Platform for Real-Time Collaborative Web Applications

CO600: GROUP PROJECT

School of Computing
University of Kent, Canterbury,
CT2 7NY, UK

Waqas Aslam
Alex Austin
James Grant
Frederick Harrington
Harry Jones

Supervised by Fred Barnes

Waqas Aslam	Alex Austin	James Grant	Frederick	Harry Jones
WA40	AA745	JHG7	Harrington FH98	HJ80

***Abstract* — The fundamental concept involves connecting to a website on a desktop device and using it as a main display. Then multiple mobile-based users connect through a mobile version of the website. The mobile users can then interact with applications while viewing their changes on the desktop screen. Our platform will run applications of varying audiences and technical complexity.**

I. INTRODUCTION

This document will outline the development of the ANEXD platform with precise technical analysis of the implementation and the problems we encounter during development. The report is separated into technologically specific sections to which we will expand on technologies implemented in addition to their role as a part of the wider platform structure. The specific sections are the frontend website, the backend engine and the application facing API. Within each of these sections we will discuss the problems we faced, our technical reasoning for our development choices and final implementation methods.

II. BACKGROUND

The project concept was devised from a video game that allows users to play together on a primary monitor while their method of control was on an additional laptop running another instance of a game. Our discussion about the game soon developed into the idea that the core concept could be improved by utilising web technologies. The majority of people have personal mobile phones making the audience for such a web application far greater than on a laptop. When the idea was developed to incorporate access anywhere online capabilities and high levels of concurrency we felt the concept was of viable complexity to become a final year project.

As the idea developed during design, we chose to construct more than just a website but to create a platform. On this platform, the development of applications could be streamlined by abstracting application developers from the core of the platform. Thus constructing a smart platform for a dynamic user base which would increase complexity and usability of the project concept.

The name for the project ANEXD is devised from the word annexed which by definition is the action of adding something extra to a subordinate element. The action of adding a mobile to a desktop felt like a level of attachment, therefore, instigating our adoption of the name. Thus the manipulated name of ANEXD was devised. The name also keeps the platform from associating to any specific audience. From the concepts inception, we chose that the platform was not solely for games but for a plethora of entertainment applications as well as corporate collaboration tools.

The website and game that instigated our interest in this field for a final project were called Jackbox[3] and Kahoot[4]. Jackbox requires the users to use laptops, phones or tablets but they must be in the same room viewing one primary monitor. Furthermore, they must be on the same wifi connection to play. Kahoot, on the other hand, is an education tool for fast paced quizzes in a classroom environment. Kahoot works extremely well with a simple lobby creation on the primary screen and access anywhere abilities on additional desktop computers. These two applications perform their intended functions very well but they lack the capability to expand through community involvement. With Kahoot, the user can create quizzes but only within a refined and strict outline. Jackbox, on the other hand, hosts hundreds of games that have been constructed by the games developers.

Our assessment of these two systems developed the concept of ANEXD with intent to create a platform that was not confined to a certain type of application, e.g. Kahoots quizzes and Jackbox's video games. Why not have corporate collaboration applications alongside user designed video games.

When dealing with multiple users, we required a platform design that contains high concurrency. To ensure we implemented a framework with high concurrency ability we researched multiple papers centred around concurrency in the web application field. We also assessed common methods of concurrency with related literature. The majority of the web applications were developed with Erlang or Go heart of their backend implementation. Many papers dictated that the efficiency of the system relied more on task and resource management. They state *"tasks, queues, and thread pools"* are paramount in concurrency reliant systems [1]. A contrary point was that the simpler the platform, the more efficiency the system is in managing resources related to high usage [2].

III. AIMS

The overall high-level aims for the project are as follows. The platform will have a user-facing frontend website, one for mobile users and one for desktop users. Our website will allow signup, login and logout capabilities. The frontend will also grant the user the ability to change their password. The frontend must uphold a high level of user experience and user interfaces design. The desktop version of the website will allow users to create lobbies on selected available applications. Mobile users will have two methods of joining a lobby, through taking a photo of the QR code or inserting the lobby code. The lobby must facilitate multiple users at one time. The mobile user will log into the lobby using an anonymous name. The desktop version must also display application data whereas the mobile version must display the controls or a level of interaction for the application.

