VERSION 1.0

MARCH 9, 2017

TECHNICAL SPECIFICATION

ANSWER BOX

PRESENTED BY: CATHAL CONROY AND CHRIS DURNING

CA326
DUBLIN CITY UNIVERSITY

CONTENTS

1. INTRODUCTION

1.1 OVERVIEW

We have developed an application on the Android platform which allows secondary school students to share their solutions to past exam paper questions. They can judge and critique others for their solutions, while simultaneously receiving feedback on their own submissions.

To use the app, students must first create an account, with which they can then sign in and freely browse the applications content. Students can vote solutions up or down, and gain reputation as a result. This reputation can be viewed as an indicator of the user's contribution to the platform. Students may also comment on other solutions, allowing them to share their opinions and suggest improvements.

Although our application relies heavily on an active internet connection, students can save any solution they want for offline use with a single click. Of course, most features available to the user will be disabled until internet access is restored.

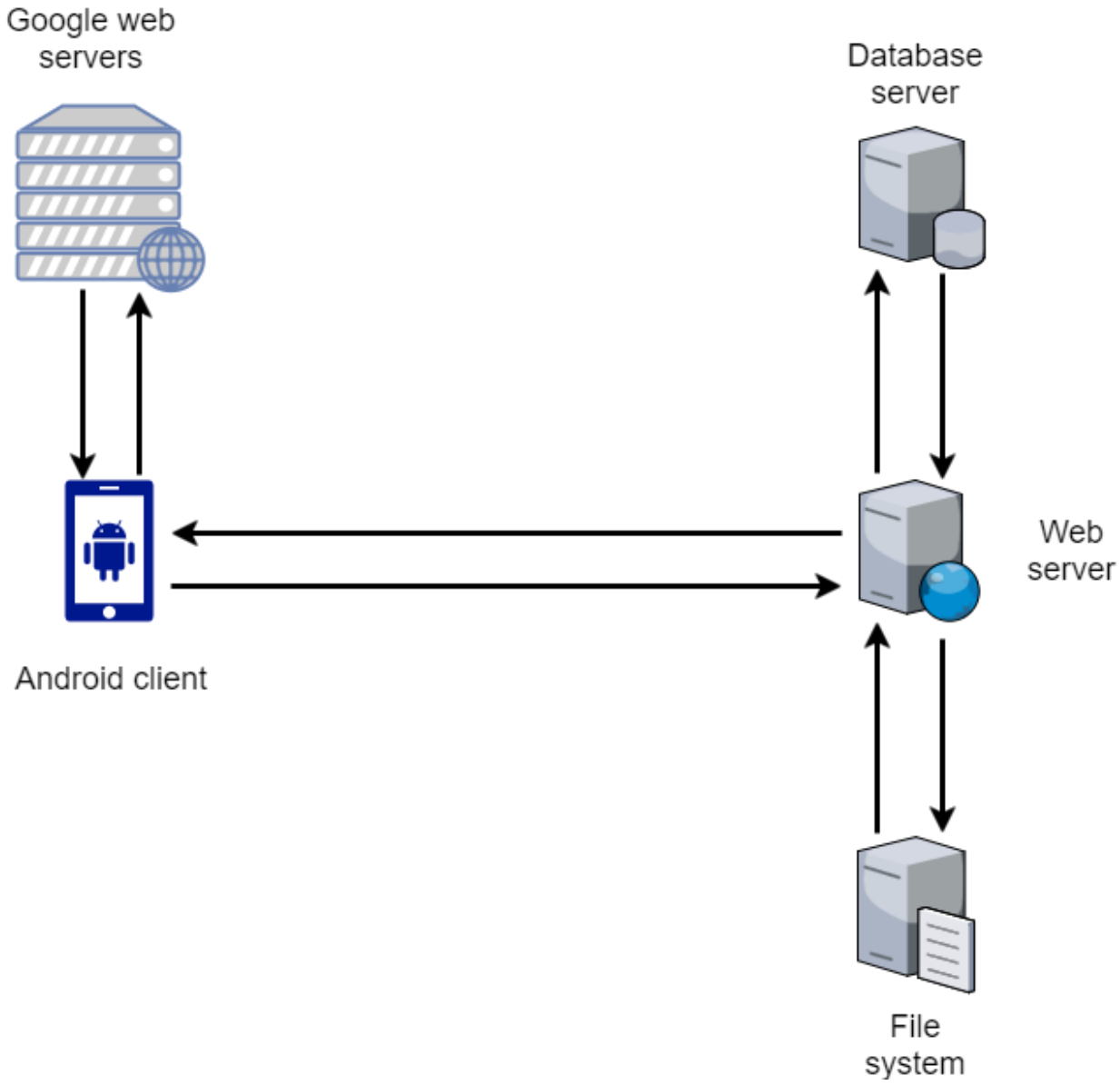
Our application communicates with one external system - our remote server. This is essentially a Linux web server running a MySQL database, with an API developed in PHP which encapsulates the underlying database and file management system.

