AQA A Level Computer Science (7517) Non-Exam Assessment

Alvin Lee (7060) Centre Number: 12456

[TownHunt Mobile App]

Table of Contents

AQA A Level Computer Science (7517) Non-Exam Assessment

Alvin Lee (7060) Centre Number: 12456

[Town Hunt Mobile App]

Analysis

Introduction

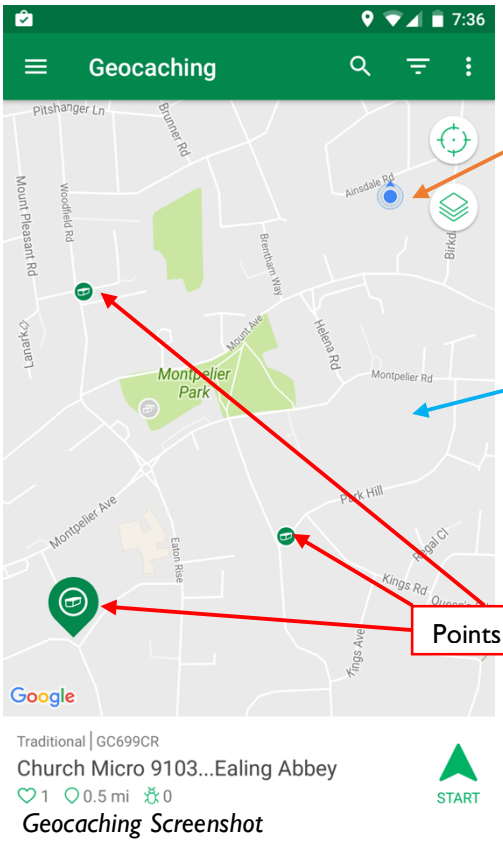
The butterfly graffiti on the sidewalk, the birthplace of a great author and a stone nose hidden in a concrete wall – these are a few quirky locations that I didn't know existed in my local area until recently. People, especially tourists, often fail to notice points of interests in areas that they visit. For my project, I aim to create a digital platform where interesting locations can be shared and discovered in an engaging way. This will come in the form of a mobile application called '*TownHunt*'.

Example of an orienteering map

At its core TownHunt will be a location-based mobile game. In location-based games the gameplay is centred around the player's location. This forces the player to physically move around an area to interact with elements in the game. For example, a treasure hunt where players must use a map to follow clues until they reach the final prize. I have always enjoyed these type of orienteering-esque games. In orienteering, players are given a map and a list of locations which they have to visit in as fast a time as possible. Whereas in scavenger hunts, players are tasked with finding as many items from a given list as possible. Both types of games force users to discover/enter areas that they might not visit normally. This is why, the game play mechanics of TownHunt will be based on scavenger hunts/orienteering.

I didn't want to create one game that is specific to one location therefore the TownHunt app will allow users to create their own scavenger hunt games (known as "Pin Packs" in TownHunt) and upload them to an in-app 'store'. In this store, other users will be able to download these scavenger hunt games and play them on their devices.

Research on Existing Location-based Mobile Games

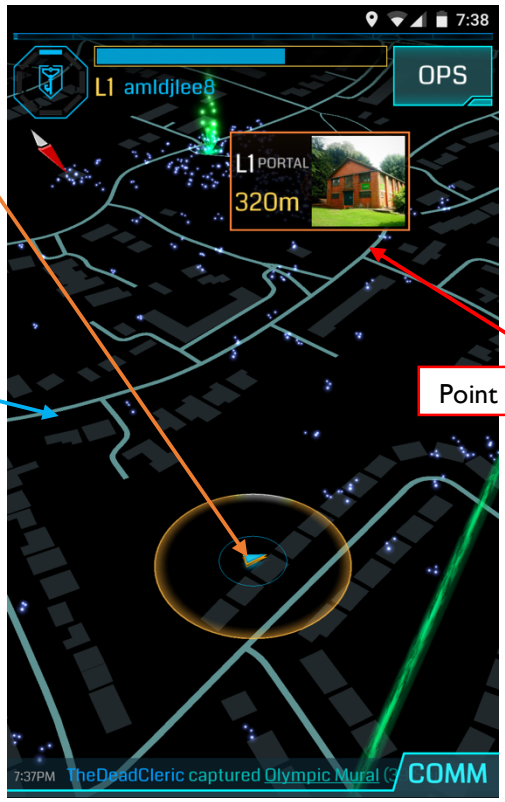


Geocaching Screenshot

User Location From GPS

Map of Area Around User

Points of Interest



Point of Interest

Ingress Screenshot

I downloaded the apps 'Ingress' and 'Geocaching', which are immensely popular location-based apps, on to my phone. I wanted to learn what makes these apps successful. Ingress is an augmented reality game developed by Niantic Labs where players have to capture 'Portals' which are located at points of interest (e.g. statues or underground stations). Geocaching, developed by Groundspeak Inc, however, involves real world logbooks/toys/trackers (caches) that are hidden at landmarks by players. In Ingress, users submit local points of interest to Niantic and, if approved, are added to the game. Users being able to create 'point of interest' objects in the game is similar to what I intend to do with the user created Pin Packs.

Both apps have very user friendly user-interfaces. The map at the forefront of the application. I will do the same. The map should always be the main focus with the 'points of interest' being clear to the user. The user's location is always displayed on screen and a navigation bar with information appears at the top of the screen. This is so users will see the most important information first. I will keep in mind these design aspects when designing TownHunt.

Explaination of 'Pin Packs' and 'Pins'

In TownHunt, 'Pins' will refer to the object the user has to find have to find e.g. the Statue of Winston Churchill. Each 'Pin' (class) will contain details like the title of the object e.g. 'The Great Statue', the hint that will be displayed to users e.g. 'Who is this statue of?', a codeword (answer which the user has to enter into the phone) e.g. 'Winston Churchill', the location of the Pin e.g. 'Parliament Square' and a point value e.g. '75pts'. A collection of 'Pins' will form a 'Pin Pack', e.g. 'Parliament Square Pin Pack' would contain Pins asking users to visit several locations around parliament. These Pin Packs will be played/created by users.

Prospective Users

(Young) Adult Smartphone Owners

The primary target audience are young adults as they are more likely to be looking for new unconventional games to play (especially when travelling/visiting new areas). Although anyone with a smartphone will be able to use this application. This age demographic like to use cleanly designed applications which are easy to use. I also chose this age demographic as my fellow peers at school are of this age category and I can ask them for continual feedback/group testing of TownHunt. Since the user must own a smartphone to run TownHunt, it is presumed that they will be very proficient in navigating smartphone user interfaces.

Mayflower Collection of Hotels

The Mayflower Collection is a chain of three boutique hotels situated in central London. Like most hotels, they want their guests to orientate themselves in the local area to they can make the most of their stay. Scavenger hunts/orienteering have often been implemented as a novel way for people to familiarise themselves with a certain location. For example, the University of Manchester[1] ran a digital scavenger hunt during Orientation week for incoming students to get accustomed to the local area.

After talking to the Juliana Boudville – the Mayflower Collection's CFO – they would be interested in using the TownHunt platform to help make their guests accustomed to the local area. Each of the three hotels could create their own 'Pin Pack' and ask the guests to download the app and play the relevant 'Pin Pack'. Guests could also play the 'Pin Pack' of an area they are visiting in London if one was available.

Staff already use digital systems on both phones and computers therefore computer experience is presumed.

Interviews with Prospective Users

I carried out two initial interviews to assess the opinions of the two main users of the application. I explained the initial concept of the app and gained feedback. From this I will be able to learn what features to implement in the app.

Interview One - Jacqueline Chua (University Student)

Jacqueline is a 24 year old adult who has lived in London for the last 3 years.
Q) Do you feel as though you have explored your local area for points of interest?

A. To be honest I could probably tell you more about areas which I visited on holiday more than my own local area. I've seen the major points of interest but I feel as though I'm missing out on local 'hidden gems'. When you walk around your own area you tend to miss out the details. The majority of my friends feel the same way. The interesting features blur into the background.

Q) Would you be interested in using Town Hunt ?

A. Yes. The application you are describing sounds like a fun way of not only being able to discover your own area but also to investigate new areas. I could also see this being a nice game to play competitively with a group of friends all playing a 'Pin Pack' together at the same time.

Q) Have you played any scavenger hunt/orienteering games before?

A. Yes. I've played a game called 'Scavengers of the Lost Ark' at the British Museum. This is where a several teams of 3-5 people had one hour to go around the museum searching for exhibits. We were given a British Museum Map and brief set of rules. On our phones, we had to go to a website and then enter codes relating to the exhibit. If the code was correct, we would gain points for our team. Different artefacts had different point values corresponding to how difficult it was to find the right answer. At the end, the team with the most points won a prize.

Q) What worked in that game?

A. The atmosphere of doing a scavenger hunt in such a public museum was a lot of fun. The fact that I was with a group of friends made it even more interesting. Through finding the codes in the exhibits, I learnt more about the exhibits. This is because to get the right code you must really scrutinise the exhibit for clues. That is why I really liked the puzzles there, they were all really fascinating.

Q) What didn't work well in that game?

A. We found it annoying to have to use both our phones and a British Library Map. It would have been nice to have the map also on our phones. The actual system they use for displaying the hints was quite clunky and unorganised.

Q) What features would you want in Town Hunt?

A. Some sort of competitive element to it. For example, a leaderboard for each 'Pin Pack'. It would be interesting to have some statistics about your own account like the number of pack you have played for example. Also, the locations you must find on the map should be very visible, this makes it more enjoyable to play.

Interview Two - Juliana Boudville (CFO of the Mayflower Collection)

This interview conducted at 20 Nevern Square, Kensington, London, SW5 9PD - the flagship hotel. Juliana has a managerial position at the Mayflower Collection.

Q) What do you do now to highlight local facilities around the hotel and orientate your guests? What are the benefits of your current system?

A. When guests check-in at the reception, the receptionist gives them a Transport For London (TFL) local area map. The receptionist would then mark on general areas of interest (copy of a map can be seen on page 157). As for benefits, apart from the receptionist, other staff members are not required and the TFL maps are free.

Q) What are the negatives of the current system?

A. The TFL maps lack a lot of detail and to the receptionist has to physically mark on points of interest on each new map. This is potentially time costly as the receptionist could be doing other tasks. These maps are only updated annually or biannually and in the ever-changing London landscape, points of interests can change very quickly

Q) Would you be interested in creating your own "Pin Pack"?

A. Yes, it sounds like an idea that we would be interested in doing. We could create tailored 'Pin Packs' for each hotel. I could see guests playing our pack to get oriented around the hotel area.

Q) What features would you like to see in Town Hunt?

A. There should be an option for guests to see all of the POIs displayed on the map so that they can refer to the application in the future. The screen for creating the 'Pin Packs' should be easy to use as we don't want staff to be confused when tasked to update/create the packs. This will ensure that this new system will also have low maintenance benefits.

Identification of users' needs

1. The app should provide a map environment to play the Pin Packs in either a competitive way (under a time limit) or a casual way (no time limit)
2. The app should provide a map environment where users can easily create new Pin Packs
3. The app should be able to store packs locally on the device
4. The app should upload Pin Packs to an online database
5. The app should provide a way to download Pin Packs from the database (a store)
6. The app should have a leaderboard for each Pin Pack, which stores scores that people achieve.

Proposed Solution

The TownHunt app will have four main sections:

1. Main game - The main screen will be a map with the user's location. A pack can be selected and played either competitively (under a time limit i.e. 'Game Time') or casually (no time limit). If competitive mode was chosen - over time, several 'Pins' (each Pin represents a point of interest e.g. a local statue) will be displayed on a digital map in a random sequence. In casual mode, all of the 'Pins' will be displayed at once. Each 'Pin' will pose a question to the player which can only be answered by visiting and identifying information on the landmark. If the user enters the correct answer the user scores points. The final score is then sent to the online leaderboard database to be stored if this play through was the first time the user has played that Pack. The leaderboard data, for scores obtained in competitive mode, can be accessed through the app and viewed.
2. Pack creation – Any player or organisation will be able to create Pin Packs. The main interface will be a map. Users can then add their own Pins with information. The packs created will automatically be uploaded to the online database as well as local device storage.
3. Store (to download packs) – This is where users will be able to search the online database for packs. If they find a pack they want to download, the app will download all of the details

of the Pack from the online database and store the Pack information into the phone's local storage. There will also be a screen where users can edit what packs are on the phone.

4. Account Information page – Users will be able to see some basic statistics of their account (e.g. total number of packs created/played). The user will also be able to logout and login to other accounts found on the database.

The app will connect to an online SQL database via a PHP API interface.

I originally chose to create TownHunt as a mobile app as the majority of the population (especially in the UK) own smartphones so creating apps means TownHunt has the potential to reach as many users as possible. Smartphones are also portable and since this is a location-based game where users have to move a lot – a smartphone app is the best medium.

Since this is a smartphone application, I had two options to produce an app for either Apple iOS or Android. I've had experience in both but due to time constraints I could only choose one. Both have excellent support communities although the Android documentation is more detailed than iOS. I decided to build an iOS application as inside the IDE (Integrated Development Environment), Xcode, it is easier and quicker to create clean GUIs (GUIs are created via a drag-and-drop interface and are controlled programmatically). Most of my peers and the workers at the hotel have iPhones so they will be able to test the app and provide feedback. This means that the app will be programmed in Apple's proprietary software Swift. This is heavily object-oriented with each screen in the app having an associated class behind it. Swift also comes with Apple's MapKit preinstalled. This will provide a map interface and the ability to add 'Pins' (annotations to the map). Swift also contains native ways to send HTTP requests (used to send data between the app and the database). In the interest of time, the app will only be developed for the 4.7 inch iPhones (iPhone 6, 6S and 7). This allows the GUI to be solely designed for the 1334 × 750 pixel screen resolution.

Server-side, I already have a personal website hosted by 1&1.co.uk. This website comes with a free MySQL database which is the reason why I chose to store the Pin Pack, leaderboard and account data on a MySQL database. For security reasons, the database cannot directly connect to an app, so a server-side bridge/interface must be programmed to enable data transfer between the database and the iOS application. I decided to program this interface with PHP as it is natively supported by 1&1.co.uk, compatible with MySQL queries and I have had some experience with PHP. PHP is also very well documented with multiple IDEs providing ways to program in PHP. For the PHP code development, I chose to use Netbeans as it is free with extensive error detection.

Objectives of the Project

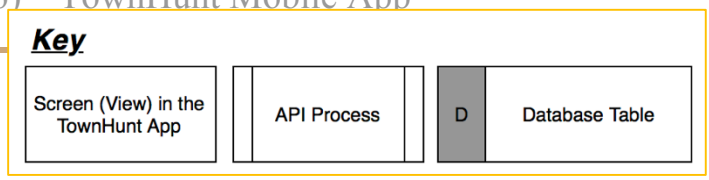
1. Storage Systems
    a. Local Storage
        i. Pin Packs, and associated details, should be able to be stored locally on the device as JSON files in an organised manner. The app should be able to read and write to these files
        ii. Linked lists should be used to store the file access information. These linked lists should be maintained and updated when new files are saved or old files are deleted
    b. Online Storage
        i. MySQL database should be created to store the following TownHunt information: User account details, Pin Pack details, Pin details and leaderboard data (scores scored for the first play-through of a pack). This information should be stored as four interlinked tables.
2. PHP API Bridge
    a. PHP API Bridge must allow data transfer between the App and the database. Data should be both retrieved and inserted into the online database via (multi-table) parameterised SQL queries. Statistics about tables should be retrieved via SQL aggregate functions. Data should be sent back to the app in the form of JSON strings.
3. Account System
    a. Users should be allowed to sign up for an account.
    b. Users should be allowed to login to an account with their login details stored locally on the device.
    c. Statistics about the user -total number of packs played/created and total number of competitive points scored- should be retrieved from the database and presented to the user.
4. Main Game
    a. Users should be able to select a pack from local storage that they want to play. This pack should be then loaded into the game and be playable.
    b. Users should be able to select which game type (competitive or casual) and the game play will change according to the selection.
    c. Users should be able to start and play a scavenger-hunt game.
        i. A map should be displayed. Icons, representing the pins, should be shown on the map. The user's location should also be displayed.
        ii. Users should be able to tap on a pin and enter the codeword. The user should be alerted if the codeword entered was correct or incorrect. If the codeword was correct points should be added to a running score total.
        iii. If casual mode was selected, all the pins in the pack should be added to the map.

iv. If competitive mode was selected then five pins should be initially added to the map, the rest of the pins should be added at random times to the map. A timer counting down should be displayed and the game should be ended when the timer reaches 0.
v. When the game ends, the final scores should be sent to the database and appended to the leaderboard table if it this was the first play-through of the pack.

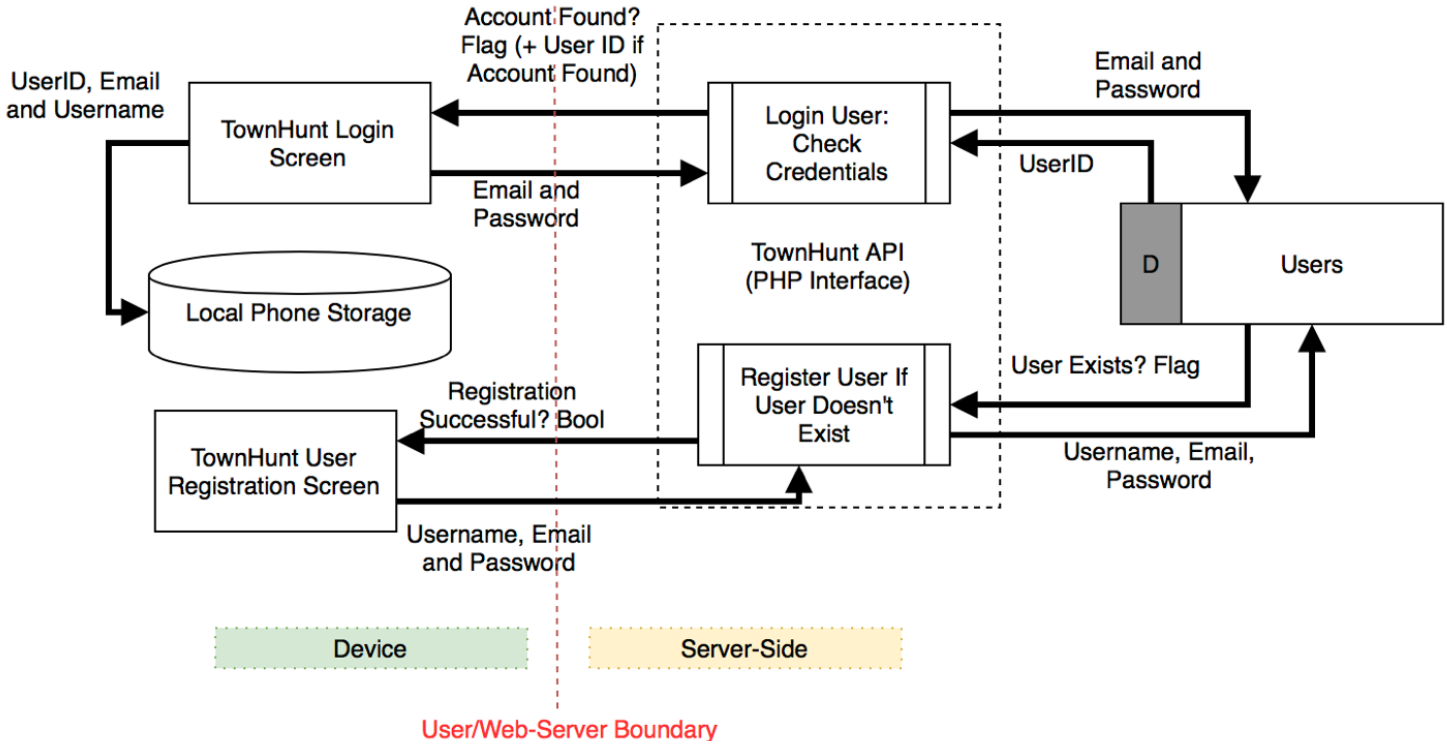
d. A pack-specific leaderboard should be accessible and display the top 10 scores for the pack as well as the score and ranking of the logged in user (if they have played the pack and aren't the creator).
e. The user should be able to zoom into their location on the map.

5. Pack Creator
    a. Users should be able to create their own pin packs.
        i. A map should be displayed. Icons, representing the pins, should be shown on the map. The user's location should also be displayed.
        ii. Users should be allowed to add/delete pins (and respective details) to/from their packs.
        iii. The user should be able to save their packs. These packs should then be uploaded the online database as well as stored locally.
    b. The user should be able to zoom into their location on the map.
6. Pack Store
    a. Users should be able to search for packs (based on the pack's name, the pack's location and or the user name of the pack's creator) stored in the online database. A results list of packs should be presented to the user.
    b. The user should be able to select a pack and download it to the phone. This pack should be stored as a JSON file.
    c. Users should be able to edit and delete pack files stored locally on the phone.