The backend must be able to save the users data so they can login at a later date. The backend manage the running of multiple lobbies alongside running of various applications. The backend will also control the instancing of the applications.

To prove the platform works we will create a set of applications as proof of concepts. They will be of varying difficulties and topic. We will also create an API that will wrap the applications providing abstraction and robustness to the platform.

IV. IMPLEMENTATION APPROACH

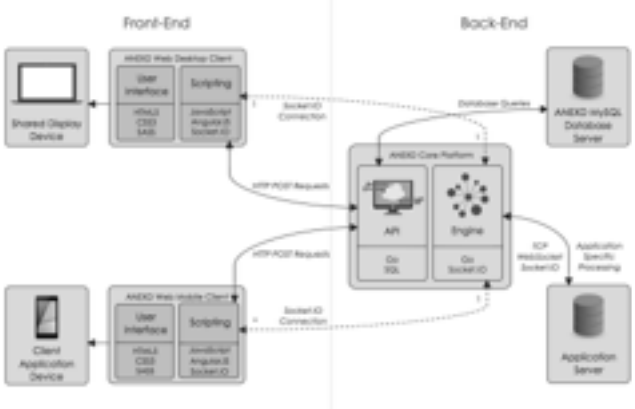
Platform Structure

This sub-section will include a high-level overview of our design plan for the structure of the system. It will not deep dive into the technologies or specifications used to implement the system. The ANEXD platform will be separated into three components; frontend, backend and application. These components will communicate to each other but will stay technologically separate.

Frontend encompasses all user-facing views, frontend data processing, backend requests and will also work to serve applications to the users. The frontend incorporates the uses of events and actions to link the business code in the backend and applications to the views.

The backend is the middle man between the frontend and both the applications and the API. It also includes the business code in the engine to both the applications and frontend. The backend has three major elements within it. The engine which is the business code for running concurrent applications, lobbies and user sessions. The second section is the API which severs as the controller for all communication between the frontend and the database. The final element is the database itself, containing static lobby, application and user data.

The final platform element is applications. This section contains the applications source code but is also wrapped with the ANEXD API. The ANEXD API aims filter an abstruse parts of application development for user-ability, robustness and security reasons. The ANEXD API aims to make it easier for the developer while abstracting them from the platforms inner workings.



Testing

Testing throughout development as well as the final product is of utmost importance. The scope of the project is an expansive one with many complexities and elements that require varying testing regimes. Therefore testing must be routine on each individual elements but also specifically on the connections between them.

A testing form was created which could incorporate the wide range of testing scenarios we would require for development. The form was intentionally left open and not restricted because many elements of the project could for example be tested by scripts whereas others could be tested by online validators or commercial products. One such example was the use of the application Postman[17] which is used to test HTTP Post and Get transactions. Some components of the platform were tested throughout development. All frontend implementations techniques were tested automatically by building tools.

Quality Assurance

The projects documentation and code is required to pass rigours quality assurance testing. A collection of documents were created to outline each implementation techniques specifically required quality testing rules.

To make sure that all documentation keeps to a set high standard, two further quality assurance documents were devised. One which outlines a set format as well as high level grammar and a second which dictates the systemic logging of meeting.

Each coding or scripting language used throughout the projects development requires a wide range of requirements therefore for each individual specific language a quality assurance checklist was created. This was to make sure that quality was upheld with each language. To strength these checklists we use external code validators, such as JsHint[6] for JavaScript or the W3C HTML validator[7]. The QA checklists were written by the group member that would be using or utilising that language the throughout the projects development.

Version Control & Repository

To deal with version control throughout the projects development we used the well known repository implementation of GitHub[5]. With a set folder structure we ensured that documents were not misplaced or saved incorrectly. GitHub also supports our development by providing automatic backups as the development directory is hosted in the cloud. GitHub grants the project the ability to see a catalogued cycle of development. Recording which members worked on what and when they committed the work. The GitHub can be found here: https://github.com/BlueHarvest28/ANEXD.

V. FRONTEND

Within this section we will asses the development of the frontend of the application. This is all user facing technologies and implementations. We will highlight the technologies used and their importance within the project. We will access the user interface design as well as the methods of communicating with the supporting components of the platform.