2.2 GLOSSARY

SQLite3 - Self-contained, server-less, zero-configuration, transactional SQL database engine MySQL - Open-source relational database management system

MySQL - Open-source relational database management system

2. SYSTEM ARCHITECTURE



Our system architecture has (for the most part) stayed true to the initial design as described in our function specification many months ago, as it has throughout our entire development process.

CLIENT

The client is an Android application running on the end user's smartphone. The physical type of device is irrelevant, as the underlying framework remains the same. Our app is targeted at Android API level 25 (7.1 - Nougat) however we have taken steps to ensure our app is backwards-compatible and will work on Android API level 19+ (4.4 - KitKat).

WEB SERVER

During the running of the application, the client will communicate with our remote server via HTTP requests. These requests will be received by a local web server (Apache v2.4.7) and handled by an API written in PHP (v7.0.16). The web server will in turn communicate with the database server and file storage as appropriate. All responses to the client will again be channelled through the web server and sent back to the client over HTTP.

Before each API call is made however, a ping is sent to Google's web servers to ensure the client has an active internet connection. Although our server has maintained an uptime of 100% (so far), no service is as fast and reliable as Google. Once the client receives a response from Google, it will continue and make the API call to our server.

DATABASE

To store personal user data, file names, solutions etc. we decided to use the MySQL relational database management system. We chose to use MySQL as it is free, open-source and it is the environment we were most familiar with, having learned basic SQL syntax in second year.

FILE SYSTEM

Here we are not referring to a file system in the conventional sense (ie Fat32 etc.), but instead referring to the scripts which store and manipulate the image files which have been uploaded by students. These files are stored in a folder accessible only by the web server's user.