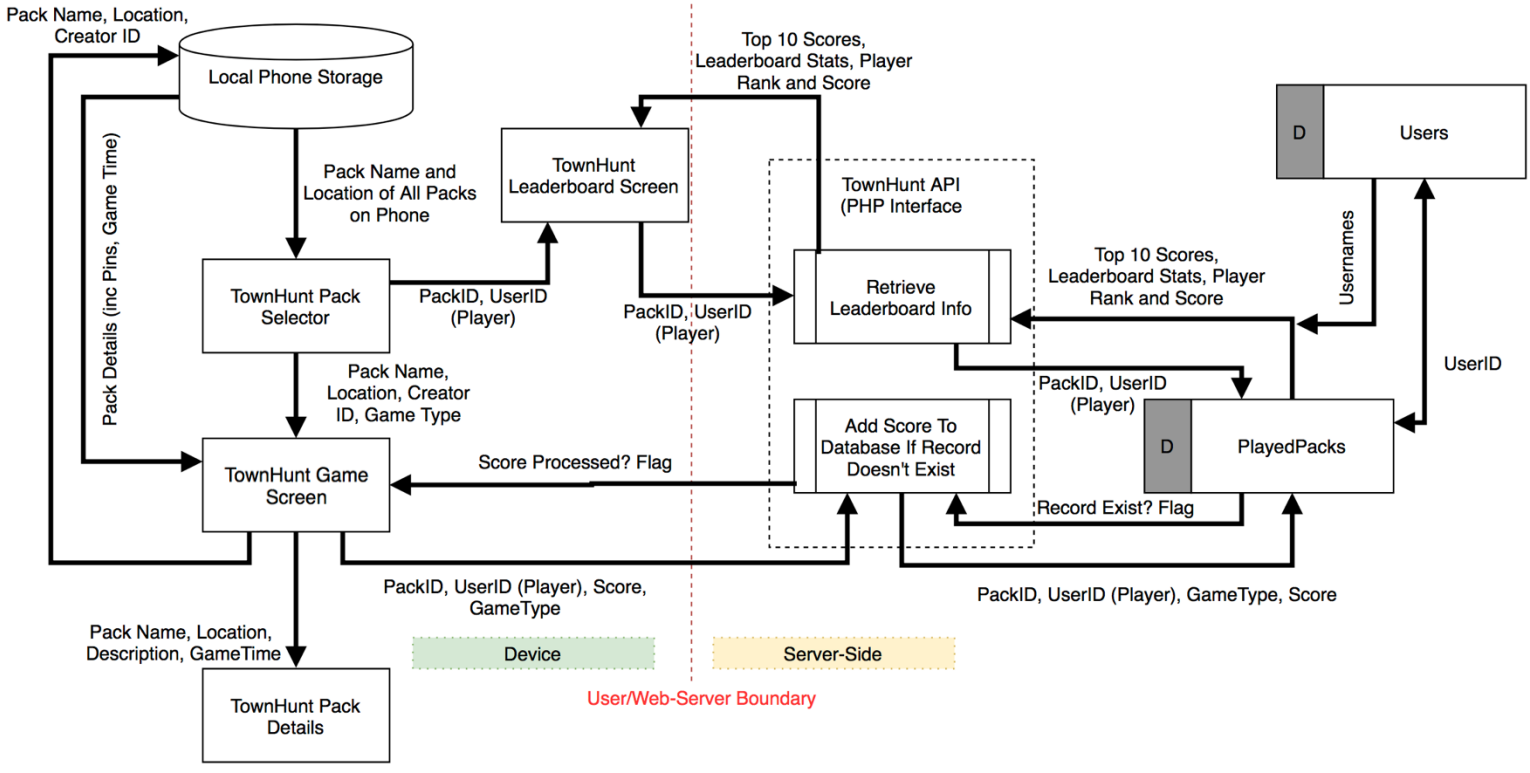
Data Flow Diagrams



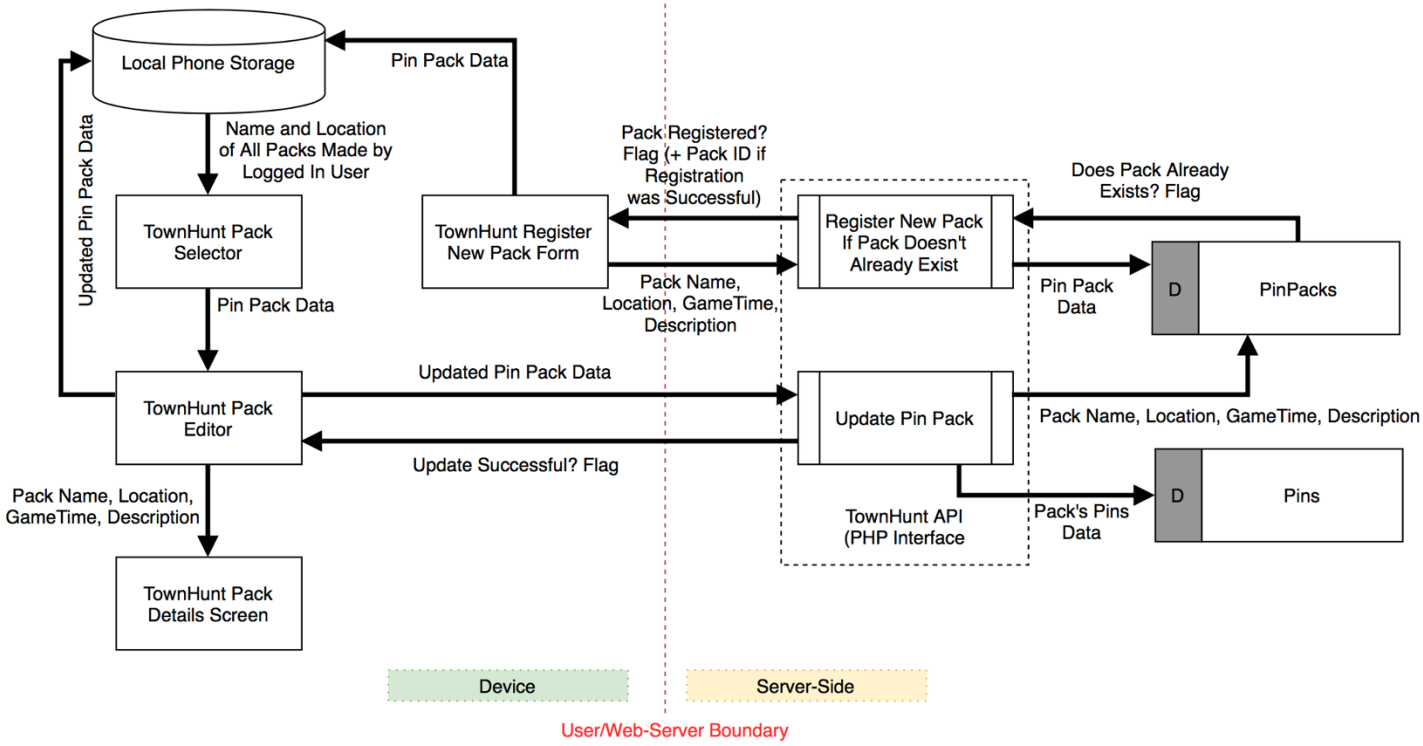
Account Login/Registration System



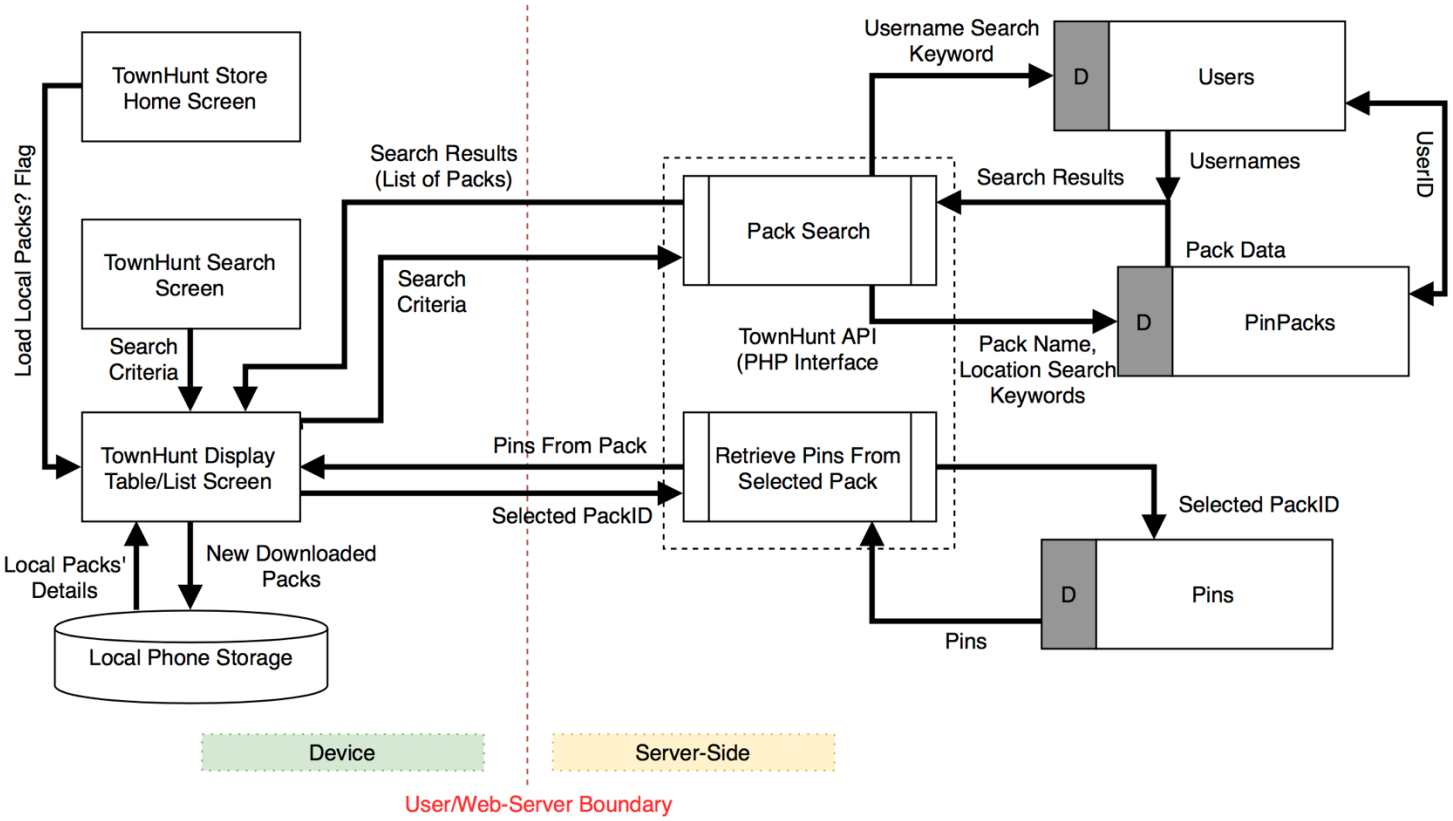
Main Game Screen/Pack Selector/Leaderboard System



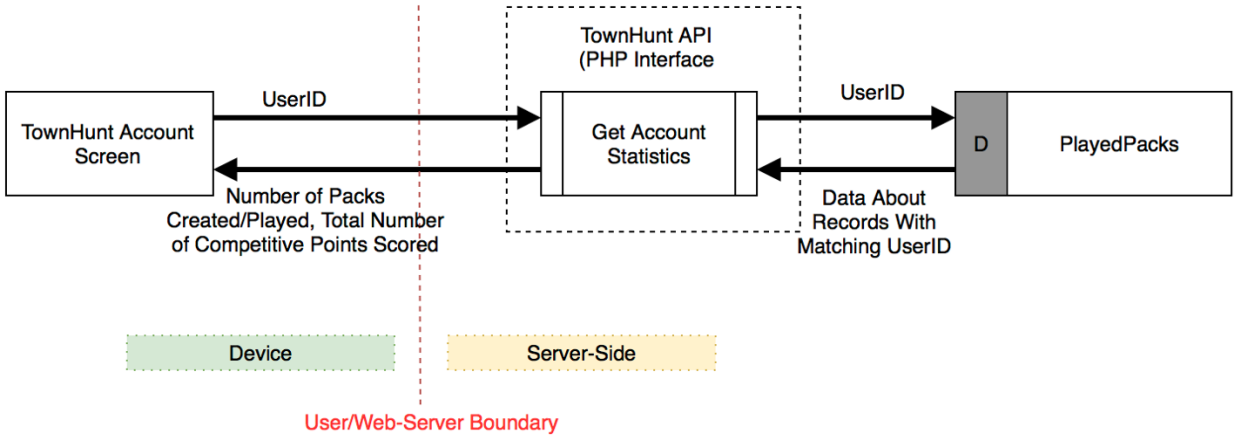
Pack Creation/Uploading Pack to Database System



Pack Search/Downloading Pack from Database System

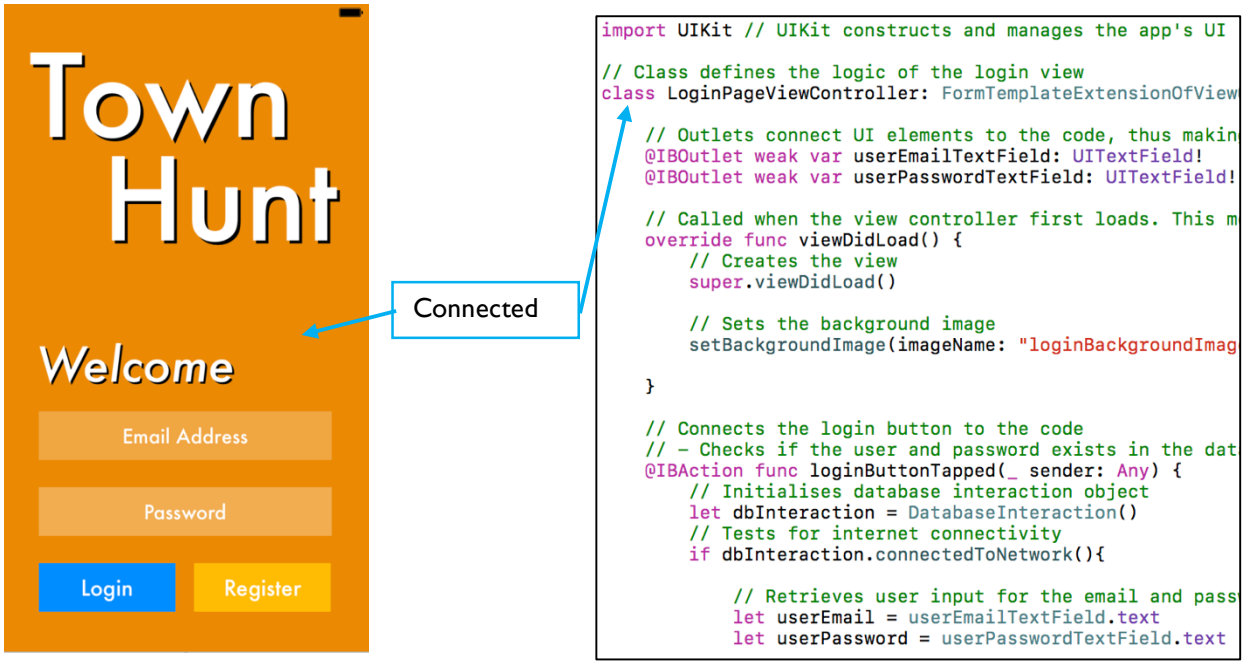


Account Statistics System



Documented Design

How Swift iOS Apps are Designed



The Login Page View

The Login Page View Controller

Swift is screen/view based. This means that you usually create the view (the user interface) first then write the code (view controller) that controls the logic of the view. A view is an object that is created on the screen which can change display after receiving touch events. For example here the Login button is connected to the function 'loginButtonTapped'. This function will execute once the login button on the view is tapped. In Swift, every view in controlled by a ViewController class. In my application every screen will have its own class.

Abstracted Overview of TownHunt App System

Start (Loading) Screen

- Login Screen
    - Loads UI (Username & Password entry boxes + Buttons)
    - Login Button
        - Entered user details are sent to the database to check if an account is found. If account is found then the account details are stored locally.
    - Register Button
        - Registration form appears where users can sign up for an account.
- Game Screen
    - Loads UI (map + buttons)
        - Checks if this is the first launch of the app.
            - If first launch, the linked lists are initialised
        - Get user location and display on map
    - Change map type button
        - If tapped, the map type changes from satellite hybrid to plain map and vice versa
    - Select map button
        - When tapped the app displays a pack selector screen. User selects pack and game type. Pack is loaded from local storage.
        - Leaderboard button
            - Gets leaderboard data for the selected pack and displays it in a table
    - Start game button (starts the game)
        - When tapped the app checks that a pack has been loaded. If a pack has been loaded the gameplay mechanics are started otherwise an error is displayed.
    - Pack details button
        - When tapped the app checks that a pack has been loaded. If a pack has been loaded a screen with the pack details (pack name, location, description and game time) is displayed

- End game button (ends the game)
- Zoom button
    - Centres map on user location

- Pack Creation Screen
    - Pack selector (displays list of all packs created by the logged in user which is on the phone). Once a pack is selected. The editing screen is shown
        - UI (map + buttons) is instantiated. The pack is loaded from local storage. Pins are displayed on the screen.
            - List all pins button
                - Loads a list of all the pins in the pack where pins can be deleted.
            - 'Add Pin' Mechanics
                - Adds Pin to the pack
            - Pack details button
                - When tapped the app checks that a pack has been loaded. If a pack has been loaded, a screen with the pack details (pack name, location, description and game time) is displayed.
                - Option to edit the pack description and game time
            - Change map type button
                - If tapped, the map type changes from satellite hybrid to plain map and vice versa
            - Finished editing button
                - After confirming that the user wants to save the changes to their pack. The updated pack is sent to the online database and is saved to local storage
    - Create new pack or edit existing pack
        - Registration form appears where a new pack can be created. These pack details are then stored on both the device and the online database.
- Pack Store Screen
    - Search for packs button
        - Displays a search form. Users enter the pack name, location and creator username search criteria.

- Search button
    - Downloads list of packs data from online database, which have values similar to the search criteria
    - Displays list of available packs
    - If user taps on a pack, the selected pack is downloaded from the database and stored in local storage. (The app checks if that pack already exists on the phone. If pack exists the user is asked if he/she wants to override it.)
- Edit packs on the phone button
    - All local packs are loaded and displayed in a table
    - If user taps on a pack the app the selected pack the user has the option to delete the pack from local storage.

- Account Page
    - Displays details about the logged in user: the user's id number, username, email, number of packs played, number of packs created and number of total competitive points scored.
    - Logout button. If this is tapped the user's information on the phone is cleared (packs downloaded remain) and login screen is presented.

Each sub screen/form presented has a back button which will return dismiss the sub screen/form and return to the previous view.

Identification of User Input and Validation Checks Needed

Form Requiring User Input	Data Item
Login Screen	Email Address
Login Screen	Password
Account Registration	Username
Account Registration	Email Address
Account Registration	Password
Account Registration	Re-enter Password
New Pack Registration	Pack Name
New Pack Registration	Location
New Pack Registration	Brief Description
New Pin Details	Pin Title
New Pin Details	Pin Hint
New Pin Details	Pin Codeword
New Pin Details	Point Value
Pack Store Search	Pack Name Fragment
Pack Store Search	Location Fragment
Pack Store Search	Creator Username Fragment

All of the fields have a non-empty validation system. If the user attempts to leave a field blank an error message will appear. This validation is needed just in case the user forgets to enter a field. However, the pack store search fields can be left empty as this would indicate that they aren't using that field to define the search criteria. For example if I was only interested in searching for a pack in London but didn't mind who it was made by or the pack name, I would only enter 'London' into the location search field and the search result would return any pack with 'London' as its location.

The majority of the fields also have a maximum length check for ease of reading. For example, I also didn't want the pin details

to be too long (maximum of 140 characters) to deter people from creating packs with long convoluted question and answers. This check is carried out by retrieving the String class' attribute 'count' (number of characters) and placing an if text.count > 140 check. If the string's character length is larger than 140, an error is raised.

Certain fields also have type validation checks for example the point value in the new pin details form will only accept an integer, as the point values must be integers. Otherwise an error occurs. This is checked via a type cast, the program tries to cast the inputted value into an integer, if this can't occur then an error message is presented to the user.

Regular Expression Pattern Matching

When users register for accounts, to ensure that account usernames only contain alphanumeric characters, I used this simple regex pattern "[A-Za-z0-9]+" to test their inputted username. The square brackets define the set of acceptable characters, in this case any alphanumeric character. The plus means that the whole string must contain at least one (or more) alphanumeric character to match. E.g. "TesTing123" would be accepted but "!@£$ser" wouldn't be.

In the case of the email addresses, I needed a way to tell if an email address was valid. I, therefore, created an email address regular expression. This regular expression can be used to test the validity of emails. "^[A-Za-z0-9._-]+@[A-Za-z0-9._-]+\\.[A-Za-z]{2,}" is the pattern. This pattern allows strings that begin ("^") with one of the allowed characters –alphanumeric characters, ".", "_" or "-" (inside of the square brackets) – followed by more allowed characters ("+"), then an @ symbol trailed by at least one more allowed characters, ending with a '.' and at least two alphabetical characters (the {2,} specifies that the string has to end with at least two alphabetical characters). This pattern allows strings such as 'alvin-lee@london.co.uk' but doesn't allow invalid email such as '@stbens.org.uk'.

I ran these regex pattern in the Swift environment and it correctly produced the desired Boolean results.

```
// Method which sees if a test string matches a regular expression pattern
private func stringTester(regExPattern: String, testString: String) -> Bool {
    let stringTester = NSPredicate(format:"SELF MATCHES %@", regExPattern)          (8 times)
    return stringTester.evaluate(with: testString)                                  (8 times)
}

// Method which returns whether the test string is a valid email address
func isEmailValid(testStr:String) -> Bool {
    let emailRegExPattern = "^[A-Za-z0-9._-]+@[A-Za-z0-9._-]+\\.[A-Za-z]{2,}"      (4 times)
    return stringTester(regExPattern: emailRegExPattern, testString: testStr)      (4 times)
}

// Method which returns whether the test string only contains alphanumeric characters
func isAlphanumeric(testStr:String) -> Bool {
    let alphanumericRegExPattern = "[A-Za-z0-9]+"                                  (4 times)
    return stringTester(regExPattern: alphanumericRegExPattern, testString: testStr)  (4 times)
}

isEmailValid(testStr: "alee@stbens.org.uk")                                        true
isEmailValid(testStr: "al-dew@tiscali-old.co")                                     true
isEmailValid(testStr: "@hotmail.com")                                              false
isEmailValid(testStr: "andrew@co.u")                                               false

isAlphanumeric(testStr: "The Quick Brown Fox")                                     false
isAlphanumeric(testStr: "Testing123")                                              true
isAlphanumeric(testStr: "123Test")                                                 true
isAlphanumeric(testStr: "£@$%^a")                                                  false
```

Tests

Corresponding Boolean result

Organisation of Online Database

Structure

The database will be cloud hosted on my personal website 'alvinlee.london' which is web hosted by the (1and1.co.uk). Data from the database will be accessible via a link (alvinlee.london/{file name}.php). This will call a PHP script that is part of the TownHunt API, the script will query the database and return a response as a JSON file which can be parsed. By storing data online, multiple users will be able to access and download packs to play.

I decided to use relational tables to store the information on the database as it allows data to be linked and accessed easily through cross-tabled SQL queries.

The four table in the MySQL relational database are as following (with sample data, column headings as field names and yellow highlight are the data type):

1. 'Pins' Table: This will store all of the pin and their respective details. The max length of 'Title', 'Hint' and 'Codeword' column values is 840 as 140 characters is the maximum length input in the app, however special characters (e.g. " double quotes) are converted to html characters (e.g. ") which are six characters long. This is to ensure that inputs with quotes do not interfere with SQL statements run by the API. Therefore the database should be able to store the theoretical maximum (140 x 6) characters.

PinID int(50)	Title varchar(840)	Hint varchar(840)	Codeword varchar(840)	CoordLongitude float	CoordLatitude float	PointValue int(10)	PackID int(50)
1	ICT 1	What is the code...	2MESTN07	51.5211	-0.306856	25	1

2. PinPack Table: This will store all of the information about the pin packs. PackName, Description and Location fields must accommodate up to 180 characters again due to HTML entities (max char length when entered in the app is 30 chars)

PackID int(50)	PackName varchar(180)	Description varchar(180)	Creator_UserID int(50)	Location varchar(180)	GameTime int(50)	Version int(20)
1	Bennies Pack	Explore the school!	2	Ealing, London	60	3

3. Users Table: This table will store all of the information about users. The passwords will be stored as (32 character) hash values for security reasons. I do not want to know what the un-hashed passwords are just in case of SQL injections/hacks.

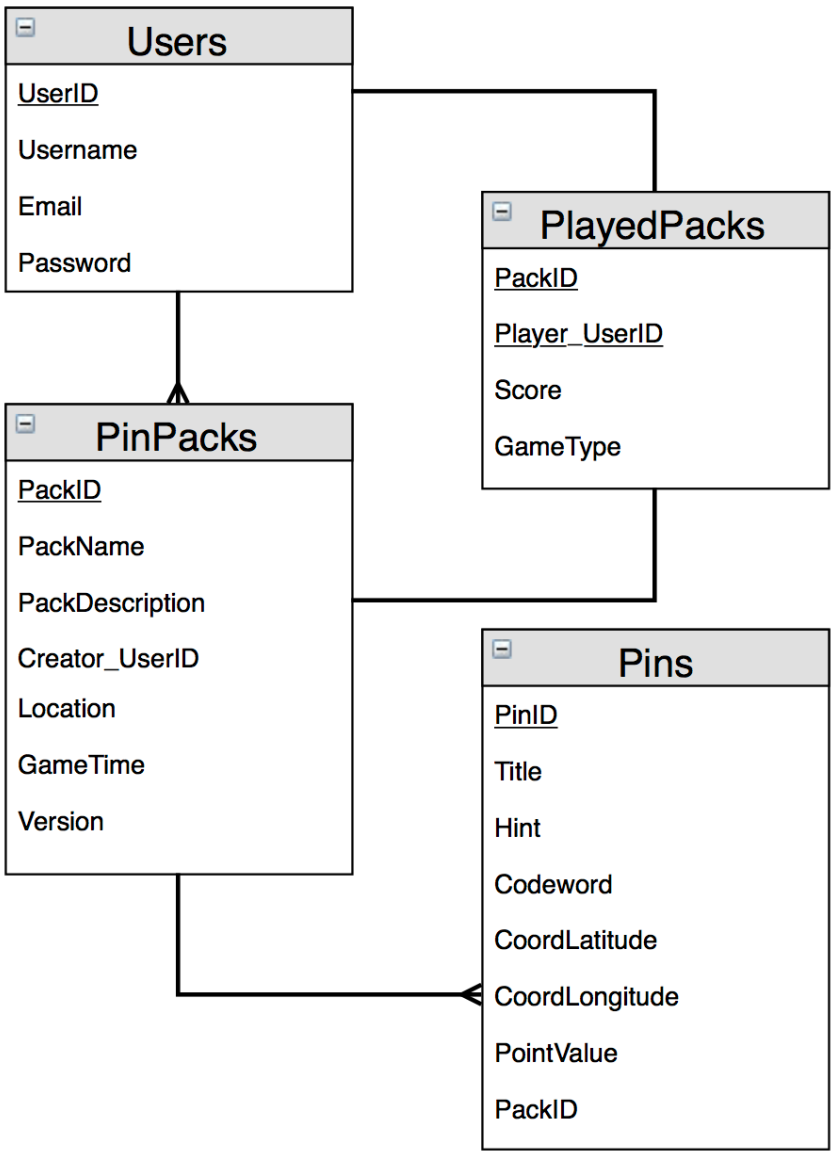
UserID int(50)	Username varchar(20)	Email varchar(100)	Password varchar(32)
1	Jonty	jc@hotmail.com	5ba7e50b29036a55cbf15e2281480c21
2	amldjlee8	alvin@gmail.com	781f357c35df1fef3138f6d29670365a

4. PlayedPacks Table: This table stores the score records of users playing the packs.

PackID int(50)	**Player_UserID** int(50)	Score int(50)	Game Type varchar(11)
1	1	175	competitive

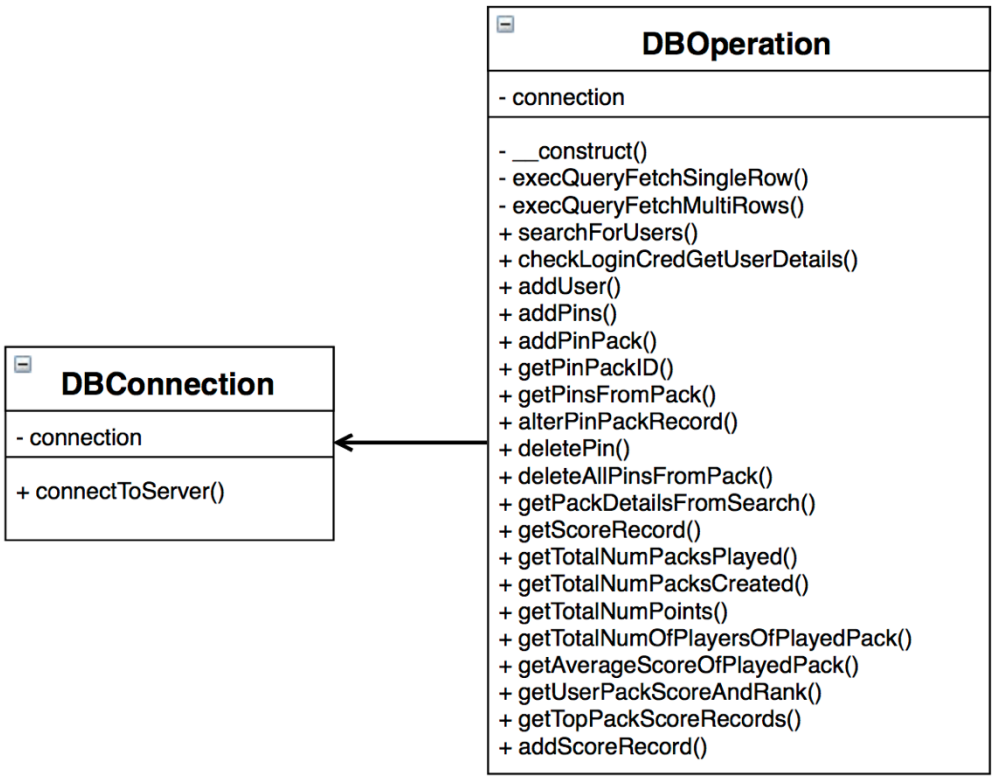
Bold & Underlined column headings are primary keys

Entity Relationship Diagram

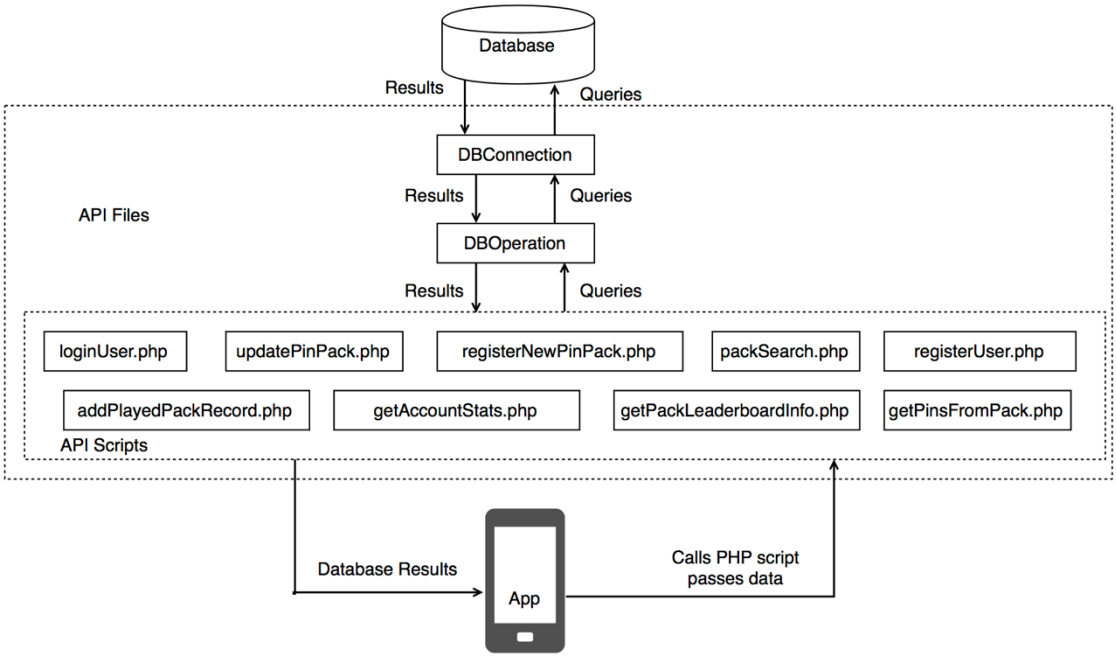


API Organisation

TownHunt API Class Diagram



API Interaction with App



Every API script will perform a separate function. For example the registerUser.php will attempt to add a new account to the user's database if one doesn't already exist. Whereas the getPinsFromPack.php will query the database for pins with a specific pack ID and return the resulting set. Creating separate scripts to carry out different functions ensures that unexpected data isn't received from or inserted into the database.

The PHP Requests, in TownHunt, are POST requests instead of GET as GET requests can be called from a URL in a browser, its not possible to POST via the address bar of a browser. It is therefore safer to use POST requests especially when dealing with sensitive information like passwords and email addresses. This means someone with the PHP script URLs can't alter the database by entering data via the address bar of a browser.

Each API script is structured roughly the same way.

