# Wander Architecture Documentation

Team Wan

June 26, 2017

## Preface

This document is the result of meetings with the customer and discussion within team wan. It documents the decisions made regarding the architecture of the Wander app and server.

## Teams

### Team Wan-App

- Ashton Spina
  - Coding the front-end
- Auke Roorda
  - Clientside SQLite database
  - Connection between application and server
- Lorena Arquero
  - Research
  - Clientside SQLite database
  - Connection between application and server
- Jake Davison
  - GUI design
  - Coding the front-end

### Team Wan-Server

- Joe Jones
  - Database design
  - Google sheet implementation
- Thomas den Hollander
  - Database design
  - Google sheet implementation
- Zamir Amiri
  - Database design

## Technologies In Use

### App-Side

The application is developed in its entirety for now in Java and XML using the Android Studio Software suite. Notable used libraries are GSon for converting to and from the server data and MPAndroidChartLibrary for drawing the graphs.

#### GUI

The GUI design will be created in Adobe Photoshop, as with the mock-up shown above. Our designer's familiarity with this program, as well as its industry reputation, are the primary reasons for this. Assets created in Photoshop will be arranged into a functioning GUI using XML in Android Studio. The nature of certain design aspects will require implementation in Java due to the way Java and XML source interacts in Android, especially when implementing design for the games themselves. GUI for the main menu and game can be seen in Figures 1 and 2. For the charts displayed by the app's feedback feature, the MPAndroid Charts library was used. Many charting libraries exist for Java, but Android's incompatibility with Swing elements called for an Android specific library. MPAndroid stood out for it's modular system of objects to pass data objects to graphs defined in the GUI, something that connected perfectly with our modular approach to DB-GUI communication.

MainMenu Example




Figure 1: An example of a vertical display for the main menu.