User Interface

Like any user-facing application, its essential we create a logical, clean, and enjoyable experience for our end users. To achieve this, we focused on not only the content we deliver, but also on the flow of the application. ANEXD went through a full design process from simple user flow diagrams to high fidelity designs and iconography. The user interface itself is built using W3C-compliant HTML5 [7] and SASS[8], which is a CSS[10] pre-processor for cleaner nesting and parent referencing.

As ANEXD is a single-page application, it is formed of a number of 'partials', which are pieces of HTML that are injected into the main view as required. This approach allowed us to maintain style and state across the application, further improving the end-user experience.

Angular.js

For our frontend workhorse language we choose JavaScript but more importantly the JavaScript Framework of Angular.js[11]. Angular.js extends HTML with directives and binds data to HTML expressions. Angular.js is mainly supported by Google but also by a plethora of other companies and individuals all with the common goal of providing a solid framework for single-page webpages.

Some frameworks provide a model view controller (MVC) or model view view-model (MVVM) implementation. Angular.js on the other hand provides a MVW or model view whatever structure. Angular provides us a lot of flexibility to neatly separate presentation logic from business logic and presentation state. HTML is wonderful for creating static web documents but to change or alter them you require hard refreshes. The Angular controller provides the behaviour behind the Document Object Model (DOM) elements. The controller causes Angular events that in turn manipulate the static HTML document. This could be by implementing callbacks when certain events occur or simply monitoring changes within the model changes. Angular does this by providing an extensive lists of directives. Some notable directives used in our implementation include ng-submit which is used to collect data from a form or set of inputs and supply said data to the controller for further processing. Another used prominently in the ANEXD implementation is ng-repeat which when given the parameters of a list from the controller will loop through the list displaying the contents. This was used by the frontend to display the members in a lobby or the applications on the main view.

By utilising Angular.js alongside HTML 5 we have been able to create what seems like multiple web pages in one, therefore removing all hard refreshes of the page. Utilising Angular event abilities with ng-shows and ng-click we can mutate the HTML dynamically, for example hiding and showing elements quickly and efficiently. When combined with booleans we were able to create one file with multiple seamless views. Within ANEXD frontend there is a singular HTML document that incorporates everything from the front-page of the website to inside the lobby view.

Angular has an extensive network of abilities where one can develop their own directives. We can also create our own directives. To reduce code and ensure no redundant code was written we developed multiple directives for our own use. These directives can be called from any controller or view on the frontend and aim to streamline actions that take place commonly without having repeated code in various controllers. For example one custom directive is require login. This custom directive checks if the user is logged into the login service.

Another important element used in our implementation is the $scope. Each Angular.js controller has a scope object. Only functions defined in the $scope object are viewable in the HTML views. For example in our implementation the scope functions are for collection of data from the view and a non scope function interacts with the backend.

A major element of the frontend implementation utilises Angular factories. Factories are like framework services in that they are injectables but factories are you can run custom code. In our platform implementation we use factories to abstract and remove business logic from the controllers. All of our factories are located in the factories.js file. These factories include implementations for socket management, API post requests and root $scopes.

During development the frontend required extensive methods of testing. Code standards testing was performed automatically by building tools that will be explained in the next section. To develop the communication methods between the frontend and the backend we required a method of testing. To do this we created a mock server to imitate the actions of the final GoLang implementation. We created this mock server in Node.js. It let us test lobby, user and applications communication events. The mock server did not contain any form of concurrency but allowed us to user test the frontend throughout development.

Grunt, NPM and Bower

To build and develop the frontend we used the service Grunt[13]. Grunt is a JavaScript task runner. Their slogan is one word: automation. By utilising Grunt we reduced the overheads when developing the fronted. Grunt runs on Node.js[12] but knowledge and understand is not required to run Grunt efficiently. Grunt brings a variety of tools together under one roof and combines them into a simple to use system. Minification[14], compilation and unit testing as well as a plethora of other tools that all can be automated and run using Grunt. After deciding what tools that are wanted by the developing party they are configured into the Gruntfile[13].

