Quizzing Bricks

D7017E project

David Eriksson, Niklas Frisk, Linus Hedenberg,
Andreas Hägglund, Simon Hietala, Martin Winbjörk,
Peter Otrebus-Larsson, William Gustafsson, Mikael Åhlén *

January 20, 2014

*daveri-9@student.ltu.se, nikfri-9@student.ltu.se, hedlin-9@student.ltu.se, andhgg-9@student.ltu.se, simhie-9@student.ltu.se, winmar-9@student.ltu.se, petotr-1@student.ltu.se, wilgus-9@student.ltu.se, mikhln-9@student.ltu.se

Contents

1 Introduction

1.1 Problem description

There are multiple devices in the mobile market today, even though this allows every user to find a device of their choice this choice can become limited due to applications on their device not allowing their associates from other platforms to share their applications. This also limits organizations trying to provide applications since they need to decide which platform to launch it on.

To solve this problem we made an efficient backend service with modern technology that allows for many users of different devices to use at the same time, at the same time it should allow for expanding upon and reusing for different types of applications. A client application for the different platforms was made to provide both a showcase of the concept as well as finding out the differences between the platforms.

1.2 Project delimitations

For this project we decided the target application to be a game. The game should have a gameboard and questions to answer, for rules see Appendix 2. This type of application allows a clear way inside the application to see when other devices are changing the state of the application while also providing a usable end result. We choose to limit our target client devices to Android API v.19, iOS 7 as well as a web application for Chrome version 29. For the mobile devices it was decided that a single API should be used for the application to provide a similar experience between the platforms. The web application should bypass the API and use the services directly, this was decided to provide early testing of the services provided by the backend as well as a fully functional client.

2 Use Cases

2.1 Login

2.1.1 Description

This use case describes how a user logs into the Quizzing Bricks application.

2.1.2 Preconditions

The user is registered on the service.

2.1.3 Basic Flow

This use case begins when a user wants to log into the Quizzing Bricks application.

1. The system asks for the username as well as the password of the user.
2. The user enters his/her username and password.
3. The backend service validates the username and password and logs the user into the application.

2.1.4 Alternative Flows

Invalid username or password. If the user would provide an invalid username or password in the basic flow the system would instead provide an error message and the user can choose to restart the use case or cancel.

2.2 Adding Friends

2.2.1 Description

This use case describes how a user adds a friend to his/her friends in the application.

2.2.2 Preconditions

The user is logged into the system.

2.2.3 Basic Flow

This use case begins when a user wants to add a friend after a login has happened in the Quizzing Bricks application.

1. The user opens the friends section of the application.
2. The user chooses the add friend option.
3. The user enters the name of the friend he/she wishes to add.
4. The backend service adds the friend to the users friends.

2.2.4 Alternative Flows

Unregistered Friend. If the friend entered by the user is not a registered user of the application, the system will respond with an error message, at this point the user can return to friends section and either try again or cancel the attempt.

2.3 Playing

2.3.1 Description

The user is playing a round in the game.

2.3.2 Preconditions

The user is a member of an existing game.
The user is logged into the application.
The user has not made a move during the round.
The user has chosen the game as the current screen in the application.

2.3.3 Basic Flow

This use case begins when the user has chosen an active game and it's the players turn.

1. The user chooses a square which has not yet been claimed and clicks on it.
2. The user chooses to receive the question.
3. The user receives a question with a 4 choice answer.
4. The user chooses the correct answer.
5. When the round is over, the user now owns that square and another round begins.

2.3.4 Alternative Flows

Postponing The Question. The user chooses to not receive the question now. The user is free to use the rest of the application or ending the session. If the user chooses not to receive the question within 72 hours, the game is forfeit.

Incorrect Answer. The user does not receive a square that round.

Question Battle. Two players in the same game chooses the same square during the same round and both players answers correctly.

1. Both players receives the option to answer another question until such time that one or both players answers incorrectly.
2. The square falls into the ownership of the user which answered correctly, if both answered incorrectly the square remains open.

2.4 Changing Platform

2.4.1 Description

The user ends a session on one device and then starts it on another device.

2.4.2 Precondition

The user is logged into the application on one device. The user has access to two or more devices that has access to the application.

2.4.3 Basic Flow

1. The user logs out of device A.
2. The user logs into device B and has access to the same things as the user had in device A.