3. HIGH-LEVEL DESIGN

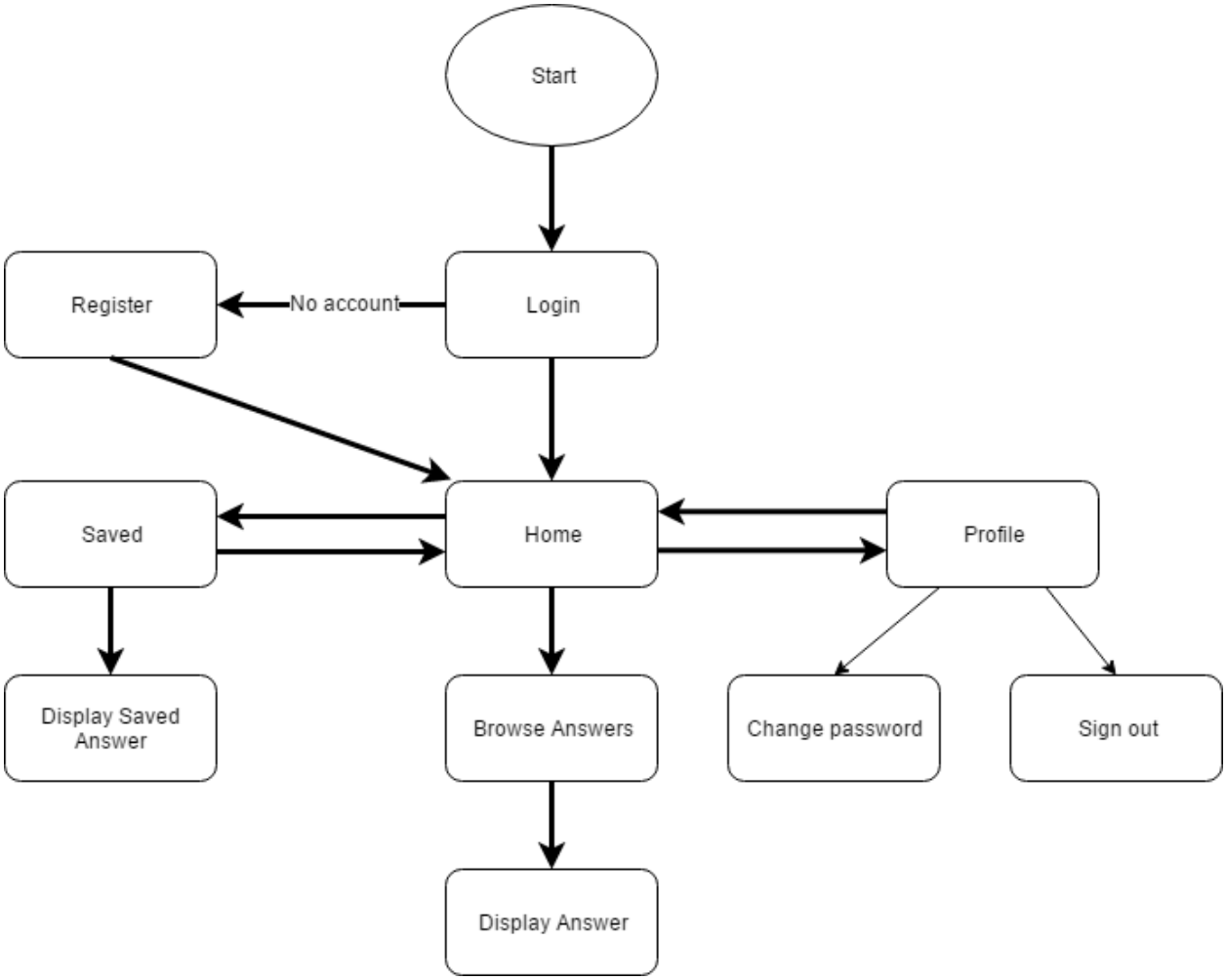
CONTEXT DIAGRAM

This diagram gives a high level view of the system and shows how it interacts with its external entities. Based on our functional spec we have completed all the large functionality tasks that we set out to do from the beginning, and this diagram at this scope displays that.


Admin
moderate system
return query
admin ID
query system
update system
Sample Answer Rating System
search
submit answer
save answer
user ID
change password
delete comment
make comment
vote on answer
report answer
delete answer
search result
User