For ANEXD our Gruntfile contents revolved centrally on automated testing and dynamic building. While developing the frontend, Grunt would monitor the files within a certain repository directory and when code was altered Grunt would dynamically re-compile, build and run the test the code before building to a local view. This aided in development because we could rapidly see our changes on a built frontend as well as getting extremely quick and precise error feedback.

During development we required the installation of multiple packages both from custom and corporate backgrounds. Therefore we required a package management system, we used the common market leader npm[12]. Using npm allowed us to easily import packaged modules quickly while also storing them efficiently.

While npm allowed for management of Node.js modules we required a method for managing components that contain HTML, CSS, JavaScript, fonts and even image files. For this we chose Bower[12]. Bower supports an extensive list of components, frameworks, libraries, assets and utilities for website development and final builds. Along with implementation of the components, bower also supports and retrieves the these modules therefore making sure that all elements of the built website is up to date at all time.

The use of Grunt, npm and Bower are to decrease overhead for the frontend developers but also by strengthening testing and quality assurance throughout development. Errors are displayed immediately which concise explanations with aim to better improve the development process. Modules are managed and supported by both npm and Bower to make building more effective while also removing the requirement of periodic updating of components by the development team.

Socket.IO and $http

When dealing with concurrent users and multiple views and instances the user of efficient communication methods is paramount. The frontend has two types of data to be sending. One to the engine and one the backend database orientated API. For each of these methods of data transmission we chose different methods of communication. These were devised with the requirements of the relationship between the three components of the platform but also in minds to the type of data being transmitted.

For communicating with the database we built a backend API to structure, manage and execute queries, insertions and removals. To communicate with this backend API we chose to use HTTP Post requests. Angular.js provides an extremely efficient core service called $http[11]. This service facilities communication with remote HTTP services via the browsers XMLHttpResquest[11] object or via JSONP[11]. This allows us to use the integrated service in the form of a function within our Angular.js controller. Within our frontend implementation we incorporated this into an Angular factory to maintain code quality. During the design phase the message standards for communication between the backend API were devised and then implemented on both ends of the communication. These message standards dictate what the event names are, what data is expected in addition to what return is required. Examples of the used event names are getGames, newUser, changePassword.

For all HTTP Post requests used in our platform implementation the payload is a JSON[11] object contains varying elements. When creating a new lobby the JSON contains, the gameID of the application being run, the userID of the creator of the lobby and the size of the lobby the user requires. The HTTP Post is sent to the API and receives either a pass or fail. Within a pass data is returned to the frontend for display and further processing. The newly created randomly generated six digit integer. This is then processed by the frontend to create the lobby URL, QR code display and lobbyID. Using Angular.js to incorporate promises on the HTTP Posts ensure we take action on anything returned from the backend API.

For all communication to the backend engine we use WebSocket. During design aimed to implement pure TCP for these connections but instead settled on the more abstracted method of web sockets, which ultimately run over TCP. The ANEXD platform utilises Socket.IO[15]

for all real-time bidirectional event-based communication. We chose Socket.IO due to its popularity, level of support and extensive knowledge base. It handles browser inconsistencies which adheres well to our aims to make the platform accessible anywhere on anything. Socket.IO is both simple to implement and handle but also extensive in its capabilities allowing for publish, subscribe systems in addition to room support. Socket.IO also instantiates automatic reconnection which helps to simplify and strengthen the robustness of the platform but increasing efficiency and user satisfaction. Like the $http posts a set of specific set message types where devised during the design stage to better create transparency between the frontend, backend and applications.

We utilise an automated testing task runner, Karma, to handle the front-end unit testing. Unit tests are written in Jasmine, meaning that they are in a human-readable syntax, based around each test's description and expectations. Tests are written for the core functionality of each major controller, including HTTP requests, which are mocked using the Angular module ngMock[11]. These tests are executed automatically via grunt serve, as well as through grunt test and grunt build. PhantomJS is a headless browser[11] on which the tests are executed.

VI. BACKEND

The backend of ANEXD encompasses a selection of slightly separated elements, written in the Go programming language. Firstly, there is the 'engine' component, which is a highly concurrent system built to service vast quantities of connected users and lobbies. Secondly, there is the backend API which manages static data from the frontend with the aim of manipulating, managing and formulating data for the insertion, removal and querying of a database. Finally, we have a database which contains all persistent information about registered users, lobbies and end point applications.

