EXPRESSQ TECHNICAL REPORT

Grant Christie, Lauren Bateson, Rorie White, Matthew Brighty, Bartosz Modelski

Use Case Models

Use Case: Customer makes an order

Actors: Customer.

Purpose: Make a lunch order for the business to process.

Overview: The customer will load the application and login with their credentials. The customer will then select which store they wish to order from and be presented with that store's menu. They will then choose which items they wish to purchase from that menu and submit their order.

Type: Essential

Preconditions: Customer has registered account.

Postconditions: Order is added to the database and the business receives the order details.

Flow of Events

Actor Action	**System Action**
1. Customer loads the application.	
	2. System asks customer for login credentials.
3. Customer fills in required login credentials: Username and Password.	
	4. System authenticates credentials.
	5. System provides list of stores.
6. Customer selects store.	
	7. System provides selected store's menu.
8. Customer selects items they wish to order from the menu.	
	9. System adds items to prospective order.
10. Customer submits their order.	
	11. System asks customer for payment.
12. Customer inputs payment information.	
	13. System adds user order to database.
	14. System sends confirmation email.

Alternate Flow of Events

Step 4: If the login credentials the customer provides are incorrect the system will return the customer to step 3 where it will ask them to provide login credentials again.

Step 10: If the customer tries to submit their order having selected no items the system will instead respond by displaying a message saying that at least one item must be selected and the customer must repeat step 8.

Step 12: If the payment information the customer inputs is invalid the system will be ask the customer for the payment information again.

Use Case: Customer Registers Account

Actors: Customer

Purpose: Register an account to make use of the application

Overview: The system will ask the customer to login or register an account in order to proceed. The user will fill out the required details and the account will be created.

Type: Essential

Preconditions: None

Postconditions: Customer account will be added to the database

Flow of Events

Actor Action	**System Action**
1. Customer loads the application.	
	2. System asks customer to fill in login details or register an account.
3. Customer fills in required details: Username, Name, Surname, Email Address, Password, Password Confirmation.	
	4. System checks email is unique and in a valid format and that the passwords match.
	5. System sends confirmation email to customer.

Alternate Flow of Events

Step 4: Error in filling out the required fields: Duplicate email, duplicate username, invalid email format or different passwords. Customer is returned to step 3.

Candidate System Architecture

The ExpressQ system will be implemented in Java. Java was chosen as the implementation language for multiple reasons:

- Java is a compiled language; compiled languages are faster and require less overhead when compared to interpreted languages.
- Java is platform independent, the system can be run on any operating system supporting the JVM without the need to port it.
- Java is the most popular programming language.[1] A popular language has a greater number of 3rd party libraries available for use and more documentation.