**In Game Example**

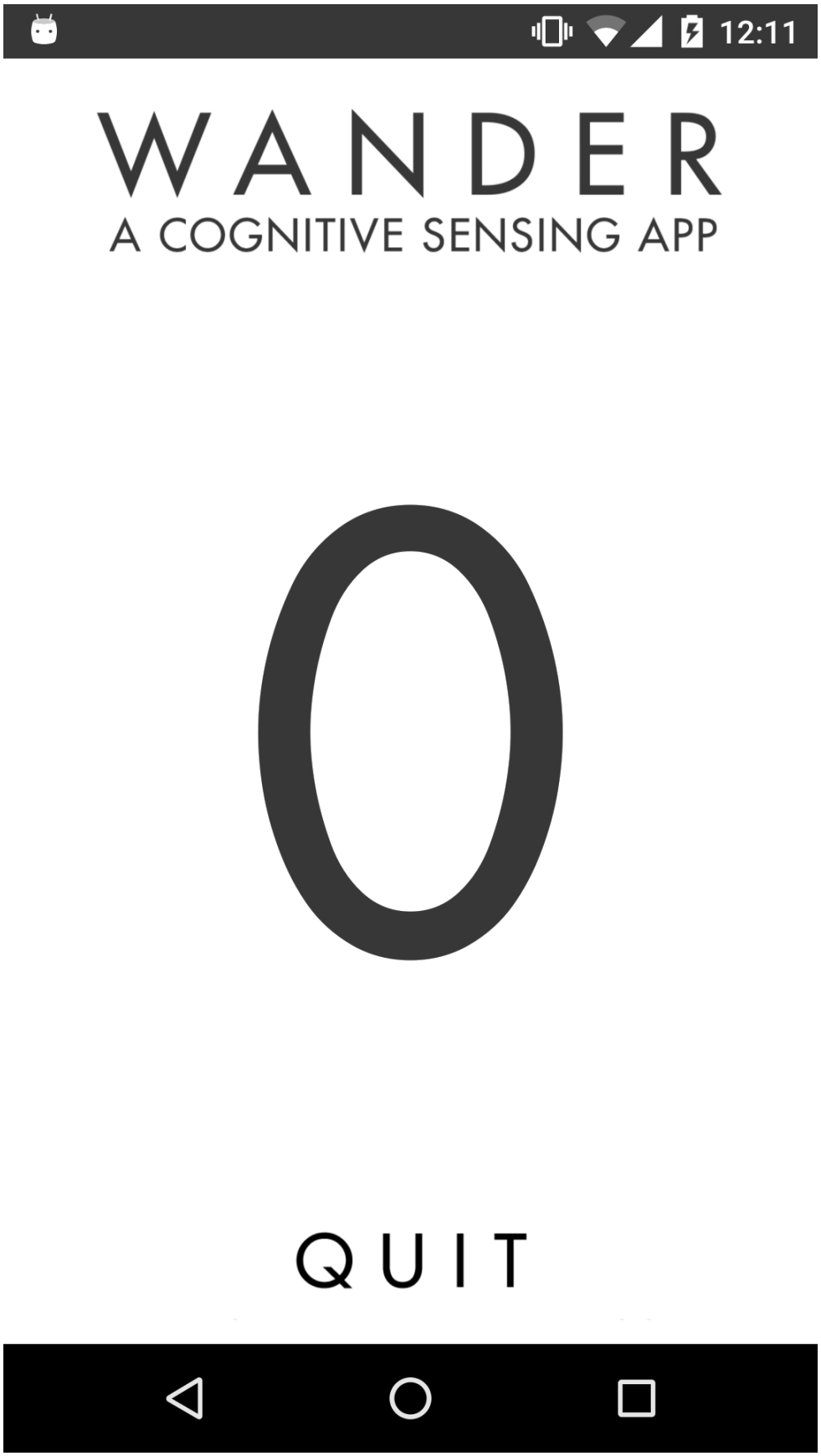




Figure 2: An example of an in game moment

### Local Database

We've decided to use SQLite for storing data on the phone for its simplicity and ubiquity. The database stores all of the relevant data measured during a game session. It is possible to retrieve all game sessions after a certain time, which is needed to upload every game session dataset.
The local database is designed in ORM. This is useful because it means it can easily be expanded and we always have a reference. It will be stored using a SQL database.

SQLite is embedded into every Android device. Using an SQLite database in Android does not require a setup procedure or administration of the database. You only have to define the SQL statements for creating and updating the database. Afterwards the database is automatically managed for you by the Android platform.
The android.database.sqlite package contains the specific classes of SQLite.

Once the data has been successfully stored, we use Androids BroadcastReciever to start uploading when there is a connection with WiFi. If there is, the data is collected from the local database and send to the Google sheet.

## Google sheet

### Setup

The data from the local database is sent to a Google sheet. The Google sheet allows scripts written in Google Apps Script which is a JavaScript-like cloud scripting language. An API URL is used, which can accept parameters to make queries as well as returning information. The Google sheet provides all of the data in a format that is usable for the customer and accessible from anywhere. Is is also more secure as we can choose who has permission to access and change the data. We have created the same tables from the local database as different sheets in the Google sheet.

# Architecture

## Application Architecture

The Application is currently composed of a main activity which is the main menu and another activity which is the game. Generally, the code of the main menu does the bulk of the interacting with other parts of the project. For example, things like notifications are initiated in the main menu and when results from games are stored to SQLite eventually it will be main menu code that takes that action through another method or class. There is also an informed consent agreement when the app is started which asks for permission to share the user's data. The confirmation of save is stored in the application's shared preferences.

### Game

The game simply randomly generates numbers within a range and displays them. One value is disallowed to be pressed while the others should be pressed. This is done by having a Handler check if a press has occurred after a delay and then change the number, giving the game a natural feel. All these values like delay, unclickable number, range of values, etc. are variables that can be modified to suit the client's preferences using a xml config file named "gameValues.xml". It is intended that eventually the game will be within a fragment so that it can be modular and other parts of the game can be swapped in for that fragment. Game performance is stored and the player will eventually be returned to the main menu storage in the local database for future use. There are also questions asked in groups throughout the game. This data is also saved and sent. The config for these intervals of these questions is not yet in the config file, but it will eventually be as well. The Handler is paused and restarted each time a question is asked to stop the game from running in the background

### Datastorage model

We've used a ORM model to design the database. The ORM model can be seen in Figure 3. This model can then be used to create the database tables as described in Figure 4. This system can be used in both the app and the Google sheet. Using the same model will also mean less inconsistencies between the two. For this model we require four tables:

- GameSession
  In this table the player id is stored alongside of which game was played at what certain point of time. Together this three types of data create a unique game session which will be represented as game session id in the other tables.

- NumberGuess
  This table stores the game session data that the client want. That includes response time of the player, whether the action done by the player was correct or not, the game session id and the time and number that was tapped on. The data of this table will also be represented by a unique id in other tables.

- QuestionAnswer
  In this table one can find the answers given by the player to the questions that were asked during the game session and at what time the answer was given.

- Question
  The question table holds all the questions that can be asked to a player. The first column $QuestionId$ represents a unique id that every question has. $Start$ is a column with boolean value. It represents the start of a sequence of questions. If a question does not have a sequence then the start value for that question will also be `TRUE`.The column next question will have the id of the next question that is asked directly after this question. The next column holds the actual question that will be asked to the player. The type of the questions asked can vary and therefore another column is needed to store the type of the question. If the question is a multiple choice question (MC) then the answer options should be stated in the next column under $MC-options$.