1. A response associative array (called array in PHP but functions more like a standard dictionary with a 'key' being associated with a value) is initialised.
2. Data is transferred from the application via a HTTP POST Request. In the form of either a JSON string or a post string. The script checks that the request was a POST request otherwise an error will be appended response array.
3. The API then checks for certain passed variables in the POST data received. If the variables are not found an error message is appended to the array and the database operation is not called.
4. If the variables were passed then a DBOperation class object is instantiated (which in turn instantiates a DBConnection object i.e. connects to the database). The DBOperation class contains a variety of methods which all contain a SQL Query statement. The API calls the relevant methods inside the DBOperation, passing the applicable variables. The SQL Query is executed on the database and returns either a result set or a query successful flag.
5. If the query was successful then the error flag in the response array is set to false, the results set is also appended to the response array if any was received. Otherwise the error flag is set to true and an error message is appended to the response array.
6. The response is then encoded as a JSON file and echoed/returned to the caller.

Example Psuedocode for the loginUser.php script

PostData = ["userEmail": "alee2@stbens.org.uk", "userPassword": "testing123"]

Start of loginUser.php *Script*

// API/Database response Associative Array/Dictionary
response ← NEW Associative Array ()
// Checks whether the server request method was POST. The API file only allows POSTed data

```
IF REQUEST METHOD == "POST" THEN
        // Attempts to retrieve post data
        userEmail ← PostData["userEmail"]
        userPassword ← PostData["userPassword"
         // Checks to see if any required variable was not passed i.e. variables are empty
        IF userEmail IS EMPTY or userPassword IS EMPTY THEN
                response["error"] ← true
                response["message"] ← "One or more fields are empty"
        ELSE
                // DBOperation Object Instantiated allowing connection to the database
                Database ←NEW DBOperation()
                // Password is hashed for security reasons
                securePassword ← HASH userPassword
                // Searches the database for the account matching the username and password via
                // DBObject
                userDetails ←Database.checkLoginCredGetUserDetails(userEmail, securePassword)
                // Checks if a account was found
                if userDetails IS NOT EMPTY THEN
                        // Login Success
                        response["error""] ← false
                        response["message"] ← "User is registered"
                        response["accountInfo"] ← userDetails
                ELSE
                        // Login Unsuccessful
                        response["error""] ← true
                        response["message"] ← "Account not found. The email and or the password is
                        incorrect"
                END IF
        END IF
ELSE
        // Data sent not via POST
        response["error""] ← true
        response["message"] ← "You are not authorised"
END IF

RETURN ENCODE response INTO JSON

END OF SCRIPT
```

SQL Queries

A total of 20 SQL parameterised queries are used in the TownHunt API. The functionality and description of all the SQL queries used can be found in the commenting of the DBOperation.php (page). The three most complex queries used in the API is the Pin Store Search query (getPackDetailsFromSearch() function in DBOperation), the query to retrieve the players score and rank in the leaderboard (getUserPackScoreAndRank() function) and the cross-table parameterised query to retrieve the top score in the leaderboard (getTopPackScoreRecords() function)

Pin Store Search Query

This query returns a list of pack details of all of the packs which match the search criteria passed into the query (the yellow highlighted sections are where the criteria strings would be inserted)

```
SELECT pinPackTable.`PackID`,
       pinPackTable.`PackName`,
       pinPackTable.`PackDescription` AS Description,
       pinPackTable.`Location`,
       pinPackTable.`GameTime` AS TimeLimit,
       pinPackTable.`Version`,
       possibleCreatorTable.`Username` as CreatorUsername,
       possibleCreatorTable.`UserID` AS CreatorID
```

1

```
FROM `db648556307`.`PinPacks` AS pinPackTable,
     (SELECT `UserID`, `Username`
      FROM `db648556307`.`Users`
      WHERE `Username` LIKE '%".{Creator Username Search Criteria}."%') AS
possibleCreatorTable
```

2

```
WHERE pinPackTable.`PackName` LIKE '%".{Pack Name Search Criteria}."%'
     AND pinPackTable.`Location` LIKE '%".{Location Search Criteria}."%'
     AND pinPackTable.`Creator_UserID` =  possibleCreatorTable.`UserID`
ORDER BY pinPackTable.`PackName` ASC"
```

3

1. This select statement defines what to retrieve in the final results set. The query will retrieve the pack's ID, name, description, location, game time and version from the PinPack table. The query will also retrieve the creator of the pack's username and user ID. The final result will be presented in as a table or an associative array as it is in the TownHunt API. For some of the field names, I have aliased e.g. (PackDescription AS Description) to match the keys of the JSON files used to store the packs on the Phone
2. The SQL query's data sources are set as the PinPacks table and a newly generated possibleCreatorTable. This possibleCreator table is generated in a subquery from the Users table. The LIKE keyword enables the search for a specific pattern (specified in the trailing quotation marks) in a column. The % is a wildcard. Placing the % before and after the search criteria means that the sub query finds any record where the search criteria appears in any position of the column's value (SQL patterns are non-case sensitive i.e. 'TeST' is the same as 'test'). %Ben% as the search pattern would result in matches for 'Benedict' or 'eben10'. The possibleCreatorTable will be filled with record where the Username is similar to the search criteria username.

```
(SELECT `UserID`, `Username` FROM `db648556307`.`Users` WHERE `Username` LIKE '%Test%')
```

Number of rows: 25

+ Options

UserID	Username
3	Test2
6	test
9	test123
10	tester

I tested this wildcard search on my test data base this was the result for possibleCreatorTable. Any record with the username similar to 'test' was taken

3. Now the query looks at the PinPacks table and searches for records where the pack name and location is similar to the search criteria and has the same user ID as a record in the possibleCreatorTable. The final result is ordered alphabetically by pack name.

If the search criteria is left blank then all that remains are two wildcard so all records will be matched and it is up to the other search criteria's to narrow down the search. If no search criteria is input then all packs will be returned.

HTTP Post Requests

Calls to the API is carried out by a user defined class called DatabaseInteraction(). This class has two methods: checkInternetAvaliability() and postToDatabase(). 'checkInternetAvaliability()' attempts to connect to a test socket IP address. If connection can be made to the socket and the device's internet communication channels are open then true is returned. If either a connection can't be made or the communication channels are closed then 'checkInternetAvaliability()' will return false. 'postToDatabase()' is a method which creates the HTTP POST request to the API/Database. This class will make use of an Apple API called URLSession, which enables data to be downloaded on a background thread. The database/API response received back will be returned as a JSON string that is converted into a dictionary.

When a view controller wants to POST to the API/Database, a DatabaseInteraction object is instantiated. The internet connectivity is then tested (checkInternetAvaliability() is called). If there is an internet connection then the POST data is created and postToDatabase() is called. The response from the database/API is then processed. If no internet connection is detected then the function that the DatabaseInteraction object is in will call itself in a recursive manner - prompting the user to connect to the internet – until an internet connection is found.

Organisation of Local Storage

Pack Data Dictionaries

The pack data, when handled inside the app, is stored in RAM as a multidimensional dictionary containing 8 key value pairs (CreatorID, Pins, Location, PackName, Version, TimeLimit, PackID and description). The "Pins" value contains all the pins in the pack and will therefore be a list of dictionaries which contain information about the pin. For example a pack data dictionary could be

```
["CreatorID": "10",
"Pins": [[
    "Codeword" : "Butterfly",
    "CoordLatitude" : "51.5263",
    "CoordLongitude" : "-0.2941",
    "Hint" : "What animal can you see drawn here?",
    "PointValue" : "100",
    "Title" : "A Mystic Sign",
],
[
    "Codeword" : "Electricity",
    "CoordLatitude" : "51.5258",
    "CoordLongitude" : "-0.293057",
    "Hint" : "What makes this block dangerous?",
    "PointValue" : "75",
    "Title" : "Concrete Block",
]], "Location": "Ealing, London, "PackName": "Hanger Hill Park Pack", "Version": 5,
"TimeLimit": "30", "PackID": "52", "Description": "Explore the park!"]
```

By using dictionaries data can easily be extracted, changed and iterated over.

JSON Files

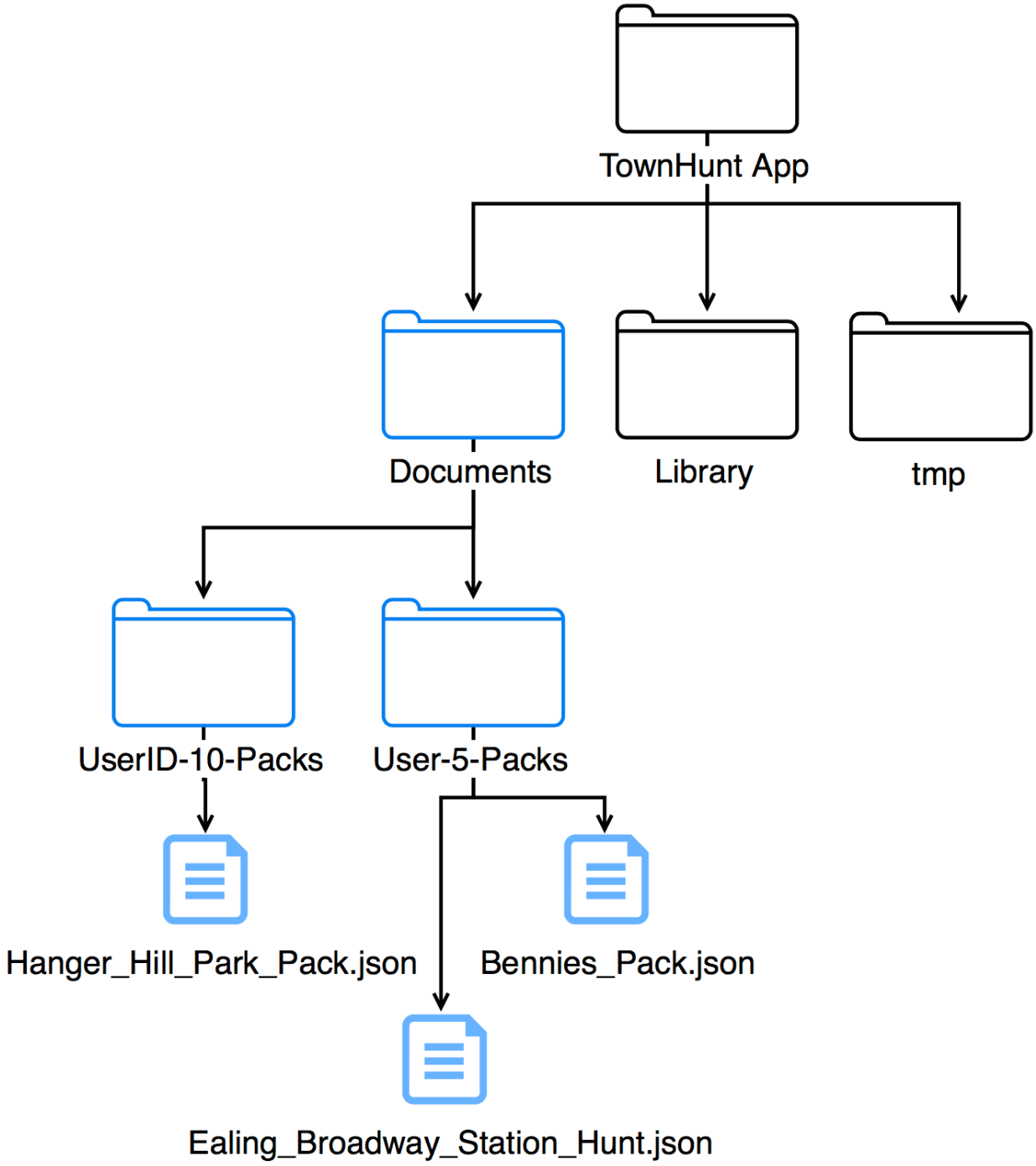
The packs data will be stored in local storage as JSON files. I chose to use JSON files as JSON is lightweight and stores data as a collection of key-value pairs which matches the structure of the pack data dictionaries. For example, the pack data dictionary above would be stored as the following JSON:

```
{
  "CreatorID" : "10",
  "Pins" : [
    {
      "Title" : "A Mystic Sign",
      "CoordLatitude" : "51.5263",
      "CoordLongitude" : "-0.2941",
      "Hint" : "What animal can you see drawn here?",
      "PointValue" : "100",
      "Codeword" : "Butterfly"
    },
    {
      "Title" : "Concrete Block",
      "CoordLatitude" : "51.5258",
      "CoordLongitude" : "-0.293057",
      "Hint" : "What makes this block dangerous?",
      "PointValue" : "75",
      "Codeword" : "Electricity"
    }
],
  "Location" : "Ealing, London",
  "PackName" : "Hanger Hill Park Pack",
  "Version" : "5",
```

```
  "TimeLimit" : "30",
  "PackID" : "52",
  "Description" : "Explore the park!"
}
```

JSON files are also quick to load and there is native support in Swift for JSON serialisation to convert dictionaries to JSON and vice versa (using the in-built JSONSerialisation class).

File Hierarchy



Each Swift application is given an area in local storage to store files. Inside the Documents folder there will be subdirectories called UserID-[Creator User ID]-Packs. There is a separate folder for each user whose packs are stored locally on the device. Inside this folder are the JSON files of all of the packs that the user has created (JSON files are named after the pack's name with spaces being replaced with underscores) E.g. in the diagram Hanger_Hill_Park_Pack.json is stored in UserID-10-Packs folder as the user, who has ID 10, is the one who created it.

Linked Lists

Linked lists will be used to keep track of the pack's filenames, display name (name-location key) and user's who have packs on the device. These lists are stored, via the UserDefaults class, into a plist found in the Library folder of the application. This plist data is loaded into the application when the app loads and is cached. This means that it is quick and easy access to information stored in UserDefaults. This is why the linked lists will be stored here.

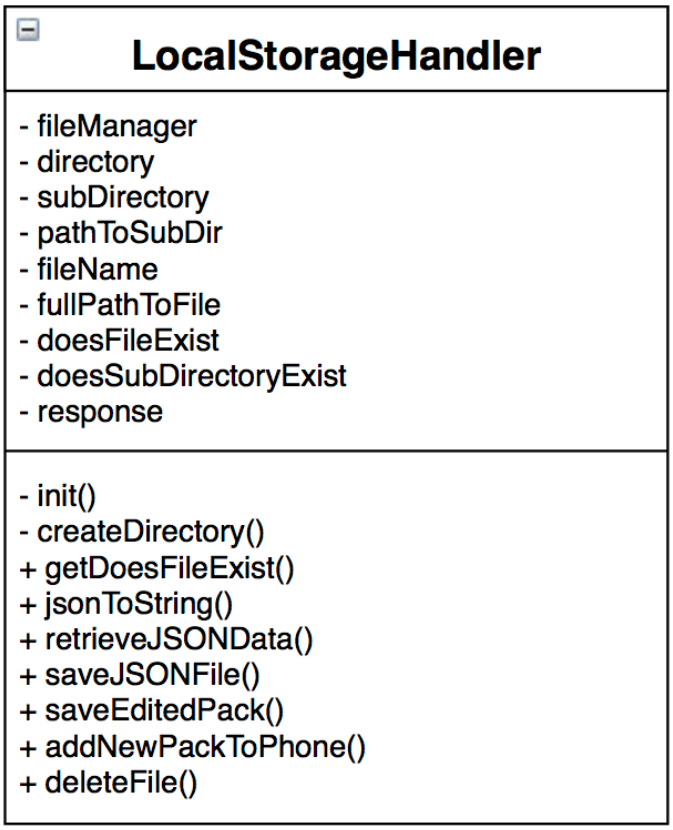
listOfLocalUserIDs = ["1", "2", "10"]

This is the main list which stores the IDs of all users whose pack is on the device. The user's subdirectory can be inferred *("UserID-10-Packs*" for example*)*. This links to dictionary 'UserID-[user ID]-Packs' which contains the display names of the pack (the string that will appear in pack selectors) as a key associated with the respective pack filename (without the extension). For example: *UserID-10-Packs = ["St Benedict's Pack – Ealing, London": "St_Benedict\'s_Pack", "Pitshanger Lane Hunt – Ealing, London": "Pitshanger_Lane_Hunt"]*

Each user id in the *listOfLocalUserIDs* has its own list. This is especially useful for the pack creator's pack selector when only the logged in user's packs should be displayed.

Key	Type	Value
▼ Root	Dictionary	(9 items)
▼ listOfLocalUserIDs	Array	(2 items)
Item 0	String	11
Item 1	String	10
▼ UserID-10-Packs ⊕ ⊖	Dictionary	(2 items)
Test Pack - Location	String	Test_Pack
Hanger Hill Park Pack - Ealing, Lon...	String	Hanger_Hill_Park_Pack
▼ UserID-11-Packs	Dictionary	(1 item)
Mayflower Hunt - London	String	Mayflower_Hunt

Sample UserDefaults plist generated inside Swift



Local Storage Handler Class

Storing files to local storage is handled by the user defined class LocalStorageHandler. It is instantiated by any ViewController that wants to interact with the local storage e.g. to load packs/JSON files or to save packs.

The LocalStorageHandler class handles the creation/deletion of files as well as maintenance of the linked lists.

The list maintenance is carried out inside the addNewPackToPhone() method and the deleteFile(). (see code on page 109 for detailed commented explanation)

Gameplay Mechanics

Pin Class

The user-defined Pin class will be a class in my project. The game play of the app relies heavily on this class to be dynamically generated. The class will extend the built in Swift classes of 'NSObject' and 'MKAnnotation'. "Through NSObject, objects inherit a basic interface to the runtime system", this is what Apple describe the NSObject class as. Since I am using Apple's native MapKit (MK) module, the Pin class will inherit from 'MKAnnotation'. Instances of 'MKAnnotations' are placed on the map with interactive capabilities. By inheriting the 'MKAnnotations' class I will be able to place instances of the Pin class onto the map allow the details of the pin to be shown once the user clicks on the pin. The other method inside the pin class returns a dictionary about the pin. This dictionary's format matches the format of how the Pins will be stored in the JSON file i.e. the key names (e.g. Title, Hint etc) are the same.

Psuedo Code:

CLASS Pin INHERITES FROM (NSObject, MKAnnotations):

title : String

hint : String

```
codeword : String

coordLongitude : Float

coordLatitude : Float

pointValue : Integer

DEF initialisation (title, hint, codeword, coordLongitude, coordLatitude, pointValue):

    self.title ← title

    self.hint ← hint

    self.codeword ← codeword

    self.coordLongitude ← coordLongitude

    self.coordLatitude ← coordLatitude

    self.pointValue ← pointValue

DEF getDictOfPinInfo():

    let dictOfPinInfo ← ["Title": self.title!, "Hint": self.hint, "Codeword": self.codeword,
    "CoordLatitude": String(self.coordinate.latitude), "CoordLongitude":
    String(self.coordinate.longitude), "PointValue": String(self.pointVal)]

    return dictOfPinInfo
```

Game

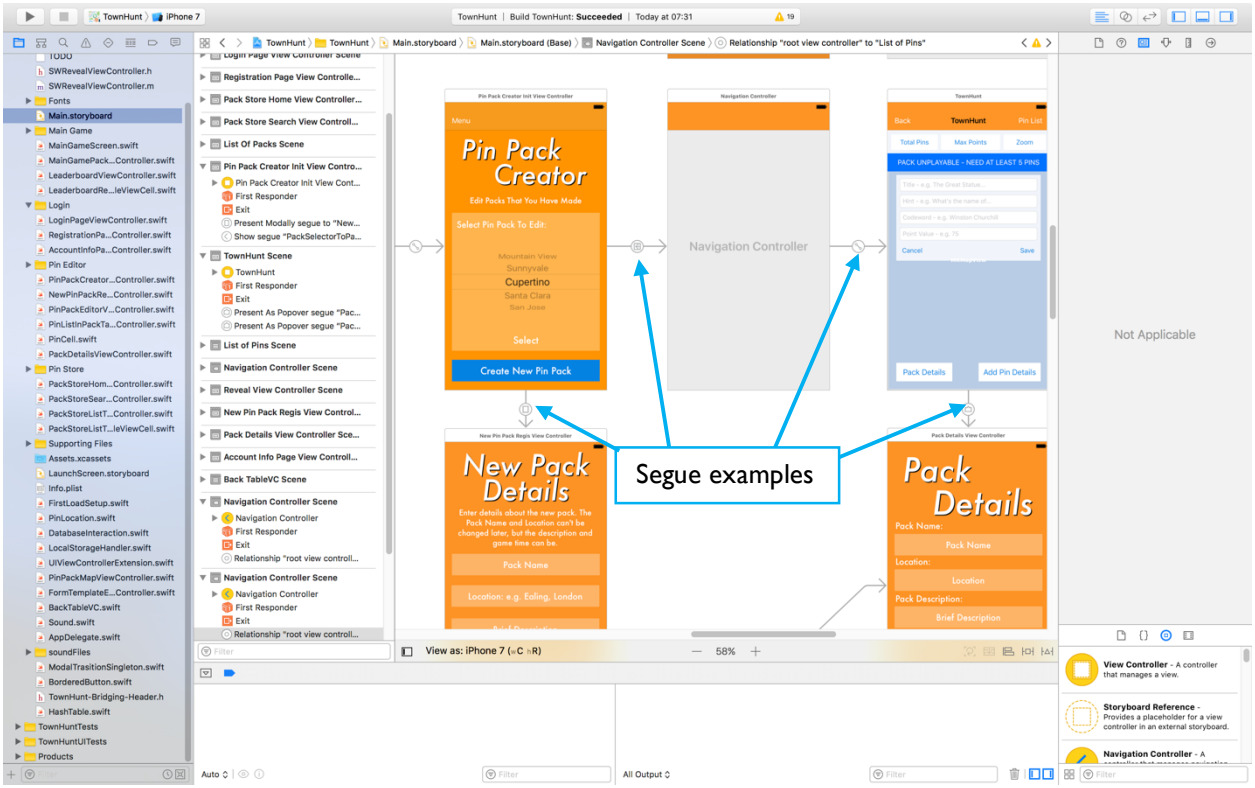
When the pack is selected to be played, the pins from the file will be inatansiated as PinLocation objects and stored in RAM as an array called gamePins. When the start button is tapped the game begins. If competitive mode was selected then a countdown timer (equal to the pack's Game Time/ Time Limit) is started and 5 pins from the gamePins array (at a random index) will be appended to the activePins array and displayed on the map. These five pins act as starting group of pins to find. The activePins array contains all the pins which are currently displayed on the map. Then over the course of the game at random times pins from the gamePins array will be moved to the activePins array. If casual mode is selected then all the pins in gamePins are appended to the activePins, thus all pins from the pack are on the map.

If the pin is tapped, text (containing the pin's hint/question will be displayed) and a textbox (for the user to enter their answer) will appear. The user's answer will be checked against the codeword attribute on the PinLocation object and the user is notified if he/she entered the correct answer. Once the correct codeword has been entered, the PinLocation object will die and be removed from the active pins list.

During the pack creation page, when a long press occurs on the map, a default Pin object with blank attributes will be created and the details will be added to the object by the user. Once saved the pin's details will be added to the pack.

HCI & Technical Solution

Xcode UI Design Interface



This is part of my application's 'Storyboard'. This is where the UI for each view was designed graphically. In the Storyboard you also have the option to set the functionality of certain objects in the view e.g. a button can be defined so that the app screen will transition to another view. The transitions between views are called 'segues' in Swift which can be set to have identifiers. Using these identifiers, the currently displayed ViewControllers can identify the target ViewController and pass data. In the section overview diagrams for this section I will be representing segues as arrows and labelling them if there is an identifier associated with the segue. Segues can also be called programmatically in the code. When a view is dismissed, the app returns to the view that presented the dimissed view, e.g. if View A segues to View B and then View B gets dismissed, the app returns to View A.

The user interface was designed to have a consistent theme throughout the application. E.g. most views share the same font type, font colour and general orange colour scheme.

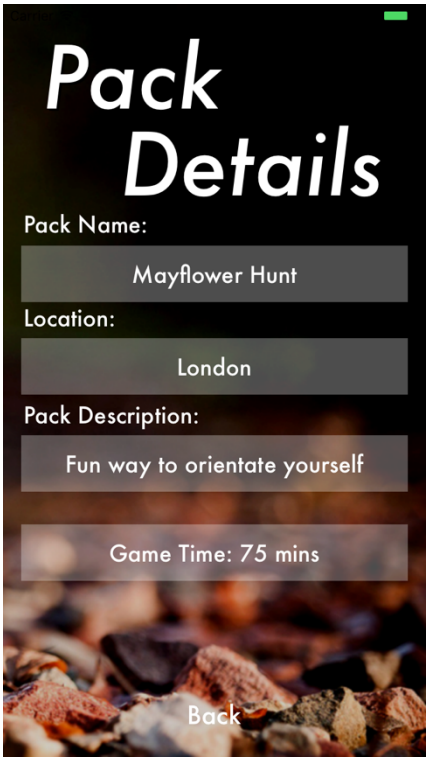
The Apple developer website's API reference guide and Swift documentation helped me to understand the pre-existing objects in Swift.

Main Game Section

Overview of View Connections & Transitions in This Section

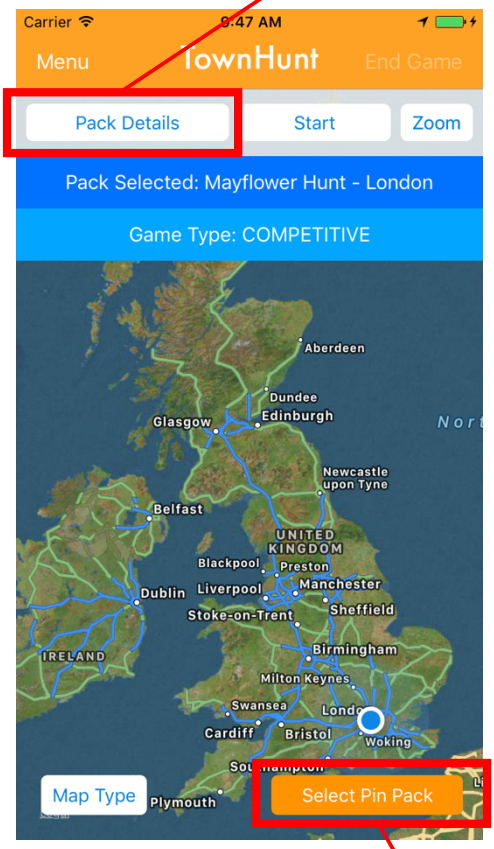
(Red arrows represents a transition (segue) caused by the button which is encapsulated by a red box. Segue identifier is in red text by the transition arrow. However, not all segues have identifiers)

MainGameScreenToPackDetail



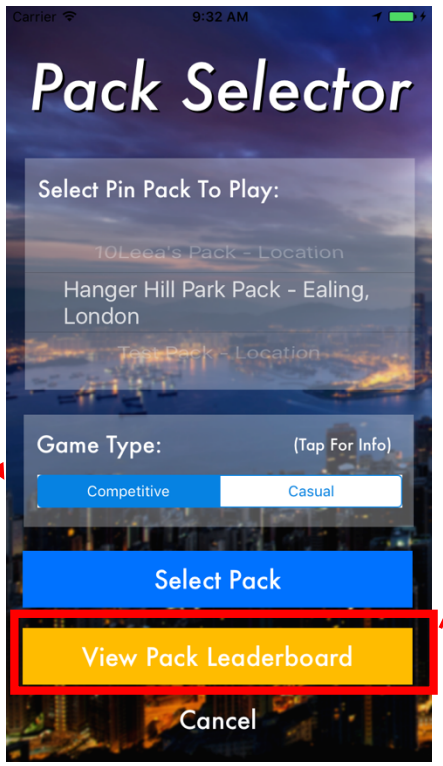
Pack Detail View

Pack Leaderboard View



Main Game Screen View

First view in this section i.e. when Main Map is selected from the menu this view appears



PackSelectorToLeaderboard

 Main Game Screen Pack Selector View

Main Game Screen View

End Game button allows the user to end the game before the timer runs out

Menu button displays the app menu. This button is disabled during the game.