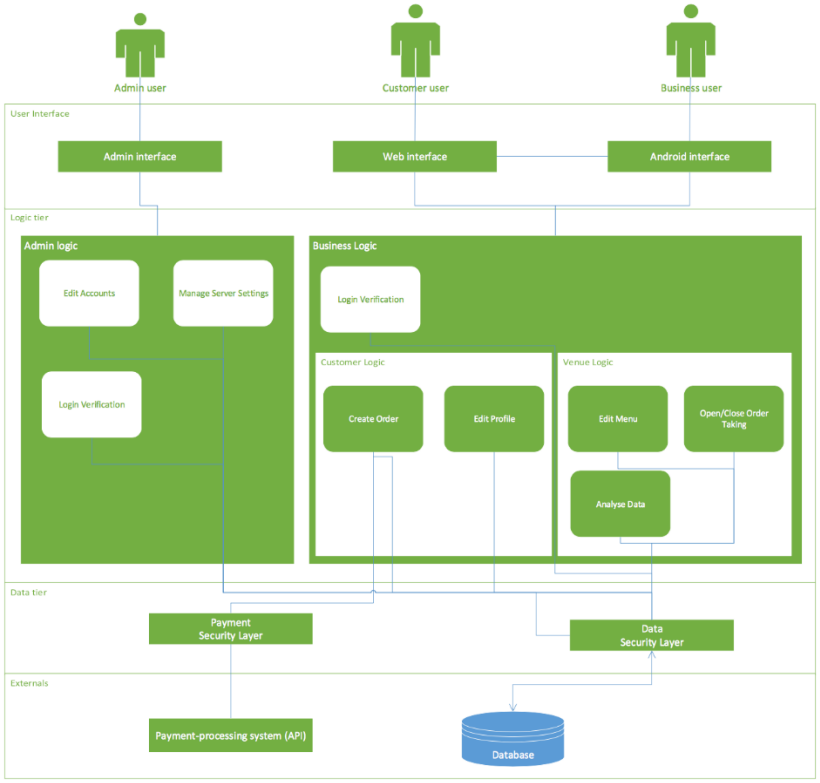

Figure 1 The System Architecture

Layered Architecture

The above image shows the architecture of our system. Our system is a four-tier layered architecture consisting of: The User Interface, the Logic Tier, the Data Tier, and the Externals.

The User Interface is our presentation layer providing both the desktop web application and the Android mobile app user interfaces. Whether the regular Web Interface or the Admin Interface is shown is dependent on the type of account logged into the web application.

[1] http://www.tiobe.com/tiobe-index/

The Logic Tier is the application layer in which the application logic is contained. This layer contains the algorithms providing the main functionality of the system, such as the ability for customers to place orders and allowing the businesses to analyse data on their customer base.

The Data Tier contains the interfaces between the main Java application in the Logic Tier and the modules external to the main application. The Payment Security Layer is an interface between the Business Logic and our chosen payment processor. The Data Security Layer is a Java Database Connectivity API which will form SQL queries needed by the business logic and will return the results of these queries as Java objects.

The external layer contains our database and our payment processor. MySQL was chosen as a database management system as our team had previous experience with it. Our payment processing will be handled by an external service, Stripe.[2]

Partitioning

Due to the complexity of the Logic Tier this layer had to be decomposed further into the Admin Logic and the Business Logic. The Admin Logic contains modules for use by the ExpressQ system administrators, allowing the management of server settings and the editing of accounts. The Business Logic is further partitioned into Customer Logic and Venue Logic sub-systems. The Business Logic contains one module for login verification which can be used by both of its sub-systems.

The Customer Logic sub-system handles all services relating to actions which the customers on the system would need to carry out, represented by the Create Order and Edit Profile modules. The Business Logic sub-system contains the modules allowing businesses to edit their menu, control whether they are currently open to new orders, and carry out data analytics on their customer base.

Partitioning this layer in such a way improves the modularity of the system. The modules within each sub-system are directly related to the purpose of the sub-system containing them, maximising system cohesion. Each partition has a clearly defined boundary based on the services it provides and do not communicate directly with each other, promoting information hiding.

Shared-Data Architecture

The ExpressQ system also makes use of a shared-data architecture. Specifically, it is a repository architecture as the shared-data store is a passive MySQL database which is queried by the accessors (the Data Security Layer). As the accessor modules can be changed independently the changeability and the maintainability of the system is increased.

[2] https://stripe.com/gb

Initial Risk Assessment

The following section details what risks may be encountered during the development of the project:

Potential Risks

Low or Medium Risk	High Risk
Large number of orders/customers	Data corruption/loss
Database limitations	Privacy/security
	Connection to database

Large Number of Orders/Customers

It is fair to say that in the early days of ExpressQ there will not be many customers or businesses using it. It will not be until the system is adequately advertised that the number of users will grow.

To counter any issues here it is possible that some research could be done on the number of customers that might be expected during peak times. With these findings, a load balancer or database cluster could be implemented to share out and balance the queries being sent to our database. This would then keep the database from being overloaded due to too many queries.

Database Limitations

Much like what was mentioned previously, carrying out research to find out the number of users that could be using the system at once can help here. Research could give us an idea of what we would need in terms of storage space.

Downtime is also another factor to take into consideration. Unfortunately, it is something that is not unavoidable. If any maintenance needs to be carried out, then the database would need to be taken temporarily down. One way to avert this is to have a second database or server for situations like this, take the main one down and temporarily replace it with the secondary while maintenance is carried out. Another solution is to monitor for when the most ideal time to take the database down is. Due to the system incorporating big data analysis we would see when business was getting the least customers, and so would know when the best time to take the database offline is to ensure that the business users do not lose out on too much income.