3 Application Design

3.1 Web

The primary focus lied within the application functionality, the design was made using Bootstrap[**?**]. The basic design of the site is a top navigation bar with a Menu dropdown containing tabs which makes the user be able to handle game, lobby and friend service options. A "Create Game" menu which contains the options to either start 2 or 4 player lobbies where the user can invite friends to their lobby before starting a game with them. There's also the possibility to start the lobby before it's full. This will fill the rest of the lobby up with other players before the game is created. Alternatively there's an option to directly put the user into a 2 or 4 player queue by using the QuickJoin option for either game mode. There's also a Username and password field in the navbar along a Register Link to be able to register/login into the site.

Figure 1: The client website during a game.

As shown in Figure 1 the game board page basically contains a player status container on the left giving information about the color of the different players and their current game status (if they're placing tiles / answering question etc) and their current score in the game. On the Right side there's a game board which is build with a grey base tile which is overwritten by the corresponding players tile when they have won that specific tile, this game board is of the size 8x8 in the current version but can easily be expanded by changing the loop variable for creating the board.

3.2 Mobile Clients

One focus in the design was that the applications should look and handle the same way. This was later determined to not be the best approach because of the differences between android and iOS design guidelines. The functionality in Android and iOS handles the same with some minor differences concerning the placement of where you can find and handle lobbies.
The only other major difference is, as mentioned before, the graphic design and what additional libraries we use for example communication.

4 Technologies

The project is built with several technologies with key components built with the programming languages Python and Scala with support of different libraries and toolkits.

4.1 Tools

4.1.1 Git

The version control system used in the project was Git[?] and hosted on `http://github.com` because we all had some previous experience with this setup. On github we used a shared organization account (`http://github.com/quizzingbricks`) with our repositories and let the contributors to fork the main repository and send pull requests to merge back into the main repository.

4.1.2 Vagrant

Vagrant[?] is a tool that lets developers create and use reproducible and portable environments. We chose to use Vagrant because we wanted all developers to share a common view with all tools pre-installed and let each developer focus on coding rather than operations and configure its own machine. The same provisioning script to install all packages in the virtual machine was also used on the server in our production environment with some additional configuration.

4.1.3 OpenStack

OpenStack[?] is an open source platform for private and public clouds similar to Amazon's Web Services (AWS) and Google's Cloud Platform. Since we used Vagrant for our local development environment the task to deploy and manage the staging and production environment for the virtual machines was easy because the same provision of installable packages was re-used in the OpenStack as well which enabled us to have the same environment in development and production.

4.2 Backend

4.2.1 Python and Gevent

The Python language[?] is a general-purpose and high-level dynamic programming language and is used for most of our internal services (read more below about services in section 4.2.3) and the web client and also our HTTP-based RESTful API that power our mobile clients. Gevent[?] is a coroutine-based networking library that enable event-driven concurrency in Python instead of thread-based concurrency. The use of gevent and non-blocking I/O enable the services and HTTP-based web and RESTful API to allow higher concurrency and more simultaneous connections which is used for websockets that the API and web client expose.

4.2.2 Scala and akka

Scala[?] is a programming language which is syntactically close to its ancestor Java and can also be compiled to Java bytecode and use Java libraries directly in its code. The powerful object-oriented concepts in Java are included in Scala, but Scala also features extensive capabilities for functional programming, which include many features found in many purely functional languages such as pattern matching, immutability and lazy evaluation. Scala also extends the type system of Java into a unified type system together with higher-order types including co- and contravariant structures. It supports immutability and provides language and library means for writing code completely free of side-effects if desired.
Included in the native library of Scala is akka[?], which is a toolkit written for the Java platform that provides a framework for writing concurrency-based code through an implementation of the actor model of synchronous computing. Actors are objects that are constructed in a hierarchical fashion to allow for supervision and failover and communicate through a message mechanism. The akka toolkit is intended to provide for scalability by allowing each actor to seamlessly serve as an abstraction for anything from a thread to a server cluster.

4.2.3 ∅MQ and Protocol Buffers

∅MQ[?] is a socket library that enhances the socket programming and includes multiple message patterns were we use request/reply and publish/subscribe. Request/reply is used for client/server communication and publish/subscribe is used as a single publisher publish messages to potentially multiple listeners. Google's Protocol Buffers[?] is a serialization format that enables multiple languages to share the same transport format and protocol. Another popular method is to use a RESTful API over HTTP with JSON as transport format. As we evaluated the combinations we ended up with the ZeroMQ and Protocol Buffer approach since the HTTP adds an overhead on the TCP and also because we designed the protocol to only use request/reply without any special features that HTTP enables but also since we needed publish/subscribe and that is not supported by HTTP.

4.2.4 Database Technologies

For persistence of data, the relational database system PostgreSQL[?] is used, which is accessed through the SQLAlchemy[?] toolkit for the Python language, and the Slick[?] toolkit for Scala. The tool Alembic[?] has been used to migrate between database versions during development.