Zoom Button centres the map on the user's location

Start Button starts the game play mechanics and displays the time remaining during the game.

Pack Details Button/Points Display: This button transitions to the pack details screen if the game isn't on. During the game this button is disabled and is used as an outlet to display the current score

Select pack displays the loaded pack's name and location (Disappears when the game starts)

Map

Game type display shows the selected game type. (Disappears when the game starts)

User location

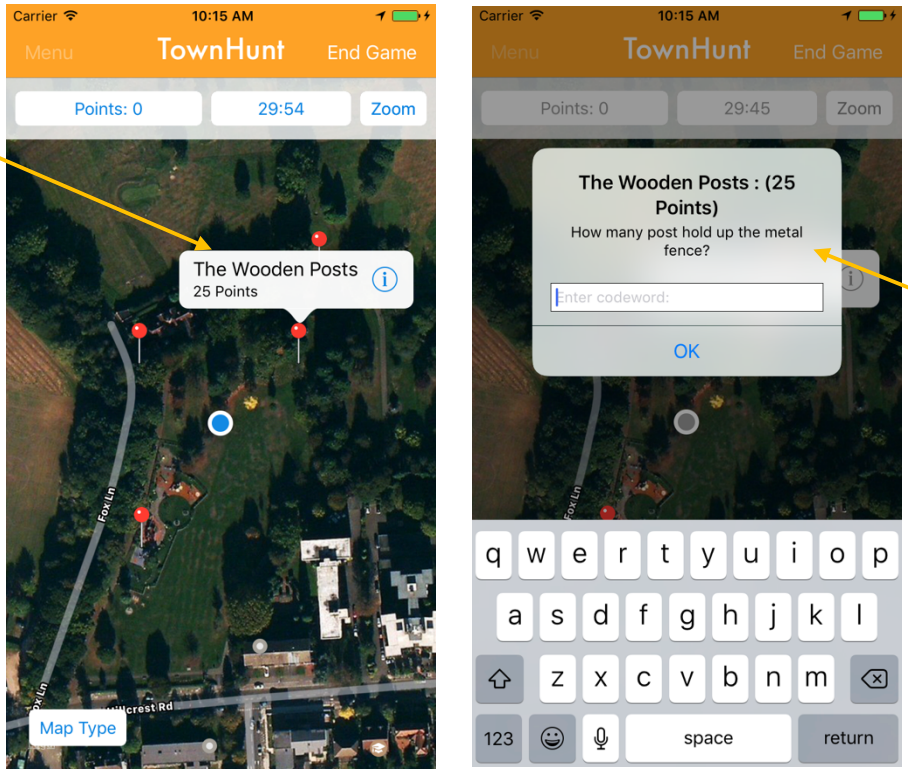
Map type button changes the map from a hybrid satellite map to a standard flat map

Select Pin Pack Button transitions to the pack selector. (Disappears when the game starts)

During the game:

Pins Appear when the game starts. Title and point value is displayed

When pin is tapped the pin details and answer box appears



Main Game Screen View Controller Code

```
//
//  MainGameScreenViewController.Swift
//  TownHunt App
//
//  This file defines the logic behind the main game screen.

import UIKit // UIKit constructs and manages the app's UI
import MapKit // MapKit constructs and manages the map and annotations

// Sets up the delegate protocol - Once the modally presented view (pack selector) has
closed, 'packSelectedHandler' is called
protocol MainGameModalDelegate {
    // Variables from the pack selector are passed into this class via this function
    func packSelectedHandler(selectedPackKey: String, packCreatorID: String, gameType:
String)
}

// Class acts as the logic behind the main game screen view. Inherits from
PinPackMapViewController, MKMapViewDelegate, MainGameModalDelegate
class MainGameScreenViewController: PinPackMapViewController, MKMapViewDelegate,
MainGameModalDelegate {

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
        // UI elements found on the top orange navigation bar
    @IBOutlet weak var endGameButton: UIBarButtonItem!
    @IBOutlet weak var menuOpenNavBarButton: UIBarButtonItem!
        // UI elements found in the info bar (bar below the navigation bar)
    @IBOutlet weak var viewBelowNav: UIView! // The background of the info bar
    @IBOutlet weak var startButton: UIButton! // Also used as an outlet to display the time
remaining
    @IBOutlet weak var pointsButton: BorderedButton! // Also used as a button which leads to
information on the selected pack details
        // The map UI element
    @IBOutlet weak var mapView: MKMapView!
        // UI elements of the pack info bars
    @IBOutlet weak var packSelectedButton: UIButton!
    @IBOutlet weak var gameTypeButton: UIButton!
        // The select pin pack button
    @IBOutlet weak var selectPinPackButton: BorderedButton!

    // Initialises attributes which will hold details about the selected pack and player
    public var filename = ""
    public var selectedPackKey = ""
    public var userPackDictName = ""
    public var packCreatorID = ""
    public var gameType = "competitive"
    private var packData: [String:Any] = [:]
    private var playerUserID: String = ""

    // Initialises attributes relating the actual game mechanics
    private var timer = Timer()
    private var isPackLoaded = false
    private var isGameOn = false
    private var countDownTime = 0
    private var points = 0
    private var timeToNextNewPin = 0
    private var activePins: [PinLocation] = [] // Holds pins which are currently displayed on
the map
    private var gamePins: [PinLocation] = [] // Holds pins which are in the selected pack
```

```
    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Check if this is the first time the app has been launched
        checkFirstLaunch()

        // Prevents the phone from going to sleep and turning the screen off
        UIApplication.shared.isIdleTimerDisabled = true

        // Connects the menu button to the menu screen
        menuOpenNavBarButton.target = self.revealViewController()
        menuOpenNavBarButton.action = #selector(SWRevealViewController.revealToggle(_:))

        // Sets up the map view
        mapView.showsUserLocation = true
        mapView.mapType = MKMapType.hybrid // Sets up the default map to a satellite map with
road names
        mapView.delegate = self // Gives this class the ability to control the map UI element

        // Checks if a user is logged in
        let isUserLoggedIn = UserDefaults.standard.bool(forKey: "isUserLoggedIn")
        // If no user is logged in the login screen is immediately called
        if(!isUserLoggedIn){
            self.performSegue(withIdentifier: "loginView", sender: self)
        }
    }

    // Checks if user has launched the app before, if not calls the initial file setup
    private func checkFirstLaunch(){
        let defaults = UserDefaults.standard
        // If app has already launched "isAppAlreadyLaunchedOnce" flag will be set i.e. not
nil
        if defaults.string(forKey: "isAppAlreadyLaunchedOnce") != nil{
            print("App already launched")
        } else {
            // "isAppAlreadyLaunchedOnce" flag is set to true
            defaults.set(true, forKey: "isAppAlreadyLaunchedOnce")
            print("App launched first time")
            // Initial set up class is called and the pack linked list system is setup
            FirstLoadSetup().initialSetup()
        }
    }

    // [------------------MAP & PIN MECHANICS-----------------------------------------------]

    // Connects the Zoom button to the code
    // - This centres the map around the user as well as increasing the magnification of the
map
    @IBAction func zoomOnUser(_ sender: AnyObject) {
        // Checks if the phone's GPS location is currently available
        if mapView.isUserLocationVisible == true {
            // If user's location is found the map region is set to the 200x200m area around
the user
            let userLocation = mapView.userLocation
            let region =
MKCoordinateRegionMakeWithDistance(userLocation.location!.coordinate, 200, 200)
            mapView.setRegion(region, animated: true)  // Updates the UI map
        } else {
```

```
            // If user's location is not found an error message is presented to the user
            displayAlertMessage(alertTitle: "GPS Signal Not Found", alertMessage: "Cannot
zoom on to your location at this moment.\n\nIf you have disabled TownHunt from accessing your
location, please go to the settings app and allow TownHunt to access your location")
        }
    }

    // Connects the Change Map button to the code
    // - Changes the map type from a hybrid satellite map to the standard map (no satellite
imagery) and vice versa
    @IBAction func changeMapButton(_ sender: AnyObject) {
        if mapView.mapType == MKMapType.hybrid{
            mapView.mapType = MKMapType.standard
            viewBelowNav.backgroundColor = UIColor.brown.withAlphaComponent(0.8) // Changes
the info bar's colour to brown
        } else {
            mapView.mapType = MKMapType.hybrid
            viewBelowNav.backgroundColor = UIColor.white.withAlphaComponent(0.8) // Changes
the info bar's colour to white
        }
    }

    // MKMapViewDelegate method which creates the annotations (Pins) and their 'callouts'
    func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView?
{
        let identifier = "PinLocation"
        // Checks if the annotation is a pin (PinLocation class)
        if annotation is PinLocation {
            // Returns a reusable annotation view located by the identifier "PinLocation".
            var annotationView = mapView.dequeueReusableAnnotationView(withIdentifier:
identifier)
            // If no free views exist then a new view is created
            if annotationView == nil {
                annotationView = MKPinAnnotationView(annotation: annotation, reuseIdentifier:
identifier) // New Annotation view
                annotationView!.canShowCallout = true // Enables callouts
                // Sets up the info button symbol on the callout
                let infoButton = UIButton(type: .detailDisclosure)
                annotationView!.rightCalloutAccessoryView = infoButton
            } else {
                // If a reusable annotation view is found then the annotation object is
retrieved
                annotationView!.annotation = annotation
            }
            // Annotation view is returned
            return annotationView
        }
        return nil // Nil is returned if the annotation is not of the PinLocation class
    }

    // MKMapViewDelegate method which is called when the pin has been tapped.
    func mapView(_ mapView: MKMapView, annotationView view: MKAnnotationView,
calloutAccessoryControlTapped control: UIControl) {
        // Retrieves details about the pin which has been tapped by accessing attributes from
the instance
        let pin = view.annotation as! PinLocation
        let pinTitle = "\(pin.title!) : (\(pin.pointVal) Points)"
        let pinHint = pin.hint

        // Creates an alert where information about the pin is displayed.
```

```
        let alertCon = UIAlertController(title: pinTitle, message: pinHint, preferredStyle:
.alert)
        // Adds an answer box to the alert. Here, the codeword found can be entered
        alertCon.addTextField(configurationHandler: {(textField: UITextField!) in
textField.placeholder = "Enter codeword:"})
        // Adds an OK button to the alert
        alertCon.addAction(UIAlertAction(title: "OK", style: .default, handler: {(action) ->
Void in
            // If the OK button is tapped, the user's entered codeword is checked
            let userInput = alertCon.textFields![0] as UITextField

            // Checks if the codeword entered matches the one found on the pin
            if (userInput.text?.lowercased() == pin.codeword.lowercased()){
                self.points += pin.pointVal // Increments point value
                self.updatePointsLabel() // Updates UI display
                pin.isFound = true

                let currentPinIndex = self.activePins.index(of: pin) // Gets index of the pin
                self.activePins.remove(at: currentPinIndex!) // Removes the pin from the
active pin index
                mapView.removeAnnotation(pin) // Removes pin from the map
                self.alertCorrectIncor(true, pointVal: pin.pointVal) // Tells the user that
the codeword entered is correct
            } else {
                self.alertCorrectIncor(false, pointVal: pin.pointVal) // Tells the user that
the codeword entered is incorrect
            }
        }))
        // Presents the alert to the user
        present(alertCon, animated: true, completion: nil)
    }

    // [---------------------- PACK SELECTOR MECHANICS----------------------------]

    // Connects the Select Pin Pack Button to the code
    // - Instantiates the pack selector
    @IBAction func selectPinPackButtonTapped(_ sender: Any) {
        // Retrieves the pack selector UI elements from the UI storyboard editor
        let storyboard = UIStoryboard(name: "Main", bundle: nil)
        let viewController = storyboard.instantiateViewController(withIdentifier
:"packSelector") as! MainGamePackSelectorViewController
        viewController.delegate = self // Sets the pack selector's delegate as the main game
screen view (current view)
        self.present(viewController, animated: true) // Presents the pack selector
    }

    // Method which retrieves the details of the pack selected from the Pack Selector
    func packSelectedHandler(selectedPackKey: String, packCreatorID: String, gameType:
String){
        playerUserID = UserDefaults.standard.string(forKey: "UserID")! // Retrieves the id of
the user logged in
        self.selectedPackKey = selectedPackKey
        self.packCreatorID = packCreatorID
        self.gameType = gameType
        // Checks if the selected pack key, pack creator id and the game type was passed
        if !(self.selectedPackKey.isEmpty || self.packCreatorID.isEmpty ||
self.gameType.isEmpty){
            userPackDictName = "UserID-\(packCreatorID)-Packs"
            let defaults = UserDefaults.standard
            // Checks if the dictionary containing the list of packs made by a specified
creator exists
```

```
            if let dictOfPacksOnPhone = defaults.dictionary(forKey: userPackDictName){
                filename = dictOfPacksOnPhone[selectedPackKey] as! String
                // Loads pack data from the file
                packData = loadPackFromFile(filename: filename, userPackDictName:
userPackDictName, selectedPackKey: selectedPackKey, userID: packCreatorID)
                // Loads the pack pins into the gamePins array
                gamePins =  getListOfPinLocations(packData: packData)
                isPackLoaded = true
                // Displays the info bars which contain details about the user selection
                    // Displays what pack was selected
                packSelectedButton.isHidden = false
                packSelectedButton.setTitle("Pack Selected: \(selectedPackKey)", for:
UIControlState())
                    // Displays what game type was selected
                gameTypeButton.isHidden = false
                gameTypeButton.setTitle("Game Type: \(gameType.uppercased())", for:
UIControlState())
            } else { // If creator's dictionary could not be found then an error message is
displayed
                displayAlertMessage(alertTitle: "Error", alertMessage: "Data Couldn't be
loaded")
            }
        } else {// If data was not passed an error message is displayed
            displayAlertMessage(alertTitle: "Error", alertMessage: "Selected pack data not
passed.")
        }

    }

    // [------------------------------ GAME MECHANICS ------------------------------]

    // Resets the game to the default game settings (specified in the selected pack details)
    private func resetGame(){

        // Resets game variables
        countDownTime = Int(packData["TimeLimit"] as! String)! * 60 // Resets countdown timer
(in seconds)
        points = 0
        isGameOn = false
        timeToNextNewPin = 0
        activePins = []
        gamePins = getListOfPinLocations(packData: packData) // Reloads gamePins with pins
from the selected pack data
        pointsButton.setTitle("Points: 0", for: UIControlState())

        // Unhides select pin pack button
        selectPinPackButton.isHidden = false
        // Re-enables menu button
        menuOpenNavBarButton.accessibilityElementsHidden = false
        menuOpenNavBarButton.isEnabled = true
        // Unhides the info bars which contain details about the user selection
        packSelectedButton.isHidden = false
        gameTypeButton.isHidden = false
    }

    // Method is called when the start button is tapped
    // - Starts the game if certain criteria are met
    @IBAction func startButtonTapped(_ sender: AnyObject) {
        // Checks if game is in progress
        if isGameOn == false{
            // Checks if pack has been loaded
```

```
            if isPackLoaded == false {
                // If no pack is loaded then an error message is displayed
                displayAlertMessage(alertTitle: "No Pin Pack Selected", alertMessage: "Tap
'Select Pin Pack' to chose a pack")
            // Checks if there is too few pins (less than 5) in the pack
            }else if gamePins.count < 4{
                // Error message is displayed
                displayAlertMessage(alertTitle: "Too Few Pins", alertMessage: "The selected
pack has too few pins please add more")
            }
            else{
                // Game is started

                timeToNextNewPin = randomTimeGen(countDownTime/5) // Random time, till a pin
is added to the map, is generated
                resetGame()
                // Hides the info bars which contain details about the user selection
                packSelectedButton.isHidden = true
                gameTypeButton.isHidden = true
                // Hides select pin pack button
                selectPinPackButton.isHidden = true
                // Disables menu button
                menuOpenNavBarButton.isEnabled = false
                // Enables end game button
                endGameButton.isEnabled = true
                // Game in progress flag is updated to true
                isGameOn = true

                // Checks if the logged in user is playing a pack made by him/her
                if playerUserID == packCreatorID{
                    displayAlertMessage(alertTitle: "You Made This Pack!", alertMessage:
"Since you created this pack, your score will not be uploaded to the leaderboard")
                }

                // Checks what game type the user wants to play
                if gameType == "competitive"{ // Competitive mode is selected
                    // Five random pins from the pack are added to the the active pins array
                    for _ in 0...3{
                        addRandomPinToActivePinList()
                    }
                    // The pins in the active pins array are added to the map as Annotations
                    mapView.addAnnotations(activePins)
                    // Timer is set up - updateTime method is called every second
                    timer = Timer.scheduledTimer(timeInterval: 1.0, target: self, selector:
#selector(MainGameScreenViewController.updateTime), userInfo: nil, repeats: true)
                    // Starts the timer
                    updateTime(timer)
                } else { // Casual mode is selected
                    // All game pins are added to the active pins array
                    activePins = gamePins
                    // All active pins are added to the map
                    mapView.addAnnotations(activePins)
                    // Indicates to the user that casual mode is being played
                    startButton.setTitle("Casual Mode", for: UIControlState())
                    displayAlertMessage(alertTitle: "Casual Mode Game", alertMessage: "There
is no time limit so take your time to explore and hunt for the pins! Once you have finished
tap 'End Game'")
                }
            }
        }
    }
```

```
    // Updates the points button's title to the current score
    private func updatePointsLabel(){
        pointsButton.setTitle("Points: \(points)", for: UIControlState())
    }

    // Updates the timer
    func updateTime(_ timer: Timer){
        // Checks if the countdown time has reached 0
        if(countDownTime > 0 && isGameOn == true){
            // Updates the timer displayed to the user
            let minutes = String(countDownTime / 60) // Calculates the minutes part of the
time left
            let seconds = countDownTime % 60 // Using modulo, Calculates the seconds part of
the time left
            // To keep the seconds display two characters long, if the seconds is less than
10, an 0 is appended infront
            var secondsToDisplay = ""
            if seconds < 10{
                secondsToDisplay = "0" + String(seconds)
            } else{
                secondsToDisplay = String(seconds)
            }
            // The start button doubles up as the timer display. It is updated with the
current time left
            startButton.setTitle(minutes + ":" + secondsToDisplay, for: UIControlState())
            // Count down time is decremented by one
            countDownTime -= 1
            print(timeToNextNewPin)
            // Checks if the time to next new pin is greater than 0
            if timeToNextNewPin > 0{
                // Time to next pin is decremented by one
                timeToNextNewPin -= 1
            // Checks if time to next new pin is 0 and there are still pins to add to the
game
            } else if (timeToNextNewPin == 0 && gamePins.isEmpty == false){
                addRandomPinToActivePinList() // Calls a method which will add a new pin to
the map
                timeToNextNewPin = randomTimeGen(countDownTime/5) // Sets a new random time
to next pin
            }

            // Checks if there are any pins on the map
            if activePins.count == 0{
                    addRandomPinToActivePinList() // Calls a method which will add a new pin
to the map
            }

        } else { // If countdown timer is 0, the game is ended
            endGame()
        }
    }

    // Creates the alert telling the user if the codeword entered was correct or incorrect
    private func alertCorrectIncor(_ isCorrect: Bool, pointVal: Int){
        var alertTitle: String = ""
        var alertMessage: String = ""

        // Checks user has found all of the pins
        if activePins.count == 0 && gamePins.count == 0{
            endGame() // Ends the game
```

```
        } else if isCorrect == false {
            alertTitle = "Incorrect!"
            alertMessage = "Try Again!"
            Sound().playSound("Wrong-answer-sound-effect") // An 'incorrect' sound is played
        // Checks if the isCorrect flag is true
        } else if isCorrect == true{
            alertTitle = "Well Done!"
            alertMessage = "\(pointVal) Points Added"
            Sound().playSound("CorrectSound") // A 'correct' sound is played
        }
        // Displays the outcome to the user
        displayAlertMessage(alertTitle: alertTitle, alertMessage: alertMessage)
    }

    // Generates (and returns) a random number between 0 and maxNum
    private func randomTimeGen(_ maxNum: Int) -> Int{
        return Int(arc4random_uniform(UInt32(maxNum)))
    }

    // Adds a random pin from the gamePin array to the map screen
    private func addRandomPinToActivePinList(){
        Sound().playSound("Message-alert-tone") // A sound is played to alert the user that a
new pin has been added
        let newPinIndex = randomTimeGen(gamePins.count) // Random pin index is selected
        self.mapView.addAnnotation(gamePins[newPinIndex]) // A pin and the random index is
added to the map
        activePins.append(gamePins[newPinIndex]) // The pin is appended to the active pins
array
        gamePins.remove(at: newPinIndex) // The pin is removed from the game pins array
    }

    // Method is called when the end game button is tapped
    // - Ends the game before the count down timer has reached 0
    @IBAction func endGameButtonTapped(_ sender: AnyObject) {
        // Sets up a confirmation alert to the user to confirm that the user really want to
the end the game
        let alert = UIAlertController(title: "End the Game", message: "Do you really want to
end the game ", preferredStyle: .alert)
        // A 'Yes' option is added to the alert
        let yesAction = UIAlertAction(title: "Yes", style: .default, handler: {(action) ->
Void in
            // If pressed then the game will end
            self.endGame()
        })
        alert.addAction(yesAction)
        // A 'Cancel' option is added, just in case the end game button was tapped
accidentally
        let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
        alert.addAction(cancelAction)
        present(alert, animated: true, completion: nil) // Alert is displayed to user
    }

    // Method which ends the game
    private func endGame(){
        Sound().playSound("Game-over-yeah") // Play a sound to indicate to the user that the
game has ended
        timer.invalidate() // Ends the timer
        startButton.setTitle("Start", for: UIControlState())
        isGameOn = false
        // Checks if the user who played the pack was the one who created it
        if playerUserID != packCreatorID{
```

```
            // If not then a method is called to add the score to the database
            addRecordToDB(score: String(points))
        }
        self.mapView.removeAnnotations(activePins) // The map is cleared of pins/annotations
        endScreen() // End screen is displayed
        resetGame()
        pointsButton.setTitle("Pack Details", for: UIControlState())
        endGameButton.isEnabled = false
    }

    // Method which displays the end screen after the game has ended
    private func endScreen(){
        // Displays "GAME OVER" and the final score to the user
        displayAlertMessage(alertTitle: "GAME OVER!", alertMessage:  "You Scored \(points).")
    }

    // Sends the score and player user id to the database
    private func addRecordToDB(score: String){
        // Retrieves pack ID
        let packID = packData["PackID"] as! String
        // Sets up the string to be posted
        let postData =
"PackID=\(packID)&PlayerUserID=\(playerUserID)&Score=\(score)&GameType=\(gameType)"

        // Initialises database interaction object
        let dbInteraction = DatabaseInteraction()
        // Tests for internet connectivity
        if dbInteraction.checkInternetAvaliability(){
            // If there is internet connectivity then the data is posted to the online
database API via the dbInteraction class on the background thread.
            // The "addPlayedPackRecord.php" API adds first time scores of a pack to the
online database
            let responseJSON = dbInteraction.postToDatabase(apiName:
"addPlayedPackRecord.php", postData: postData){ (dbResponse: NSDictionary) in

                // Retrieves the database response
                let isError = dbResponse["error"]! as! Bool
                let dbMessage = dbResponse["message"]! as! String

                // Displays a message to the user indicating whether the score was received
                DispatchQueue.main.async(execute: { // Returns to the main thread
                    if isError{ // If there is an error it is displayed to the user
                        self.displayAlertMessage(alertTitle: "Error", alertMessage:
dbMessage)
                    } else{
                        print("Score successfully sent to database API")
                    }

                })
            }
        } else{ // If no internet connectivity, an error message is displayed asking the user
to connect to the internet
            let alertCon = UIAlertController(title: "Error: Couldn't Connect to Database",
message: "No internet connectivity found. Please check your internet connection",
preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Try Again", style: .default, handler: {
action in
                // Recursion is used to recall  addRecordToDB until internet connectivity is
restored
                self.addRecordToDB(score: score)
            }))
```

```
            // Alert is presented to user
            self.present(alertCon, animated: true, completion: nil)
        }
    }

    // [----------------------------System Mechanics----------------------------------]

    // System function which is called to check whether the transition to the next view
should be allowed
    override func shouldPerformSegue(withIdentifier identifier: String?, sender: Any?) ->
Bool {
        // Checks if a segue identifier was passed
        if let segueIdentifier = identifier {
            // Check if identifier was "MainGameScreenToPackDetail"
            if segueIdentifier == "MainGameScreenToPackDetail" {
                // Checks if a pack is loaded
                if isPackLoaded == false {
                    // If no pack is loaded then an alert message is displayed
                    displayAlertMessage(alertTitle: "No Pack is Loaded", alertMessage: "Tap
'Select Pin Pack' to chose a pack")
                    return false // Transition to the pack detail view is not allowed
                // Checks if the game is on
                } else if isGameOn {
                    // If the game is in progress
                    return false // Transition to the pack detail view is not allowed
                }
            }
        }// Otherwise the // Transition to the pack detail view is allowed
        return true
    }

    // System function which is called when the view is about to transition (segue) to
another view. This enables data to be passes between views
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Checks that the segue identifier is "MainGameScreenToPackDetail"
        if segue.identifier == "MainGameScreenToPackDetail" {
            // Checks if the next view controller is of the PackDetailViewController class
            if let nextVC = segue.destination as? PackDetailsViewController{
                // Passes data from the current main game view to the next pack detail view
                //  - Passes the pack details without the pins
                var packDetails = packData
                packDetails.removeValue(forKey: "Pins")
                nextVC.packDetails = packDetails as! [String : String]
                nextVC.isPackDetailsEditable = false // Doesn't allow the pack details to be
edited
            }
        }
    }
}
```

Main Game Screen Pack Selector View

Main Game Screen Pack Selector View Controller Code

```
//
//  MainGamePackSelectorViewController.swift
//  TownHunt App
//
//  This file defines the behaviour of the pack selector view

import UIKit // UIKit constructs and manages the app's UI

// Class acts as the logic behind the pack selector view. Inherits from
FormTemplateExtensionOfViewController
class MainGamePackSelectorViewController: FormTemplateExtensionOfViewController,
UIPickerViewDelegate, UIPickerViewDataSource {

    // This view's optional delegate is specified as of the MainGameModalDelegate class
    var delegate: MainGameModalDelegate?

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
```

