Prototype description

Victor Dahlberg, Philip Laine, William Björklund,
Jimmie Berger, Fred Hedenberg, Edvin Tobiasson

October 2016

Contents

1 Overview

The application includes a few big and essential parts as well as some smaller and less essential ones. The most important parts (divided into the corresponding activities/classes) will be explained and discussed in their own subsections.

Main Menu

Figure 1: The main menu.

This is the first thing you'll see upon starting the application. Pressing, "Inställningar" will move you to the settings activity. In the settings activity you can change the voice speed and log in a user or create a user. If pressing "Flerspelarläge" while not logged in, will take you to a login-activity where you as in settings, can log in or create a user. This is since every user need some unique identifier. "Enspelarläge" will get you to Choose Game Menu, described below, where you can choose a game to play. "Profil" shows the statistics of your recent singleplayer and multiplayer games. Mostly a "speak" button is offered next to a text in case the user is not able to read. We didn't add the speak button to the menu because we believe the symbol next to an option represents the specific option well.

Choose Game Menu

Ideally this menu would be filled with various mini games with the goal to develop your Swedish skills. Unfortunately there aren't, as of now there are two games. Sentence Game and Memory, which are both described below. The games are supposed to be short but rewarding.

Sentence Game

Sentence Game is a game where you are given a sentence with a missing word, a picture representing the complete sentence and four options of words to complete

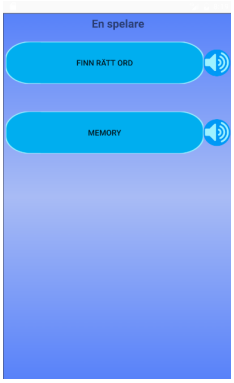

Figure 2: The Menu for choosing a game.

Figure 3: The sentence game.

the sentence so that it makes sense. You choose a word by dragging it to part in the sentence where there is a word missing. If you click a word, you will hear the word. If you click the sentence, you will hear the sentence. These are consciously chosen features made so that the user can learn Swedish even though she/he can't read, or just to learn pronunciation of the words.

Memory Game

The memory game is designed with the intention of learning words. Clicking on a card will show either a word or an image. The objective is to expand you vocabulary by matching a word wih its corresponding image. There is also a "Lyssna" button in order to satisfy the needs of the illiterate users.

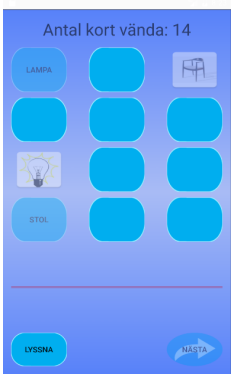

Figure 4: The memory game.

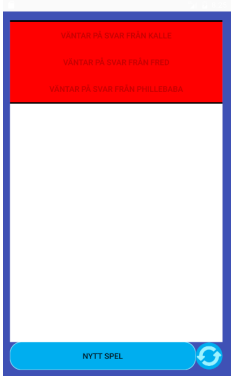

Figure 5: The multiplayer landing page.

Multiplayer Landing Page

The Multiplayer Landing Page is the first page you arrive to if you are logged in as a user. This page show all of your current and unstarted games. Finished games can be seen in the "Profil" activity available in the Main Menu. The games shown in the multiplayer landing page are ordered under three different categories. On the top games where you have been challenged but not yet responded are shown. You can either accept or deny a challenge. Secondly games where it is your "turn" are showed. Multiplayer games are played in turns; first the challenger answers the questions, when he/she is done, the challangee answers the questions. As of know only the sentence game has multiplayer support. Thirdly games where it is the other players turn are shown. And lastly games where you have challanged a user but the challangee has not yet responded to the challenge are shown. Games where it is your turn are shown in green while games where it is not your turn or you are not able to do anything are shown in red.

The Multiplayer Landing Page also has two buttons at the bottom. The universally known update button which gets new information from the database about your games and updates the view accordingly, and "Nytt spel" button. By pressing Nytt Spel you will be forwarded to another view. This is a classic searchfield with a scrollview of the results. From the results of the search you can challenge a user. If you for example type "Ka", a button with the text "Utmana Kalle" will appear which you can also click to challenge the user Kalle. You will then come back to the Multiplayer Landing Page.

2 Design rationale

API-level

We chose Ice Cream Sandwich (API 15), android 4.0.3, as the minimum supported API-level. This is a fairly old API but suits the intended user well. Since the intended user will mostly likely have an old phone running an old version of android we had to get maximum coverage. API 15 and newer has a coverage of just over 97%, which is a high enough coverage to make it a comfortable choice.

Backend

The app was going to include a online component to allow users to challenge each other in an asynchronous fashion. This means that we needed some sort of backend to allow for the data to be synced between devices, and to keep track of the game states. The easiest and most common solution would be to use Firebase as it offered everything in one service, and would not require any additional logic outside of the client. This would on face value offer a quick turnaround from prototype to an actual working online component. The benefit of quick initial development time did not out way the issues which could be faced later down the road with more complicated user stories which would use the backend. Firebase also had the issue of not having native support for storing arrays, even if it was a JSON backed storage solution. This meant that a one-to-many relationship could have complications in practice, and manually inserting data challenging. Firebase is not a bad solution just not a good fit when more complicated data relationships are implemented.