4.3 iOS 7

The iOS client needs to be written in Objective C and the only choice you have is trying to use a third party development tool that translates code into objective c. The problem with these tools is that they actually restrict you even further than Apple does with their tools and you end up with an application that doesn't feel good enough, so we decided to develop directly in Xcode using objective c. Apple provides a foundation framework that actually includes all the things we needed for our application from Strings to url requests and json parsing so no use for external libraries has arisen.

4.4 Android

Android uses java and xml to make applications, the graphical design is most often done in xml and uses java for functionality. But the graphic can be made in java as well, which allows for dynamic layouts compared to

xml's static layouts. This project focused on developing for Android phones running version 4.0 (Ice Cream Sandwich) and above. The main reason for this decision is that the percentage of devices running older version is around 20%[?] and by excluding the backwards compatibility more focus could be put on making a better ui using the newer API:s. This also gives our application more of a modern look which is harder to mimic with the older API versions. As for the development environment there are two official choices offered by Google, Android Studio and Eclipse. For this project we choose to work with Eclipse since Android Studio is still in beta. But as a IDE Eclipse has some flaws and one of the major ones is the amount of files that are included in each project which makes backing up files an unnecessary hassle.

4.5 Web

4.5.1 HTML5

HTML Is a markup language used for structuring and presenting information on the Internet. HTML5[?][?] is the latest iteration of the HTML standard. And is more closely working with JavaScript (JS) and Cascading Style Sheets (CSS). This technology has mainly been used for the above reason that is structuring and presenting the information on our web platform. Where Bootstrap has been the backbone in the design with the navigation bar layout and grid layouts for the body of the webpage and basic clean CSS. In addition to this JS has been used for additional functionality on the game board page.

4.5.2 Websockets

Websockets[?] is a protocol that got full-duplex communication channel of a single TCP connection. In the web applications it have been used to receive message containing information about changes of the game and player states in the backend. When these messages was received they called JS functions that updated the game board as soon as a full game round have been completed, they are also used to change the player status for a specific player whose state was changed.

4.5.3 Flask

Flask[?] is a micro framework written in Python and is based on Werkzeug WSGI toolkit and the template engine Jinja2[?]. Flask is used to implement the HTTP end-points for both the web application and the RESTful API that is used by the mobile clients.

5 System Architecture

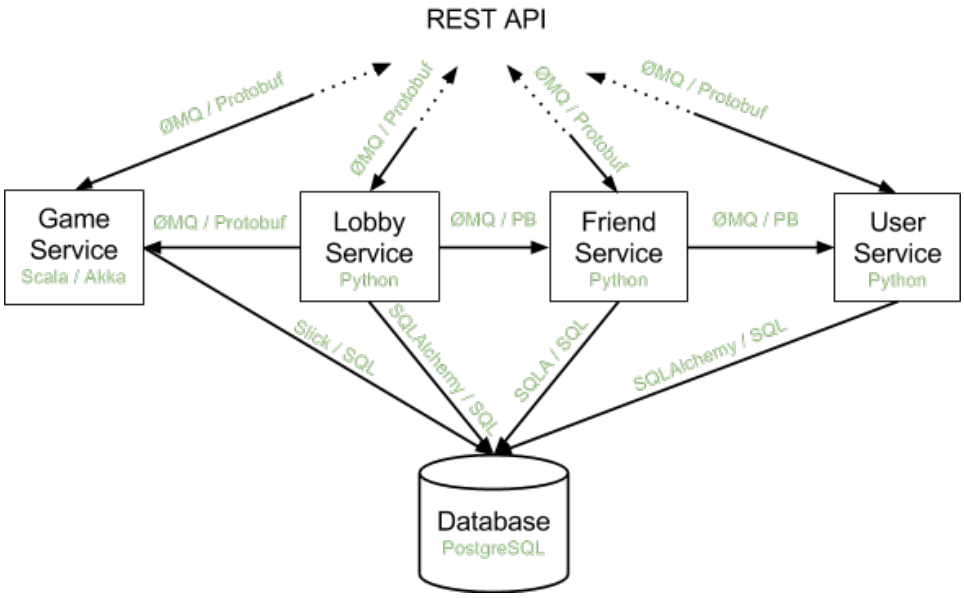

Figure 2: The infrastructure of the server backend. Each arrow indicates communication between respective services and green text indicates the main technologies used for the various entities.

5.1 Backend

The backend is divided into smaller parts and composed as services that run as independent processes, where each service handles a distinct part of the business logic. To enable services to talk between each other ZeroMQ is used as the transport layer and Protocol Buffers is employed as the serialization or transport format to enable different languages to share the same protocol.