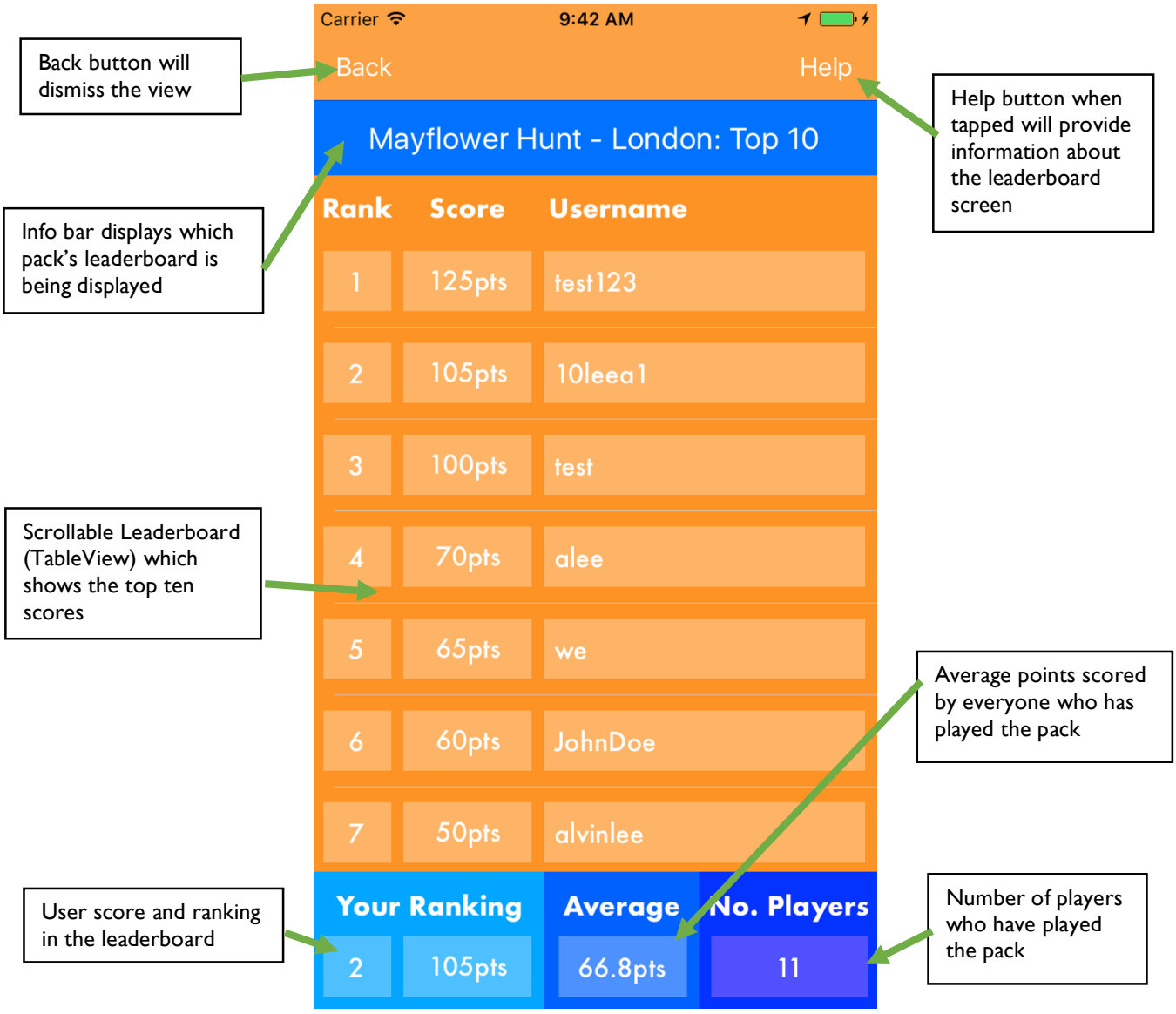
```
    @IBOutlet weak var packPicker: UIPickerView! // The slot-machine-esque selector which
displays all of the local packs
    @IBOutlet weak var gameTypeSegCon: UISegmentedControl! // The game type binary
(segmented) selector
    @IBOutlet weak var selectMapButton: UIButton! // The 'Select Map' button
    @IBOutlet weak var viewLeaderboardButton: UIButton! // The 'View Leaderboard' button

    // Initialises attributes which will hold details about all local packs, selected pack
and pack options
    private var allPacksDict = [String: String]() // Holds the pack name-location key and the
creator user ID of all local packs
    private var pickerData = [String]() // Stores data (pack name-location keys) which will
be shown in the picker
    private var selectedPickerData: String = "" // Holds the selected picker data (pack name-
location key)
    private var gameType = "competitive" // Stores which game type is selected

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Sets the background image
        setBackgroundImage(imageName: "packSelectorBackground")

        // Sets up the pack picker
        self.packPicker.delegate = self // Gives this current class the ability to control
the picker UI element
        self.packPicker.dataSource = self // The data source of the picker is set as this
current class
        setUpPicker() // Calls a function to the set up the picker
    }

    // [------------------------------ Pack Picker Mechanics --------------------------]

    // Populates the picker with the names of local packs
    private func setUpPicker(){
        let defaults = UserDefaults.standard
        // Retrieves the list of user ids whose packs are found in local storage
        let listOfUsersOnPhone = defaults.array(forKey: "listOfLocalUserIDs")
        // Checks if there is no users whose packs are on the phone
        if !(listOfUsersOnPhone!.isEmpty){
            // Iterates over every id and retrieves all packs on the phone
            for userID in listOfUsersOnPhone as! [String]{
                let userPackDictName = "UserID-\(userID)-Packs"
                // Checks that the user has packs on the phone
                if let dictOfPacksOnPhone = defaults.dictionary(forKey: userPackDictName) {
                    // Appends all pack name-location keys and their creator user id to the
allPacksDict
                    for pack in Array(dictOfPacksOnPhone.keys){
                        self.allPacksDict[pack] = userID
                    }
                }
            }
            // Sorts the picker data in alphabetical order by the pack name-location key
            self.pickerData = Array(allPacksDict.keys).sorted{ $0.lowercased() <
$1.lowercased() }
            // Sets the initial selected element as the first value of the picker
            self.selectedPickerData = pickerData[0]
```

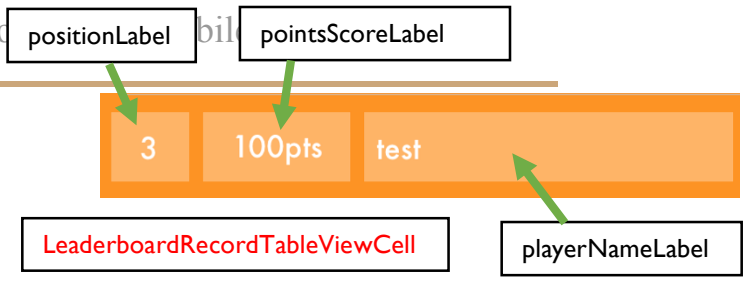
```
        // If there are no local packs, pack interaction buttons are hidden and a message is
shown to the user
        } else if pickerData.isEmpty {
            // The only value in the picker is set to "No Packs Found"
            self.pickerData = ["No Packs Found"]
            self.selectMapButton.isHidden = true // User is unable to select a pack as there
aren't any
            self.viewLeaderboardButton.isHidden = true // Leaderboard button is hidden as
there are no packs
        }
    }

    // PickerView method which sets the number of columns of data in the picker
    func numberOfComponents(in pickerView: UIPickerView) -> Int {
        // The picker used in the app only has one column to display the pack name-location
key
        return 1
    }

    // PickerView method which sets the number of rows of data in the picker
    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) ->
Int {
        // Number of rows is determined by the number of elements in the pickerData array
        return self.pickerData.count
    }

    // PickerView method which sets the picker data to display for a certain row
    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component:
Int) -> String? {
        // Returns the corresponding pack name-location key
        return self.pickerData[row]
    }

    // PickerView method that retrieves the item that is currently selected
    func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component:
Int) {
        // "selectedPickerData" is set to the pack name-location key
        self.selectedPickerData = pickerData[row]
    }

    // PickerView method that sets up how the text of each row should be
    func pickerView(_ pickerView: UIPickerView, viewForRow row: Int, forComponent component:
Int, reusing view: UIView?) -> UIView {
        // Instantiates a label
        let label = UILabel(frame: CGRect(x: 0, y: 0, width: 330, height: 30));
        label.lineBreakMode = .byWordWrapping; // Text won't go off the screen and instead
wrap
        label.numberOfLines = 0 // Allows the label to have unlimited lines
        label.text = pickerData[row] // Sets the text to display as the pack name-location
key
        label.textColor = UIColor.white // Sets the text colour to white
        label.font = UIFont.systemFont(ofSize: CGFloat(20)) // Sets the font to 20 pixels
        label.sizeToFit() // Makes the frame of the label fit the size of the text
        return label // Returns the label to display
    }

    // PickerView method that sets the height of each row
    func pickerView(_ pickerView: UIPickerView, rowHeightForComponent component: Int) ->
CGFloat {
        return 50 // 50 pixels is returned
    }
```

```
    // Connects the Select Pack Button to the code
    // - If the select button is tapped then the view is dismissed and the selected pack data
is returned to the main game view
    @IBAction func selectPackButtonTapped(_ sender: Any) {
        // Checks to see if this view has a delegate
        if let delegate = self.delegate {
            // Passes the selected pac data to the main game view
            delegate.packSelectedHandler(selectedPackKey: self.selectedPickerData,
packCreatorID: self.allPacksDict[self.selectedPickerData]!, gameType: gameType)
            // Dismisses the pack selector view
            self.dismiss(animated: true, completion: nil)
        } else{ // Displays error message
            displayAlertMessage(alertTitle: "Error", alertMessage: "Pack Selector has no
Delegate")
        }

    }

    // [ ---------------------- Game Type Selector Mechanics ----------------------------]

    // Connects the game type selector to the code
    // - Changes the gameType attribute
    @IBAction func indexChanged(_ sender: UISegmentedControl) {
        // A binary switch changes the game type from competitive to casual depending on what
the user has selected
        switch gameTypeSegCon.selectedSegmentIndex
        {
        case 0:
            gameType = "competitive"
        case 1:
            gameType = "casual"
        default:
            break
        }
    }

    // [----------------------------System Mechanics---------------------------------------]

    // Connects the game type info button to the code
    // - Display info explaining what the two game modes are
    @IBAction func tapForInfoButtonTapped(_ sender: Any) {
        let displayMessage = "Competitive mode: 5 Pins will be initially appear with more and
more pins being added as the game progresses. You will have to hunt for the pins under a time
limit. \n\nCasual mode: All of the pins in the pack will appear. Hunt them all in your own
time with no time limit. \n\nYour first competitive playthrough score will be added to the
leaderboard. If your first playthrough was casual, no future scores for that pack will count
in the leaderboard"
        displayAlertMessage(alertTitle: "Competitive or Casual?", alertMessage:
displayMessage)

    }

    // Connects the cancel button to the code
    // - Closes the view when the cancel button is pressed
    @IBAction func cancelButtonTapped(_ sender: Any) {
        self.dismiss(animated: true, completion: nil)
    }

    // System function which is called when the view is about to transition (segue) to
another view. This enables data to be passes between views
```

```
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Checks that the segue identifier is "PackSelectorToLeaderboard"
        if segue.identifier == "PackSelectorToLeaderboard" {
            // Retrieves the navigation controller of the target view
            let destNavCon = segue.destination as! UINavigationController
            // Checks that the target controller is of the class LeaderboardViewController
            if let targetController = destNavCon.topViewController as?
LeaderboardViewController{
                // Passes selected pack data to the leaderboard view
                targetController.selectedPackKey = self.selectedPickerData // Pack name-
location key
                targetController.packCreatorID = self.allPacksDict[self.selectedPickerData]!
// Pack creator id
            }
        }
    }
}
```

Pack Leaderboard View





Pack Leaderboard Cell Code

```
//
//  LeaderboardRecordTableViewCell.swift
//  TownHunt App
//
//  File defines the prototype leaderboard
record cell

import UIKit // UIKit constructs and manages the app's UI

// Class defines the prototype cell that each leaderboard record cell is based upon. Inherits
from UITableViewCell
class LeaderboardRecordTableViewCell: UITableViewCell {

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
    @IBOutlet weak var positionLabel: UILabel!
    @IBOutlet weak var pointsScoreLabel: UILabel!
    @IBOutlet weak var playerNameLabel: UILabel!

    // Called when the cell is loaded into the table
    override func awakeFromNib() {
        // Sets up the functionality of the cell
        super.awakeFromNib()
    }

    // Called when the cell is selected in the table
    override func setSelected(_ selected: Bool, animated: Bool) {
        // Changes the colour of the cell
        super.setSelected(selected, animated: animated)
    }

}
```

Pack Leaderboard View Controller Code

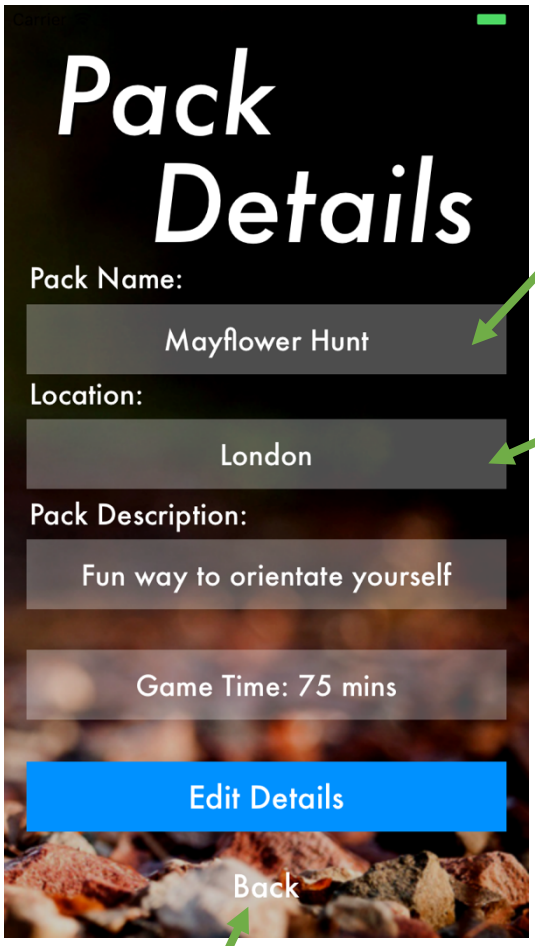
```
//
//  LeaderboardViewController.swift
//  TownHunt App
//
//  This file defines the behaviour of the leaderboard view

import UIKit // UIKit constructs and manages the app's UI

// Class acts as the logic behind the leaderboard. Inherits from UIViewController,
UITableViewDelegate, UITableViewDataSource
class LeaderboardViewController: UIViewController, UITableViewDelegate, UITableViewDataSource
{

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
    @IBOutlet var leaderboardTable: UITableView! // The table view containing the leaderboard
info
    @IBOutlet weak var packLeaderboardHeaderLabel: UIButton! // Info bar below the nav bar
    @IBOutlet weak var userPositionLabel: UILabel! // Logged in user's position in the
leaderboard
    @IBOutlet weak var userPointsScoreLabel: UILabel! // Logged in user's score in the
leaderboard
    @IBOutlet weak var averagePointsScoredLabel: UILabel! // Average score of all the players
who have plaeyd the pack
```

```
    @IBOutlet weak var numberOfPlayersLabel: UILabel! // Total number of plays who have
played the pack

    // Initialises attributes which will hold details about the selected pack, the associated
leaderboard and the user
    public var selectedPackKey = "" // Pack name-location key
    public var packCreatorID = ""
    private var packID = ""
    private var userID = ""
    private var leaderboardRecords = [[String:String]]()

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Sets up the leaderboard table
        leaderboardTable.delegate = self // Gives this current class the ability to control
the table UI element
        leaderboardTable.dataSource = self // The data source of the table is set as this
current class
        getUserID()
        getPackID()
        getLeaderboardData()
    }

    // Retrives the logged in user's id
    private func getUserID(){
        userID = UserDefaults.standard.string(forKey: "UserID")!
    }

    // Retrieves the pack id of the selected pack
    private func getPackID(){
        // Checks that the selectedPackKey and packCreatorID is not empty otherwise an error
message is displayed
        if !(self.selectedPackKey.isEmpty || self.packCreatorID.isEmpty){
            let userPackDictName = "UserID-\(packCreatorID)-Packs"
            let defaults = UserDefaults.standard
            if let dictOfPacksOnPhone = defaults.dictionary(forKey: userPackDictName){
                // Opens file and retrieves the pack data and hence the pack id
                let filename = dictOfPacksOnPhone[selectedPackKey] as! String
                let packData = PinPackMapViewController().loadPackFromFile(filename:
filename, userPackDictName: userPackDictName, selectedPackKey: selectedPackKey, userID:
packCreatorID)
                packID =  packData["PackID"] as! String
            } else{ // Error message displayed
                displayAlertMessage(alertTitle: "Error", alertMessage: "Data Couldn't be
loaded")
            }
        } else{ // Error message displayed
            displayAlertMessage(alertTitle: "Error", alertMessage: "Selected pack data not
passed")
        }
    }

    // Prepares the leaderboard data to be displayed in the UI
    private func setLeaderboardData(data: [String:Any]){
        // Sets the leaderboard records to the top 10 scores retreived from the database
        leaderboardRecords = data["topScoreRecords"] as! [[String : String]]
        // Refreshes the leaderboard tables displayed to the user
        leaderboardTable.reloadData()
```

```
        // Sets up pack info and the info about the logged in user's ranking/score
        let userRank = data["userRank"]! as! String
        let userScore = data["userScore"]! as! String
        let averageScore = String(describing: data["averageScore"]!)
        let numOfPackPlayers = data["numOfPlayersOfPack"]! as! String
        // Updates the UI display
        setNonTableLabels(userRank: userRank, userScore: userScore, averageScore:
averageScore, numPlayers: numOfPackPlayers)
    }

    // Updates the UI labels displayed to the user
    private func setNonTableLabels(userRank: String, userScore: String, averageScore: String,
numPlayers: String){
        packLeaderboardHeaderLabel.setTitle("\(selectedPackKey): Top 10", for:
UIControlState())
        userPositionLabel.text = userRank
        userPointsScoreLabel.text = "\(userScore)pts"
        averagePointsScoredLabel.text = "\(averageScore)pts"
        numberOfPlayersLabel.text = numPlayers
    }

    // Retrieves leaderboard data from the database
    private func getLeaderboardData(){
        // Initialises database interaction object
        let dbInteraction = DatabaseInteraction()
        // Tests for internet connectivity
        if dbInteraction.checkInternetAvaliability(){

            // Sets up string to be posted to the database api
            let postData = "packID=\(packID)&userID=\(userID)"

            // The post string is posted to the online database API via the
DatabaseInteraction class on the background thread
            // The "getPackLeaderboardInfo.php" API retrieves all of the leaderboard
information about a pack from the online database
            let responseJSON = dbInteraction.postToDatabase(apiName:
"getPackLeaderboardInfo.php", postData: postData){ (dbResponse: NSDictionary) in

                // Retrieves the database response
                let isError = dbResponse["error"]! as! Bool
                var errorMessage = ""

                // Prepares the error message if there were errors
                if isError{
                    for message in dbResponse["message"] as! [String]{
                        errorMessage = errorMessage + "\n" + message
                    }
                }
                // Returns the execution flow to the main thread
                DispatchQueue.main.async(execute: {
                    if isError{ // If there is an error it is displayed to the user and the
view is dismissed
                        let alertCon = UIAlertController(title: "Error", message:
errorMessage, preferredStyle: .alert)
                        alertCon.addAction(UIAlertAction(title: "Ok", style: .default,
handler: {action in
                            self.dismiss(animated: true, completion: nil)}))
                        self.present(alertCon, animated: true, completion: nil)
                    } else{ // If no errors then the data received is processed and displayed
to the user
```

```
                        self.setLeaderboardData(data: dbResponse as! Dictionary<String, Any>)
                    }
                })
            }
        } else{ // If no internet connectivity, an error message is displayed asking the user
to connect to the internet
            let alertCon = UIAlertController(title: "Error: Couldn't Connect to Database",
message: "No internet connectivity found. Please check your internet connection",
preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Try Again", style: .default, handler: {
action in
                // Recursion is used to recall the getLeaderboardData function until internet
connectivity is restored
                self.getLeaderboardData()
            }))
            self.present(alertCon, animated: true, completion: nil)
        }

    }

    // TableView method that sets the number of rows in the table
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        // Number of rows in the table is the same as the number of records in the
leaderboardRecords array
        return leaderboardRecords.count
    }

    // TableView method which defines what to display in each cell
    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        // Returns a reusable table-view cell object based on the prototype cell defined in
the UI storyboard
        let cell = tableView.dequeueReusableCell(withIdentifier: "leaderboardRecordCell",
for: indexPath) as! LeaderboardRecordTableViewCell

        // Sets up the label in the cell with information about each record
        let record = leaderboardRecords[indexPath.row]
        cell.positionLabel?.text = String(indexPath.row + 1)
        cell.pointsScoreLabel?.text = "\(record["Score"]!)pts"
        cell.playerNameLabel?.text = record["Username"]!

        return cell
    }

    // Connects the back buton in the navigation bar to the code
    // - Dismisses the view when the back button is tapped
    @IBAction func backNavButtonTapped(_ sender: Any) {
        self.dismiss(animated: true, completion: nil)
    }

    // Connects the help button in the navigation bar to the code
    // - Displays an alert informing the user about the leaderboard
    @IBAction func helpButtonTapped(_ sender: Any) {
        displayAlertMessage(alertTitle: "Help: About the Leaderboard", alertMessage: "The top
ten competitive scores of the pack are displayed in the table. At the bottom you can see some
statistics about the pack scores - like the average score and the number of people who have
played the pack competitively.\n\nIf you have played the pack competitively then your ranking
and score also appears at the bottom.\n\nNB If two players score the same score, the user who
joined TownHunt first will be ranked higher")
    }
}
```

Pack Detail View

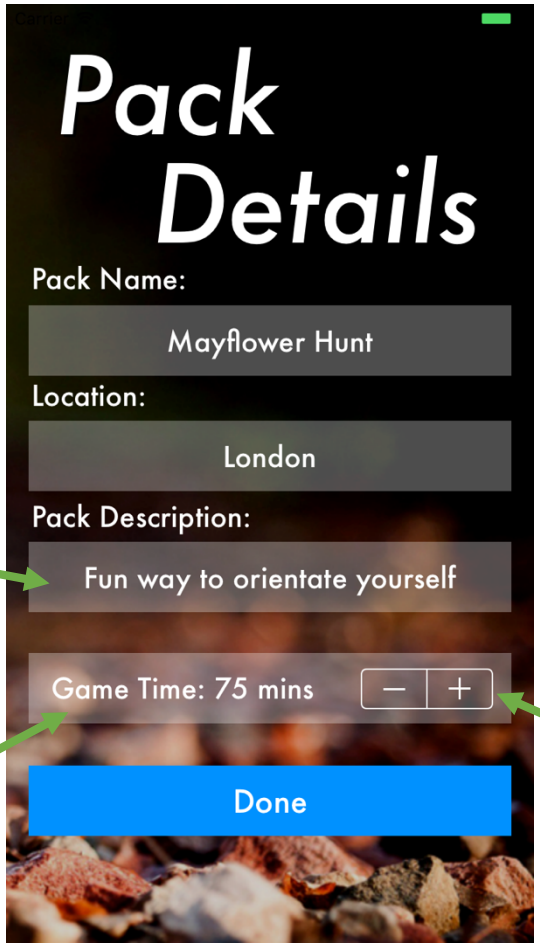


Pack Name Display. Displays the name of the pack loaded

Tapping Edit Details Button in the top screen leads to the bottom screen (edit mode). Similarly tapping the Done button will lead to the top screen (end edit mode).

Pack Location Display. Displays the location of the pack loaded

Back button will dismiss the view



Pack Description Display. Displays the description of the pack loaded. If pack details screen was instantiated via the pack creator the pack description is editable

Stepper increments or decrements the game time by 15mins with a minimum time of 15mins and max of 300 mins (these settings were specified in the storyboard)

Pack game time Display. Displays the description of the pack loaded. If pack details screen was instantiated via the pack creator the pack description is editable

Pack Detail View Controller Code

```
//
//  PackDetailsViewController.swift
//  TownHunt App
//
//  This file defines the behaviour of the pack details view controller

import UIKit // UIKit constructs and manages the app's UI

// Class controls the Pack Details View. Inherits from FormTemplateExtensionOfViewController
class PackDetailsViewController: FormTemplateExtensionOfViewController {

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
        // All are outlets displaying information about the pack
    @IBOutlet weak var packNameTextField: UITextField!
    @IBOutlet weak var briefDescriptionTextField: UITextField!
    @IBOutlet weak var locationTextField: UITextField!
    @IBOutlet weak var timeControlStepper: UIStepper!
    @IBOutlet weak var gameTimeTextLabel: UILabel!
    @IBOutlet weak var gameTimeStaticLabel: UILabel!
        // Connects the back and edit detail button
    @IBOutlet weak var backButton: UIButton!
    @IBOutlet weak var editDetailButton: UIButton!

    // Initialises public attributes
    public var isPackDetailsEditable = true // Is Editable? flag
    public var packDetails = [:] as [String: String] // Holds the pack details

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Sets the background image
        setBackgroundImage(imageName: "packDetailsBackground")

        // Sets the initial game type label value
        timeStepperAction(Any.self)
        // Sets up the form labels with the pack details
        setUpForm()

        // Checks if the is editable flag is false
        if isPackDetailsEditable == false{
            // Pack become uneditable as the edit detail button is hidden
            editDetailButton.isHidden = true
        }
    }

    // Updates the game time label with the current value of the stepper
    @IBAction func timeStepperAction(_ sender: Any) {
        gameTimeTextLabel.text = "Game Time: \(Int(timeControlStepper.value)) mins"
    }

    // Displays the pack details to the user via the UI outlets
    private func setUpForm(){
        changeFormEnabledStatus() // Disables the ability to edit the textfields
        // Sets pack detail labels
        packNameTextField.text = packDetails["PackName"]!
        briefDescriptionTextField.text = packDetails["Description"]!
        locationTextField.text = packDetails["Location"]!
```

```
        let timeLimit = packDetails["TimeLimit"]!
        // Two game time labels for aesthetic reasons, one is shifted to the left when the
stepper is added to the form
        gameTimeTextLabel.text = "Game Time: \(timeLimit) mins"
        gameTimeStaticLabel.text = "Game Time: \(timeLimit) mins"
        timeControlStepper.value = Double(Int(timeLimit)!) // Sets the value of the stepper
        backButton.isHidden = false
    }

    // Changes the fields from being disabled (user-uneditable) to enabled (user-editable)
and vice versa
    private func changeFormEnabledStatus(){
        // Enables/Disables the pack description and pack location from being edited
        briefDescriptionTextField.isUserInteractionEnabled =
!briefDescriptionTextField.isUserInteractionEnabled
        locationTextField.isUserInteractionEnabled =
!locationTextField.isUserInteractionEnabled
        // Hides/Unhides the game time stepper as associated label
        timeControlStepper.isHidden = !timeControlStepper.isHidden
        gameTimeTextLabel.isHidden = !gameTimeTextLabel.isHidden
        backButton.isHidden = !backButton.isHidden
    }

    // Updates pack details with the new data provided by the user
    private func updatePackDetailDict(){
        // Retrieves user input
        let packDescrip = briefDescriptionTextField.text!
        let packLocation = locationTextField.text!
        let gameTime = String(Int(timeControlStepper.value))

        // Changes the pack details array to reflect the updated pack details values
        packDetails["Description"] = packDescrip
        packDetails["Location"] = packLocation
        packDetails["TimeLimit"] = gameTime
    }

    // Enables/Disables the pack details to be edited
    @IBAction func editDetailButtonTapped(_ sender: Any) {
        // Checks which 'mode' the view is in
        if editDetailButton.title(for: UIControlState()) == "Edit Details"{
            // Alert displayed to user indicating that only the pack description and game
time can be edited
            displayAlertMessage(alertTitle: "Editing Pack", alertMessage: "Only the
description and game time can be edited")
            gameTimeStaticLabel.text = ""
            editDetailButton.setTitle("Done", for: UIControlState()) // Sets the edit details
button's text to 'Done'
            changeFormEnabledStatus() // Changes fields to enable editing
        } else{ // User has finished editing
            //Checks the character length of the pack description
            if((briefDescriptionTextField.text!.characters.count) > 30){
                //Displays pack description character length error message
                displayAlertMessage(alertTitle: "Pack Description is Greater Than 30
Characters", alertMessage: "Please enter a description which is less than or equal to 30
characters")
            }else{
                gameTimeStaticLabel.text = gameTimeTextLabel.text
                editDetailButton.setTitle("Edit Details", for: UIControlState()) // Sets the
edit details button's text to 'Edit Details'
                changeFormEnabledStatus() // Changes fields to disable editing
            }
```

