(20F) 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 here in this document will be on Midterm(s) and the Final Exam!**

Assignment 2's submission is to be uploaded to the Assignment 2 Brightspace submission folder. It does not matter if it is a zip-file containing a zip-file containing ... many-levels-of-nested -zip-files or if you upload the required artifacts individually, Brightspace will handle things. If you have a 100% perfect solution but it is not in the correct folder, you will receive a mark of zero ☹ If you know that you made mistake, please upload it to the correct folder ASAP.

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
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*** – a ***Single Page Application***

Note: in Brightspace, I have uploaded a short 1:00 minute video showing – approximately – what the application should look like. You may have different styling or different ways to solving the challenges of Assignment 2, but the basic C-R-U-D functionality must be working and everything is on a single page.

Grading for Assignment 2

Here is a screen capture of the Brightspace Rubric that describes the marking scheme for this assignment (sometimes Brightspace makes it difficult to 'see' the Rubric):

Criteria	**Missing/Poor** 0 points	**Below Expectations** 0.5 points	**Meets or Exceeds Expectations** 1 point	**Criterion Score**
Java Code	Submission is empty, or code does not compile, or code is too similar to that in the 'starter' .zip file, indicating little-to-no progress has been made	• missing Javadocs • poorly formatted • Student's name is not at top of Java files	All requirements met: • JPA Model class(es) • JSF classes - Managed Beans, Validators, etc. • EJBs	/ 1
JPA Entity/Table annotations at top of class	improper annotations	• some functions work • but missing some functionality	Fully-functional JPA Entity: • annotations correct • working JPQL for named query	/ 1
JPA Annotation in body of class	improper annotations	• some functions work • but missing some functionality	Fully-functional JPA Entity: • mappings for existing properties • new member fields and mappings for them	/ 1
JPA EntityListeners		• some functions work • but missing some functionality	Audit properties full functional	/ 1
EntityManager	not injected into proper component	• some functions work • but missing some functionality	Fully-functional with respect to EntityManager	/ 1
JSF Managed bean view	NewCustomerView has improper annotation(s)	• some functions work • but missing some functionality	NewCustomerView fully-functional with respect to CustomerController and Customer XHTML	/ 1
JSF Hide/Show 'Add New Customer' view	'Add New Customer' does not logger between hide/show 'Add New Customer' view does not work	'Add New Customer' view • some functions work • but missing some functionality	'Add New Customer' View fully-functional with respect to CustomerController and customerList.xhtml	/ 1
JSF Hide/Show inline-editing	rows do not toggle between show and editing	• some functions work • but missing some functionality	Hide/Show for inline-editing fully-functional with respect to CustomerController and customerList.xhtml	/ 1
JSF View Refresh	refresh of dataTable does not work	• some functions work • but missing some functionality	Refresh of dataTable fully-functional with respect to CustomerController and customerList.xhtml	/ 1
JSF Validation	field validation missing or does not work	• some functions work • but missing some functionality	Field validation • standard validators • custom validators fully-functional with respect to CustomerController and customerList.xhtml	/ 1
Total				/ 10

Note – the emphasis in Assignment 2 (vis-*à*-vis Assignment 1) is on showing things working (or somewhat working: there are a lot of opportunities for part-marks!)

Submitting Assignment 2

{These instructions are virtually identical as those for Assignment 1}

You **must** use the file from Brightspace ***20FAssignment2_starter.zip*** to start your solution – extract its contents to some work folder on your hard drive. ***Before*** you import the project into Eclipse, please change the **name** of the project to include your Student name and number (this is done in two places). Open the file **pom.xml** in a regular text-editor (eg. Notepad++) and find the **<name>** entry (close to the top of the file):

```
<groupId>com.algonquincollege.cst8277</groupId>
<artifactId>acme-customers</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>war</packaging>
<name>studentNumber_studentName_ACME Customers JSF JPA</name>
<description>Mavenized JSF JPA example</ description>
```

Additionally 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

The starter project is an Eclipse Maven project, choose the 'Existing Maven Projects' Import wizard.

