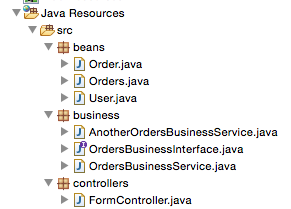
**Short Assignment/In-Class Activity #3 – EJB’s & CDI**

# Short Assignment 3a: Design and Implement Enterprise Java Beans

Objective:

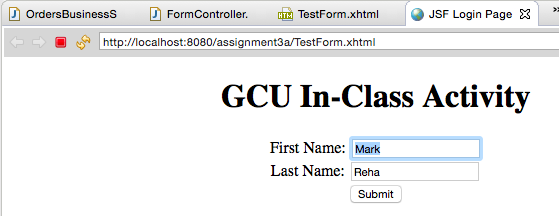
1. Create business services using Enterprise Java Beans.
2. Inject concrete implementation of business service using CDI.

Activity Directions:

Study the EJB and CDI example code from the reading for this Topic. This activity will add code to the previous activity.

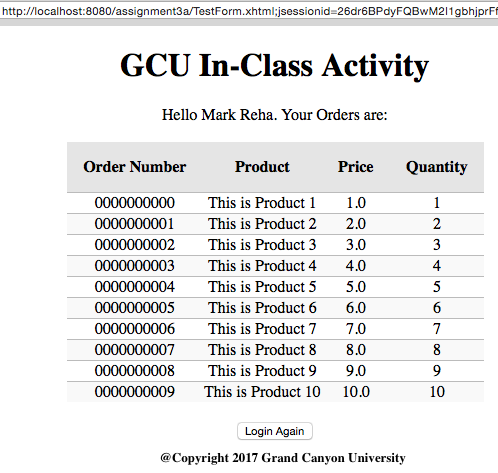
In this activity, we are going to have the following class and package structure. The beans and controller packages should already be done from previous work on this project:

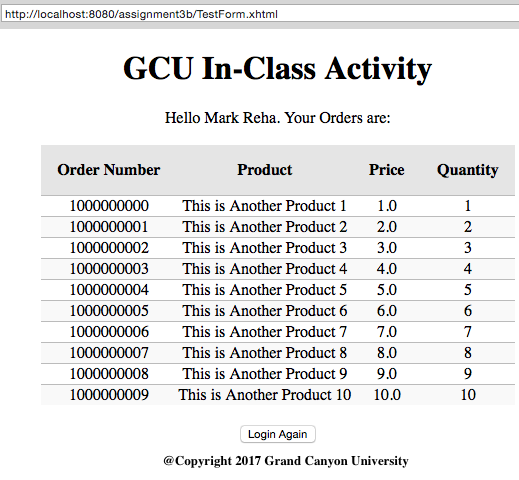
The new material on this project mostly happens in the **business** package.

You can see that there are two different versions of of the business logic class: OrdersBusinessService and AntherOrdersBusinessService. With a small settings change, we will be able to modify our app to use one or the other of these classes.

Preview of how the app will work:

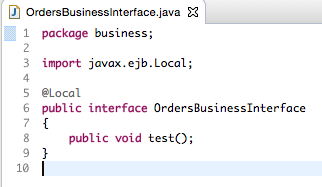
1. Login page.



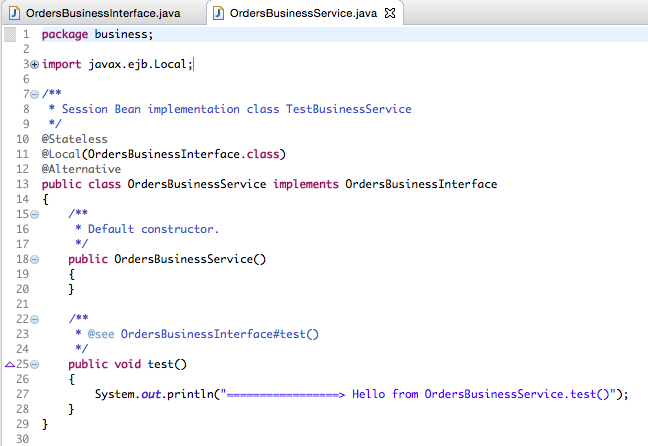
1. Results page on version #1 of the business logic.
2. This is version #2 of the app. Notice the app will look the same, but there are different output results coming from the class controlling the data.