ZeroMQ readily allows for implementation of message patterns such as the request/reply model and the publish/subscribe paradigm, and we have used the former pattern for the primary communication between processes and the latter pattern for websocket push notifications. An easily extendable architecture for the Python processes is built based on an inheritable base class that together with a shared protocol structure allows for quick development of new services. The architecture also allows for expedient definition of new methods that are callable by RPC messaging via ZeroMQ and Protocol Buffers.

5.1.1 Game Service

Built with scalability in mind, the game process handles all the game logic in individual games, as well as the creation and maintenance of new games, and the pruning of old games. Internally, the game utilizes a load broker pattern for load balancing to allow many ZeroMQ requests to be handled simultaneously, and uses Akka to parallelize the requests. The Slick API is used throughout to save game state to the database.

5.1.2 Lobby Service

This is a service that handles the creation of games. This service has functionality to create a lobby, remove a lobby and start a game. When a user has created a lobby the user can send invitations to friends and the friends can accept or deny these invitations. This service is used when a user checks which lobbies said user is a member of.

5.1.3 User Service

This service handles the users of the system. The service has functionality to authenticate, create and get users. The service is used when a user is created and when a user logs in. When a user tries to add another user as a friend this service is used to make sure that the provided email address for the friend really exists.

5.1.4 Friends Service

This is a service that handles the users connection with friends. The service gives the functionality to add a friend, remove a friend or to list all friends. The service is used when a user manages the friends list. It is also used while creating a game. The user can choose friends from the current friends list whom the user wishes to play with and invite them to a game lobby.

5.2 Web/RESTful API

As can be seen in Figure 2 the REST API is in the top and can call the different services with protocol buffers messages over ZMQ. The Web application Flask server is just like the REST API located in the top and use the services in the same manner as the REST API.

5.3 Mobile Clients

The mobile clients communicate with the backend using the web api, this communication is made using simple http-requests to different urls with custom headers such as user-token and sometimes providing data using a body of json. The responses are either just a verification that the request was made successfully or a body of json with information. Creating and handling these requests and responses in each and every view directly would be tedious and unnecessary so the focus and most work for each client has been creating a communication manager that handles these requests as easily as possible. We wanted these communication managers to be made in such a way that you could easily expand upon them when new features are added to the backend. The android and iOS client has each achieved this in a different way but achieved it nonetheless.

5.4 Security

The security for the systems can be seen as two different parts. The first one is the security for the RESTful API that mobile clients consume and produce data to and the other is the web frontend.

5.4.1 RESTful API

Today the most popular security scheme used by platforms and applications is the OAuth 2.0[1] which is used by the larger actors to provide public available API's, such as Facebook, Google or Github. The main difference related to our project is that we don't expose a public API in that sense, since we don't plan to allow 3rd party developers to create their own clients. Because our API is private, our authentication scheme is more simple than the fit-for-all OAuth 2.0 scheme.

The security scheme for our mobile clients is that the user provide the email and password in the mobile client and transmit the data over the network to the RESTful API to exchange the email/password-credentials for an access token. The access token is then used instead of the email and password during requests to the RESTful API by the mobile client when it needs to acts on behalf for the user.

The access token is a signed token that contains the user id for the authenticated user and a timestamp when the token were obtained which means that the end-user can read the values from the token, but not change it due the signature unless the end-user in some way have got access to the secret key that is kept on the server. To limit the eventual spread of access tokens, they are only valid during a limited period and in our case it is 7 days and after that the user is required to request a new token by the email and password credentials. During development and the demos running on the production server, all communication have been done without TLS, but in a real scenario TLS is of course required to protect the users data that is transmitted over the wire.

5.5 Web frontend

The authentication scheme on the frontend is based on session cookies which means that the user provides email and password and is then logged in by storing a signed cookie, similar to the approach described in the RESTful API security, but simpler and without the timestamp. The web frontend is protected to the most common security vulnerability Cross-Site Scripting[2] (XSS) since the templating system provided by Flask (Jinja2) by default escapes the output and no known security vulnerabilities exist as today in

[1] http://oauth.net/2/
[2] https://www.owasp.org/index.php/XSS

Jinja2 that enables XSS. Other common vulnerabilities such as Cross-Site Request Forgery[3] (CSRF) is not secured by default and such attacks are possible to do as today, like add or remove friends and possibility to fetch or answer a quiz question.