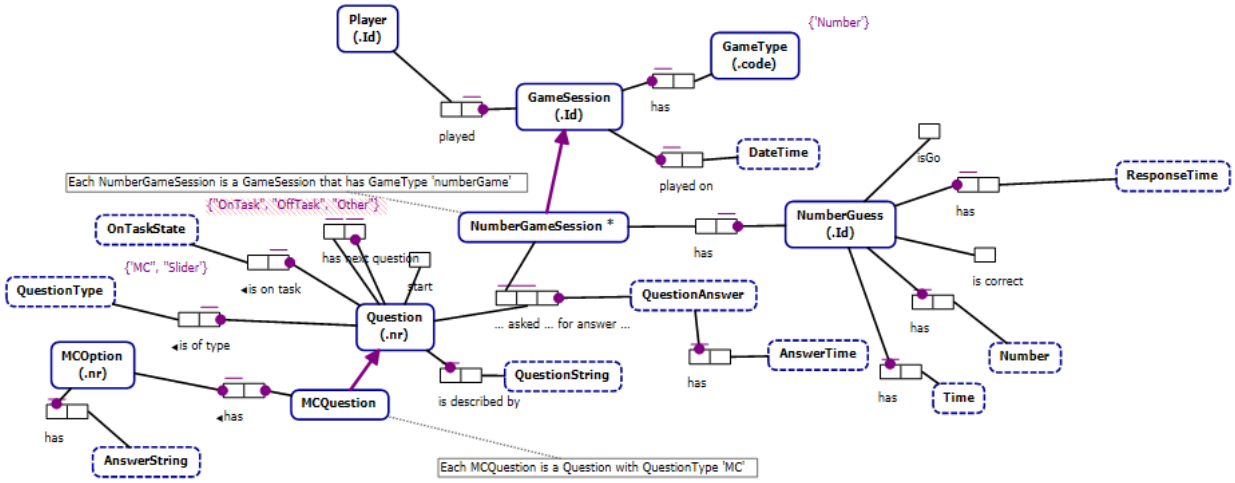


Figure 3: The ORM model for the database.

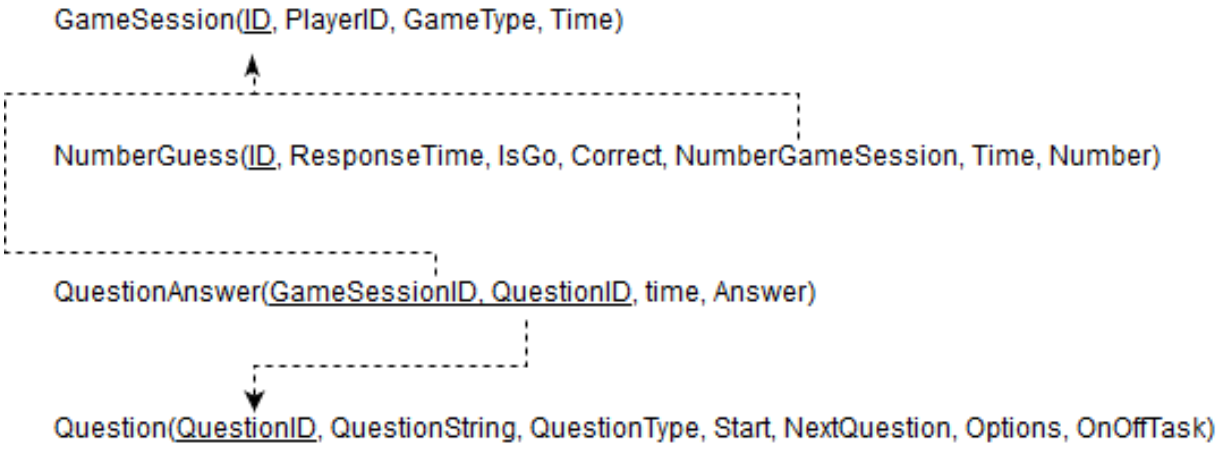


Figure 4: The tables for the database.

### Communication with the server

Once the data is stored in the database, the app will look for a connection with the Google sheet in order to send its data. Once a connection is established, the data will be sent. Upon confirmation the app will pop up a little message saying that the data has been sent. Since the data is not time-sensitive, the app will try to wait for a convenient time to send the data over wifi.
To identify the user we will store a Universally Unique ID, which is generated on the phone using Java's UUID generator. This ID will be sent with every dataset so user data on the Google sheet can be mapped to different users while the users remain anonymous.
We will consider using Android's sync adapter framework to send the data to the web server. This framework helps manage and automate data transfers, and coordinates synchronization operations across different apps. When you use this framework, you can take advantage of several features:

1. Plug-in architecture
2. Automated network checking. The system only runs your data transfer when the device has network connectivity.
3. Improved battery performance.
4. Automated execution. Allows you to automate data transfer based on a variety of criteria, including data changes, elapsed time, or time of day. In addition, the system adds transfers that are unable to run to a queue, and runs them when possible.