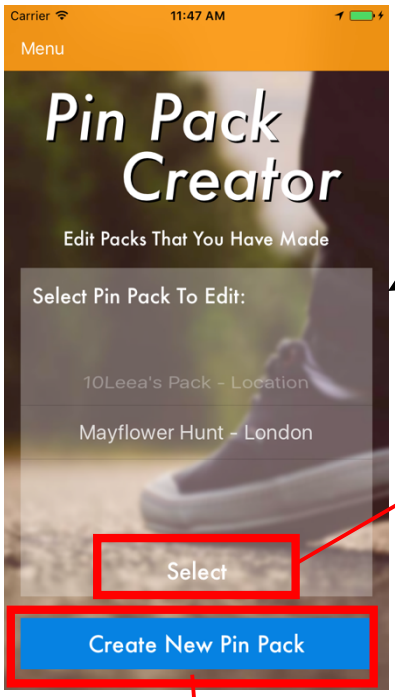
```
        }
    }

    // Connects the back button to the code
    // - Dismisses the pack details view and returns the (updated) back to the view that
presented the pack details view
    @IBAction func backButtonTapped(_ sender: Any) {
        updatePackDetailDict() // Updates the pack details to reflect the user input
        // Checks if the presenting view controller was a navigation controller
        if let destNavCon = presentingViewController as? UINavigationController{
            // Checks if the final destination view controller is the pin pack editor
            if let targetController = destNavCon.topViewController as?
PinPackEditorViewController{
                // Passes the pack details back to the pin pack editor
                for key in Array(packDetails.keys){
                    targetController.packData[key] = packDetails[key]! as AnyObject?
                }
            }
        }
        // Dismisses the pack details view controller
        self.dismiss(animated: true, completion: nil)
    }
}
```

Pin Pack Creator Section

Overview of View Connections & Transitions in This Section

(Red arrows represents a transition (segue) caused by the button which is encapsulated by a red box. Segue identifier is in red text by the transition arrow. However, not all segues have identifiers)

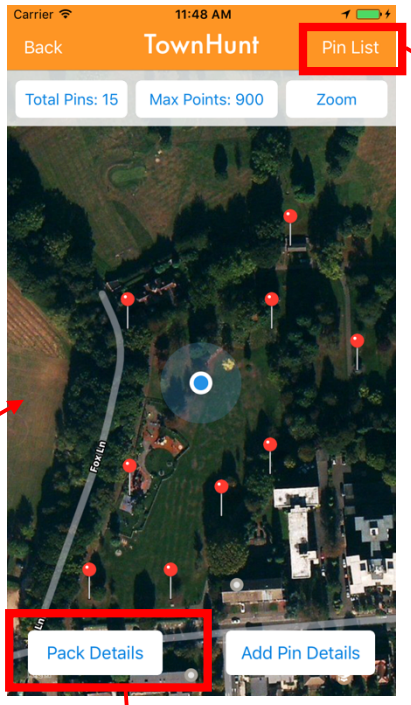
Pack Creator Initial View



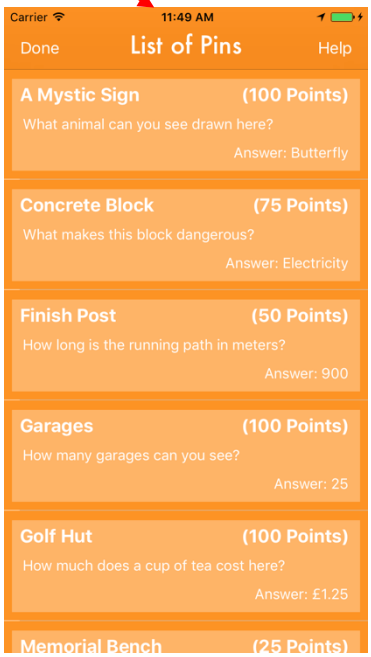
First view in this section i.e. when Pin Pack Creator is selected from the menu this view appears

PackSelectorTo PackEditor

Pack Editor View

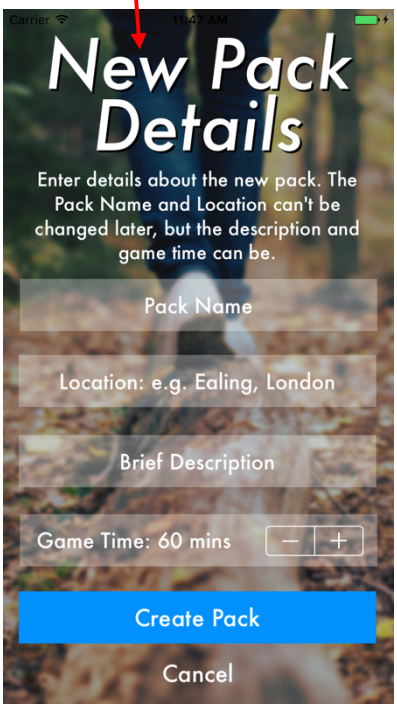


PackEditorToListOfPins

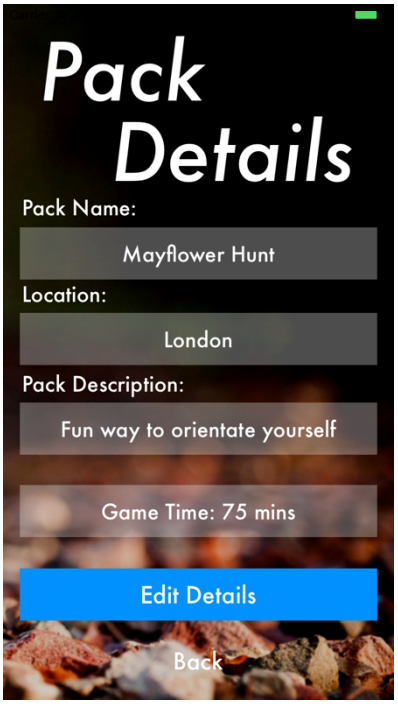


Pin List Table View

PackEditorToPackDetail

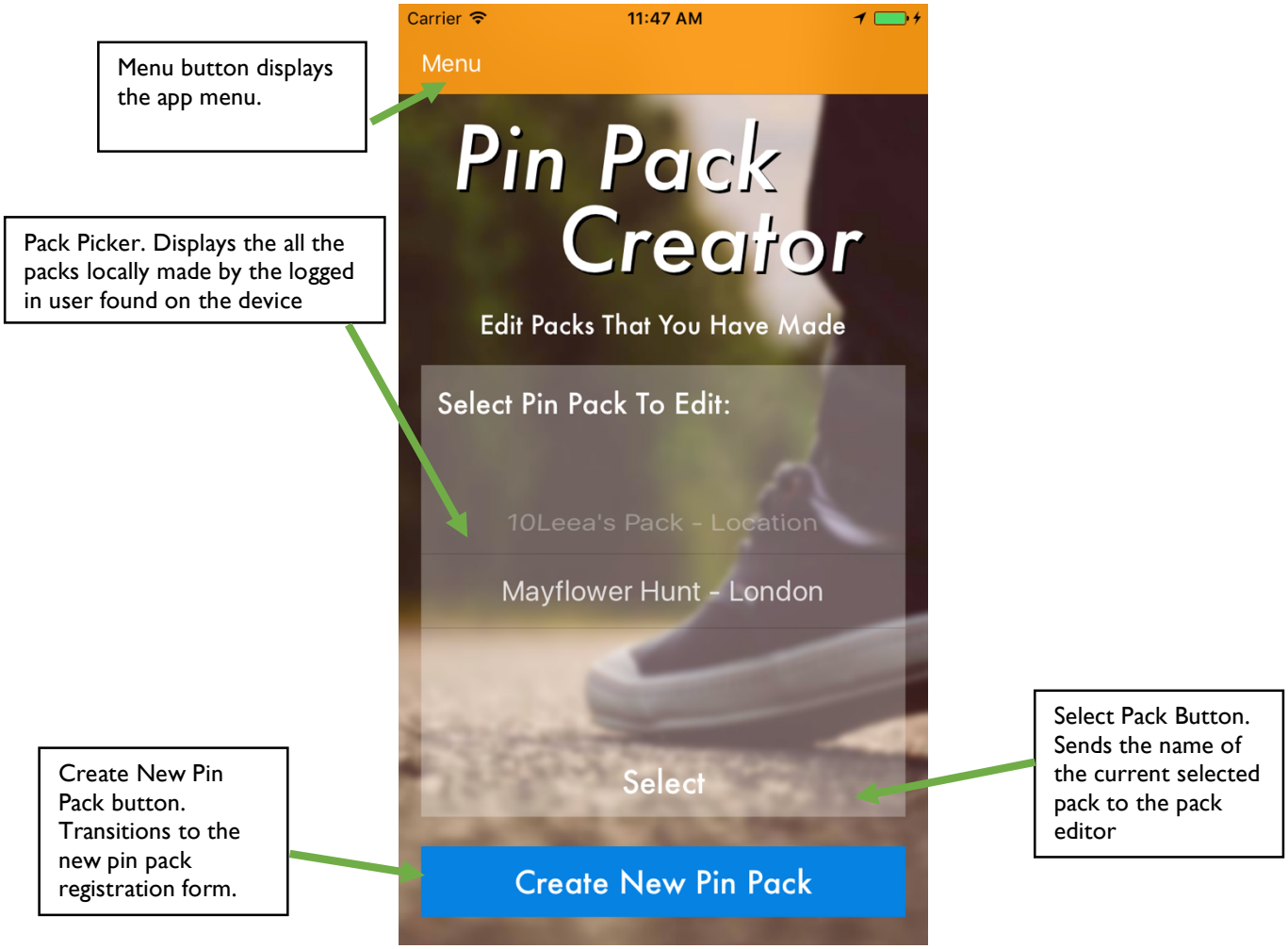


New Pin Pack Registration View



Pack Detail View

Pack Creator Initial View (Pack Selector)



Pack Creator Initial View Controller Code

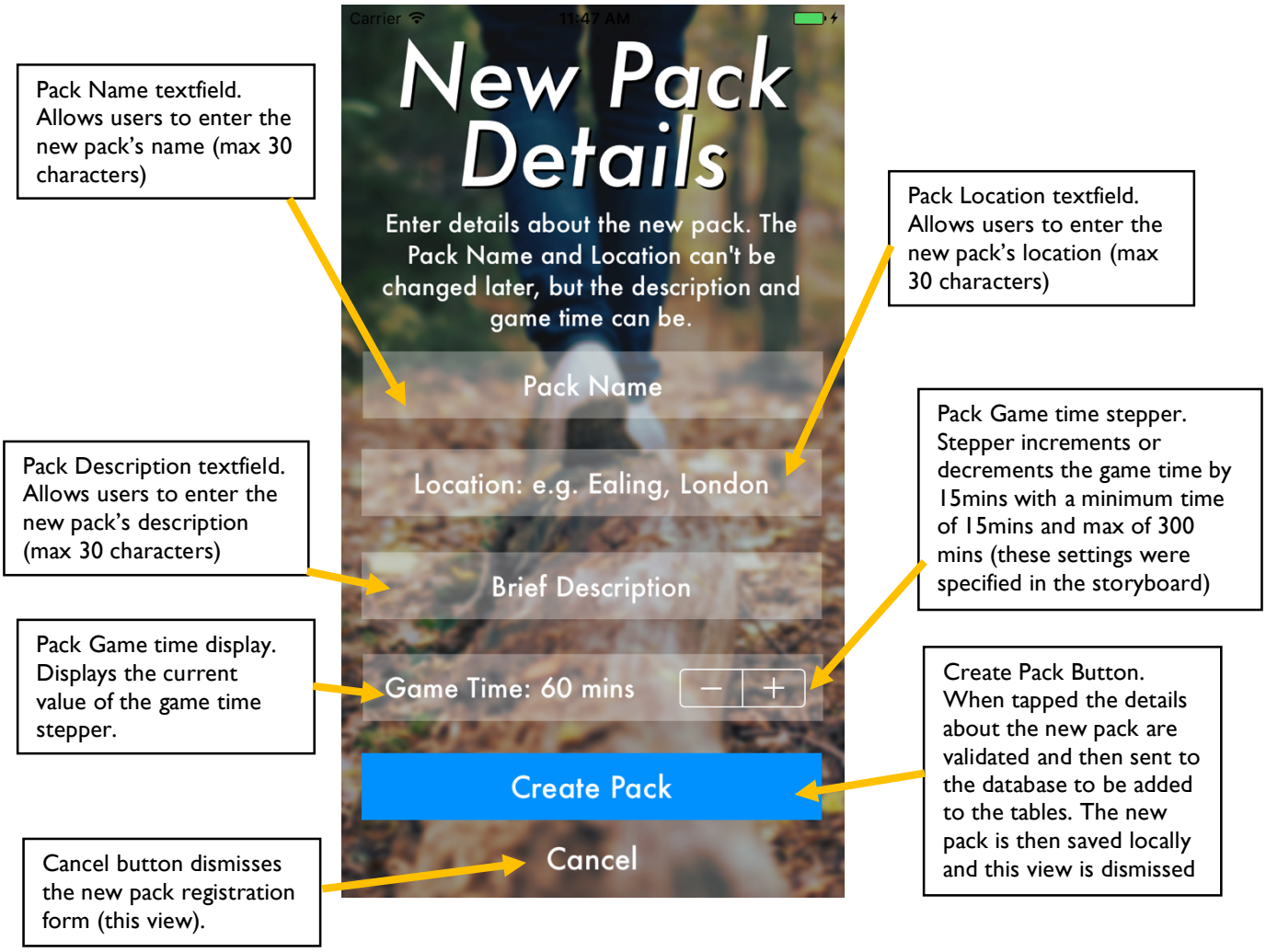
```
//
//  PinPackCreatorInitViewController.swift
//  TownHunt App
//
//  This file defines the behaviour of the pick pack creator initial home screen

import UIKit // UIKit constructs and manages the app's UI

// Class controls the UI elements and logic of the pack creator initial screen. Inherits from
UIViewController, UIPickerViewDelegate
class PinPackCreatorInitViewController: UIViewController, UIPickerViewDelegate,
UIPickerViewDataSource, ModalTransitionListener {

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
    @IBOutlet weak var selectButton: UIButton!
    @IBOutlet weak var menuOpenNavBarButton: UIBarButtonItem! // Menu button on the nav bar
    // The slot-machine-esque selector which displays all of the local packs made by the
logged in user
    @IBOutlet weak var packPicker: UIPickerView!
```

```
    // Initialises attributes relating to the local packs created by the logged in user
stored on the phone
    private var pickerData: [String] = [String]() // Stores the data that will populate the
picker
    private var userPackDictName = "" // Internal dictionary holding the filenames of the
local packs created by the user
    private var selectedPickerData: String = ""
    private var userID = ""

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Singleton which sets up an event listener (listening for the new pin pack
registration form
        // to be dismissed) in this instance of the class
        ModalTransitionMediator.instance.setListener(listener: self)

        // Connects the menu button to the menu screen
        menuOpenNavBarButton.target = self.revealViewController()
        menuOpenNavBarButton.action = #selector(SWRevealViewController.revealToggle(_:))

        // Sets the background image
        setBackgroundImage(imageName: "createPackBackground")

        // Sets up the pack picker
        self.packPicker.delegate = self // Gives this current class the ability to control
the picker UI element
        self.packPicker.dataSource = self // The data source of the picker is set as this
current class
        setUpPicker() // Calls a function to the set up the picker
    }

    // Populates the picker with the names of local packs made by the logged in user
    private func setUpPicker(){
        let defaults = UserDefaults.standard
        userID = defaults.string(forKey: "UserID")! // Retrieves logged in user's id
        userPackDictName = "UserID-\(userID)-Packs"
        // Checks that the dictionary which contains the local packs made by the user exists
        // If this doesn't exist than no local packs made by the user exists
        if let dictOfPacksOnPhone = defaults.dictionary(forKey: userPackDictName) {
            // Sets the picker display data as the pack name-location key, sorted
alphabetically
            self.pickerData = Array(dictOfPacksOnPhone.keys).sorted{ $0.lowercased() <
$1.lowercased() }
            // Sets the initial selected element as the first value of the picker
            self.selectedPickerData = pickerData[0]
            self.selectButton.isHidden = false // User is able to select a pack as there
aren't any
        // If there are no local packs, pack selector button is hidden and a message is shown
to the user
        } else {
            // The only value in the picker is set to "No Packs Found"
            self.pickerData = ["No Packs Found"]
            self.selectButton.isHidden = true // User is unable to select a pack as there
aren't any
        }
    }

    // PickerView method which sets the number of columns of data in the picker
```

```
    func numberOfComponents(in pickerView: UIPickerView) -> Int {
        // The picker used in the app only has one column to display the pack name-location
key
        return 1
    }

    // PickerView method which sets the number of rows of data in the picker
    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) ->
Int {
        // Number of rows is determined by the number of elements in the pickerData array
        return self.pickerData.count
    }

    // PickerView method which sets the picker data to display for a certain row
    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component:
Int) -> String? {
        // Returns the corresponding pack name-location key
        return self.pickerData[row]
    }

    // PickerView method that retrieves the item that is currently selected
    func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component:
Int) {
        // "selectedPickerData" is set to the pack name-location key
        self.selectedPickerData = pickerData[row]
    }

    // PickerView method that sets up how the text of each row should be
    func pickerView(_ pickerView: UIPickerView, viewForRow row: Int, forComponent component:
Int, reusing view: UIView?) -> UIView {
        // Instantiates a label
        let label = UILabel(frame: CGRect(x: 0, y: 0, width: 330, height: 30));
        label.lineBreakMode = .byWordWrapping; // Text won't go off the screen and instead
wrap
        label.numberOfLines = 0 // Allows the label to have unlimited lines
        label.text = pickerData[row] // Sets the text to display as the pack name-location
key
        label.textColor = UIColor.white // Sets the text colour to white
        label.font = UIFont.systemFont(ofSize: CGFloat(20)) // Sets the font to 20 pixels
        label.sizeToFit() // Makes the frame of the label fit the size of the text
        return label // Returns the label to display
    }

    // PickerView method that sets the height of each row
    func pickerView(_ pickerView: UIPickerView, rowHeightForComponent component: Int) ->
CGFloat {
        return 50 // 50 pixels is returned
    }

    // ModalTransitionalListener protocol function which is called when the presented view
(new pack registration page) is dismissed
    func modalViewDismissed(){
        self.navigationController?.dismiss(animated: true, completion: nil)
        self.setUpPicker() // Retrieves the local packs created by the user including the
newly registered pack
        self.packPicker.reloadAllComponents() // Reloads the picker display
    }

    // System function which is called when the view is about to transition (segue) to
another view. This enables data to be passes between views
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
```

```
        // Checks that the segue identifier is "PackSelectorToPackEditor"
        if segue.identifier == "PackSelectorToPackEditor"{
            // Checks if the segue destination view controller is a Navigation controller
            if let navigationController = segue.destination as? UINavigationController{
                // Checks if the target view controller is the pin pack editor
                if let nextViewController = navigationController.topViewController as?
PinPackEditorViewController{
                    // Data passed
                    nextViewController.selectedPackKey = self.selectedPickerData // Pack
name-location key
                    nextViewController.userPackDictName = self.userPackDictName // User packs
dictionary name
                    nextViewController.userID = self.userID // User id
                }
            }
        }
    }

}
```

New Pin Pack Registration View



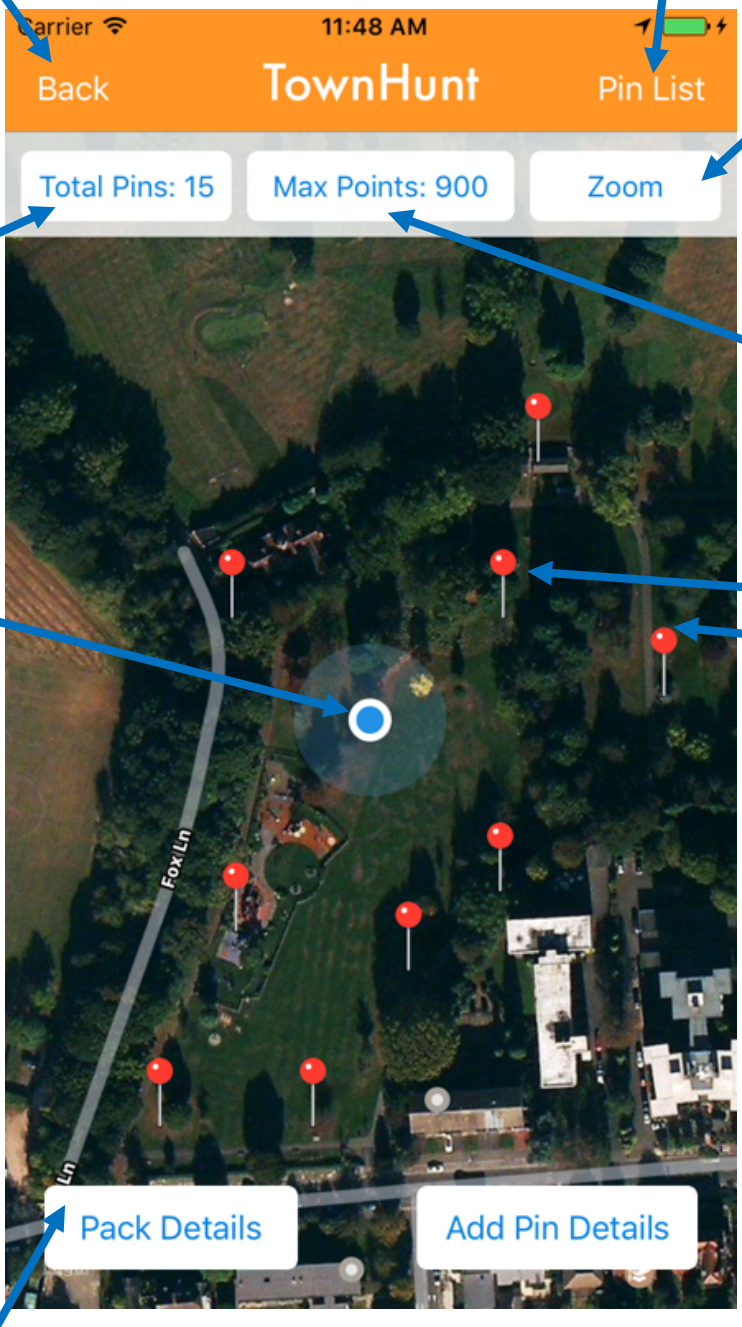
New Pin Pack Registration View Controller Code

```
//
//  NewPinPackCreationViewController.swift
//  TownHunt App
//
//  This file defines the behaviour of the new pin pack registration page

import UIKit // UIKit constructs and manages the app's UI

// Class controls the new pin pack view. Inherits from FormTemplateExtensionOfViewController
class NewPinPackRegisViewController: FormTemplateExtensionOfViewController {

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
    @IBOutlet weak var packNameTextField: UITextField!
    @IBOutlet weak var briefDescripTextField: UITextField!
    @IBOutlet weak var locationTextField: UITextField!
    @IBOutlet weak var gameTimeTextLabel: UILabel!
    @IBOutlet weak var timeControlStepper: UIStepper! // Two buttons ("+" and "-") which can
be used to (de)increment the game time

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Sets the background image
        setBackgroundImage(imageName: "newPackDetailsBackgroundImage")

        // Sets the initial game type label value
        timeStepperAction(Any.self)
    }

    // Connects the stepper to the code
    // - Function is called every time the stepper is tapped. It updates the game time label
    @IBAction func timeStepperAction(_ sender: Any) {
        gameTimeTextLabel.text = "Game Time: \(Int(timeControlStepper.value)) mins"
    }

    // Connect the create pack button to the code
    // - Registers new pack with the database and saves the new pack to the phone
    @IBAction func createPackButtonTapped(_ sender: Any) {
        // Initialises database interaction object
        let dbInteraction = DatabaseInteraction()
        // Tests for internet connectivity
        if dbInteraction.checkInternetAvaliability(){

            // Initialises pack data variables obtained from the registration form
            let packName = packNameTextField.text
            let packLocation = locationTextField.text
            let packDescrip = briefDescripTextField.text
            let creatorID = UserDefaults.standard.string(forKey: "UserID")
            let creatorName = UserDefaults.standard.string(forKey: "Username")
            let gameTime = String(Int(timeControlStepper.value))

            // Check for empty fields
            if((packName?.isEmpty)! || (packLocation?.isEmpty)! || (packDescrip?.isEmpty)!) {
                // Display data entry error message
                displayAlertMessage(alertTitle: "Data Entry Error", alertMessage: "All fields
must be complete")
```

```
            //Checks the character length of the pack name
            } else if((packName?.characters.count)! > 30){
                //Displays pack name character length error message
                displayAlertMessage(alertTitle: "Packname is Greater Than 30 Characters",
alertMessage: "Please enter a pack name which is less than or equal to 30 characters")

            //Checks the character length of the pack location
            } else if((packLocation?.characters.count)! > 30){
                //Displays pack location chararacter length error message
                displayAlertMessage(alertTitle: "Pack Location is Greater Than 30
Characters", alertMessage: "Please enter a location in less than or equal to 30 characters")

            //Checks the character length of the pack description
            } else if((packDescrip?.characters.count)! > 30){
                //Displays pack description chararacter length error message
                displayAlertMessage(alertTitle: "Pack Description is Greater Than 30
Characters", alertMessage: "Please enter a description which is less than or equal to 30
characters")

            } else{
                // A post string is posted to the online database API via the
DatabaseInteraction class on the background thread
                // The "registerNewPinPack.php" API attempts to add the new pin pack details
to the online database and retrieves the new pin pack's id
                let responseJSON = dbInteraction.postToDatabase(apiName:
"registerNewPinPack.php", postData:
"packName=\(packName!)&description=\(packDescrip!)&creatorID=\(creatorID!)&location=\(packLoc
ation!)&gameTime=\(gameTime)"){ (dbResponse: NSDictionary) in

                    // Default error variables initialised
                    var alertTitle = "ERROR"
                    var alertMessage = "JSON File Invalid"
                    var isNewPackRegistered = false

                    // If a database error exists, the  database response message is
presented to the user
                    if dbResponse["error"]! as! Bool{
                        alertTitle = "ERROR"
                        alertMessage = dbResponse["message"]! as! String
                    } // If there is no database error the JSON file is saved
                    else if !(dbResponse["error"]! as! Bool){

                        // Prepares pack details/info to save to local storage
                        let fileName = packName?.replacingOccurrences(of: " ", with: "_")
                        let packInfo = dbResponse["packData"] as! NSDictionary
                        let packID = packInfo["PackID"]! as! String

                        // The NSDictionary which will become the JSON file
                        let jsonToWrite = ["PackName": packName!, "Description":
packDescrip!, "PackID": packID, "Location": packLocation!, "TimeLimit": gameTime, "Creator" :
creatorName!, "CreatorID": creatorID!, "Version": "0", "Pins": []] as [String : Any]

                        // Stores the pack details (JSON file) to local storage via the
LocalStorageHandler class inside the logged in user's folder
                        let storageHandler = LocalStorageHandler(fileName: fileName!,
subDirectory: "UserID-\(creatorID!)-Packs", directory: .documentDirectory)
                        if storageHandler.addNewPackToPhone(packData: jsonToWrite as
NSDictionary){
                            // If storage was successful then the 'successful' alert details
are prepared
                            alertTitle = "Thank You"
```

```
                            alertMessage = "Pack Successfully Created"
                            isNewPackRegistered = true
                        } else{ // If storage was unsuccessful then the 'unsuccessful' alert
details are prepared
                            alertTitle = "ERROR"
                            alertMessage = "Couldn't Save Register Pack"
                        }
                    }
                    // Returns the execution flow to the main thread
                    DispatchQueue.main.async(execute: {
                        // Displays a message to the user indicating the
successful/unsuccessful creation of a new pack
                        let alertCon = UIAlertController(title: alertTitle, message:
alertMessage, preferredStyle: .alert)
                        alertCon.addAction(UIAlertAction(title: "Ok", style: .default,
handler: { action in
                            // Checks if pack was successfully created
                            if isNewPackRegistered{
                                self.dismiss(animated: true, completion: nil) // Exits
registration page
                                // Lets the pack selector know that the form has been
dismissed
                                ModalTransitionMediator.instance.sendModalViewDismissed(model
Changed: true)
                            }}))
                        self.present(alertCon, animated: true, completion: nil)
                    })
                }
            }
        }else{ // If no internet connectivity, an error message is displayed asking the user
to connect to the internet
            let alertCon = UIAlertController(title: "Error: Couldn't Connect to Database",
message: "No internet connectivity found. Please check your internet connection",
preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Try Again", style: .default, handler: {
action in
                // Recursion is used to recall the createPackButtonTapped function until
internet connectivity is restored
                self.createPackButtonTapped(Any.self)
            }))
            self.present(alertCon, animated: true, completion: nil)
        }

    }

    // Connects the cancel button to the code
    // - Dismisses registration form when tapped
    @IBAction func cancelButtonTapped(_ sender: Any) {
        self.dismiss(animated: true, completion: nil)
    }

}
```

Pin Pack Editor View



Back Button causes an alert to be displayed. This alert asks if the user wants to save the changes to the pack. 'Yes' options saves the changes. 'No' doesn't save the changes. Tapping either dismisses the editor. A third option 'Cancel' dismisses the alert

Pin List Button. Tapping this causes the app to transition to the List of Pins Table View. All of the current pins are passed to the table view.