Data Corruption/Loss

Data corruption or loss has a relatively high risk despite what one might think. If a user's account was lost then it is only a small inconvenience, it would be simple enough to create a new account. On the other hand, if a store's menu is somehow lost then that store may lose faith in us as a company. They would be missing out on a profit from potential customers if their menu has nothing to select, and so nothing for the customer to purchase.

Ideally, we hope a situation like this would never occur but precautions must be taken for the worst-case scenario. To do this we will backup data stored on the primary database which will be updated daily to ensure the least amount of data is lost if such an event was to occur.

Privacy/Security

Although any customer's credit card details will be handled by an external system, their username and passwords will still be stored in our databases. Encrypting this data correctly and keeping it secure is of high priority.

For example, in the case of Stripe, the user can choose to save their card details for use again on a later date. This is done by trading over the card details for a token in return which takes form as an ID for the customer and is then associated with their account. Now, in theory, if someone else were to gain access to a customer's details then they could potentially use their credit card fraudulently.

We believe working to prevent any leaks will assist on building up trust between both businesses and customers. If either discovers that the system is insecure then they would be very reluctant to purchase it. Overall, this would be a great loss for us as the software developers.

To prevent this risk any sensitive data will encrypted and set up in such a way that only those who are authorised to do so can view the data.

Connection to Database

Since the system will be designed to be used both on desktop and mobile platforms, we cannot guarantee that customers will always have an adequate connection to the internet. After all, the nature of a mobile phone is to be mobile and constantly on the move. To try and cut down on the number of timeouts that customers may encounter while using the application a higher connection timeout value could be used. This would allow the application to wait a longer time before timing out, giving it more of a chance to connect.

Initial Project Plan

Schedule

Week 1 - 17th October

Our team discussed ideas for the system and performed a feasibility analysis to ensure that our preferred idea would be possible. After this we put together a first draft of functional and nonfunctional requirements. All functional requirements were ranked in order of importance to the system. We also decided on using Java for the system and making use of the Spark web application framework for the prototype.

Week 2 - 24th October

Designed our database's class diagram and created a first draft of our system's architectural structure. We went back to our initial functional and nonfunctional requirements at this stage to ensure they would all be satisfied by our proposed architecture and to make them more clear and concise.

Week 3 - 31st October

At this stage, we took our first architectural design and refined it into a multitier architecture which had four layers: Presentation tier, Logic tier, Data tier and the Externals. All the modules within the architecture at this point were still very abstract and needed to be partitioned however we still assessed the coupling and cohesion for this architectural model.

We also looked at our more critical requirements in greater depth. Creating use case models for each along with a flow of events charts to illustrate how the user and system would interact with one another during use.

Week 4 - 7th November

We iterated back over our multitier architectural design and partitioned the complex logic modules into various submodules so we could better assess coupling and cohesion between each submodule and with the other layers of the application.

We also planned out how to approach the implementation of our prototype. We split the prototype into 5 main tasks:

1. Create the database and SQL Queries
2. Create the Java methods that interact with with database to allow our application and database to communicate with each other
3. Create the view and a protocol for communicating with the android app for our system.
4. Create the views for the prototype and create the classes which will make use of the methods create in part 2.
5. Create the android app that will be able to scan QR codes and communicate with the server.

These will be discussed later in the section as these are major milestones for our prototype. Each member of our team was assigned one of these tasks to complete along with two sections of the documentation. Once a team member finished his or her code, they were then assigned to one of the other stages to assist.

Week 5 - 14th November

We went back to our functional requirements and performed black box testing for each one, using either equivalence partitioning or boundary case testing techniques depending on the requirement's inputs. The database and the Java methods for allowing the communication between the database and application were completed allowing the other 3 parts to be worked on as they depend on these parts of the system.

Week 6 - 21st November

Completed our proof of concept and documentation. Iterating back through all the previous stages to ensure all the separate sections of our documentation were consistent with one another.

Below is a GANTT chart that indicates how long we spent on each task during the development of our prototype. The requirements and use case section were revisited during much of the project which is why a considerable amount of time was spent on them.

