(21F) CST8277 Enterprise Application Programming

Assignment 2

Please read this document carefully – if your submission does not meet the requirements as stated here, you may lose marks even though your program runs. Additionally, this assignment is also a teaching opportunity – **material presented in this document will be on the quizzes and/or the final exam!**

Submission

Zip your project folder and submit on Brightspace/Activities/Assignments/Assignment 2.
You do not need to answer the questions in this document, they are there to guide you.

Theme for Assignment 2

After completing this assignment, you will have achieved the following:

1. Use JSF in conjunction with new EE components: EJBs and JPA
2. Create a model class for JPA with all appropriate mappings/annotations
3. Use a JSF `@ViewScoped` managed bean
4. Add validation to JSF views
5. Use Bootstrap CSS framework to layout and style the Employee Directory application as an ***SPA*** **(*Single Page Application)***

Grading and Requirements for Assignment 2

1. There are TODO markers in the assignment.
2. Complete the **index.xhtml**.
    a. There are rendered tags missing which must be added. Read the documentation in the code. Look for visible keywords.
    b. Some components are also broken which need fixing. Look for TODO in the documentation.
    c. Use **f:validator** and **f:validateLength** to add validator to **h:inputText**. Look for validator in the documentation.

-1 to -5 per mistake depending on the severity of the mistake. Out of 100.

Submitting Assignment 2

Import

1. **Unzip** your project.
2. In **Eclipse** go to **File/Open Projects from File System**.
3. **Navigate** to your **project folder**.
4. When importing there should **only be 1 project selected**.

Important

1. After opening the project in your Eclipse.
2. Right-click your **project** and go to **Properties**.
3. Go to **Deployment Assembly**.
4. Click **Add** and choose **Java Build Path Entries**.
5. Click **Next** and choose **Maven Dependencies**.
6. Click **Finish**.

Remember, if you right-click Maven->Update Project, you have to do the above steps again.

Your Info

You **must** use the skeleton project provided for this assignment to start your solution.

In the folder **`src/main/resource`**, there is a file called **`Bundle.properties`** which contains constants used in the UI – please change the default values for:

```
footer.labsection=Lab Section 301
footer.studentnumber=040123456
footer.studentname=Jane Doe
```

Note – your name should be as it appears in ACSIS.

Your submission must include:

- **Code** – completed project that compiles, has good indenting, not overly complicated and is efficient.
    - The code must demonstrate your understanding of how a JSF application works.
- Eclipse has an ‘export’ function to create an external ‘archive’ – i.e. a zip file – of your project. Please export the project to a file that follows the naming: ***`studentLastName_studentFirstName_studentNumber_21SAssignment2.zip`***

Starting Assignment 2

Project starts with many errors. That is fine. You need to find and fix the issues. There are hints such as TODO markers in the code which can help you.

Update the DB

Attached to the assignment on Brightspace, there is a DB script (**databank.sql**) which you must run on your DB again.

Updating the Application - Additional EE Components: EntityManager

EntityManager

To connect the DAO to the DB, we need to **inject** a reference (into the DOAImpl) using the `@PersistenceContext` annotation:

- **Q1: What is the folder/name of the standard JPA descriptor document (config)?**
- **Q2: What is the name of the Persistence Unit?**
  (Hint – look in the 'non-Java' Maven 'resources' folder)

Java Persistence API (JPA)

The PersonPojo model needs additional JPA annotations in order to work. It also needs new member fields to handle the `CREATED`/`UPDATED`/`VERSION` DB columns. The type for the `CREATED`/`UPDATED` member fields should be `java.time.Instant`.

```
...
@ViewScoped
@Entity(name="some-entity-name")
@Table(name="some-table-name")
@Access(AcessType.???)
@EntityListeners({PersonPojoListener.class})
@NamedQueries(
    @NamedQuery(name=PersonPojo.PERSON_FIND_ALL, query = "select something"),
    ...
)
public class PersonPojo implements Serializable {
    private static final long serialVersionUID = 1L;

    public static final String PERSON_FIND_ALL =
        "Person.findAll";
...
```

- **Q3: What should the `@Entity` name be?** What would it be if the name is not set?
- **Q4: What should the `@Table` name be?** What would it be if the name is not set?

- **Q5: What is the JPQL query string** for the @NamedQuery 'Person.findAll' that retrieves all persons from the database?
- **Q6: What are the column names** for firstName, lastName? What would JPA 'think' the DB column names are if the annotations were not set?
- **Q7: If annotations are on fields what should @Access(AcessType.??) be?**
- **Q8: What type should the member field version be?**

Audit Columns

The purpose of the **CREATED**/**UPDATED** columns is to answer the following questions:

- When was the record *created*?
- When was the record last *updated*?

These columns are typically referred to as 'Audit' columns, they allow you to know when a given row changed. This has obvious benefits in production systems where fraud or security breaches are a concern. Often, DBAs like to solve this problem with *database triggers* or logic that causes a table row to be updated whenever a particular action takes place:

```
CREATE OR replace TRIGGER set_create_date_for_orders
  BEFORE INSERT ON orders
  FOR EACH ROW
BEGIN
    -- Update create_date column to current system date
    :new.create_date := SYSDATE;
END;
```

The above **PLSQL** code (**P**rocedural **L**anguage for **SQL**) works only on Oracle databases. Accomplishing the same task across multiple databases can greatly increase the complexity of an application when support of multiple database vendors is required.