The Go Programming Language

Through our requirements specification, it was identified that we would require a system that was capable of processing large amounts of data, coming from multiple inputs simultaneously. With the intention of not compromising things such as processing latency, it became obvious that a language with native concurrency would be necessary for optimal results. We also did thorough research on the potential technologies and libraries that would be required; most significantly in our decision to use the widely adopted WebSocket library Socket.IO.

Statistical benchmarking showed that the concurrent language Erlang in fact performed the best at handling the largest quantity of WebSocket messages per second, with Go not far behind in second place[23]; the significance being that these two languages use the CSP concurrency model. Other languages such as Node.JS, Python, Java, etc. performed considerably worse, especially under heavy load, adding significant latency or even crashing completely. Erlang however did not have an up-to-date library to support the latest version of Socket.IO, while Go has one which is being regularly maintained by a small team[24]. This, along with the knowledge that Go has all the functionality to support our other requirements as a systems language; made it the optimal choice to build our backend on.

Upon learning and implementing the system using Go, we discovered that it offered many other powerful features natively built into the language, which further support our engine. These include native slice and map data structures, an effective interface and struct system, communication channels and most importantly 'goroutines'; which are extremely lightweight concurrent processes which language developer Google have stated are capable of being run in the millions simultaneously without degrading performance[17]. Additionally, the Socket.IO library we have used uses goroutines to process incoming and outgoing messages, which is unique to the Go implementation of this library, making use of the language's strengths, and synergising well with our implementation of the engine.

Go is a relatively new and rapidly growing systems programming language, which natively supports the CSP concurrency model as seen in Erlang, combining with the imperative and structured paradigms of languages like C, providing a strong typing system and the use of low level features such as pointers.

Engine

The engine component of the system is the logistical core for applications running on our platform, which is primarily responsible for managing lobby sessions, connected users, and communication with the application specific server which is responsible for the application processing. The engine removes many concerns for the application developers, most significantly including the need to manage the connections of many users in a session, potentially large amounts of message processing overhead, and the management functionality of a user lobby system.

Its functionality is primarily split into four parts, the first of which being a lobby manager, which is the first point of access via the API backend component. The Manager struct uses a Map data structure to store all running instances of application lobbies, and all of the functionality for creating and destroying them. It also provides the first point of access for Socket.IO connections, with the aim of associating it with a respective User instance within an existing lobby.

The second part is the Lobby instance itself, which uses a Slice data structure for managing Users within a lobby. A slice was deemed a good choice here as it allows constant access time by the use of indexing – by which Users are conveniently indexed by their player number within the session. It also contains multiple communication channels, two of most significant being one for receiving Command messages; which are primarily used to modify lobby data, and one for forwarding processed messages from users to the

application server. The lobby also manages the connection to the application server, and supports both TCP and WebSocket connections.

The third part of the engine is the User instances, each representing a single connected user within a lobby. This struct provides the functionality to instantiate the incoming Socket.IO events from users, as well as emitting data to them on the frontend. Like the Lobby itself, each user has a communication channel used to receive and forward messages to the frontend.

The fourth and most technically significant component are the Message types. There are three main types of message; MessageServer, MessageUser and Command, which are all interface types, requiring a processing function for a struct to implement. There are many structs defined within the program, all with unique functionality for processing their respective instructions. The communication channels used within the other components send messages defined by the specification of one of the three interface types. Using run-time polymorphism (and in some cases for multiple interface implementation), the underlying message type will perform it's expected functionality immediately, as the function it runs is implicit to it's underlying struct type. Not only does this implementation make the system very modular and maintainable, but it removes the processing overhead of determining the message type. Additionally, these messages are intended to be run as a goroutine; meaning that all messages and operations are processed concurrently, unless the thread is blocked by a mutual exclusion lock when simultaneously attempting to modify the data structure.

Overall, the implementation of the engine has successfully made use of Go's strengths whenever applicable, and uses an efficient architecture; the majority of which is of constant time complexity, in addition to making full use of Go's concurrency model to achieve as close to real-time processing as possible; a primary ambition for the platform.