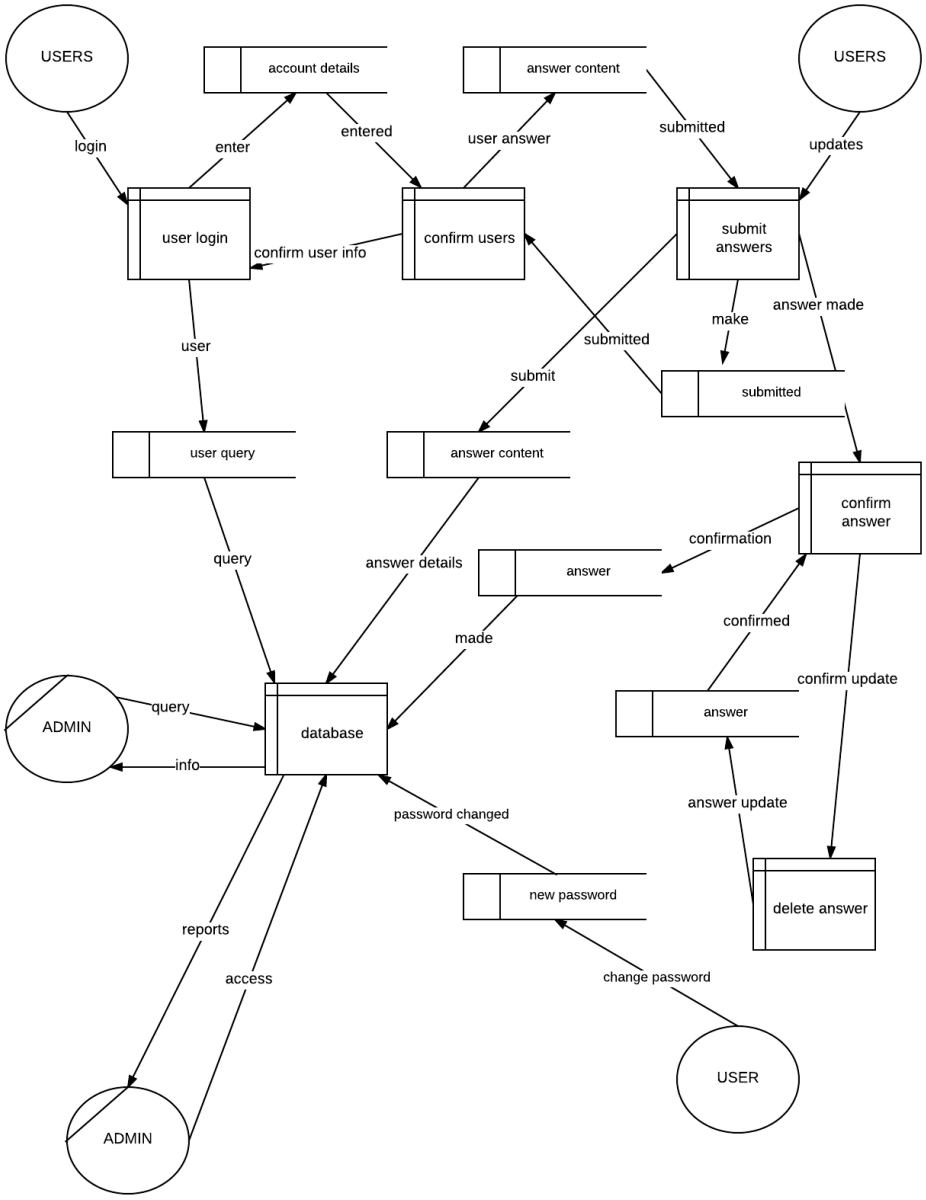
LOGICAL-FLOW DIAGRAM

This logical-flow diagram describes how the flow of our application from a very high level. We can see how the user navigates throughout the app in order to access each feature.