We can implement a solution using JPA that works across **all** databases. In the previous section, we discussed adding two new member fields to handle the **CREATED**/**UPDATED** columns. To populate the audit member fields automatically, we use a JPA 'listener' that is invoked whenever a particular event occurs – there are seven events (Hint! Quizzes/final exam material):

1. @PrePersist – executed before the EntityManager *persist* operation is actually executed
2. @PreRemove – executed before the EntityManager *remove* is actually executed
3. @PostPersist – executed after the EntityManager *persist* operation (INSERT SQL already committed)
4. @PostRemove – executed after the EntityManager *remove* operation (DELETE SQL already committed)
5. @PreUpdate – executed before the UPDATE SQL is executed
6. @PostUpdate – executed after the UPDATE SQL is committed
7. @PostLoad – executed after an entity has been loaded into the current @PersistenceContext or when entity has been refreshed (EntityManager *refresh* operation)

```
import javax.persistence.PrePersist;
import javax.persistence.PreUpdate;
...
    @PrePersist
    public void setCreatedOnDate(arg) {
        // TODO
    }
    @PreUpdate
    public void setUpdatedOnDate(arg) {
        // TODO
    }
...
```

The `PersonPojo` class is missing the `@EntityListeners` annotation, it needs to be added. Please refer to the **JSF Regional Inventory 2** sample project discussed during Week 5 lecture.

- **Q9: What is the argument to** `setCreatedOnDate/setUpdatedDate` **?**

Updating the Application – JSF, EJB, and JPA

The JSF controller is un-aware that some new EE components and APIs (EntityManager and JPA) are now associated with the PersonDao managed bean, its API has changed only slightly. The PersonDaoImpl is also simplified with no direct manipulation of the DB using SQL, all DB operations are delegated to the `EntityManager`.

Our new app uses JPA which needs to be configured and initialized. Luckily, when running inside an EE application server (such as Payara), this is simple:

- In **`payara-resources.xml`**, specify the Payara JDBC connection pool and its mapping to a JNDI name (already done in Assignment 1, no need to change for Assignment 2).
- In the JPA standard persistence deployment descriptor file (that is, **persistence.xml**) (what was the answer to above **Q1** and **Q2**?), ensure that the entry `<jta-data-source>jndi-name</jta-data-source>` is properly setup.

(Note: In the upcoming Assignment 3, we will see how configuring and initializing JPA outside of Java EE is much more complicated!)

Updating the Application - JSF Navigation and SPA

In Assignment 1, the Person application provided C-R-U-D capabilities across three (3) JSF views:

1. `list-people.xhtml`
2. `add-person.xhtml`
3. `edit-person.xhtml`

Even though the application is a simple 'C-R-U-D' list-view, navigating between views can become quite difficult. Amongst the various controller methods are hard-coded strings `"add-person?faces-redirect=true"` and `"list-people?faces-redirect=true"` What if, for example, a 'Sign In' functionality needs to be added to the application:

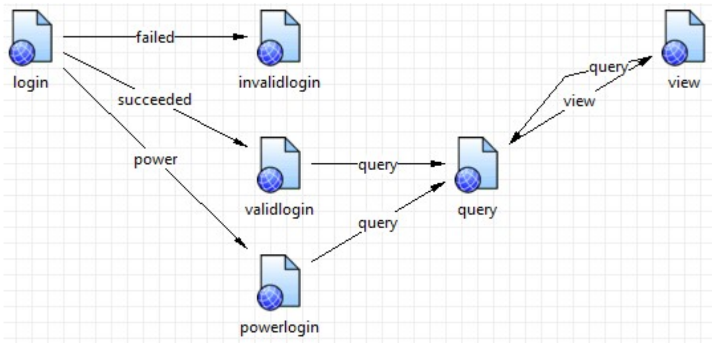

Figure 1. Credit to Russell Bateman 'A short treatise on JSF'(https://bit.ly/2RToMnQ)

JSF supports separating controller code from navigation decisions by allowing the developer to specify navigation rules in the **`faces-config.xml`** file:

```
<navigation-rule>
    <display-name>query</display-name>
    <from-view-id>/query.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>submit</from-outcome>
        <to-view-id>/view.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
<navigation-rule>
    <display-name>view</display-name>
    <from-view-id>/view.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>query</from-outcome>
        <to-view-id>/query.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Even with this separation, it is easy to see that the navigation rules for a fully-featured application would become very hard to maintain. This is one of the motivating factors in the move to single-page applications (SPA), where the web app stays on a single page instead of loading new pages from the server, rewriting its content either via direct DOM manipulation or asynchronously using AJAX calls.[§]

In JSF, the primary way to manage this is to ensure that 'outcomes' of actions invoked on various components ie. `<h:commandButton action="#{customerController.someOutcome()}"/>` is to return a `null String`, or the method signature is changed to return `void`. Over on the 'View' side of things, the primary mechanism to help the user interact with the app is to toggle – hide/show – various on-screen artifacts via their `rendered` attribute (which is present on almost all JSF components), delegating control of the toggle-state to the controller, or in the case of editing an individual row of `<h:dataTable>`, to the model object itself:

- **Q10: What must be added to `PersonController` toggle hide/show the 'Add New Employee' view?**
- **Q11: What must be added to `PersonPojo` toggle hide/show editing?** If a new member field is added to `PersonPojo`, what about the JPA mappings? Should the state of this member field be stored in the DB? If not, how does one prevent that from happening?

Updating the Application - JSF `@ViewScoped` Helper Class

In Assignment 1, to hold the "to-be-created"/"to-be-updated" person data, we either used 'spare' fields in `PersonController` or used the (`@Inject`'d) `sessionMap` to hold them. In Assignment 2, we introduce a new class `NewPersonView` to specifically handle this responsibility.

- **Q12: What annotation(s) should we add to `NewPersonView`?**
- **Q13: What field(s) should we add to `NewPersonView`?**
- **Q14: How is `NewPersonView` used in `index.xhml`?**

- end -

[§] Wikipedia (2020). *Single-page application* (retrieved 2020/02/02 https://en.wikipedia.org/wiki/Single-page_application)