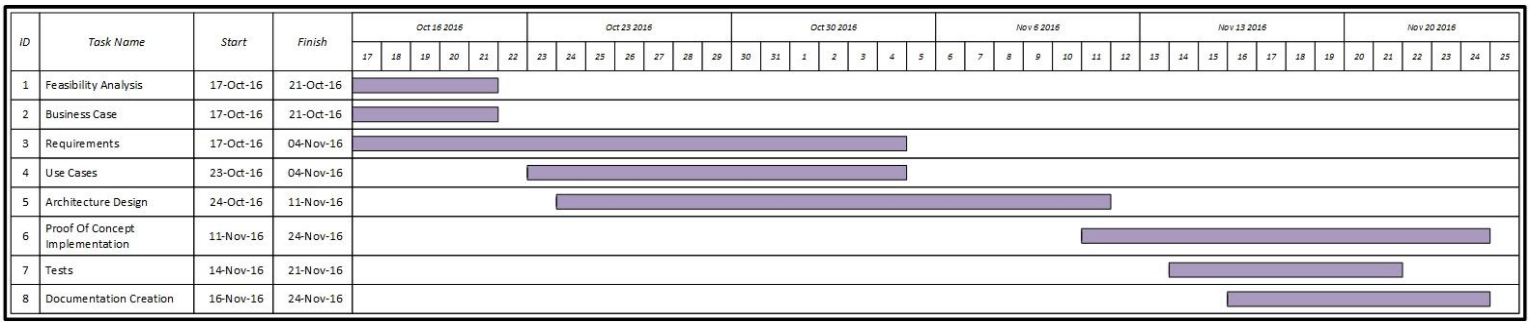

Figure 2 Project GANTT Chart

Milestones

As mentioned earlier in the scheduling section these are the 5 major steps that needed to be completed for our proof of concept. The ordering of these steps is important as the later steps are dependent on the completion of those that came before them in the list.

1. Create the database and SQL Queries - This needs to be setup so that the application can easily request and store data needed when in operation. This step needs to completed in order for the rest of the steps in the prototype to be functional.
2. Create the Java methods that interact with database - Once the database has been setup we can move onto the next stage of development. The implementation of this step will allow us to communicate with the database via the application. In the prototype the main use for this is to retrieve items from the database.
3. Create the view and a protocol for communicating with the android app for our system. - This stage is needed so that the android app can be displayed to the user and so step 5 can be easily integrated.
4. Create the views for the prototype - This step is the main part of the prototype. It encompasses what the user will see and be able to do when using the application. For the prototype the priority is to allow the user to select an item to purchase and be given a QR code to scan. This step makes use of the methods created in the second milestone.
5. Create the android app - This is the final milestone that will make use of the view in step 3 and the generated QR code in step 4 and will complete the ordering process for the user.

The Proof of Concept

This project from the very beginning could have been done in a different way – the simple one. Every component of our project might be accomplished at smaller cost, using either on-hand solutions (e.g. ORM instead of JDBC wrapper) or technologies we are acquainted with (e.g. Python instead of Java). Although returning from the deep end is never a process one possesses full control over, it turned out to be rewarding in the long-term. We will enter next semester having piece of software, which can comprise a core of an application meeting all quality requirements, and majority of the skills needed to build it.

One of the things, that caused us many difficulties was high interconnectedness of our project and resulting in permanent need for communication. As mentioned earlier, proof of concept was split into 5 object-like components. Such organisation leads to very cohesive and reliable code, but also high dependence of e.g. person creating JDBC wrapper on person designing database. Given the pressure of time, the only solution was to use agreed-on, existent interfaces, planned together, but without any similar experience. Despite the best of intentions, some miscommunication and inconsistencies could not be avoided. For example changes to be made in the design of the database were discovered by troubles that arose while plugging the ready JDBC wrapper into web page. Similar things usually come unexpectedly and can be off-putting, but in the end were always effectively solved by good cooperation.