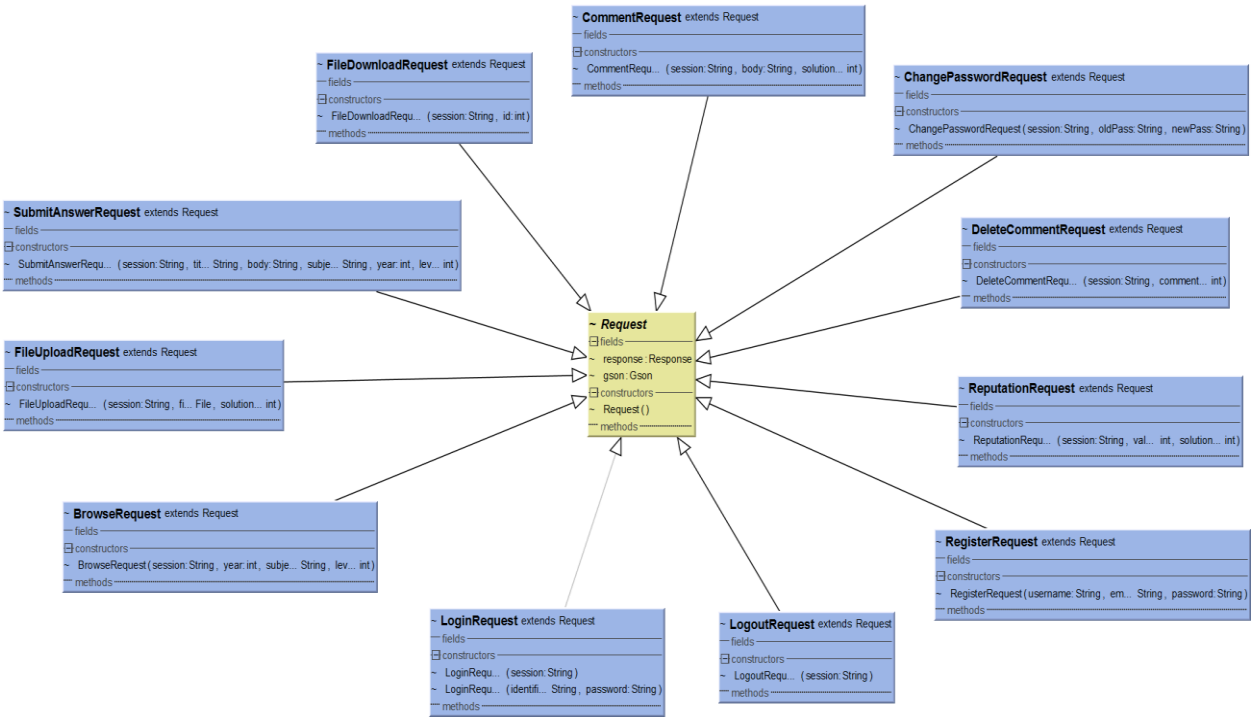
DATA-FLOW DIAGRAM

The data flow diagram is more low level than the context diagram, so you can analyze exactly where the data flow begins and follow it right through until it reaches its end destination. The data flow always begins with the user when they log in either by creating an account or by using an account that has already been created. The data flow always ends at the database on our very own web server which stores and updates data, while giving the admin access to analyze and query the database in order to make updates and identify reported content.

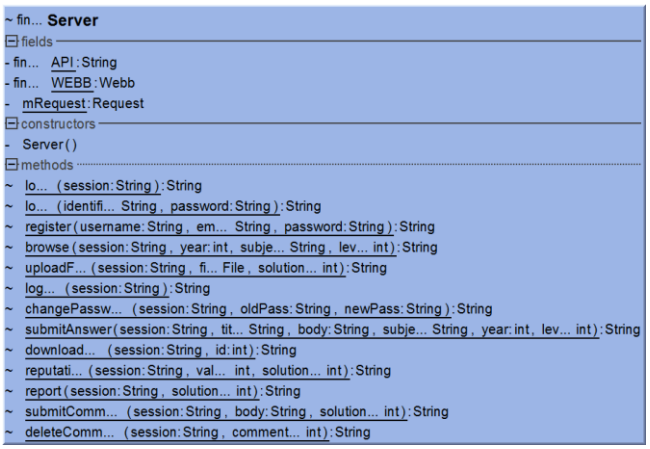


CLIENT-SERVER COMMUNICATION

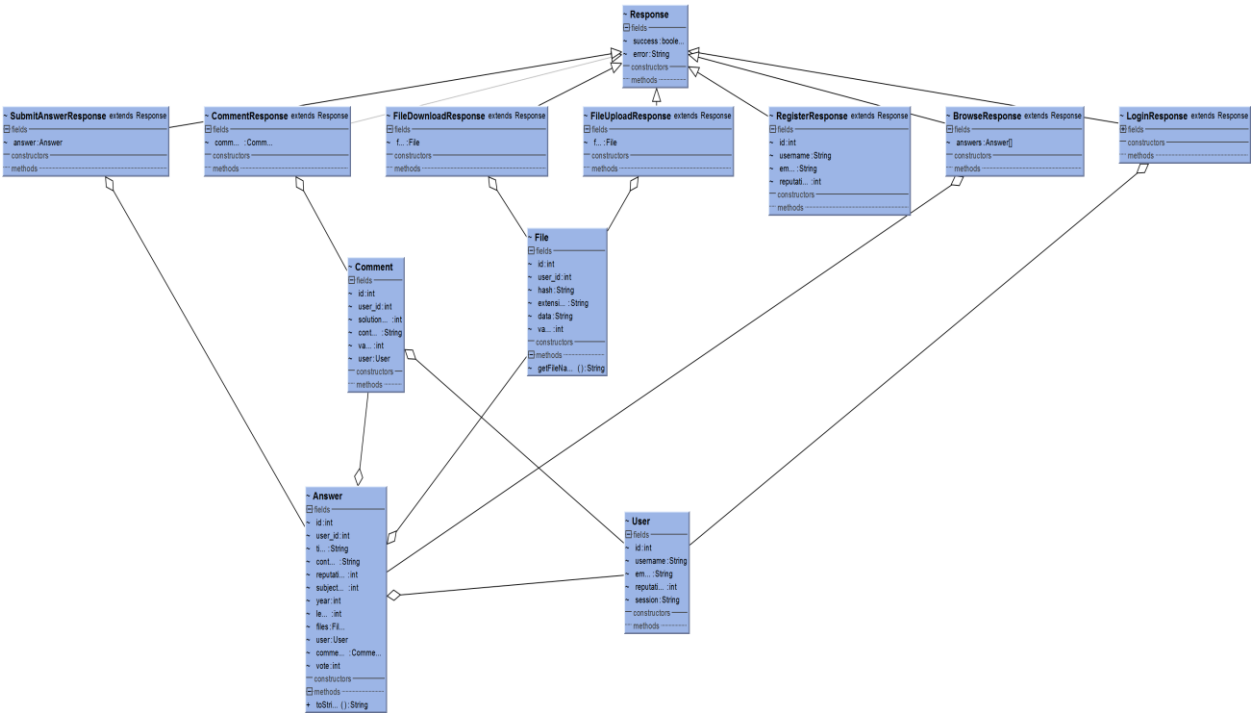
When the client needs to use the API, it makes an API call over HTTP as we have seen before. We decided to encapsulate this "request" in an abstract Request class. To make the call then, we simply instantiate whichever child class of the Request class which suits our needs. Let's look at a specific example. Let's say we want to sign in to the application. Once we gather the user's username and password, we pass these to the LoginRequest constructor, which in turn makes a call to the appropriate static Server method.