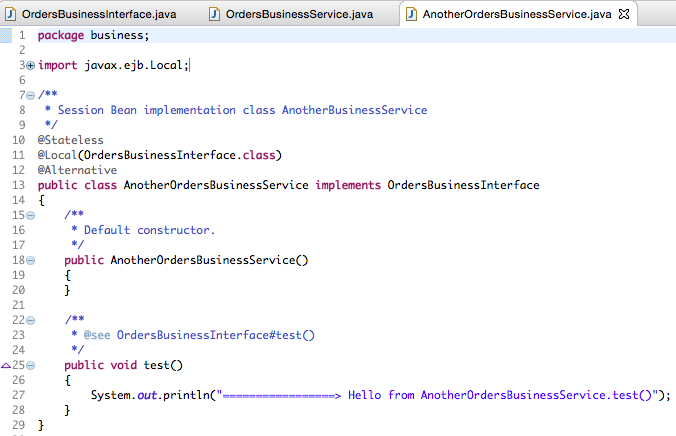
**Instructions**

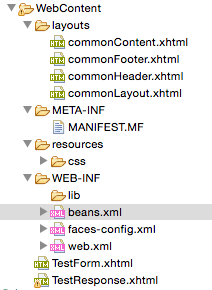
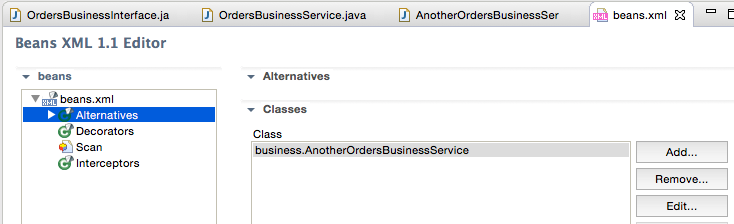
Open the previous version of this project. Add and update the following components in the project:

1. EJB Business Service Interface
   * Create a Java Interface named OrdersBusinessInterface in a new business package. Add the @Local annotation above the interface declaration (make sure to import the javax.ebj.Local package). Add a public void test() method to the interface.

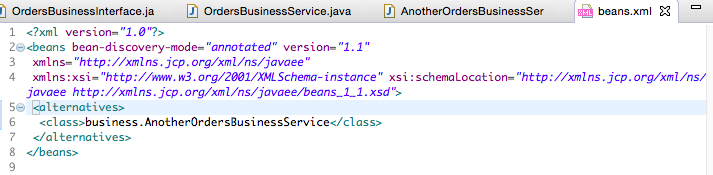
This Interface will be used to implement two different versions of classes that can process the application’s business logic.

