade, to be handled programmatically by hibernate using the defined relationship in step (3)  1. Create detail entity class   Use usual hibernate annotations (@Entity, @Table, @Id, @GeneratedValue, @Column, etc.)   1. Create parent entity class 2. Aside from the usual columns, create a field for the detail entity:  * annotated with @OneToOne(specifying cascade type) * annotate with @JoinColumn(specifying the fk db column name)   Note: JoinColumn = fk column   1. Create main app 2. When instantiating main entity, set its field for detail entity 3. Start transaction 4. Persist the main entity (main details will be created as well in db) 5. Commit transaction |

#### Note on Bi-directional annotation use with @OneToMany/@ManyToOne

Where the 2 tables below have a relationship:

|  |  |
| --- | --- |
| **Table has foreign key?** | **Bi-directional annotation in Entity fk property** |
| Yes | @OneToOne/@ManyToOne(cascade)  @JoinColumn(“<db column name of fk>”)  Use these annotations on other table property on this entity class. |
| No (this table has the pk referred to by fk in other table) | @OneToOne/@OneToMany(mappedBy=”<property name of fk property on entity having the foreign key>”, cascade, fetch)  Use these annotations on other table property on this entity class. |

#### Note on uni-directional annotation use with @OneToMany/@ManyToOne

Where the 2 tables below have a relationship:

|  |  |
| --- | --- |
| **Table has foreign key?** | **Uni-directional annotations** |
| Yes | @OneToOne/@ManyToOne(cascade)  @JoinColumn(“<db column name of fk>”)  Note: this entity has no reference to other table (since it is uni-directional) |
| No (this table has the pk referred to by fk in other table) | @OneToOne/@OneToMany  @JoinColumn(“<db column name of fk in other table>”)  Use these annotations on other table property on this entity class. |

#### One-to-One, bi-directional

|  |
| --- |
| 1. Create db tables 2. Detail table has pk  * same steps as One-to-One, uni-directional   Note: no need to add fk field to parent table id   1. parent table has pk, fk to detail table (owning side)  * same steps as One-to-One, uni-directional  1. Create detail entity class 2. same steps as One-to-One, uni-directional 3. Add field for parent entity & annotate:  * @OneToOne, define attributes:   + mappedBy: specify fk property in parent that is referencing the detail id   + cascade: specify cascade operations that can now be effected to parent table  1. Create parent entity class  * same steps as One-to-One, uni-directional  1. Create main app 2. same steps as One-to-One, uni-directional 3. when creating records, associate parent & detail records:  * set parent details * set detail parent |

#### One-to-Many, bi-directional

|  |
| --- |
| 1. Create db tables 2. parent table has pk 3. Detail table has pk, and fk column to parent table’s pk 4. Create entity class: 5. Create parent entity class    * Aside from usual fields, add field for detail class & annotate:  * @OneToMany, define attributes:   + mappedBy: detail entity property name   + cascade: specify cascade ops that will be effected to detail records   + fetch: lazy or eager. Use lazy if separate detail records retrieval is desired when retrieving parent record.  1. Create detail entity class    * Aside from usual fields, add field for parent class & annotate:    * @ ManyToOne, define attribute:    * cascade: specify cascade ops that will be effected to parent record  * @JoinColumn, define attribute: * name: db column name of fk column in detail table  1. Create main app  * when creating records, associate parent & detail records: * set parent details * set detail parent |

#### Default Fetch Types

|  |  |
| --- | --- |
| **Mapping** | **Default Fetch Type** |
| @OneToOne | Eager |
| @OneToMany | Lazy |
| @ManyToOne | Eager |
| @ManyToMany | Lazy |

|  |  |
| --- | --- |
|  |  |

### **Hibernate Advanced Mappings**

#### One-to-One, uni-directional

|  |
| --- |
|  |