**An Online Payment Service**

1. Introduction

This assignment is about the design and implementation of a web-based, multi-user payment service using Java Enterprise Edition (Java EE) technologies. The system is a much simplified version of PayPal. Through a JSF-based web interface, users should be able to send money to other registered users (e.g. using their registered email address as their unique identifier), request money from other registered users and manage their own account (e.g. look at their recent transactions). Super-users (i.e. admins) should be able to access all user accounts and transactions. Optionally, you will deploy your application on the cloud (e.g. on Amazon AWS, Microsoft Azure, or any similar infrastructure).

After successfully completing the assignment, you will have demonstrated that you can:

* design and implement user interfaces using Java Server Faces
* design and implement business logic using enterprise Java beans (EJBs)
* design and implement a secure multi-user system

2. Project Description

Online payment services, such as PayPal, allow users to connect their online accounts to their bank accounts, debit and credit cards. In such systems, users are usually able to transfer money from their bank accounts to the online account, receive payments to this account from other users, push money from the online account to their bank accounts etc.

For simplicity, we will assume that, for this project, all registered users start with a specific amount of money (e.g. £1000 pounds) and no connections to bank accounts exist.

***Note: this is pretended money and no connection to real sources of money should exist.***

Each user has a single online account whose currency is selected upon registration. A user can select to have their account in GB Pounds, US dollars or Euros. In that case, the system should make the appropriate conversion to assign the right initial amount of money (e.g. if the baseline is £1000, then the initial amount should be 1000 \* GBP\_to\_USD\_rate US dollars).

A user can instruct the system to make a direct payment to another user. If this request is accepted (i.e. the recipient of the payment exists and there are enough funds), money is transferred (within a single Java EE transaction) to the recipient immediately. A user should be able to check for notifications regarding payments in their account.

A user can instruct the system to request payment from some other user. A user should be able to check about such notifications for requests for payment. They can reject the request, or, in response to it, make a payment to the requesting user.

Users can access all their transactions, that is, sent and received payments and requests for payments as well as their current account balance.

An administrator can see all user accounts and all transactions.

Currency conversion must be implemented by a separate RESTful web service (see Section 3.3). The actual exchange rates will be statically assigned (hard-coded) in the RESTful service source code.

3. System Architecture

3.1. Web Layer

The web layer consists of a set .xhtml (facelets) pages through which users and administrators interact with the web application.

Users should be able to:

* View all their transactions
* Make direct payments to other registered users
* Request payments from registered users

Administrators should be able to see:

* user accounts
* all payment transactions

and register new administrators

CDI Beans must not access any persistent data from the database. They should delegate all business logic to the service layer.

3.2. Service Layer

The service layer consists of a set of Enterprise Java Beans (EJBs) that implement the business logic for the system. EJBs should support Java EE transactions so that data integrity is preserved. You should utilise container-managed transactions. That is, your code doesn't need to cope with opening, committing or roll-backing transactions. You will only need to annotate your EJBs with the appropriate transaction attributes (or leave the default behaviour, if appropriate).

The service layer is responsible for accessing the data (persistence) layer. Persistence (JPA) entity managers must be injected into your EJBs. Access to persistent data must only take place through these entity managers.

3.3. Data Layer

The data layer consists of a relational database and JPA entities. To simplify deployment and configuration you must use JavaDB as your Relational DataBase Management System (RDBMS). JavaDB is an RDBMS that is installed with GlassFish.

Your data model should be written as a set of persistence entities (JPA). Upon deployment, JPA will create the actual relational database tables for you. Access to the database must always take place through manipulating JPA entities. Do not access the database directly using JDBC.

3.3. Web Services

You must implement a REST Service that is accessed by the service layer. The service will be deployed on the same server but accessed from the service layer in the standard way (i.e. through HTTP).

A currency conversion RESTful web service that responds only to GET requests. The exported resource should be named *conversion*, in a path like the following:

*baseURL/conversion/{currency1}/{currency2}/{amount\_of\_currency1}*

The RESTful web service should return an HTTP response with the conversion rate (currency1 to currency2) or the appropriate HTTP status code if one or both of the provided currencies are not supported.

e.g. GET *baseURL/conversion/{currency1}/{currency2}* HTTP/1.1 should return a status ok response with a very simple response (e.g. in JSON that says that 1.00 GBP = 1.21217 EUR). Writing a REST client should be straightforward. Check the following links for more information:

[https://eclipse-ee4j.github.io/jakartaee-tutorial/jaxrs-client.html (Links to an external site.)  
 (Links to an external site.)](https://eclipse-ee4j.github.io/jakartaee-tutorial/jaxrs-client.html)[http://stackoverflow.com/questions/2793150/using-java-net-urlconnection-to-fire-and-handle-http-requests (Links to an external site.)](http://stackoverflow.com/questions/2793150/using-java-net-urlconnection-to-fire-and-handle-http-requests)

3.4. Security

The online payment service is a multi-user web application. A user must be logged-in in order to interact with the system. Users should not be able to see other users' information nor access pages and functionality for administrators. Administrators access their own set of pages, through which can have access to all users information. Users and administrators should be able to logout from the web application.