Backend API

The Backend was written in Golang as this allows direct communication with the engine. Golang is also a very quick concurrent language which will allow many API requests to be handled gracefully.

This part of the platform allows the frontend to retrieve persistent information from the database such as application, user and lobby information. This is achieved by serving an API in which the front end accesses via the use of HTTP POST requests. This is then handled by the Backend and processed, where the data is fetched and altered, and returned accordingly.

Some security measures were added prohibit anyone from using the backend API. To make a successful request the user must first be logged in our frontend login service where the user will then be given a key which the user must then use on all subsequent requests. This is handled in the Angular.js of the frontend and is completely abstracted from the user. This key will last for two days of inactive use where at which point you must then get a new key. Furthermore, when deleting a specific user, the password must be supplied, this ensures that the intended user that is being deleted is correct.

Some data security measures have been implemented. The first is for user passwords to never be sent back over to the front end. User passwords are then encrypted with the MD5[20] hashing algorithm along with a random generated salt which the Backend stores. The password sent to the Backend from the frontend is also MD5 encrypted and combined with the utilisation of a simple salt to make it more difficult for an attacker to obtain transported information.

To allow access to the API from anywhere it has been hosted on a service called Openshift[16], which is a free service that allows us to deploy our Golang server on the cloud available from anywhere. The service requires us to use a gear, which we edit with our Golang code and then deploy via a private Git. This is a separate repository from central project development repository. Within this Openshift repository is the ANEXD Engine in addition to the Backend API.

Simple worth through of how the API will handle a request. First the request will come in and get handed off to the relevant method so /newLobby will call the newLobby() function in the code. The request will then be decoded. This is done using one of Golang's standard libraries "encoding/JSON". Simply pass in the data structure and it will parse it into a GoLang map type which can then be used.

Once this is done a check will be done with the cookie field to check that it is valid. If so, the process will continue otherwise it will terminate and send a failed message. Before continuing a check is made to ensure that the parsed game and creator values exist in the database in there respective table. An error will be thrown if not. A check is then done to see if there is already an existing lobby with the creator from the request. If it does exist that lobby is considered to be unused and will be removed.

Next the 6 digit unique number needs to be created so we use the Golang rand library to generate between 1 and 999999 so up to 6 digits and if number doesn't meet the required int length a zero is appended to the front. The rest of the parameters in the request are then checked that they exist so for lobby it is creator game and size. A method called required params then appends to the query string when required using a pointer and creates a args array which is what is passed to the database function as the parameters.

All database functions are then executed using two libraries one is the driver for setting up the connection and handling any errors which is a Github contribution. The other is a Golang standard library called 'database/ sql' which makes the request to the already connected database using the driver. The results of this are handled and the response is sent accordingly. A fail sends a fail message constructed with a simple map method and parsing back to JSON string. A success appends the relevant data and constructs a different map parsed in the same way using the "encoding/JSON" library. This is then sent in the response from the according request.

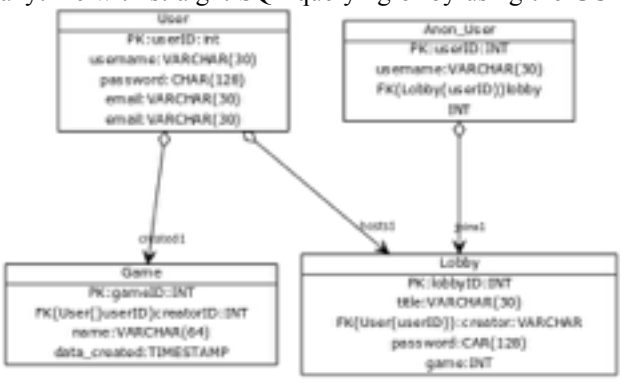
When creating the backend there were a few challenges. The first one was the previous implementation which was in Node.js. The problem with this was when the backend would then want to communicate with the Engine it would have to establish a connection through something like TCP or another mode. This would slow everything down and added an extra unnecessary level so the decision was made to move it into Golang. This however mean rewriting everything that had been done and that we now had to learn and move everything into Golang. The time constraints for learning and making this was a challenge.

Challenges within the scope of the backend were ensuring that any potential error that could occur would always send an error/fail response and not crash and return nothing. The security measure of adding a salt added an interesting challenge and step in retrieval and adding users.

