ONLINE PAYMENT SERVICE REPORT
Web Applications and Services

3 TIER ARCHITECTURE MODEL

The design of this application consists of three independent tiers which do not expose dependencies related to their implementation.

The presentation tier contains XHTML Facelets that only interface with the business tier using the expression language to either pass client data to the business tier or call a business tier method where the presentation tier then display's the data returned from that method. The Facelets interface with the business tier without being concerned how the business logic is implemented, i.e. how the method it is calling is implemented. The Facelets are also unaware of the type of database being used in the data tier i.e. where the data is coming from and how it is being retrieved.

The business tier connects the presentation tier to the data tier and contains JSF Beans and Enterprise Java Beans. JSF Beans are used to receive and transform data from the presentation tier into respective Java objects which can be manipulated programatically. JSF Beans can then either pass the presentation tier data to Enterprise Java Beans by calling and passing data to one of the Enterprise Java Beans' respective methods, here the Enterprise Java Beans then interface with the data tier using the Java Persistence API and JPQL to manipulate the database with the presentation tier data.

Alternatively if the presentation tier wants to retrieve data from the data tier, the business tier passes this data back to the JSF bean that called the EJB method, who then passes the data back to the presentation tier.

The business tier is unaware of the user agent of the client in the presentation tier that made the request to it and is unaware of the type of database used in the data tier, it is only responsible for passing presentation tier data to the data tier and retrieving data from the data tier in order to pass it back to the presentation tier.

In order to design for change, the business tier contains two local interfaces which contain all the method signatures required by an administrator and a payment service user and provides two implementations of those interfaces in the form of Enterprise Java Beans. This provides loose coupling between different functionalities.

The data tier contains a relational database consisting of several Entities, represented as Java objects. The Enterprise Java Beans in the business tier communicate with these entities in the database using the Java Persistence API and JPQL in order to update their data or retrieve their data, however data tier specifics such as the relationships between tables are hidden from the business tier and subsequently the presentation tier as well. This design ensures that If the data tier were to be migrated to a NoSQL database later on, the Facelets in the presentation tier and the business logic would not have to be changed.

MODEL VIEW CONTROLLER DESIGN PATTERN

The Controller is the FacesServlet which is hidden and guides where HTTP requests and responses go at a lower level, along with the JSF Beans. The JSF Beans receive requests from the user via XHTML Facelets who then communicate with Enterprise Java Beans (which are part of the model) to either instruct the model to change or it may instruct the Enterprise Java Beans to invoke functionality of the model such as retrieving data.

The View of this application consists of XHTML Facelets and are connected to the model through JSF Beans in order to render model data to the user. The views are connected to the JSF Beans via the Expression Language.

The Model consists of Enterprise Java Beans, the Java Persistence API and the database and is responsible for managing data and responding to requests about it's state from the JSF Beans and therefore the view. The Model also follows instructions from the JSF Beans to manipulate the database i.e. to insert a new entry into the database.

SECURING THE APPLICATION

Declarative Security was implemented to provide authorisation by adding security constraints in a deployment descriptor so that web pages are restricted to authorised principles/groups. Form based authentication using the JDBC realm was used in order store credentials in a database rather than the file realm which stores credentials in a text file, the certificate realm was not used as client certificates are not being added because authentication takes place using a username and password, rather than certificates.

EJB methods are annotated with the appropriate role based declarations so that authorised principles/groups could only use business methods they are authorised for. In this application role based annotations are not frequently used because when users are authenticated, they only have access to authorised functionality, this is achieved by implementing Programmatic Security to check whether the authenticated user is an admin or a normal user and then redirect them to their appropriate home pages where they can then only perform their authorised actions as these are only visible.

Alternatively the design could be changed so that all functionality could have been seen by all authenticated users, then if a user was not authorised to perform administrative functionality, access would be denied, this would still ensure only authorised users would have access to only what they authorised to do. The benefits of providing role based declarations in EJB methods mean there is flexibility to adapt to this approach if needed.