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: ***studentNumber_studentName_20FAssignment2.zip***
- Code style: every class method and every member field needs Javadoc comments
- Every class file has a multiline comment block at the top giving the name of the file (and typically the Instructor's name, too) – you **must** add your Student Name and number
    - ***Important*** – your name ***must*** appear at the top of each and every Java source code file submitted
    - You should **NOT** to do this at the top of XHTML/XML files:

```
<!-- Student Name (040123456) -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
...
```

    - in fact, ***any*** line(s) above <?xml... will cause the project to **fail** to deploy 😟

Starting Assignment 2

You will notice that the starter project has a number of ❌'s in the 'Problems' tab – that is *done on purpose*. You must find and fix the problems and extend the project to provide full C-R-U-D capabilities. **Note: Even after getting rid of the ❌'s, the Application will not deploy!**

Update the CUSTOMER table

In the **`Scripts`** folder, there is a file called **`fixCustomers.sql`** In Assignment 1, we ignored the Db columns **`CREATED`**, **`UPDATED`** and **`VERSION`**. For Assignment 2, we will arrange for JPA to handle those fields; however, they may be null/empty so the following line of SQL will fix that:

```
UPDATE CUSTOMER SET CREATED=CURRENT_TIMESTAMP,
UPDATED=CURRENT_TIMESTAMP, VERSION=1;
```

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?**
- **Q2: What is the name of the Persistence Unit?** (Hint – look in the 'non-Java' Maven 'resources' folder)

Java Persistence API (JPA)

The Customer 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 these new member fields should be `java.time.LocalDateTime`.

```
import static
com.algonquincollege.cst8277.customers2.model.CustomerPojo.ALL_CUSTOMERS_QUERY_NAME;
...
@Entity(name = "some-entity-name")
@Table(name = "some-table-name")
@Access(AcessType.PROPERTY)
@EntityListeners({CustomerPojoListener.class})
@NamedQueries(
    @NamedQuery(name=ALL_CUSTOMERS_QUERY_NAME, query = "select something")
)
public class CustomerPojo implements Serializable {
    private static final long serialVersionUID = 1L;

    public static final String ALL_CUSTOMERS_QUERY_NAME =
        "allCustomers";
...
```

- **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` '`allCustomers`' that retrieves all Employees from the database?
- **Q6: What are the column names** for `firstName`, `lastName`? What would JPA 'think' the Db column name is if the annotation was not set?
- **Q7: Because of `@Access(AcessType.PROPERTY)`, where are the annotations for `created/updated/version` located?**
- **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 refered 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, DBA's like to solve this problem with *triggers*, 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! Midterm/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 `CustomerPojo` class is missing the `@EntityListeners` annotation, it needs to be added.

- **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 CustomerDao Managed bean, its API has changed only slightly. The CustomerDao is also simplified with no direct manipulation of the Db using SQL, all Db operations delegated to the `EntityManager`.

Our new app uses JPA which needs to be configured and initialized. Luckily, when running inside an EE Application Server, 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 deployement descriptor file (What was the answer above to **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 ACME Customer Application provided C-R-U-D capabilities across three (3) JSF views:

1. `list-customers.xhtml`
2. `add-customer.xhtml`
3. `edit-customer.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-customer?faces-redirect=true"` and `"list-customer?faces-redirect=true"` What if for example a 'Sign In' functionality need to be added:

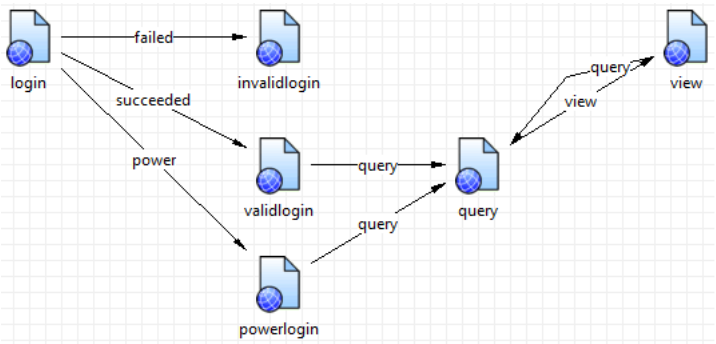

Figure 1 credit Russell Bateman 'A short treatise on JSF' (https://bit.ly/2RToMnQ)

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 manipulate 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 the `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 `CustomerController` toggle hide/show the 'Add New Employee' view?
- **Q11**: What must be added to `CustomerPojo` toggle hide/show editing? If a new member field is added to `CustomerPojo`, 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 toBeCreated/toBeUpdated customer data, we either used 'spare' fields in `CustomerController` or used the (`@Inject`'d) `sessionMap` to hold them. In Assignment 2, we introduce a new class `NewCustomerView` to specifically handle this responsibility

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

- fin -

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