Another complexity was an issue with the access control headers which caused the browser to block the requests. This was solved by an override of the MUX("gorilla/MUX")[16] contributed library. This was implemented so that the Backend would work with Openshift. Once the application was at a useable state and was starting to be used by rest of the platform it was crucial that any additions didn't break existing functionality and was kept consistent. If something did break it would have to be fixed immediately to ensure that it did not slow development elsewhere.

Database

The platform need to retain a constant set of persistent data. Everything that passes through the backend API is destined to interact in someway with the database. Either through insertion, deletion, updating or by queries. With the final build of the platform the lobby, applications and desktop users information is stored in the database. The database itself is hosted with GoDaddy. GoDaddy supports our platform with a MySQL database in addition to a management system called phpMyAdmin[18]. This allows us to browse the contents of the database at anytime with straight SQL querying or by using the GUI.



During development we changed the scope of the backend database. At first anonymous users from the mobile version of the platform had their information stored in the database. Anonymous users are ultimately volatile in nature, we felt that we did not benefit from having static records of their existence. Anonymous users are now handled completely by the GoLang backend engine. Therefore the table Anon_Users stills exists in all design documentation but does not feature in the final implementation.

VII. APPLICATIONS

The applications section encompasses the proof of concepts for the ANEXD Platform. For the applications themselves we chose to develop a selection of concepts that contained varying; audiences, complexities and use cases. We created six it total, an application for creating quizzes, one for playing said quizzes in addition to a four player real-time tanks arcade game. Finally two external API based visitation tools. The final two POCs utilised the Soundcloud[21] and YouTube APIs[22]. The applications were developed with the ANEXD API, which is our implementation technique for abstracting the application developer from the ANEXD Platform.

ANEXD API

To simplify the process for application developers, we produced an API for abstracting communications with the engine. The API provides functions for sending and receiving messages from the game server, and also handles users leaving the application. Messages can be processed in response to an initial message on the same event, with a promise callback similar to that of $http or at any time using Angular's $digest cycle via $scope.$watch [11].

The API abstracts the ANEXD platform internals from the application developer. The developer is not required to understand our lobby or user systems, nor do they need to program for the handling of players. The API also removes all aspects of communication handling from the developer. All they are required to code is to specify their subjective message event names. All WebSocket control is performed by the backend engine. The application receives a POST request containing a URL that they save. They emit URL which in turn instantiates the socket connection. The application is abstracted from the platform so that the developer does not know who it they emitting to or receiving from. The engine records and handles the sockets, which it does by utilising rooms to keep track of which game server is being played by which users.

During the development of the proof of concepts the use of a mock server was used to imitate the GoLang engine. The mock server which is the same mock server used in frontend testing was also used to test the capabilities of each application. The mock server allowed the applications to talk to both mobile and desktop users in the same method the final GoLang implementation would.

Proof of Concept

We wanted to create the platform with the ability of interfacing with any application written in any language.

A few rules must be met for an application to be able to interface with the platform, the most important being the language must support WebSocket or TCP connection capabilities. The second is that the application is written with extensive threading or concurrency capabilities, to allow the application to create multiple instances efficiently. Languages such as Java, Python and Erlang all contain multi-threading ability and therefore are extremely well suited for this role. Though our proof of concepts are written in Node.js. Though not an overall concurrent language at heart, Node.js is designed to build scalable network applications, therefore making it a simple and ideal technology for use in our proof of concepts.

VIII. Conclusion

We are exceptionally pleased with how the project has materialised. Not only have we met the vast majority of our aims, but in some cases we have surpassed them; ANEXD is technologically impressive and visually stunning; our documentation is broad, consistent, and encompasses all major stages of the project.

ANEXD also has a great deal of potential for the future, thanks to the backend's scalability and the multitude of monetisation avenues should it ever become a paid service.

The frontend successfully met requirements in terms of user-facing functions, and features an excellent user experience. The backend's concurrency capabilities exceed expectations, and considerations made in the design of the architecture allow for exceptional scalability, with most functionality running at constant time complexity.