6 Result

6.1 User Experience

6.1.1 Login / Register

For the login we choose to use the email as the users unique identifier and a password for the means of authentication. The main reason for going with a username and password authentication instead of a oauth solution is because this solution can be used without having a third party account. Implementing an oauth solution that everyone can use is time consuming because you would need to have more than one implementation (like Google+, Facebook or Twitter login methods) and therefore not fitting for the time scope of the project. This means that a username and password system would need to be implemented in case a user has no accounts on the given services. The main drawback with this kind of login method is that it requires the user to remember a new password instead of using an already existing username and passphrase.

6.1.2 Friends Management

To make the process of inviting friends to the game more easy we choose to implement a friends list feature.
Friends are added to the friends list by entering their email address and after that they can be invited to any lobby by just a single click or tap. Our implementation does not require the other party to accept the friends request (like in a social network) and the major reason for that is because our friends list is more suppose to behave like a shortcut for inviting friends to the game. Having the other party accepting a friends request is therefore unnecessary since our application doesn't limit how users can send invitations to each other. This is however a feature that needs to be considered if the

[3] https://www.owasp.org/index.php/CSRF

number of social features increase in the application (like a chat feature for instance).

6.1.3 Lobby

The lobby was chosen to allow users to make a party of friends before going into a game. The owner of a lobby may choose to invite how many friends

he/she wishes for, however there are only 2 or 4 slots that may be filled (the owner takes one slot), how many slots there are depends on the game type. The remaining users may not join the lobby when it has become full. The owner may however start the game at any time, in such an occasion the lobby will become locked from joining by invitation.
The lobby will then be placed in a queue depending on game type and how many players are currently in the game. The lobby then becomes merged with other lobbies until such a time that all slots are filled, the lobby will then be sent to the game service to become a game.

6.1.4 Game

When a game starts it will be added to the games list where it can have two different kinds of states, waiting for other players and ready to place brick.
The game board itself is represented by bricks which is placed in an eight by eight square. White bricks are untaken places on the board and the players bricks are represented in different colors. The board view also includes a list showing which colors the players have.
When a user wants to place a brick on the board he/she first picks a position on the board, if it's a legal move the user will be prompted with a question. In this state the user can either answer correctly or incorrectly to the question and also time out if a answer has not been given after 20 seconds. When a question times out it will act as a false answer. If more than one player has chosen the same brick and answered correctly to the question there will be a "Question Battle" to determine which player gets the brick. In a question battle the players will each get a new question, if a player answers incorrectly to a question they are out. If more than one player answers correctly to a question they will all be given a new question. The question battle continues until there are either one player or no one left. In the case of the second outcome (where all of the users has answered incorrectly to the current question) no one will receive the brick and it will

be left untaken in the second round.
If a player has answered the question correctly the brick on the chosen position will either get their color or a lighter version of it. The second option acts as a placeholder when there still are players in the game that haven't picked a place to place their bricks. This is to to give a visualisation to the placing of the brick and to inform the player that it might be a question battle over it.
When all the players have placed their bricks and all the question battles are over a new round will begin.

6.1.5 Cross Platform

The application created for this project is available on a total of three platforms and a user can switch between them. All of the platforms have the same features so you don't lose any part of the core experience when on one of the applications. The big difference between all of the clients is that they have different user interfaces as we chose to follow the platform guidelines for user interfaces.
Because the state of the game is controlled by the server it means that a user can log in to any of the three devices and resume their game where they left of. This gives the users more freedom in how they chose to interact with the application, whether it being on a big screen pc or a portable device on the go. It also means that a user can, for instance, start a new game on the web browser at home and play the next round on the daily commute on the bus.

6.2 Project Goals

6.2.1 Scalability

Through the use of APIs such as Gevent and Akka, we have provided a scalable sub-architecture inside each process that allows the backend to efficiently utilize available computing resources to serve a very large number of clients simultaneously. Therefore, we feel that we have achieved our initial goal to provide the necessary level of scalability for a project of this character.

6.2.2 Extensibility

Since we have purposely divided the backend into subcomponents implemented as processes and have provided an infrastructure that is constructed with extensibility in mind, we have provided ample and powerful means to readily expand the backend with new processes and modules. We deem that we have satisfied our objective of being able to easily extend the system with new functionality.

7 Discussion

7.1 Development Issues

When we planned the project we decided that with so many members we could create both an Android and an iOS client to prove how cross platforming would work. This meant setting two people to work on the android client and one on the iOS client thinking that the clients would be done in good time before the end of the course. Unfortunately we did not consider the steep learning curve of creating a full application with little previous experience. This meant that the three people set to work on the clients were pretty much locked in to clients with too much work spent to scrap them when we realised that they would take the full extent of the course to complete. So in hindsight we would probably have gone with only one of the mobile clients.