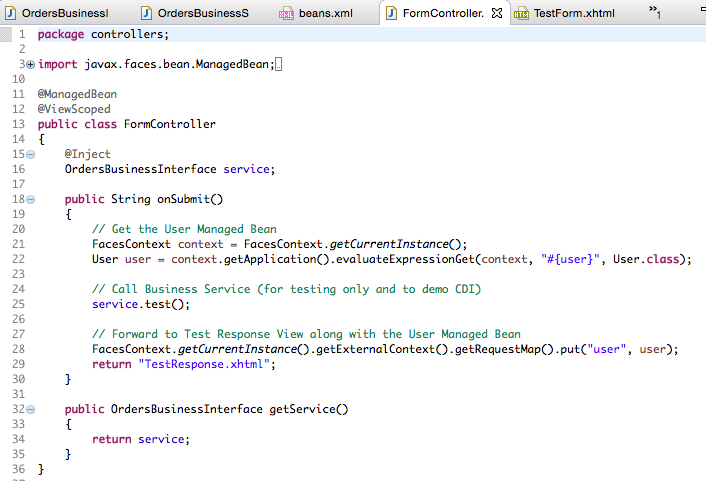
2. Orders Business Service version #1

* + Create an EJB by right clicking on the project and selecting the New->Other... ->EJB->Session Bean (EJB 3.x) menu options.
  + Create the EJB in the business package with a class name of OrdersBusinessService.
  + Use the Stateless state type, with a Container transaction type, and implements the OrdersBusinessInterface interface.
  + Add the @Alternative annotation above the class declaration (make sure to import the javax.enterprise.inject.Alternative package).
  + Implement the test() method that simply prints “Hello from the OrdersBusinessService” using the System.out.println.

1. Orders Business Service version #2
   * Create an EJB by right clicking on the project and selecting the New->Other... ->EJB->Session Bean (EJB 3.x) menu options.
   * Create the EJB in the business package with a class name of AnotherOrdersBusinessService, using the Stateless state type, with a Container transaction type, and implements the OrdersBusinessInterface interface.
   * Add the @Alternative annotation above the class declaration (make sure to import the javax.enterprise.inject.Alternative package).
   * Implement the test() method that simply prints “Hello from the AnotherOrdersBusinessService” using the System.out.println.
2. EJB Configuration:
   * Add EJB configuration file, beans.xml, to the WEB-INF directory. A beans.xml file can be added by right clicking on the WEB-INF directory and selecting the New->beans.xml File menu options.
   * Open the beans.xml file and select the Alternatives bean option. Add the business.OrdersBusinessService as an alternative EJB.

Switch the view from tree to source and you will see the XML code:





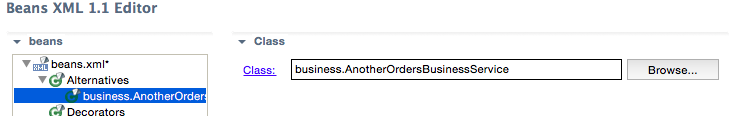
1. JSF Login Controller:
   * Add a class scoped variable service of type OrdersBusinessInterface. Add the @Inject annotation above the variable declaration (make sure to import the javax.inject.Inject package).
   * In the implementation of the onSubmit() call the services test() method.

Deploy the code and execute the application specifying the TestForm.jsf using the internal IDE browser. Submit the form with valid data.

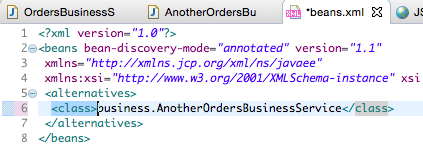
The test string from the OrdersBusinessService should be visible on the Console.



1. Now we will modify the beans.xml file to utilize the second version of the business service implementation. EJB Configuration:
   * Open the beans.xml file and select the Alternatives bean option. Edit the existing entry and change the alternative EJB to the business.AnotherOrdersBusinessService class.



or in xml view



Deploy the code and execute the application specifying the TestForm.jsf using the internal IDE browser.

Submit the form with valid data. Take a screenshot. The test string from the AnotherOrdersBusinessService should be visible on the Console.



# Short Assignment 3b: Using Enterprise Java Beans

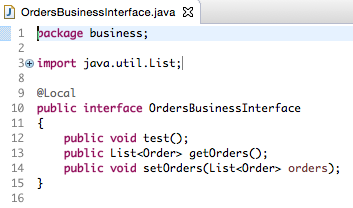
Objective:

1. Create business services using Enterprise Java Beans that implements business logic.
2. Inject concrete implementation of business service using CDI.