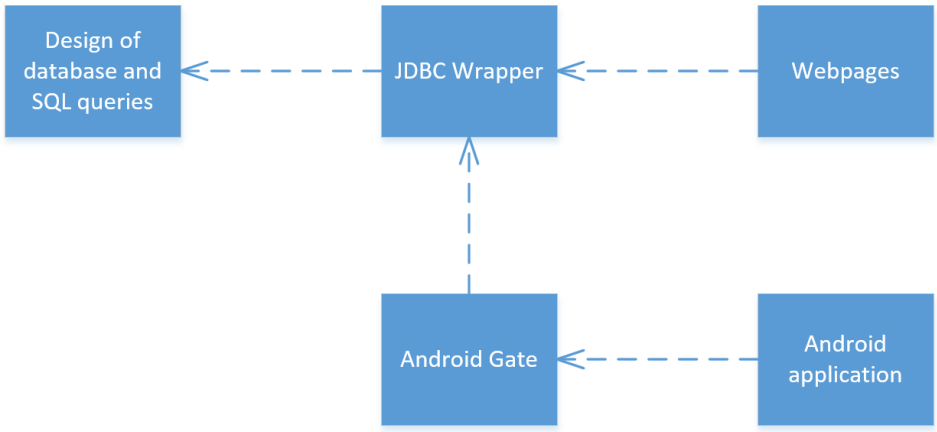

Figure 3 Dependencies between assigned tasks

A different considered approach assumed implementing functionalities – everyone implements queries, web pages, anything else needed to satisfy his/her requirement and only this. That was rejected, because lower quality of components would be unavoidable. Moreover, even though all parts depend less on others, certain problems must be solved multiple times and synergy in the team is significantly lower.

Another issue concerned the selection of a web framework. This decision was not easy, as it had to be both business- and feasibility-oriented. In other words, an overly simple framework, without

SSL support or a poorly scalable one would not make business sense, while for example the widely used in the industry Spring introduces tens of new concepts. Some are necessary to program at all, most to program properly.

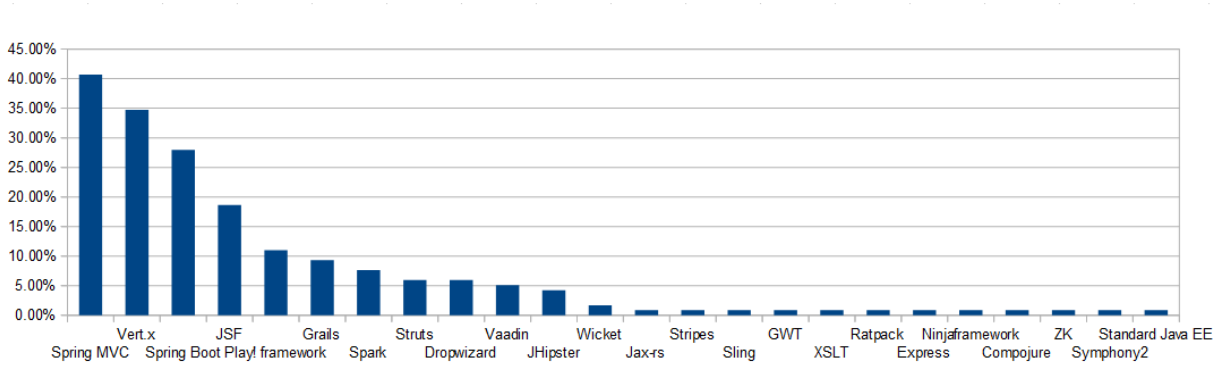

Figure 4 Results of unofficial survey about used web framework in Java

Our final choice was Spark. It seemed to bridge the gap between completely new, unreliable frameworks and professional ones. We were not mistaken regarding its functionalities, but did not consider other factors like available examples, community and documentation. The documentation was especially impoverished and this impeded our work and forced to learn through reading open-source projects. Studying others' code is surely a very developing activity for us, but also incredibly demanding in terms of time. Although we have already possessed most of the essential knowledge, it remains an open question if Spark is to be used for the full version.

Last problem described in this paper arose where the least expected. Android is a very young system, however during last years it has undergone a real metamorphosis. From the first, dramatically unstable releases it gradually improved its quality and market share. Nowadays it has over 75% of the market (according to Statista) and was assessed as far more reliable than iOS (by Blancco Technology Group). The score is even more impressive given how many completely different devices use Android compared to 15 models of iPhones. This tremendous change strongly influenced developments of apps - current libraries, examples and documentation are more consistent than ever. The more surprising was the fact, that QR codes scanner which used latest libraries and drew on official examples failed to pass tests on all devices with Android 7.0+.

The solution turned out to be strictly technical and not of central interest in this paper (different processing of permissions in newer versions), but taught us an excellent example that testing should never be skipped. Even for certainly correct components.