Another problem we had was the difficulty in getting something to work as early as possible without being completely done with it. Most features we implemented were unusable until they were finished since we set to make them as good as we wanted them to be for the finished product from the beginning. This is probably where most of us have learned most from our mistake since it is so much more useful to have a bare bone working prototype to share with the team than nothing at all until you have a finished product. Especially since you are unlikely to have achieved a finished product without testing it with the team.

In both the android and the iOS client the focus has been to get all features working and not in trying to make a beautiful designed interface shared between devices. So for each client we've stuck to the standard way of making a simple application. This has lead to some minor differences between the

mobile clients but we believe that trying to for example recreate the iOS layout in the android phone or vice versa would only confuse the user more since an android or iOS user expects an application to work in a certain way when using an android or iOS device.

7.2 Future Work

The most prominent work would be to actually launch the application. What is stopping us today is that we have no questions for our quiz game, for this purpose we would need to buy a service providing this. For the application designs on web and the mobile clients more time could be spent on graphical design to make the clients look better, also reducing the clicks needed to do things inside the application by using "smart" patterns or additional settings so you for example, automatically accept invite from a player etc. For the game itself, one can think of developing additional game modes that you can choose to play.

A API

The API[**?**] is segmented into four different parts each describing the main responsibility of that section.

A.1 Users

A.1.1 /api/users/

Sent
Post email="email@example.com", password="password"
Response
200 ok
Response fail
{"errors":{"message":"Internal service error", "code": "000"}}
{"errors":{"message":"This mail is already taken","code":"101"}}
{"errors":{"message":"Missing email or password","code":"102"}}
{"errors":{"message":"Service not available", "500"}}
{"errors":{"message":"Error code not defined","code:"0"}}

A.1.2 /api/users/login/

Sent
Post email="email@example.com", password="password"
Response
{"token":"abcdefghijklmnopqrstuvwxyz1234567890"}
Response fail
{"errors":{"message":"Wrong email or password","code":"010"}}
{"errors":{"message":"Internal service error","code":"000"}}
{"errors":{"message":"E-mail or password is missing, check your data", "1"}}
{"errors":{"message":"Service not available", "500"}}
{"errors":{"message":"Error code not defined","code:"0"}}

A.1.3 /api/users/me/

Sent
Get token="<tok_id>"
Response
{"id":"<u_id>", "email":"<mail_address>"}
Response fail
{"errors":{"message":"The token is invalid, please login again","code":"1"}

A.2 Friends

A.2.1 /api/users/me/friends/

Sent
Get token="<tok_id>"
Post token="<tok_id>", friend = "<email>"
Response
Get: {friends : [{"id":"<u_id1>","email":"<mail_address>"},..]}
Post: Plaintext "OK"
Response fail
{"errors":{"message":"Missing required friend parameter", "004"}}
{"errors":{"message":"no such user exists","code:"0"}}
{"errors":{"message":"The token is invalid, please login again","code":"100"}}
{"errors":{"message":"Service not available", "500"}}
{"errors":{"message":"Internal service error", "000"}}

A.2.2 /api/users/me/friends/<user_id>/

Sent
Delete token="<tok_id>"
Response
Plaintext "OK"
Response fail:
{"errors":{"message":"no such user exists","code":"011"}
{"errors":{"message":"The token is invalid, please login again","code":"100"}
{"errors":{"message":"Service not available", "500"}}

{"errors":{"message":"Internal service error", "500"}}

A.3 Lobby

A.3.1 /api/games/lobby/

Sent
Get token="<tok_id>"
Response
{"lobbies": [
{
"l_id": <l_id>,
"owner" : <bool>,
"size": <size_int>,
"invited_count":<int>,
"accepted_count" : <int>
}, ...
]}
Response fail
{"errors":{"message":"The token is invalid, please login again","code":"100"}
{"errors":{"message":"Service not available","code":"500"}}
{"errors":{"message":"Internal service error","code":"500"}}

A.3.2 /api/games/lobby/

Sent
Post token="<tok_id>" size="<size>"
Response
{"lobby": {
"l_id" : <l_id>,
"size" : <int>,
"owner" : <bool>,
"players" : [{
"u_id" : <u_id>,
"u_mail" : <string>,
"status" : "accepted"/"waiting"

},...]
}
}
Response fail
{"errors":{"message":"The token is invalid, please login again","code":"100"}
{"errors":{"message":"Missing required parameter size","code":"004"}}
{"errors":{"message":"Service not available","code":"500"}}
{"errors":{"message":"Internal service error","code":"400"}}
{"errors":{"message":"Internal server error","code":"500"}}