You will need to implement and support:

* Communication on top of HTTPS for every interaction with users and admins
* Form-based authentication (simple using the file realm, in a jdbcRealm where users can subscribe - full marks will be given to the second approach (see Section 4.))
* Logout functionality
* Declarative security to restrict access to web pages to non-authorised users
* Declarative security to restrict access to EJB methods

4. Mark Allocation

4.1. Web Layer (20%)

15% - Full marks will be given if all required .xhtml are written and correctly connected with CDI backing beans in a way that makes sense even if no other functionality is implemented at the service and data layer. The set of correctly implemented JSF pages includes .xhtml pages required to perform security-related actions.

5% - Full marks will be given if all required conversions and validations are done. This highly depends on the way you design your pages. In most cases, standard validations and conversions should be enough. Full marks will be given to assignments which support full and correct page navigation by explicitly specifying navigation rules in a faces-config.xml file.

**Important Note**: The appearance of web pages will not be marked. If you want, you can use frameworks like PrimeFaces or RichFaces  that build on JSF, or custom .css files, but this is not part of this assignment.

4.2. Service Layer (20%)

Full marks will be given if all required business logic is implemented in a set of Enterprise Java Beans, which must include appropriate annotations for supporting JTA transactions, if and when required.

Users should be able to (15%):

* View all their transactions
* Make direct payments to other registered users
* Request payments from registered users

Administrators should be able to (5%):

* view all user accounts and balances
* view all payment transactions
* register more administrators

4.3. Persistence Layer (10%)

Full marks will be given if all access to application data is handled through JPA Entities. A correctly configured persistence.xml file is required along with annotations for defining JPA entities. Annotations are required to define associations among different entities (e.g. one-to-many, many-to-many) wherever this is required.

4.4. Security (20%)

10% - Form-based authentication

* Full marks will be given if users can register, login and logout. This can happen using a jdbcRealm which is linked to JavaDB in order to register and authenticate users. An admin must be registered in the system when deploying (and, therefore, creating the DB tables)

4% - Declarative security for access control when navigating through .xhtml pages

* Access to .xhtml pages must be restricted to authorised actors. You need to add security constraints in the deployment descriptor.

4% - Declarative security for accessing EJB functionality

* EJBs must be annotated appropriately (along with annotation-based role declarations) so that EJB functionality can be accessed my authorised actors (users and admin)

2% - Initial administration registration

* Upon deployment, a single administrator account (username: admin1, password:admin1) must be present. You can implement that through a singleton EJB that is instantiated upon deployment or by using a simple SQL script when the persistence unit is deployed. Only an administrator can register more administrators through the restricted admin pages.

4.5. Web Services (10%)

Full marks will be given if the REST web service is correctly implemented (using the correct URI structure presented above).

4.7. One out of two options (20%)

For the last 20% of the marks you can select between 2 options.

4.7.1 Report

Write a report (up to 1500 words), critically assessing the strengths and limitations of your implementation utilising your understanding of the underlying technologies. The report must consider the following points:

* (5%) How your design fits with the 3-tier architectural model and the model-view-controller software pattern
* (5%) The strengths and weaknesses of your chosen methods for securing the application, comparing your approach to other options
* (5%) How your design could be extended so that your server is not a single point of failure
* (5%) How your system would deal with concurrent users accessing functionality and data

4.7.2 More Programming

* + 10% - Implement the DAO and DTO access patterns. In real world applications, the service layer never accesses the persistence layer directly. Back-end storage resources may change (e.g. new relational or non-relational databases may be added) over time and the service layer code must be independent of such changes. The most common software pattern for implementing such independence is the DAO pattern. DTO objects are also used instead of moving entity objects across layers. Check below for more information and extra resources:
    - * http://www.oracle.com/technetwork/java/dataaccessobject-138824.html
      * http://www.oracle.com/technetwork/java/transferobject-139757.html
  + 10% - All transactions must be timestamped by accessing a 'remote' Thrift timestamp service (which is deployed on the same server as your system). The service should return the current date and time to your system when requested by the Enterprise Java Bean. The Thrift server can be implemented as a deployable EJB which uses a separate thread to accept time-stamping requests at port 10000.

4.8. (Optional) Deployment on the Cloud (5% bonus)

You can optionally deploy your application on the Cloud and get an extra 5%. The maximum mark you can get for this assignment is 100% (i.e. you will still get 100% (and not 105%) even if your submission is perfect and you deployed your application on the Cloud). To do so, you must successfully deploy and run the application on e.g. Amazon EC2 virtual machine (any other framework would work fine, too). It is up to you to get access to the Cloud infrastructure (e.g. through awseducate) and configure the virtual machine with all required software. In order to get full marks, you must submit screenshots of the commands you issued on the console to run Glassfish and JavaDB, the security configuration, JDBC pool and data source configuration, and screenshots of the application running on the cloud where the URI of the application is shown. In order to verify that you have indeed deployed the application, I may ask you, during the marking period, to run the server and deploy your application for me to test it.

If you want to use Amazon AWS, you can use AWS Educate. In order to get access to Amazon AWS, you must first register as a Sussex student here (no credit card is required if you select the AWS educate starter account):