There are plentiful of BaaS solutions out there which could have also been considered, but that would probably result in more time being spent on exploring their features. It was easier to just build everything from the ground up, instead of outweighing the pros and cons of every solutions available. This also offers the ability to host other media such as images which could be used in certain situations. Ubuntu 16.0.4.1 LTS was chosen due to the stability of the OS and the abundant third party support, which comes in handy when other dependencies can be used easily. The backend has many components working together doing their small part in a request process. Those components include

the web server, app and database. NGINX was chosen as the web server to route the request as it is light weight and simple to configure. Another option would be Apache which has more features than NGINX, but are not needed. Apache works great in instances where PHP is used which isn't the case here.

The request are processed in a Node.js application, as it is very simple and there are already libraries which help with path parsing. Its a lightweight solutions which offers scalability through pm2, a process manager which runs the application instance. This allows for the option to scale up with more instances, with automatic load balancing if needed. The applications uses MySQL as data storage as it is one of the most stable and supported options. There are many flavours of SQL out there but MySQL became the final options as it was the simplest. It was just easier to work with a database which was known to work and wouldn't cause issues, so the focus on being light weight was ignored in this instance.

There are other options out there for programming languages and environments to use but they would be experimental at best. This meaning that a working product could not be promised in the given time frame. Kotlin and Swift were both two options which could be used as the application language. They both have the benefit of being compiled applications which may have made them faster than Node.js, and additionally offered type safety. This could have been nice in the long run as it would have given the ability to catch simple mistakes at compile time instead of experiencing them later on, making them harder to pinpoint.

The image routing solution could have been done better but the performance increase would be minimal in these small use cases. Currently the images are fetched by calling an endpoint and giving an id of the image. Which in turn express parses and returns a corresponding image from the internal file system. A better way may have been to allow NGINX handle these requests and route them directly into the file system. Which would skip a layer and therefore increase the speed.

Volley

For communication with the server and database we chose to use the network library Volley as aid. More info on volley can be found at:
https://developer.android.com/training/volley/index.html.
Volley was an obvious choice of library since it is very easy to use. It comes with features and support for requesting data as JSON or Images, which is exactly how our information is stored in the database. Volley manages network requests automatically, eliminating any need for AsyncTask. Since volley makes networking in android very easy, we were able to focus on the logic and usage of data retrieved, rather than retrieving data.

It's worth noting the we didn't use the official version of Volley. The offical volley-version had some bugs that made volley to not correctly set headers and content-type of a request as meant to. Instead we used:
https://github.com/mcxiaoke/android-volley, which is a version of volley were these issues have been solved.

Storing information

You are able to shut down the application while logged in, and then start application and still be logged in. This is possible by saving information about the last logged in user in a SharedPreference file. Note that the password is never stored, only username and user id. A users password is only stored in the database after using a SHA1 hash on it. Logging in just compares the stored hash with the SHA1 hash of the password suggestion and gives a response according to the output of the compare.

2.1 Protocol

Communication between the client and server was done with HTTP request following a Rest like model. Parameters were sent as Json in the body of the request to make issues with percent escaping easier. All responses were in turn just Json data with objects corresponding to those requested, or nothing in cases where that was applicable. All Json was in accordance to the RFC 4267 as that was the highest possible standard to be supported by both parties. Using RFC 7159 may have been better as it allows for single value Json objects, which reduces data transfer and parsing time. This could be changed at a later date and implemented easily.

Appendix A is the documentation for all of the possible endpoints on the server side, and the parameters required. These endpoints can then be tested with tools like curl to see what the response will be, or to simulate another client.

3 Testing

We haven't run any automated tests for our application. The reason is that we feel writing various Unit-tests etc is too time-consuming to be a viable option in such a short time-span as this project had. It would take time to properly learn it and find useful ways to use it. Instead we have, during development and implementation of each feature, test it by trying to break it. This may also be seen as time-consuming but given such a small application as ours, we felt it was the proper choice as far as testing goes.

GitInspector and FindBugs

The output files of FindBugs and GitInspector are uploaded to the Github repository and are easily viewed using for example https://htmlpreview.github.io/. In the HTML-file of the Findbugs output, no filter has been used for the R.java file, even though the file name says the contradictory.

4 Appendix A

POST /user/create - Creates a new user.
    username
    password

POST /user/authenticate - Signs user in.
    username
    password

GET /user/id - Returns the user object with the corresponding id.

GET /user/all - Lists all users.

POST /user/id/challenge - Challenges a user.
    context_id
    type

GET /game/id - Returns the game object with the corresponding id.

GET /game/list - Lists all the users games.
    context_id

POST /game/id/accept - Accepts a challenge.
    context_id

POST /game/id/decline - Declines a challenge.
    context_id

GET /game/id/rounds - Returns GameType objects related to the round.

POST /game/id/progress - Reports progress on the round.
    correct

GET /type/id.
    count

GET /image/id - Returns the image with the id.

GET /data/id - Returns an actual image file corresponding to the id.