As a result of Authorisation, this implies a need for Authentication because there needs to be a way to authenticate principles to allow them to have access to specific resources. HTTPS is enabled in order to provide data secrecy, message integrity and authentication. When authentication occurs, usernames and passwords are not sent using plaintext, so malicious users can't understand username and passwords. When users register, passwords are passed to a secure hash function (SHA1) in order to generate a hashed value of the password, this is then stored in the database rather than the plaintext password, so that if malicious users were to gain access to the database, then the actual passwords cannot be known.

GlassFish adds a self signed certificate upon deployment, however because the developer is not trusted, the web browser displays a warning indicating the web page may be malicious. In order to mitigate this warning a certificate could be signed by a trusted third party.

If Basic HTTP authentication was only used for enforcing access control, then username and password pairs would only be encoded in the authorisation field of the HTTP GET request, these would not be hashed or encrypted and would therefore not provide data secrecy unlike HTTPS.

ENSURING THE SERVER IS NOT THE SINGLE POINT OF FAILURE

The application would need to be scaled up from a single server to a high availability server cluster consisting of multiple servers that replicate the application across multiple servers rather than a single server. A load balancer is need, which efficiently distributes HTTP requests from the client to the the server cluster in order prevent overloading a single server within the cluster with HTTP requests, this will minimise the possibility of a server failing. The load balancer therefore ensures high availability for the application by only distributing HTTP requests to those servers in the server cluster that are able to send a HTTP response quickly.

This approach ensures that if a single server were to fail due to a hardware or software failure then another server would take control of the application, therefore the application will still be available for users to access as it has been replicated on another server. If the single server fails, it not only prevents users to access the application, but may cause data loss as well and data is the most important asset of any web application.

MANAGING CONCURRENT ACCESS OF FUNCTIONALITY AND DATA

Optimistic Concurrency Control where transactions get access to tentative versions of objects has been implemented in order to ensure transactions perform as a single, indivisible unit whilst maintaining Atomicity, Consistency Isolation and Durability, ensuring transactions are serially equivalent in order to prevent the lost update and inconsistent retrievals problem and preventing the dirty reads and premature writes problems, because committed data is always accessed.

Optimistic Concurrency Control was implemented by adding a version annotation to a field in the AppUser entity containing the balances of accounts (as this may be accessed concurrently by multiple transactions). If two transactions are modifying the balance of a user concurrently, whenever the balance is modified by one transaction, the version number of the AppUser entity increases, then another transaction checks if this version number is the same as it was when the transaction was first initiated, if it is the same then the transactions commit successfully, otherwise if they are not the same, the application will abort and rollback the transactions to their previous state due to a conflict, this check takes place in methods that may be concurrently accessed in the UserServiceBean EJB, who catches an EJBTransactionRolledBackException and notifies the calling JSF bean to display an error message to the client notifying them if a conflict occurred and instructs users to retry their transaction when a conflict occurs.

An issue with Optimistic Concurrency Control is that there may be a chance of starvation, whereby multiple transactions repeatedly access the same data and perform an operation, then when committing changes, they repeatedly fail due to a conflict, resulting in transactions being repeatedly aborted, however this is rare for this particular application, for this reason Optimistic Concurrency Control was preferred over Pessimistic Concurrency Control which places a long term lock on transactions until they are complete, preventing other transactions from manipulating the data until the lock has finished, however in this application the likelihood of two transactions accessing the same object at the same time is low, therefore this approach would end up decreasing the performance of the application, which is why Optimistic Concurrency Control is implemented.

REFERENCES

1. Wikipedia (2017) ‘Single point of failure’ [Online] Available at: https://en.wikipedia.org/wiki/Single_point_of_failure#cite_note-1 [Accessed 27th April 2017]

2. F5 Networks Inc (2017) ‘Glossary and Terms: Load Balancer’ [Online] Available at: https://f5.com/glossary/load-balancer [Accessed 27th April 2017]

3. Wikipedia (2017) ‘Basic access authentication’ [Online] Available at: https://en.wikipedia.org/wiki/Basic_access_authentication [Accessed 1st May 2017]

4. Oracle (2017) ‘Overview of Entity Locking and Concurrency’ [Online] Available at: http://docs.oracle.com/javaee/7/tutorial/persistence-locking001.htm [Accessed 2nd May 2017]