Zoom Button centres the map on the user's location

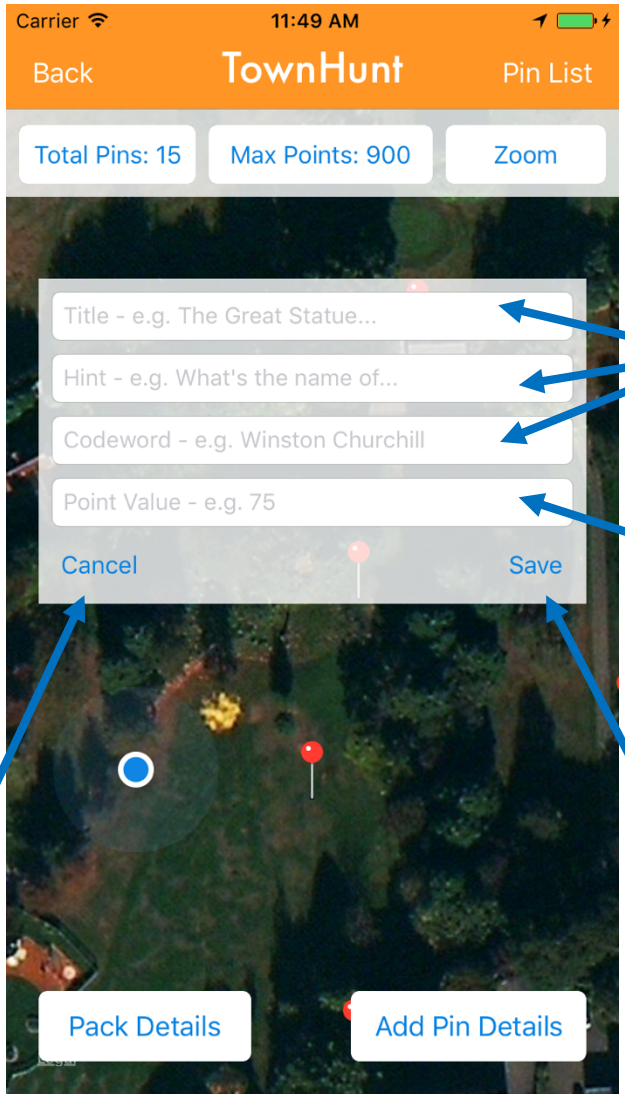
Total Pins Display. This shows the total number of pins currently in the pack

Max Points display. This shows the sum of all the (Pin's) point values currently in the pack

User location taken from their GPS

Pins. All of the pins in the pack are displayed

Pack Details button. This will transition to the pack details view, passing the pack details.



A long tap on the map will drop a generic pin (instantiate an empty PinLocation object). Then the Add Pin Details can be tapped. This centres the map around the new pin and presents the user with a form.

New pin's title, hint and codeword textfields. The details about the new pin can be inputted. (max character length 140, can't be left empty)

New pin's point value textfield can be inputted. When the user taps on this field, a keyboard with only numbers will appear. However, integer type validation still occurs and will raise an error if non-integer is entered (max char length 10, can't be left empty)

Cancel Button. When tapped, the form is dismissed and new pin is removed from the map.

Save Button. When tapped the textfield values are retrieved and added to the PinLocation object which is then added the pack pins array.

Info Bar. If the number of pins is less than 5, this info bar becomes visible and displays a warning that the pack is unplayable

Pack Editor View Controller Code

```
//
//  PinPackEditorViewController.swift
//  TownHunt App
//
//  This file defines the logic behind the pin pack editor

import UIKit // UIKit constructs and manages the app's UI
import MapKit // MapKit constructs and manages the map and annotations

// Class controls the pin pack editor view. Inherits from PinPackMapViewController,
MKMapViewDelegate
class PinPackEditorViewController: PinPackMapViewController, MKMapViewDelegate{

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
        // UI elements found in the info bar (just below the orange nav bar)
    @IBOutlet weak var viewBelowNav: UIView! // Background of the info bar
    @IBOutlet weak var totalPinsButtonLabel: BorderedButton!
    @IBOutlet weak var maxPointsButtonLabel: BorderedButton!
    @IBOutlet weak var packUnplayableWarningButton: UIButton! // Warning that the pack is
currently unplayable

        // UI elements which relate to the new pin information form
    @IBOutlet weak var addPinDetailView: UIView! // The form container
    @IBOutlet weak var pinTitleTextField: UITextField!
    @IBOutlet weak var pinHintTextField: UITextField!
    @IBOutlet weak var pinCodewordTextField: UITextField!
    @IBOutlet weak var pinPointValTextField: UITextField!
        // The map UI element
    @IBOutlet weak var mapView: MKMapView!

    // Initialises attributes which will hold details about the selected pack and creator
    public var selectedPackKey = ""
    public var userPackDictName = ""
    public var userID = ""
    public var filename = ""
    public var packData = [:] as [String: Any]
    public var gamePins: [PinLocation] = []

    // Initialises details about the new pin
    private var newPLat = 0.0 // New Pin latitude coordinate
    private var newPLong = 0.0 // New pin longitude coordinate
    private var isNewPinOnMap = false
    private var newPinCoords = CLLocationCoordinate2D(latitude: 0.0, longitude: 0.0)
    private let newPin = MKPointAnnotation() // New pin map annotation

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Starts the process of retrieving the selected pack data from local storage
        loadInitialPackData()

        // Sets up an event listener (listening for the pin list to close). If the event is
detected then the annotations are refreshed
        NotificationCenter.default.addObserver(self, selector:
#selector(PinPackEditorViewController.refreshAnnotations(_:)),name:NSNotification.Name(rawVal
ue: "load"), object: nil)
```

```
        // Sets up detector to detect where a long press on the screen has occurred. At this
point on the screen a pin will be dropped
        let longPressRecog = UILongPressGestureRecognizer(target: self, action:
#selector(MKMapView.addAnnotation(_:)))
        longPressRecog.minimumPressDuration = 1.0 // Press has to be a min of 1 second
        mapView.addGestureRecognizer(longPressRecog) // Adds the detector to the map

        // Updates the pack info labels
        updatePackLabels()

        // Setting up the map view
        mapView.showsUserLocation = true // Sets up the default map to a satellite map with
road names
        mapView.mapType = MKMapType.hybrid
        mapView.delegate = self // Gives this class the ability to control the map UI element
        mapView.addAnnotations(gamePins) // Adds the pins already in the pack to the map

    }

    // Updates the pack info labels
    private func updatePackLabels(){
        // New pin count is determined and displayed
        totalPinsButtonLabel.setTitle("Total Pins: \(gamePins.count)", for: UIControlState())
        // Checks if the pack is playable i.e. contains more than 5 pins
        // If a pack isn't playable a warning sign will appear on screen
        if gamePins.count >= 5{
            packUnplayableWarningButton.isHidden = true
        } else{
            packUnplayableWarningButton.isHidden = false
        }
        // Calculates the max number of points available in the pack
        var maxPoints = 0
            // Loops through each pin and appends the point value to a running total
        for pin in gamePins{
            maxPoints += pin.pointVal
        }   // Displays the point total
        maxPointsButtonLabel.setTitle("Max Points: \(maxPoints)", for: UIControlState())
    }

    // [---------------------------- Pin Mechanics -------------------------------]

    // Refreshes all of the pins (annotations) on the map
    func refreshAnnotations(_ notification: Notification){
        mapView.addAnnotations(gamePins) // Pins in the pack are added to the map
        updatePackLabels()
    }

    // Resets the new pin details form
    private func resetTextFieldLabels(){
        isNewPinOnMap = false
        // Clears the form
        pinTitleTextField.text = ""
        pinHintTextField.text = ""
        pinCodewordTextField.text = ""
        pinPointValTextField.text = ""
    }

    // Adds a new pin (annotation) to the map if there isn't already one currently on the map
    func addAnnotation(_ gestureRecognizer:UIGestureRecognizer){
        // Checks if there is already a new pin on the map
        if isNewPinOnMap == false{
```

```
            // Gets coordinates of the long tap
            let touchLocation = gestureRecognizer.location(in: mapView)
            newPinCoords = mapView.convert(touchLocation, toCoordinateFrom: mapView)
            newPin.coordinate = newPinCoords // Sets the new pin's coord as where the long
tap occured
            mapView.addAnnotation(newPin) // Adds pin to map
            isNewPinOnMap = true
        }
    }

    // Connects add pin details button to the code
    // - Presents a form to the user where info about the new pin can be entered
    @IBAction func addPinDetailsButton(_ sender: AnyObject) {
        // Checks if there is a new pin on the map
        if isNewPinOnMap == true{
            // Zooms and centres on the new pin
            let region =
MKCoordinateRegionMakeWithDistance(CLLocationCoordinate2D(latitude:  newPinCoords.latitude +
0.0003, longitude:  newPinCoords.longitude), 100, 100)
            mapView.setRegion(region, animated: true)
            // Presents the form
            addPinDetailView.isHidden = false
        } else{ // If no new pin is detected an error message is presented
            displayAlertMessage(alertTitle: "No New Pin On The Map", alertMessage: "A new pin
hasn't been added to the map yet. Long hold on the location you want to place the pin")
        }
    }

    // Connects the cancel adding new pin button to the code
    // - Removes the new pin and dismisses the form to add details
    @IBAction func cancelAddPinDetButton(_ sender: AnyObject) {
        // Dismisses the form
        addPinDetailView.isHidden = true
        view.endEditing(true)
        // Removes the new pin from the map
        mapView.removeAnnotation(newPin)
        // Resets the pin detail form text fields
        resetTextFieldLabels()
    }

    // Connects the save new pin button to the code
    // - Validates the new pin details and appends it to the pack pins array
    @IBAction func saveAddPinDetButton(_ sender: AnyObject) {
        // Checks if the point value entered was an integer
        if let pointNum = Int(pinPointValTextField.text!){

            // Retrieves user input for the pin details
            let pinTitle = pinTitleTextField.text!
            let pinHint = pinHintTextField.text!
            let pinCodeword = pinCodewordTextField.text!

            // Check for empty fields
            if((pinTitle.isEmpty) || (pinHint.isEmpty) || (pinCodeword.isEmpty)) {
                // Displays data entry error message
                displayAlertMessage(alertTitle: "Data Entry Error", alertMessage: "All fields
must be complete")

            //Checks the character length of the pin title
            } else if pinTitle.characters.count > 140{
                // Displays pin title character length error message
```

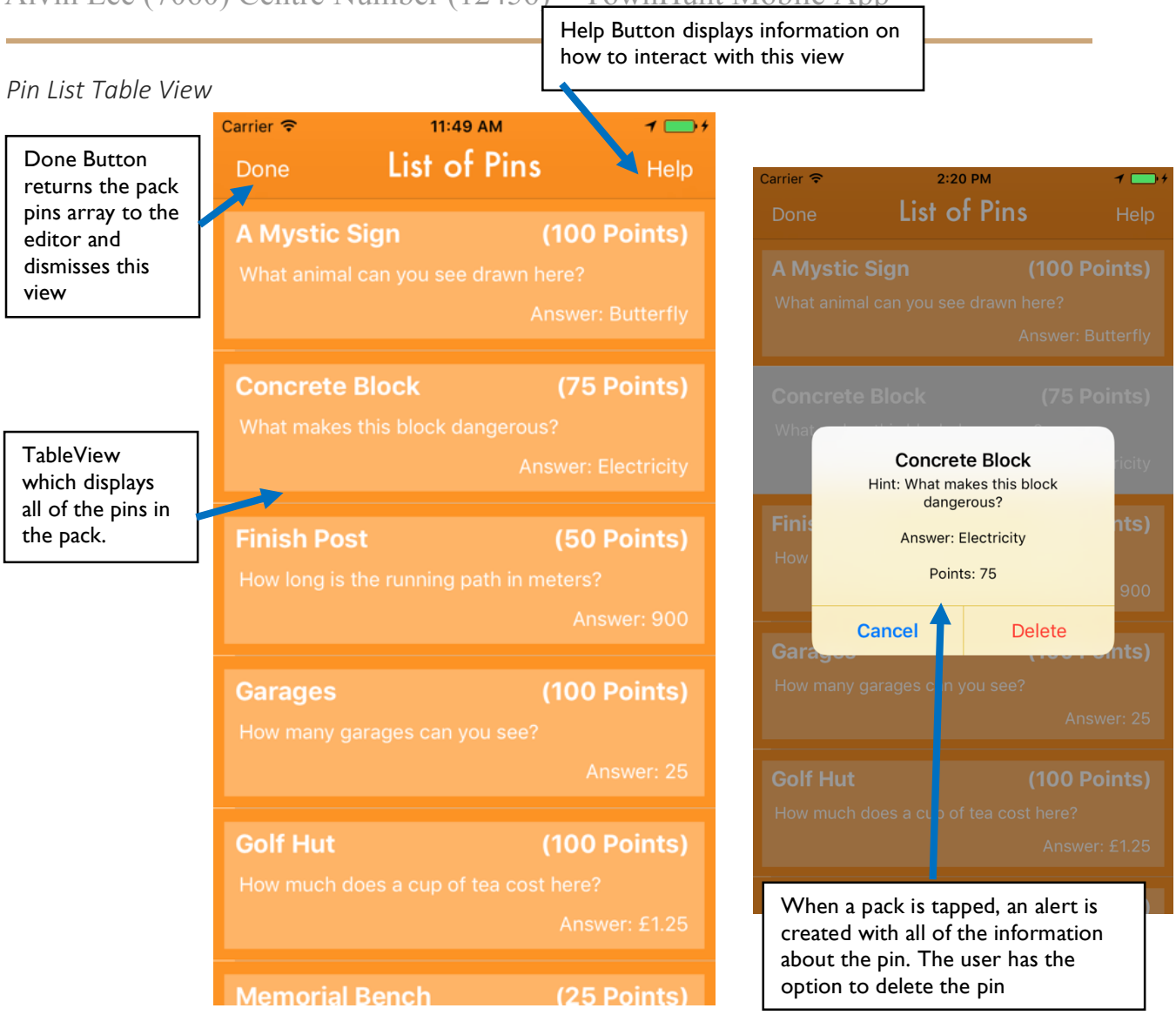
```
                displayAlertMessage(alertTitle: "Pin Title is Greater Than 140 Characters",
alertMessage: "Please enter a title which is less than or equal to 140 characters")

            //Checks the character length of the pin hint
            }else if((pinHint.characters.count) > 140){
                // Displays pin hint character length error message
                displayAlertMessage(alertTitle: "Pin Hint is Greater Than 140 Characters",
alertMessage: "Please enter a hint which is less than or equal to 140 characters")

            //Checks the character length of the pin codeword
            }else if((pinCodeword.characters.count) > 140){
                // Displays pin codeword character length error message
                displayAlertMessage(alertTitle: "Pin Codeword is Greater Than 140
Characters", alertMessage: "Please enter a codeword which is less than or equal to 140
characters")

            //Checks the character length of the pin value
            }else if((String(pointNum).characters.count) > 10){
                // Displays pin value character length error message
                displayAlertMessage(alertTitle: "Pin Value is Greater Than 10 Characters",
alertMessage: "Please enter a point value which is less than or equal to 10 characters")
            } else { // Pin is validated
                // Creates an instance of a PinLocation with the new pins details
                let pin = PinLocation(title: pinTitle, hint: pinHint, codeword: pinCodeword,
coordinate: newPinCoords, pointVal: pointNum)
                // Adds the new pin to the pack pins array
                mapView.addAnnotation(pin)
                mapView.removeAnnotation(newPin)
                addPinDetailView.isHidden = true // Hides the pin details form
                resetTextFieldLabels() // Resets the pin details form
                gamePins.append(pin)
                updatePackLabels()  // Updates pack stats
                view.endEditing(true)
            }
        } else {
            // Displays invalid point value type error
            displayAlertMessage(alertTitle: "Invalid Point Value", alertMessage: "Please
enter a number (integer) into the point value")
        }
    }

    // Retrieves the selected pack details and loads it into the view
    private func loadInitialPackData(){
        let defaults = UserDefaults.standard
        // Checks that the dictionary which contains the local packs made by the user exists
        if let dictOfPacksOnPhone = defaults.dictionary(forKey: userPackDictName){
            filename = dictOfPacksOnPhone[selectedPackKey] as! String // Retrieves file name
            packData = loadPackFromFile(filename: filename, userPackDictName:
userPackDictName, selectedPackKey: selectedPackKey, userID: userID) // Retrieves pack info
            gamePins =  getListOfPinLocations(packData: packData) // Retrieves pins in the
pack
        } else{ // Error message is displayed indicating that there was an error in loading
the file
            displayAlertMessage(alertTitle: "Error", alertMessage: "File Couldn't be loaded")
        }
    }

    // Method which saves the current pack to both local storage and the online database
    private func savePack(){
        // Initialises database interaction object
        let dbInteraction = DatabaseInteraction()
```

```
        // Tests for internet connectivity
        if dbInteraction.checkInternetAvaliability(){

            // Prepares the pack data to save
            var jsonToWrite = packData
            var pinsToSave = [[String: String]]()
            // Retrieves a dictionary about the info of each PinLocation Object and appends
it to a 'pinToSave' list
            for pin in gamePins{
                pinsToSave.append(pin.getDictOfPinInfo())
            }
            jsonToWrite["Pins"] = pinsToSave

            // Stores the pack details (JSON file) to local storage via the
LocalStorageHandler class in the logged in user's folder
            let storageHandler = LocalStorageHandler(fileName: filename, subDirectory:
"UserID-\(packData["CreatorID"]!)-Packs", directory: .documentDirectory)
            let localStorageResponse = storageHandler.saveEditedPack(packData: jsonToWrite as
[String : Any])

            // If there is an error with saving the json file, the user is presented with an
alert with details of the error
            if (localStorageResponse["error"] as! Bool){
                displayAlertMessage(alertTitle: "Error", alertMessage:
localStorageResponse["message"] as! String)
            } else{ // File successfully saved

                // The data to send to the database is received and set up for POSTing to the
API
                let dataToPost = localStorageResponse["data"] as! [String: Any]

                // Converts dictionary into a JSON string
                let convertedDataToPost = "data=\(storageHandler.jsonToString(jsonData:
dataToPost))"
                print(convertedDataToPost)

                // If there is internet connectivity then the data is posted to the online
database API via the DatabaseInteraction class on the background thread.
                // The "updatePinPack.php" API attempts to update the pin pack details
                let responseJSON = dbInteraction.postToDatabase(apiName: "updatePinPack.php",
postData: convertedDataToPost){ (dbResponse: NSDictionary) in

                    // Instantiates the alert details
                    var alertTitle = ""
                    var alertMessage = ""

                    // The alert details is set depending on if there was a database error
                    if dbResponse["error"]! as! Bool{
                        alertTitle = "Error"
                        alertMessage = dbResponse["message"]! as! String
                    } else if !(dbResponse["error"]! as! Bool){
                        alertTitle = "Success"
                        alertMessage = dbResponse["message"]! as! String
                    }
                    // Returns the code execution flow to the main thread
                    DispatchQueue.main.async(execute: {
                        // Displays a message to the user indicating the
successful/unsuccessful creation of a new pack
                        let alertCon = UIAlertController(title: alertTitle, message:
alertMessage, preferredStyle: .alert)
```

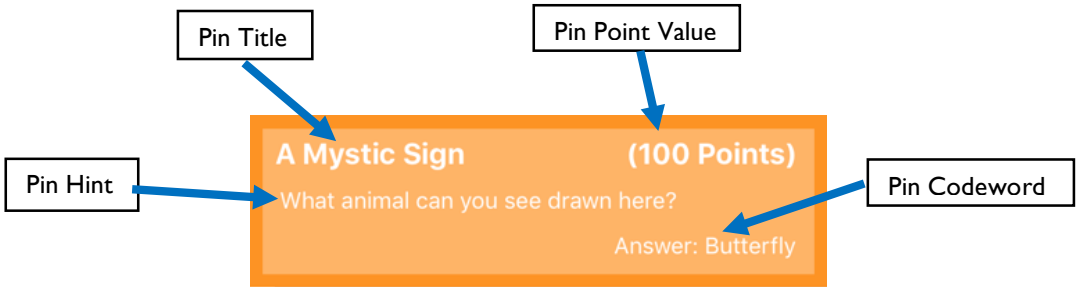
```
                        alertCon.addAction(UIAlertAction(title: "Ok", style: .default,
handler: { action in
                                // Dismisses view
                                self.dismiss(animated: true, completion: nil)
                            }))
                        self.present(alertCon, animated: true, completion: nil)
                    })
                }
            }
        }else{ // If no internet connectivity, an error message is displayed asking the user
to connect to the internet
            let alertCon = UIAlertController(title: "Error: Couldn't Connect to Database",
message: "No internet connectivity found. Please check your internet connection",
preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Try Again", style: .default, handler: {
action in
                // Recursion is used to recall the savePack function until internet
connectivity is restored
                self.savePack()
            }))
            self.present(alertCon, animated: true, completion: nil)
        }

    }

    // [----------------------------System Mechanics--------------------------------------]


    // Connects the Zoom button to the code
    // - This centres the map around the user as well as increasing the magnification of the
map
    @IBAction func zoomButton(_ sender: AnyObject) {
        // Checks if the phone's GPS location is currently available
        if mapView.isUserLocationVisible == true {
            // If user's location is found the map region is set to the 200x200m area around
the user
            let userLocation = mapView.userLocation
            let region =
MKCoordinateRegionMakeWithDistance(userLocation.location!.coordinate, 200, 200)
            mapView.setRegion(region, animated: true)  // Updates the UI map
        } else {
            // If user's location is not found an error message is presented to the user
            displayAlertMessage(alertTitle: "GPS Signal Not Found", alertMessage: "Cannot
zoom on to your location at this moment\n\nIf you have disabled TownHunt from accessing your
location, please go to the settings app and allow TownHunt to access your location")
        }
    }

    // Connects the back button to the code
    // - Dismisses the view, with or without saving the pack depending on the user's choice
    @IBAction func backButtonTapped(_ sender: Any) {
        // An exit menu is displayed
        let alertCon = UIAlertController(title: "Do you want to save the changes to your pack
before leaving?", message: "Select 'Cancel' to return to the editor", preferredStyle:
.actionSheet)
        alertCon.addAction(UIAlertAction(title: "Yes", style: .destructive, handler: { action
in
            // If the user opts to save the pack then the savePack method is called
            self.savePack()
        }))
        alertCon.addAction(UIAlertAction(title: "No", style: .default, handler: { action in
```

```
            // If the user opts to not save the pack then the view is dismissed without any
changes being saved
            self.dismiss(animated: true, completion: nil)
        }))
        alertCon.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: nil)) //
Cancel returns to the editor
        self.present(alertCon, animated: true, completion: nil)
    }

    // System function which is called when the view is about to transition (segue) to
another view. This enables data to be passes between views
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Checks that the segue identifier is "PackEditorToListOfPins"
        if segue.identifier == "PackEditorToListOfPins" {
            // Retrieves the segue destination's navigation controller
            let destNavCon = segue.destination as! UINavigationController
            // Checks if the next view controller is of the PinListInPackTableViewController
class
            if let targetController = destNavCon.topViewController as?
PinListInPackTableViewController{
                targetController.listOfPins = gamePins // Pack pins array is passed
                mapView.removeAnnotations(gamePins) // Map annotations are cleared in the
editor
                gamePins = [] // Game pins array is reset
            }
        // Checks that the segue identifier is "PackEditorToPackDetail"
        } else if segue.identifier == "PackEditorToPackDetail" {
            // Checks if the next view controller is of the PackDetailsViewController class
            if let nextVC = segue.destination as? PackDetailsViewController{
                // The current pack's details are passed without the pins
                var packDetails = packData
                packDetails.removeValue(forKey: "Pins")
                nextVC.packDetails = packDetails as! [String : String]
            }
        }
    }

}
```

Pin List Table View



Pin List Table View Cell



All cells in the table have the same layout. This layout was defined by a prototype cell. For each pin, a new PinCell is instantiated and populated with the pin's details before being added to the table

PinCell Class

```
//
//  PinCell.swift
//  TownHunt App
//
//  File defines the prototype pin detail cell

import UIKit  // UIKit constructs and manages the app's UI

// Class defines the prototype cell that each pin detail cell is based upon. Inherits from
UITableViewCell
class PinCell: UITableViewCell {

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
    @IBOutlet weak var codewordLabel: UILabel!
    @IBOutlet weak var pointValLabel: UILabel!
    @IBOutlet weak var hintLabel: UILabel!
    @IBOutlet weak var titleLabel: UILabel!

    // Called when the cell is loaded into the table
    override func awakeFromNib() {
        // Sets up the functionality of the cell
        super.awakeFromNib()
    }

    // Called when the cell is selected in the table
    override func setSelected(_ selected: Bool, animated: Bool) {
        // Changes the colour of the cell
        super.setSelected(selected, animated: animated)
    }

}
```

