Web application report

# Introduction

This report covers the implementation of a Java EE payment transaction application. Because this is a multi-user application, it is imperative that it is designed to be as secure and scalable as possible.

# Architectural model

The application can be designed using the model view controller (MVC) design pattern:

1. Data Layer
   * This layer represents the model and is responsible for taking the data created by a user interacting with the system and storing it in the database.
2. Web Layer
   * This layer represents the view and is responsible for generating the HTML code that is read by the browser
   * This tier also handles form inputs and validation, ensuring the correct data types are inputted
   * Finally, this tier contains calls to the controller tier, returning data that can be updated in the view or stored in the database
3. Service Layer
   * This layer represents the controller and holds all of the business logic for the application.
   * The business logic is implemented in a set of Enterprise Java Beans (EJB’s)
   * The EJB’s are responsible for taken data from the web layer (e.g. a username entered by the user) and from the database (e.g. a user with a matching username) and performing some action (such as the returning the users balance)

In accordance with the 3-tier architechtual model, these layers can be written such that each layer is independent and does not expose dependancies to the application. Furthermore, each tier could potentially run on a completely different machine and network and the application would function the same. Aditionally each tier can be changed with minimal disruption to the other tiers, for example, the data layer could be edited to use a MySql database instead of JavaDB and the other layers would not need any editing.

As this application involves transferring (virtual) currency, personal user data and storing of passwords, it is imperative that malicious users can not get access to data that they are not entitled to. Therefore, this application uses several methods provided by glassfish and Java EE to ensure the security of the system.

# User authentication

When a user creates an account, they input a username, password, full name and a currency. The password field uses the *<h:inputSecret>* tag which hides the characters inputted from the screen. However, this only prevents an attacker looking over the users shoulder, the data is still passed to the JSF backing bean in plain text. If this were stored in the database, it would still be a huge security risk. Therefore, the *registerUser* function hashes the password using the SHA-256 algorithm before storing it in the database.

To handle the authentication of a user when logging in, form based authentication is used. A jdbcRealm linked to the database allows glassfish to securely check (using TLS) the hashed password value matches the one stored in the database. The browser then caches an authentication cookie, creating a session in which the user can freely access protected pages.

By using the *Confidential* transport guarantee mode, the application is forced to use the secure HTTPS protocol for all pages of the site. This reduces the effectiveness of man in the middle attacks as the data would need to be decrypted to be understood. However, this application uses a self signed certificate which will cause most browsers to view it as insecure. This could be overcome by using a trusted certificate, although this is usually not free.

When the application is first started, a admin user is created by a singleton EJB called ‘InitAdmin’. As this file contains the username and password of the admin in plain text, it could be a potential security risk should an attacker get hold of the source code. However, as the only code viewable by a regular user is the generated HTML, this is not a big risk. To overcome it, the value “admin1” could be hashed and directly stored by an SQL script instead of hashing each time.

Another approach that could be used is the file realm, manually inputting user details into the glassfish console. This would allow an administrator to control access of the system without any knowledge of the source code or java. However, it would make it difficult for users to register as they would have to have access and knowledge of the glassfish console.

To ensure the .xhtml pages intended for administrators are only accessed by admins, the application makes use of the *SecurityRoles* feature of Java. When a user is registered in the database, they are assigned to a group (admins or users). Admins and users are both permitted to access user pages, but only admins can access admin pages. Furthermore, certain methods in the JSF backing beans are restricted to admins as they return information regarding multiple users.

# Scaling the application

As the application may need to handle many users at once, certain provisions have to be made to ensure that the system can handle the load.

## JTA Transactions

The ‘TransactionService’ EJB is responsible for making transactions and transaction requests. As these operations involve multiple steps, it is vital that the transaction does not stop halfway through without rolling back to the state before it started. For example, if a user transfers £10 to another user and the transaction is interrupted after £10 has been deducted from the senders account but before the £10 has been added to the recipients account, the transaction should restore the £10 to the sender before issuing a failure message. To achieve this, the ‘REQUIRES\_NEW’ transaction attribute annotation is applied to the transaction service. This makes every call to a transaction service method run in a separate transaction, ensuring that no matter how many users are using the system, if a transaction fails it can be rolled back safely. As the application uses container managed transactions, the system will automatically roll back transactions, so manual implementation of it is not required.

The alternative to container managed transactions is bean managed transactions, in which manual implementation of opening, committing and roll backing transactions is necessary. This gives more flexibility to the programmer but for this application it is unneeded.

## Handling server failure

Currently, although the Application can preserve data integrity during transactions, if too many people were to access it and the server failed, the application would not be accessible by any user. To overcome this problem, glassfishes clustering abilities could be utilised. This would allow the application to be run on multiple servers, with a load balancing tier managing the traffic between each one. If one server were to fail, the transactions would be rolled back and the users would be directed to one of the other available servers dramatically improving the availability and scalability of the application.

Furthermore, the application could take advantage of the geographic locations of these servers by using a content delivery network (CDN) service. This would result in a user request being directed to the edge server nearest to them in the DNS lookup phase. As data takes a long time to travel over long distances, this would improve the speed of transactions and reduce waiting times for all users.