Reflection

Reflecting on the project as a whole, we notice the mistakes we made early on. Particularly, we should have spent more time analysing our requirements, formulating our specifications, and discussing the communication and connection between the frontend and the backend. In some cases, these mistakes resulted in wasted time and discarded code, like the initial API which was written in Node.js, only to be rewritten in Go as a result of miscommunication. However, this change was still constructive, and massively improved the end result.

By developing the frontend and backend in parallel, the two teams were able to work almost entirely independently, which helped productivity. Unfortunately, this independence resulted in more difficulties when connecting the two components together. If more effort had been made to keep the two components synchronised throughout development, less time would've been spent connecting them.

In conclusion, we have thoroughly enjoyed the challenge this project has brought us, and are proud of our accomplishments.

IX. Acknowledgements

Fred Barnes - *F.R.M.Barnes@kent.ac.uk*

Who we thank for all his help and technological wisdom throughout the projects development in addition to his support on group dynamics and software development.

X. Bibliography

[1] Rob von Behren, Jeremy Condit and Eric Brewer, Why Events Are A Bad Idea, Last Accessed: 20th March 2016, http://www.cs.berkeley.edu/~brewer/papers/threads-hotos-2003.pdf - 2003

[2] Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler, A Design Framework for Highly Concurrent Systems, Last Accessed: 20 March 2016 , https://www.eecs.harvard.edu/~mdw/papers/events.pdf

[3] jackboxgames, jackboxgames, Last Accessed: 19th March 2016, http://jackboxgames.com/

[4] Festus Olatoye, Evaluation of Online Formative Assessment in the Classroom: A Comparative Case Study of Kahoot and Socratives, Last Accessed: 10th March 2016, https://www.academia.edu/11171554/Evaluation_of_Online_Formative_Assessment_in_the_Classroom_A_Comparative_Case_Study_of_Kahoot_and_Socratives

[5] Github, Github Showcases, Last Accessed: 20th March 2016, https://github.com/explore

[6] Rick Waldron, Caitlin Potter, Mike Sherov, Mike Pennisi, and Luke Page, JsHint About, http://jshint.com/about/, Last Accessed: 21st March 2016,

[7] W3C, Markup Validator Service, https://validator.w3.org/, Last Accessed: 19th March 2016

[8] Ben Frain, Sass and Compass for Designers, 2013, Last Accessed: 10th March 2016

[9] Constantine Stephanidis, Margherita Antona, Universal Access in Human-Computer Interaction, 2013, Last Accessed: 22 March 2016

[10] Ben Frain, Responsive Web Design with HTML5 and CSS3, 2012, Last Accessed: 10 March 2016

[11] Ari Lerner, Ng-Book - the Complete Book on Angularjs, 2013, Last Accessed: 11 March 2016

[12] Pedro Teixeira, Professional Node.js: Building Javascript Based Scalable Software, 2012, Last Accessed: 8 March 2016

[13] Jaime Pillora, Getting Started with Grunt: The JavaScript Task Runner, 2014, Last Accessed: 2 March 2016

[14] Minification https://en.wikipedia.org/wiki/Minification_(programming), Last Accessed 3 March 2016

[15] Rohit Rai, Socket. IO Real-Time Web Application Development, 2013, Last Accessed 19 March 2016

[16] Eric D. Schabell, OpenShift Primer, 2012, Last Accessed: 21 March 2016

[17] Shiju Varghese, Web Development with Go, 2015, Last Accessed: 27 March 2016

[18] https://www.getpostman.com/, Last Accessed: 25 March 2016

[19] Marc Delisle, Mastering Phpmyadmin 3.4 for Effective MySQL Management, 2012, Last Accessed: 21 March 2016

[20] William Stallings, Cryptography and Network Security: Principles and Practice, 2014, Last Accessed: 21 March 2016

[21] Soundcloud API, https://developers.soundcloud.com/docs/api/guide, Last Accessed: 8 March 2016

[22] Youtube API, https://developers.google.com/youtube/js_api_reference, Last Accessed: 22 March 2016.

[23] Eric Moritz, http://eric.themoritzfamily.com/websocket-demo-results-v2.html, Last Accessed: 21 March 2016

[24] googollee, https://github.com/googollee/go-socket.io, Last Accessed: 21 March 2016