Server is a singleton class. It contains only static methods, and should not be instantiated. It is responsible for making the various HTTP requests to the server, returning the JSON response as a String.



We then use Google's GSON library to de-serialize the JSON response into an actual Java object. To accommodate this, we developed a set of Response classes, which represent the data returned by the server. These classes contain information about the request (such as whether the request was successful or not), and of course the data returned.



SAVING ANSWERS

As per our project specification, users must able to save solutions for offline use. To achieve this, we realized we would have to replicate parts of our remote database on the user's device. A full-blown relational database would be overkill in this case, and would cause serious performance issues for a device such as smartphone, therefore we went with SQLite3.

When the user clicks the save button, the solution's information is stored in this local database. Now, whenever the user navigates to the saved answers tab, solution data will be pulled from this local database instead of the remote database.

OFFLINE MODE

When an API call is made but the client fails to receive a ping response from Google's servers, offline mode is activated. This essentially disables any features within the application which require internet access, such as browsing for new answers. The user will still be able to navigate to their saved answers, and access all of the content which they have stored for offline use. Offline mode is turned off when any API call is made and internet access if found to have been restored.

CACHING SYSTEM

While we were developing offline mode, we decided to build a caching system. The problem we identified was that solutions with images were being downloaded every time the solution was opened. With modern smartphones, these images could be easily upwards of 5MB, posing a problem to low end devices, or whenever the internet connection is slow. We figured that if a user is going to download an image once, they may as well keep it stored on their device in case they open that same solution again in the future. Therefore, we modified our DisplayAnswerFragment so that before it attempts to download an image, it first checks the image hash against its local database. If there is a matching entry, we attempt to retrieve it from the local file system.

MANAGING IMAGE FILES

TAKING A PICTURE

When the user attempts to take a photo with their device's camera, we make a request to the device's default camera application and allow it do manage the process. We then retrieve the image data and metadata from the application and process it ourselves.

SELECTING IMAGES FROM THE GALLERY

If a user prefers to select one or more images from their gallery, we make a similar request to the device's default gallery application. We read through this data and store it in an ArrayList for further processing. Android has quite a strict set of security features, and so to allow our application to process these files, we must first request permission and declare this permission throughout the processing.

UPLOADING IMAGES

Performing POST HTTP requests with the Android framework is tricky. Sending binary data is even trickier. To overcome this, we prepare each file for uploading by encoding each image into Base64 byte by byte. We then send the data over to our server as a String, one FileUploadRequest per file.

STORING IMAGES

This process is much the same on both client and server side. Firstly, the received image is decoded from Base64 and compressed into a Bitmap. The server extracts the file's extension type and generates a unique hash, storing both in the files table. The image is then stored in a folder only accessible by the Apache user.

MYSQL DATABASE

The MySQL database used to store data on the server is very much integral to our whole application. It stores everything from our users' personal details, to uploaded image information, to our application content. All the data in our database is stored in first normal form, meaning we have minimal unnecessary data replication.

comments

Column	Type	Null	Default	
id	int(11)	No		
user_id	int(11)	No		
solution_id	int(11)	No		
content	varchar(1000)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	id	54	A	No	

files

Column	Type	Null	Default	
id	int(11)	No		
user_id	int(11)	No		
hash	varchar(32)	No		
extension	varchar(5)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	id	38	A	No	

reports

Column	Type	Null	Default	
id	int(11)	No		
user_id	int(11)	No		
solution_id	int(11)	No		
resolved	tinyint(1)	No	0	