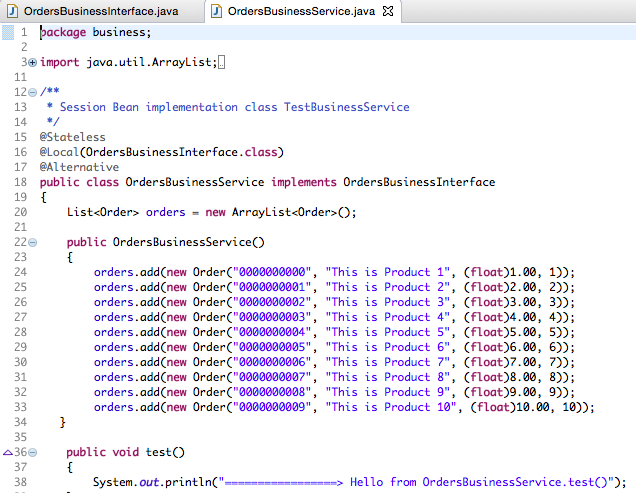
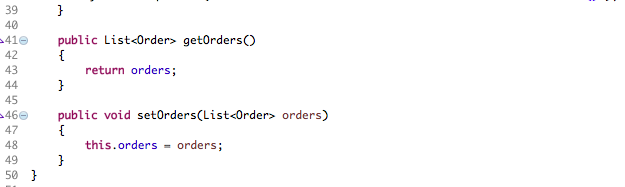
Activity Directions:

Study the EJB and CDI example code from the reading for this Topic. This activity will add code to the previous activity.

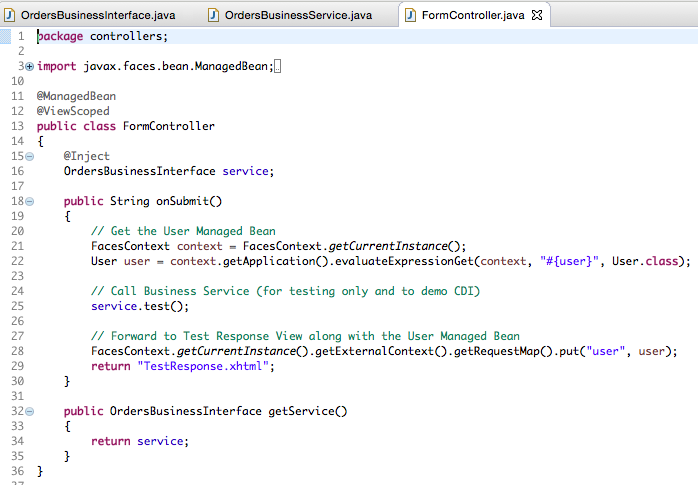
Add and update the following components in the project:

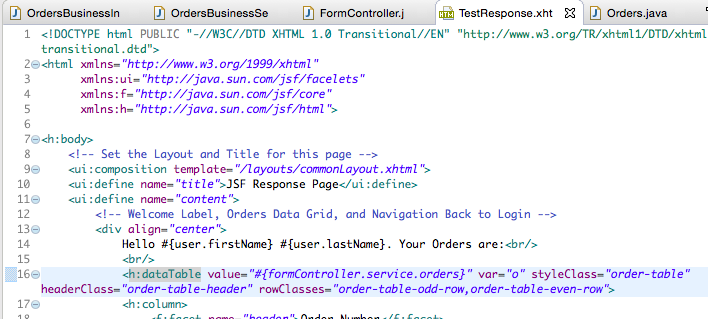
1. EJB Business Services:
   * Add the following public methods to the OrdersBusinessInterface:
     + public List<Order> getOrders();
     + public void setOrders(List<Orders> orders);
   * Declare a class scoped variable *orders* of type List<Order> in the OrdersBusinessService implementation class and AnotherOrdersBusinessService implementation class.

This is the first of two implementations.

* + Implement the getOrders() method and setOrders() method in the OrdersBusinessService implementation class and AnotherOrdersBusinessService implementation class that gets or sets the value of the class scoped orders variable.
  + In the default constructor of the OrdersBusinessService implementation class and AnotherOrdersBusinessService implementation class initialize the orders list with “dummy” test data. Use your imagination. In a real app, this java class accesses the database, does complex computations and finally returns a result.