A.3.3 /api/games/lobby/<l_id>/

Sent
Get token="<tok_id>"
Response
{"lobby":
{"l_id"="<l_id>",
"size":<size_int>,
"owner":True/False,
"invited_count": <int>,
"accepted_count": <int>,
"players[4]" : [{
"u_id" : <u_id>,
"u_mail" : <string>,
"status": "accepted"/"waiting"},...
]}
}
Response fail
{"errors":{"message":"The token is invalid, please login again","code":"100"}
{"errors":{"message":"Lobby does not exist","code":"226"}
{"errors":{"message":"You are not permitted to that lobby","code":"42"}
{"errors":{"message":"Internal service error","code":"400"}}
{"errors":{"message":"Service is not available","code":"500"}}

[4]Should be sorted as Owner>Accepted>Waiting

A.3.4 /api/games/lobby/<l_id>/accept/

Sent
POST token="<tok_id>"
Response
Plaintext "OK"
Response fail
{"errors":{"message":"There exists no such lobby","code":"226"}}
{"errors":{"message":"You are not permitted to that lobby","code":"42"}}
{"errors":{"message":"The lobby is full","code":"225"}}
{"errors":{"message":"Service not available","code":"500"}}

A.3.5 /api/games/lobby/<l_id>/deny/

Sent
Post token="<tok_id>"
Response
Plaintext "OK"
Response fail
{"errors":{"message":"The token is invalid, please login again","code":"100"}}
{"errors":{"message":"The lobby is full","code":"225"}}
{"errors":{"message":"There exists no such lobby","code":"226"}}
{"errors":{"message":"You are not permitted to that lobby","code":"227"}}
{"errors":{"message":"Service not available","code":"500"}}

A.3.6 /api/games/lobby/<l_id>/invite/

Sent
Post[5] token="<tok_id>", invite = { "invite" : [<u_id1>, <u_id2>, ...]}
Response
Plaintext "OK"
Response fail
{"errors":{"message":"Required JSON body is missing or bad type, check your content-type","code:"0"}}

[5]Header should be `Content-Type = "application/json"`

{"errors":{"message":"Required JSON body is missing or bad type","code":"004"}}
{"errors":{"message":"The token is invalid, please login again","code":"100"}
{"errors":{"message":"There exists no such lobby", "code": 226}}
{"errors":{"message":"You are not permitted to that lobby","code":"42"}
{"errors":{"JSON object invite should be an array with only integers","code":"004"}}
{"errors":{"message":"Service not available","code":"500"}}
{"errors":{"message":"Internal service error","code":"0"}}

A.3.7 /api/games/lobby/<l_id>/start/

Sent
Post token="<tok_id>"
Response
Plaintext "OK"
Response fail
{"errors":{"message":"The token is invalid, please login again","code":"100"}
{"errors":{"message":"There exists no such lobby","code":"226"}
{"errors":{"message":"You are not permitted to that lobby","code":"42"}
{"errors":{"message":"Service not available","code":"500"}}

A.3.8 /api/games/lobby/<l_id>/end

Sent
Post token="<tok_id>"
Response
Plaintext "OK"
Response fail
{"errors":{"message":"The token is invalid, please login again","code":"100"}
{"errors":{"message":"You are not permitted to that lobby","code":"42"}
{"errors":{"message":"Service not available","code":"500"}}

A.4 Games

A.4.1 /api/games/

Sent
Get token=<"tok_id">
Response
{"games":[{"id": <int>, "size": <int>, "state": <int>}, ...]}
Response fail
{"errors":{"message":"The token is invalid, please login again","code":"100"}
{"errors":{"message":"Service is not available","code":"500"}}

A.4.2 /api/games/<g_id>/

Sent
Get token=<"tok_id">
Response
{"gameId" : <int>,
"players": [{
"userId" : <int>,
"email" : <string>,
"state[6]" : <int>,
"x" : <int>,
"y" : <int>,
"score" : <int>,
"answeredCorrectly" : <bool>
}, {...},...
],
"board" : [<int>, ..]}
Response fail
{"errors":{"message":"Game service not available","code":"500"}}

[6] 0 can place
1 can get question
2 is answering
3 waiting
4 Lost
5 Won

A.4.3 /api/games/<g_id>/play/move

Sent

Post token="<tok_id>", x="<x_coord>", y="<y_coord>"

Response

200 ok

Response fail

{"errors":{"message":"Some of the required parameters x and y are missing","code":"004"}

{"errors":{"message":"Service not available","code":"500"}}

{"errors":{"message":"Game service not available","code":"500"}}

{"errors":{"message":"The token is invalid, please login again","code":"100"}

A.4.4 /api/games/<g_id>/play/question

Sent

Post token="<tok_id>"