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	id	0	A	No	

solutions

Column	Type	Null	Default	
id	int(11)	No		
user_id	int(11)	No		
title	varchar(40)	No		
content	varchar(10000)	Yes	*NULL*	
reputation	int(11)	No	0	
subject_id	int(11)	No		
year	smallint(4)	No		
level	tinyint(1)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	id	1016	A	No	

solutions_files

Column	Type	Null	Default	
id	int(11)	No		
solution_id	int(11)	No		
file_id	int(11)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	id	34	A	No	

subjects

Column	Type	Null	Default	
id	int(11)	No		
name	varchar(40)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	id	30	A	No	

users

Column	Type	Null	Default	
id	int(11)	No		
username	varchar(20)	No		
email	varchar(60)	No		
password	varchar(60)	No		
reputation	int(11)	No		
session	varchar(32)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	id	17	A	No	
username	BTREE	Yes	No	username	17	A	No	
				email	17	A	No	

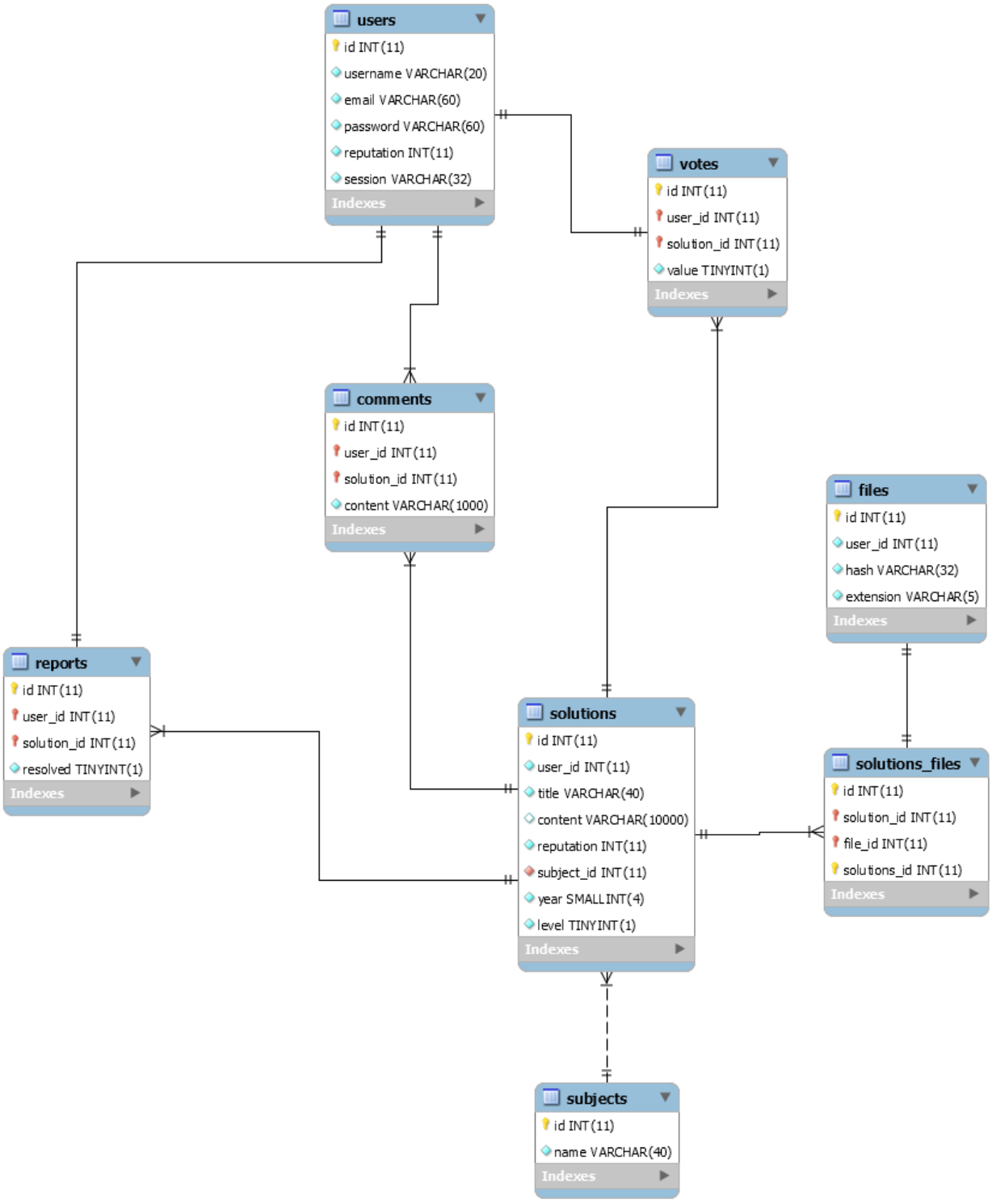
votes

Column	Type	Null	Default	
id	int(11)	No		
user_id	int(11)	No		
solution_id	int(11)	No		
value	tinyint(1)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	id	18	A	No	

ENTITY RELATIONSHIP MODEL



4. PROBLEMS AND RESOLUTIONS

ACTIVITIES VS FRAGMENTS

Arguably the biggest and most painful challenge we faced was using Fragments over Activities. Here's some context to the problem: An Activity represents a single, focused action that the user can perform. Almost all Activities interact with the user, and are presented as full-screen UIs. These are very easy to implement:

```
Intent intent = new Intent(this, MyNextActivity.class);

startActivity(intent);
```

The problem with this is that over-using Activities where they aren't appropriate for simplicity's sake is poor design, and quite simply, an easy way out. If at any stage of your app there is a single UI component which remains on the screen throughout various Activities, then you should be using Fragments. A Fragment is a piece of an application's user interface or behavior that can be placed in an Activity. In its core, it represents an operation or interface that is running within a larger Activity.

Just about every app that exists uses a navigation bar of some part either at the top or the bottom of the screen. You use this navigation bar to jump between different areas of your app, and it remains in-place regardless of where in the app you navigate to.

This might seem like a subtle feature to have for the effort it requires to maintain, however the result is a far snappier and more professional looking app.

IMPLEMENTING FRAGMENTS