Invent different data for the other business service class AnotherOrderBusinessService.

* + Initialize the values of the orders list differently between the two service classes.

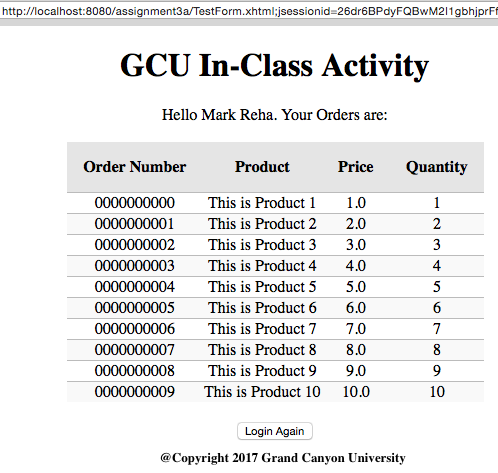
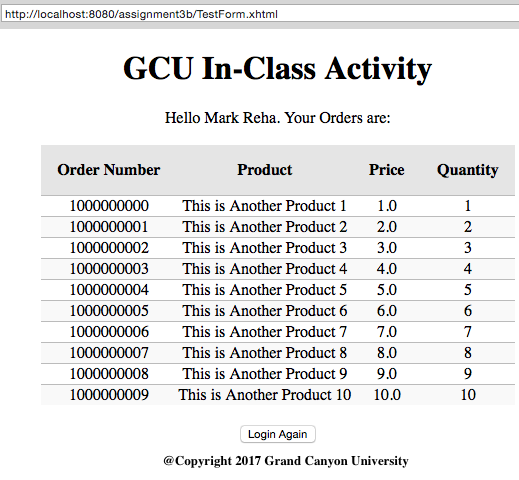
1. JSF Login Controller:
   * Add a public method getService() that returns the service class scoped variable in the FormController.java file
2. JSF Results Page:
   * In the Data Table change the Managed Bean that returns the Orders List from #{orders.orders} to #{formController.service.orders} see line 16.

Deploy the code and execute the application specifying the TestForm.jsf using the internal IDE browser. Submit the form with valid data. Take a screenshot. The Orders List from the AnotherOrdersBusinessService should be displayed.

Change the Business Service implementation by only changing the EJB configuration:

1. EJB Configuration:
   * Open the beans.xml file and select the Alternatives bean option. Edit the existing entry and change the alternative EJB to the business.OrdersBusinessService class.

Deploy the code and execute the application specifying the TestForm.jsf using the internal IDE browser. Submit the form with valid data. Take a screenshot. The Orders List from the OrdersBusinessService should be displayed. Two different sets of data are shown here.

# Short Assignment 3c: Design and implement Enterprise Timer Beans

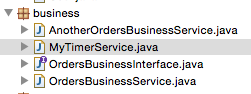
Objective:

1. Create scheduled business service using Enterprise Java Beans that implements business logic.

Activity Directions:

Study the Timer EJB and CDI example code from the reading for this Topic. This activity will add code to the previous activity.

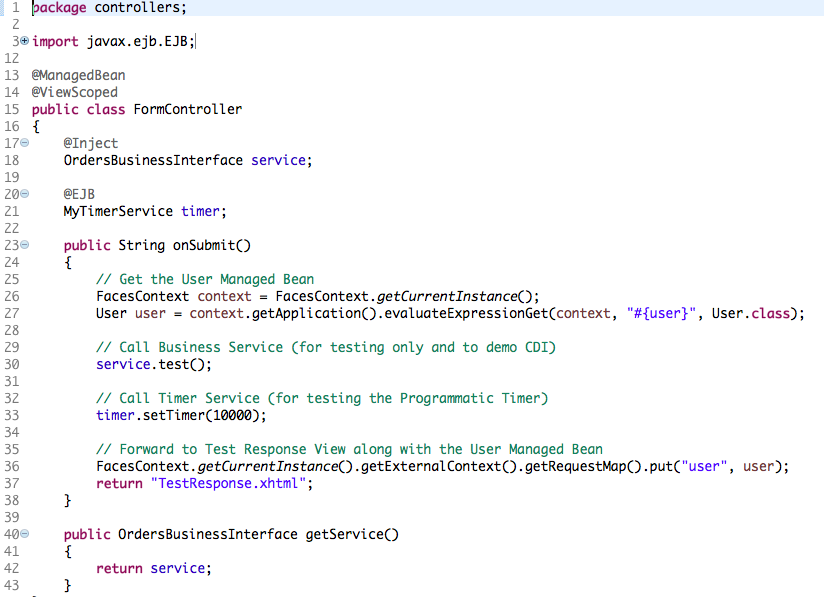
Add and update the following components in the project:

1. EJB Timer Service:
   * Create a Timer EJB by right clicking on the project and selecting the New->Other...->EJB->EJB Timer menu options. Create the EJB in the business package with a class name of MyTimerService.



* + Create a class scoped variable named *logger* of type private static final Logger and initialize the logger variable using the Logger.getLogger(“business.MyTimerService”) method call (make sure to import the java.util.logging.Logger package).
  + In the code generated for the scheduledTimer() method change the schedule of the hour to 0-23 and use the logger.info() method instead of the System.out.println() call.
  + Create a class scoped variable named *timerService* of type TimerService (make sure to import the javax.ejb.TimerService package). Add the @Resource annotation above the timerService variable declaration (make sure to import the javax.annotation.Resource package).
  + Add a public method to the EJB setTimer(long interval) that returns void. In the method implementation call the timerService.createTimer(interval, “My Timer”).
  + Add a public method to the EJB programmicTimer(Timer timer) that returns void and using the logger.info() method to print a message for this method. Add the @Timeout annotation above the method declaration (make sure to import the javax.ejb.Timeout package).

1. JSF Login Controller:
   * Add a class scoped variable named *timer* that is of type MyTimerService. Add the @EJB annotation above the variable declaration (make sure to import the javax.ejb.EJB package).
   * In the onSubmit() method call the setTimer(10000) method on the *timer* variable.



Deploy the code and execute the application specifying the TestForm.jsf using the internal IDE browser. Submit the form with valid data. Take a screenshot of the console output. The scheduled timeout logging statement should be printed every 10 seconds and the programmatic timeout logging statement should be printed 10 seconds after the form was submitted.

Explore different scheduled timer and programmatic timer settings. In your team project what business logic can you think of that might be applicable to be implemented as part of a scheduled service?

Developer Notes & Best Practices:

* You can use either the @EJB or @Inject annotation to inject implementations into your class using CDI. A flexible approach is to simply always use the @Inject annotation which allows you to inject an EJB or any POJO managed bean (using an Interface class declaration).
* Unless warranted you should always specify a local EJB to avoid the network overhead when you simply need to make a local reference.
* Get in the habit of following good package naming and class naming conventions. Put all your Business Service EJB classes in a packaged called *business*. All business services should be designed using an Interface class which then gives you the ability to use CDI dependency injection at runtime along with configuration to specify the concrete implementation class for your business service. You should also consider using Postfix names for your Business Service classes. End your Business Service class names with *BusinessService* and any Interfaces class names with an *Interface*.
* Inspect the packages you import into any Business Services class to ensure that you are not “injecting” presentation layer technologies into your business layer.

Deliverables and Submission:

1. A Project report containing the following:
   * Cover sheet with name of class, assignment, date, and your name.
   * BitBucket URL.
   * Brief theory of operation explaining the approach to design and implementation, including a detailed list of all classes, methods, variables, and pages.
   * Project report with all screenshots showing execution of applications.
2. Upload code to BitBucket
3. Upload code as a zip file to LoudCloud
4. Upload Project report to LoudCloud.

This assignment utilizes a rubric. Please review the rubric prior to beginning the assignment to become familiar with the expectations for successful completion.