Our database will store data which includes time of day, which game the user is playing, score and reaction times for the game, answers to questions and a unique id to identify the player.

## Google sheet Architecture

### Data Visualization

The data is supplied in a raw format allowing the customer to process this conveniently. This requirement is automatically satisfied by the usage of Google Sheets.

### API

The Wander API is created using the Google Apps Scripting system.

To receive the questions from the database, the following URL can be used: here. When a `GET` request is send to it a JSON string, formatted as seen in figure 5, is returned. Parsing this string will give the questions to be asked within the game, although this feature isn't yet implemented and the questions are currently hard coded. Questions come in two types, 'Slider' and 'MC'. In the first case, the answer is chosen with a slider from a continuous range of values. In the second case, multiple choice options are supplied of which one can be chosen.

To send a query to the database system, the same URL is used with a `POST` request.

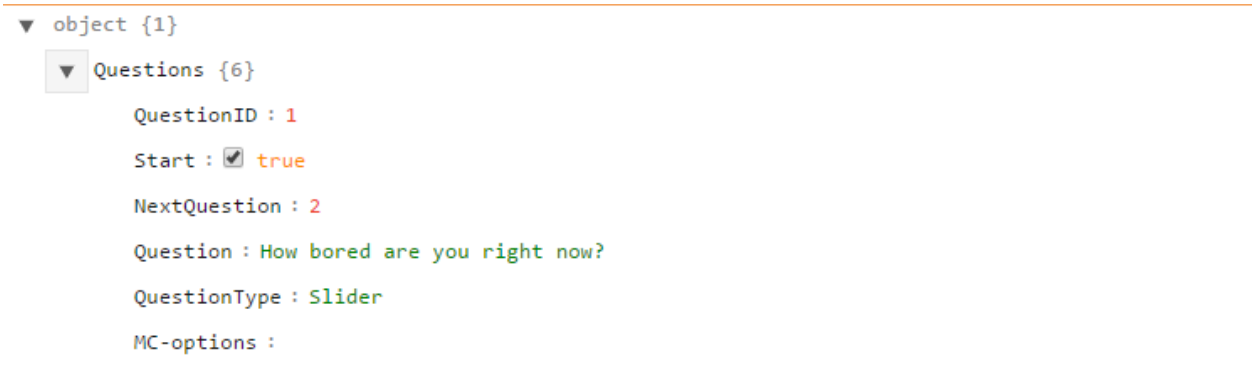


Figure 5: A example question of the JSON format accepted in a query.

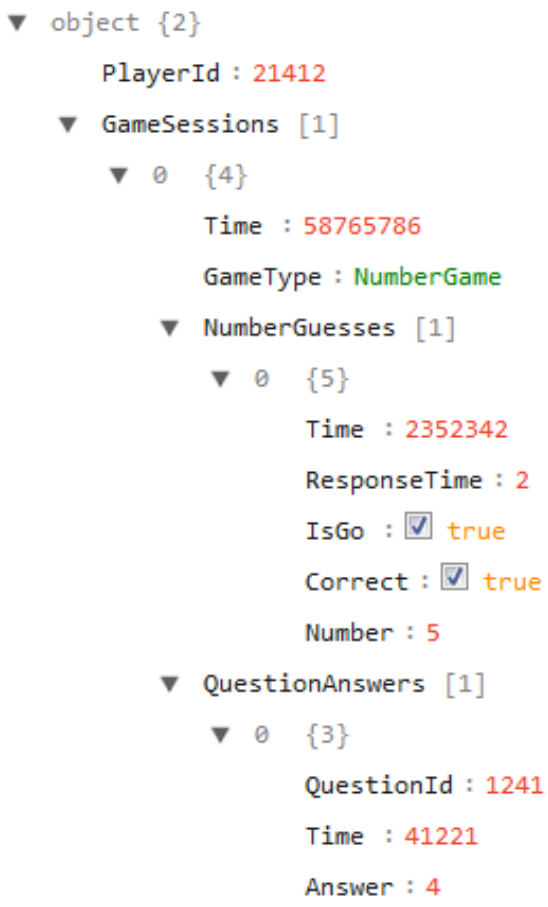


Figure 6: An example of the JSON format accepted in a query.

It should be supplied with one parameter; *data*, which is a JSON formatted object that contains the data to be stored.
Figure 6 shows an example of this data object.

# Modification

Click Here to Modify this Document on Overleaf
Upload the modified document over the old one on Basecamp.