The main difficulties we had with fragments was trying to keep track of them. Generally using fragments means using android back stack. A string called a tag is placed onto the stack when a fragment is created, so you can access that instance of that fragment some time again in the future and find it in the state you left it in. For us the android stack wasn't compatible with our bottom navigation bar design so we decided to create our own. We have three stacks, one for every tab on the bottom navigation bar and a different tag for each stack, which is used to identify the fragments associated with that stack. A HashMap was used to map the tags to the stacks.

HTTP REQUESTS

Android's support for HTTP requests is limited. Well, technically it's pretty much unlimited and unrestricted in that Android provides a very powerful, but unfortunately a very low-level API for making HTTP requests, with the HTTPURLConnection class. It works well for GET requests, but is notoriously difficult for dealing with POST requests (and even more so when sending binary data). We stumbled upon a small, lightweight HTTP wrapper client on GitHub developed by David Webb, which enables easier sending of data. This, combined with the Base64 encoding of image files, enabled us to communicate with our server smoothly.

OFFLINE MODE

We encountered several problems when developing our saved answer system. Firstly, we realized that MySQL would not be appropriate in this area, so we had to research an alternative. MySQLite3 was the solution, but it came with its own troubles. We had no real way to administrate the database as it does not provide any form of management system. We could not even access the files on our test device as the file the database is stored in is accessible only with a root user, which we did not have access to. Our solution was to build an emulated device on our computer and manually open a shell as root through the emulator.

Initially, we didn't even plan for an "offline mode", however we quickly realized the problem with that approach when every single API call crashed the app because of lack of internet access. To further the problem, Android doesn't provide any built-in APIs whatsoever to check for an internet connection. Therefore, we came up with the idea of pinging Google's web servers on each API call.

5. INSTALLATION GUIDE

REQUIREMENTS

- Android smartphone, KitKat or higher (Android v4.4+)
- “app_release.apk” file, located in project root folder

INSTALLATION STEPS

1. Enable installation from unknown sources
    a. As our app is not on the Play Store, you need to ensure your device can install apps from unknown sources
    b. Navigate to Settings > Security (and fingerprint) > Unknown sources > ON
2. Download the “app_release.apk” file in our project’s root folder
3. Locate the .apk file on your device and install by opening it

UNINSTALLING THE APP

Navigate to Settings > Apps > Answer Box > Uninstall

6. JAVADOCS

You can find a copy of our generated Javadoc at: http://api.cathal.xyz/javadocs