Response

{"question":"<String>" "alternatives":["<a1>","<a2>",..] }

Response fail

{"errors":{"message":"Service not available","code":"500"}}

{"errors":{"message":"Game service not available","code":"500"}}

A.4.5 /api/games/<g_id>/play/answer

Sent

Post token="<tok_id>", answer="<Int>"

Response

{ "isCorrect" : <bool> }

Response fail

{"errors":{"message":"Missing required parameter answer","code":"400"}

{"errors":{"message":"The token is invalid, please login again","code":"100"}

{"errors":{"message":"Service not available","code":"500"}}

{"errors":{"message":"Game service not available","code":"500"}}

A.4.6 /api/games/<g_id>/events

Sent

Post token="<tok_id>", answer="<Int>"

Response

```
{ "type" : "player_change" ,
"payload": {
    "player": {
        "id" : <u_id>,
        "state[7]": <s>,
        "score": <int>
        }
    }
}
```

or

```
{ "type": "board_change",
"payload": {
    "board": [<b1>,..],
    "players": [{
        "id" : <u_id>,
        "state" : <s>,
        "score": <int>
    }, {...}, ...]
}
```

Response fail

```
{"type":"unknown","payload":{"msg_type":<msg_type>}}
{"type":"type","player_change":{"payload":{"player":
    {"id":<id>,"state":<state>,"score":<int>}}}
```

[7] see section A.4.2

B Game Rules

A game allows for either two or four players and consists of a series of turns, where at the start of each turn, each player decides where to place a brick on a game board of size 8x8. After each player has chosen where to put their brick, the player is able to request a question. After having received the question, they have to answer it correctly within a small amount of time to be allowed to place their brick on the chosen position. After this, a new turn starts. If two or more players have chosen the same position, each of those players will be asked more questions until no more than one person has answered correctly to all of the questions. When a player gets his brick added to the board, he gets a number of points added to his score that is equal to the total number of contiguous bricks that can be traced through the newly acquired position, in a linear fashion, horizontally, vertically and diagonally. The player with the highest score when the score count of any player has reached 50 has won the game.

C Deployment

The deployment of the system is divided into the Backend and Web/RESTful API and the mobile clients, Android and iOS.

C.1 Backend and Web/RESTful API

In the project the usage of Vagrant as the local development environment and OpenStack in production was used as described in the technologies section but neither of these are required to install and run the system. This deployment guide will go through how to install system dependencies and running the systems. This guide expect the operational person to have a machine ready with a 64-bit version of Ubuntu 12.04 or better.

C.1.1 System environment

With a running machine with Ubuntu 12.04 or better the first step is to fetch the latest version of the source code by downloading the tarball for the master branch in the repository quizzingbricks-core from github.com.

In the terminal, execute the following command that fetch the tarball and extracts the content followed by moving into the folder.

```
$ cd ~/quizzingbricks
$ wget https://github.com/quizzingbricks/quizzingbricks-core/archive/master.tar.gz
$ tar xzvf master.tar.gz
$ cd quizzingbricks-core
```

With the source code available, the initial step is to install all system dependencies with the provided shell script named *bootstrap-dev.sh*. The installation require root permissions as all following operations in this guide. This will take a moment (expect at least $10 - 15$ minutes) depending on system to install all system dependencies.

```
$ sudo -i
$ sudo chmod +x bootstrap-dev.sh
$ ./bootstrap-dev.sh
```

The next step is to install the python packages required for the python applications in the project.

```
$ sudo python setup.py install
```

With the dependencies installed, the next step is to do a database migration to create the tables and indices required for the system. This is done via the command *alembic*.

```
$ alembic upgrade head
```

With the database ready, the next step is to start each service. The python services are started via the *quizctl.py*-file in the bin-directory and the Scala application is started via Simple build tool (sbt).

The python services or applications provided in the system are *web*, *webapi*, *userservice*, *lobbyservice*, $friendservice$ and *pubsub*. To start each application, execute the following command, notice the & after the command since we place the application in the background.

```
$ python bin/quizctl.py SERVICENAME
```

To start the scala application, move into the folder *src/scala/gameprocess/* and execute the following commando.

```
$ _JAVA_OPTIONS="-Xms256M -Xmx512M -Xss1M -XX:+CMSClassUnloadingEnabled"
 sbt "run-main GameProcess"
```

Now should all components for the backend and web be running and access the web frontend on port 5000 and the RESTful API on 8100. Example http://127.0.0.1:5000 and http://127.0.0.1:8100.

C.2 iOS

C.3 Android