Pin List Table View Controller Code

```
//
//  PinListInPackTableViewController.swift
//  TownHunt App
//
//  This file defines the behaviour of the pin list table view

import UIKit // UIKit constructs and manages the app's UI

// Class controls the PinListInPackTable View. Inherits from UITableViewController
class PinListInPackTableViewController: UITableViewController {

    // Initialises attributes which will hold the list of pack pins
    public var listOfPins: [PinLocation]! = []

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        listOfPins = listOfPins.sorted{ ($0.title)!.lowercased() < ($1.title)!.lowercased() }
    }

    // TableView method that sets the number of columns in the table
    override func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }
```

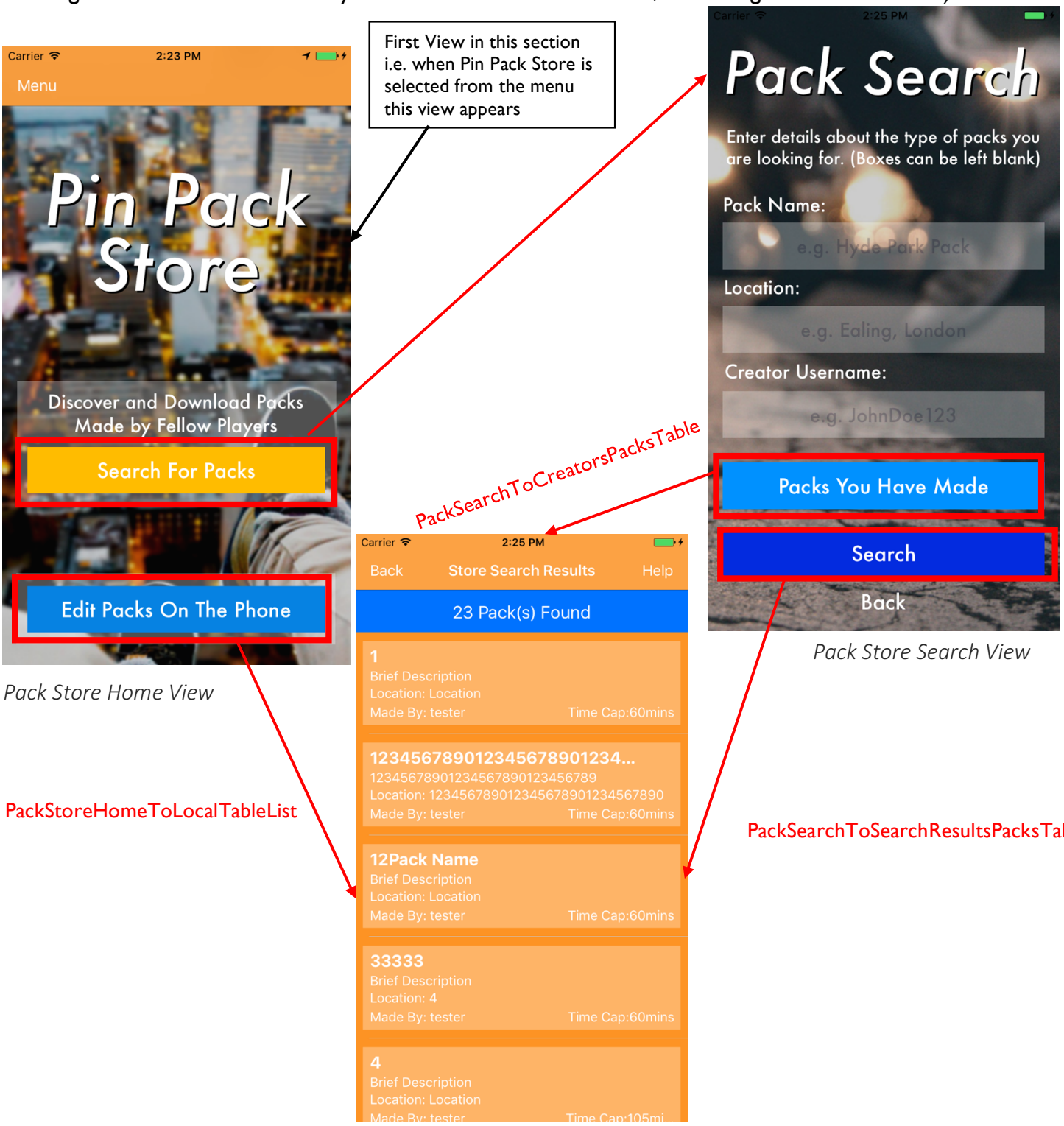
```
    // TableView method that sets the number of rows in the table
    override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) ->
Int {
        // Number of rows in the table is the same as the number of pins in the pack
        return listOfPins.count
    }

    // TableView method which defines what to display in each cell
    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
PinCell {
        // Returns a reusable table-view cell object based on the prototype cell defined in
the UI storyboard
        let cell = tableView.dequeueReusableCell(withIdentifier: "PinCell", for: indexPath)
as! PinCell

        // Sets up the label in the cell with information about each record
        let pin = listOfPins[(indexPath as NSIndexPath).row]
        cell.titleLabel.text = pin.title
        cell.hintLabel.text = pin.hint
        cell.codewordLabel.text = "Answer: \(pin.codeword)"
        cell.pointValLabel.text = "(\(String(pin.pointVal)) Points)"
        return cell
    }

    // TableView method which is called when a row, in the table, is tapped
    override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
        // Retrieves the associated pin
        let selectedPin = listOfPins[indexPath.row]
        // Presents the user with an alert with the details of the pin and the option to
delete the pin
        let alertCon = UIAlertController(title: selectedPin.title, message: "Hint:
\(selectedPin.hint)\n\nAnswer: \(selectedPin.codeword)\n\nPoints:
\(String(selectedPin.pointVal))", preferredStyle: .alert)
        alertCon.addAction(UIAlertAction(title: "Delete", style: .destructive, handler:
{action in
            self.deletePin(indexPath: indexPath) // Deletes the pin
        }))
        alertCon.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: nil))
        self.present(alertCon, animated: true, completion: nil)
    }

    // TableView Method which enables each tow in the table to be slid left
    override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
        // Checks if the user clicked on the delete button after sliding the row to the left
        if editingStyle == UITableViewCellEditingStyle.delete {
            deletePin(indexPath: indexPath) // Deletes the pin
        }
    }

    // Method which deletes the pin from the pack pins array and the table
    private func deletePin(indexPath: IndexPath){
        listOfPins.remove(at: (indexPath as NSIndexPath).row) // Removes pin from pack pins
array
        tableView.deleteRows(at: [indexPath], with: UITableViewRowAnimation.automatic) //
Removes pin form table
    }

    // Connects the Done button to the code
    @IBAction func doneButtonNav(_ sender: AnyObject) {
```

```
        // Checks if the presenting view controller was a navigation controller
        if let destNavCon = presentingViewController as? UINavigationController{
            // Checks if the final destination view controller is the pin pack editor
            if let targetController = destNavCon.topViewController as?
PinPackEditorViewController{
                // Passes the pack pins back to the pin pack editor
                targetController.gamePins = listOfPins
            }
        }
        // Notifies the pin pack editor that the pin list view has been dismissed
        NotificationCenter.default.post(name: Notification.Name(rawValue: "load"), object:
nil)
        self.dismiss(animated: true, completion: nil) // Dismisses the pin list view
    }

    // Connects the help button to the code
    // - Displays in info alert about the pin list view
    @IBAction func helpButtonTapped(_ sender: Any) {
        displayAlertMessage(alertTitle: "Help", alertMessage: "Here is a list of all the pins
in the pack. Swipe left on a pin and tap delete to remove the pin from the pack. Tap a pin to
find out more information about the pin. There is an option to delete the pin here as
well.\n\nOnce you are finished, tap 'Done' to return to the map.")
    }
}
```

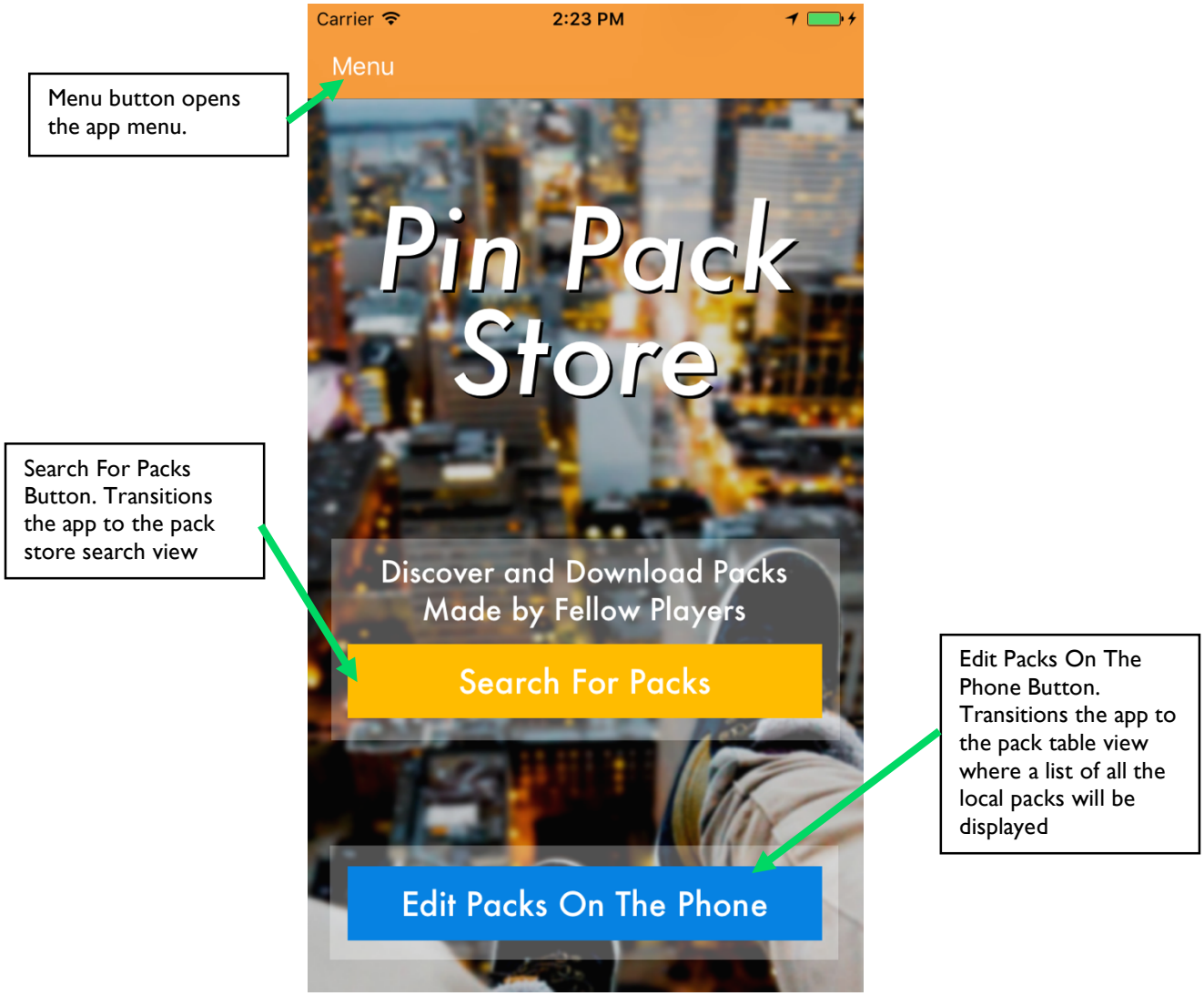
Pin Pack Store Section

Overview of View Connections & Transitions in This Section

(Red arrows represents a transition (segue) caused by the button which is encapsulated by a red box. Segue identifier is in red text by the transition arrow. However, not all segues have identifiers)



Pack Store Home View

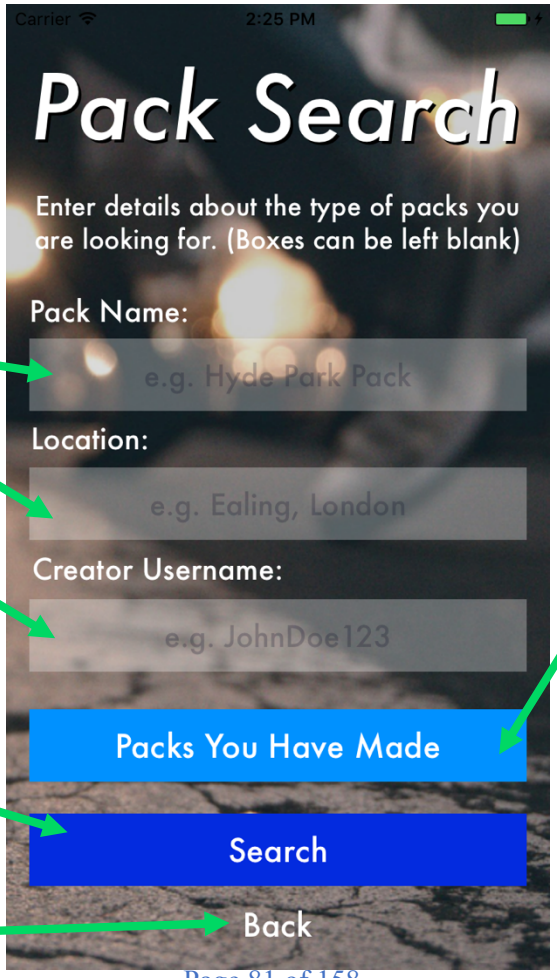


Pack Store Home View Controller Code

```
//
//  PinStoreHomeViewController.swift
//  TownHunt App
//
//  This file defines the logic of the pack store home

import UIKit // UIKit constructs and manages the app's UI

// Class controls the behaviour of the pack store home view. Inherits from UIViewController
class PackStoreHomeViewController: UIViewController {

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
    @IBOutlet weak var menuOpenNavBarButton: UIBarButtonItem! // The menu button

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
```

```
        // Creates the view
        super.viewDidLoad()

        // Sets the background image
        setBackgroundImage(imageName: "packStoreHomeBackground")

        // Connects the menu button to the menu screen
        menuOpenNavBarButton.target = self.revealViewController()
        menuOpenNavBarButton.action = #selector(SWRevealViewController.revealToggle(_:))
    }

    // System function which is called when the view is about to transition (segue) to
another view. This enables data to be passes between views
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Checks that the segue identifier is "PackStoreHomeToLocalTableList"
        if segue.identifier == "PackStoreHomeToLocalTableList"{
            // Checks if the segue destination view controller is a Navigation controller
            if let navigationController = segue.destination as? UINavigationController{
                // Checks if the target view controller is the pin pack store table view
                if let nextViewController = navigationController.topViewController as?
PackStoreListTableViewController{
                    nextViewController.loadLocalPacksFlag = true // Sets the table view's
load local packs flag as true
                    navigationController.title = "List of Local Packs" // Changes the title
of the table view
                }
            }
        }
    }
}
```

Pack Store Search View



Pack Search Criteria fields. Here the user can enter in fragments of the pack name, location and or creator username.

Search Button. When tapped initiates the retrieval of the textfield data and creates a post string, with the search criteria

Back Button dismisses this search form.

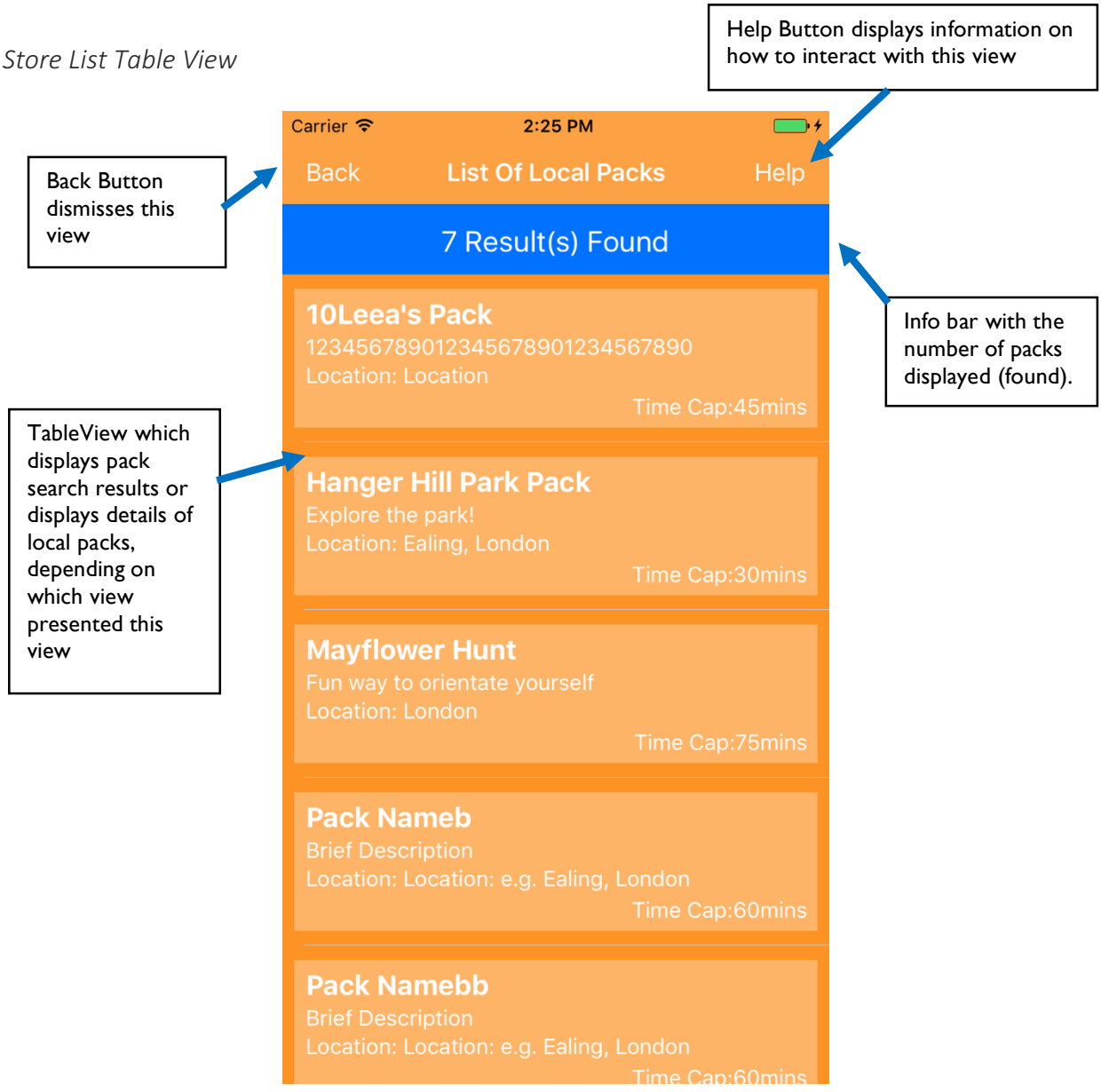
Packs You Have Made Button. When tapped sets the creator username search criteria as the logged in user's username. A post string is then created

Pack Store Search View Controller Code

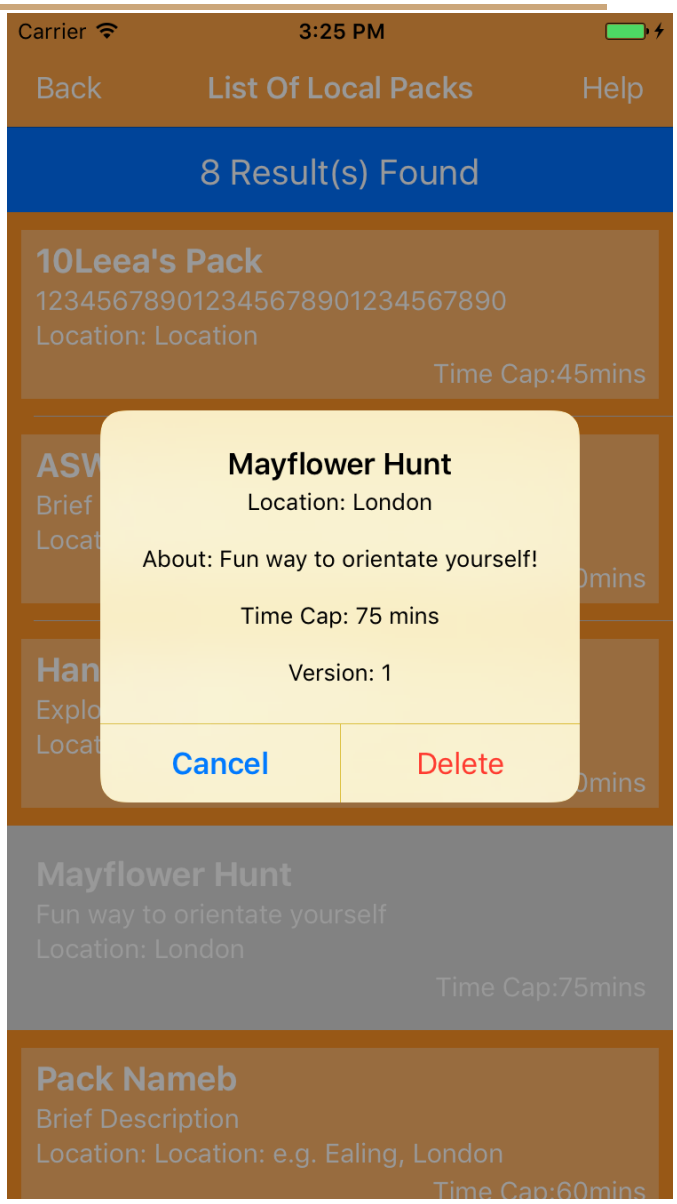
```
//
//  PackSearchViewController.swift
//  TownHunt App
//
//  This file defines the logic of the pack store search

import UIKit // UIKit constructs and manages the app's UI. Inherits from
FormTemplateExtensionOfViewController

// Class controls the pack store search view. Inherits from FormTemplateExtensionOfViewController
class PackStoreSearchViewController: FormTemplateExtensionOfViewController {

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
        // Search textfields
    @IBOutlet weak var packNameTextField: UITextField!
    @IBOutlet weak var locationTextField: UITextField!
    @IBOutlet weak var creatorTextField: UITextField!

    // Attribute containing the search string
    public var searchDataToPost = ""

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Sets the background image
        setBackgroundImage(imageName: "packStoreSearchBackground")
    }

    // Called when 'Pack You Have Made' button tapped
    // - Sets the text fields as empty except the creator text field which is set to the username
of the logged in user
    private func prepareSearchForCreatorsPack(){
        packNameTextField.text = ""
        locationTextField.text = ""
        creatorTextField.text = UserDefaults.standard.string(forKey: "Username") // Retrieves the
username of the logged in user
        prepareSearchDataToPost() // Prepares the search string
    }

    // Generates the search string/data which will be posted to the online database (API)
    private func prepareSearchDataToPost(){
        // Retrieves user input
        let packNameFrag = packNameTextField.text! // Pack name search fragment
        let locationFrag = locationTextField.text! // Location search fragment
        let creatorUsernameFrag = creatorTextField.text! // Creator username search fragment
        // Generates the search string/data
        searchDataToPost =
"usernameFragment=\(creatorUsernameFrag)&packNameFragment=\(packNameFrag)&locationFragment=\(locat
ionFrag)"
    }

    // System function which is called when the view is about to transition (segue) to another
view. This enables data to be passes between views
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Checks if the segue destination view controller is a Navigation controller
        if let destNavCon = segue.destination as? UINavigationController{
            // Checks if the target view controller is the pin pack store table view
            if let targetController = destNavCon.topViewController as?
PackStoreListTableViewController {
                // Checks that the segue identifier is "PackSearchToCreatorsPacksTable"
```

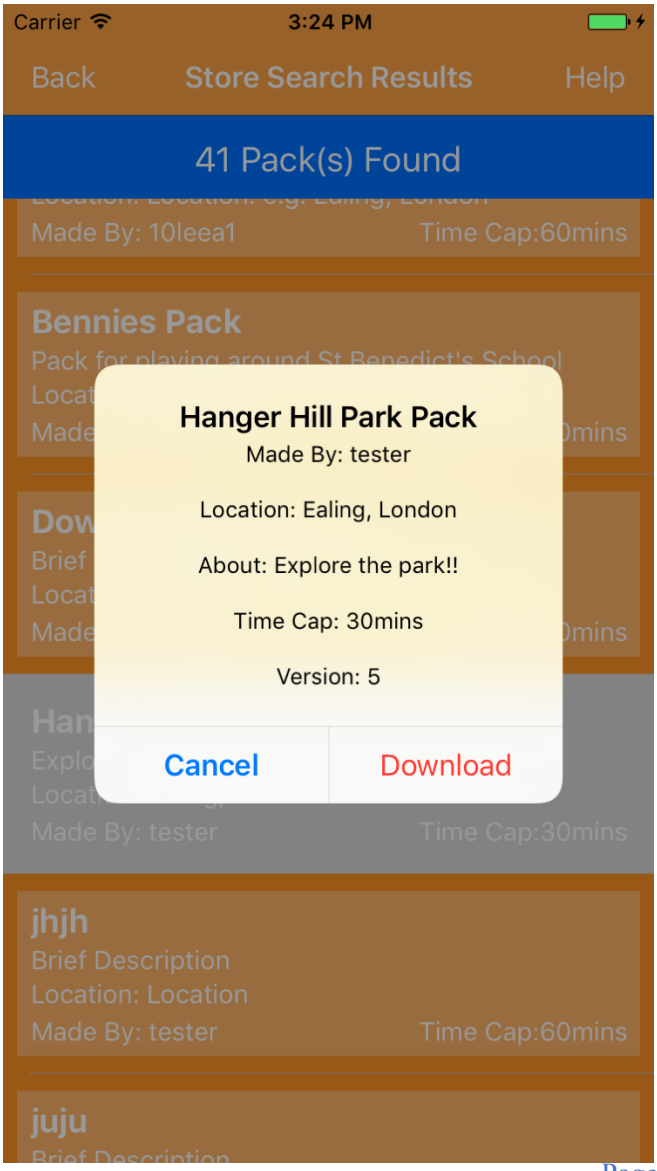
```
                if segue.identifier == "PackSearchToCreatorsPacksTable"{
                    prepareSearchForCreatorsPack()
                // Checks that the segue identifier is "PackSearchToSearchResultsPacksTable"
                } else if segue.identifier == "PackSearchToSearchResultsPacksTable"{
                    prepareSearchDataToPost()
                }
                // Passes the search string to the pack store table view
                targetController.searchDataToPost = self.searchDataToPost
            }
        }
    }

    // Connects the back button to the code
    // - Dismisses the pack store search view when tapped
    @IBAction func backButtonTapped(_ sender: Any) {
        self.dismiss(animated: true, completion: nil)
    }

}
```

Pack Store List Table View



When a pack is tapped an alert is created with all of the information about the pack. If local packs are listed, the user has the option to delete the pin (see image to the right). If a database search result is listed then the user has the option to download the pack (see image below)





Pin Pack Store List Table View Cell Code

```
//  PackListTableViewCell.swift
//  TownHunt App
//
//  File defines the prototype pack detail cell

import UIKit // UIKit constructs and manages the app's UI
// Class defines the prototype cell that each pack detail cell is based upon. Inherits from
UITableViewCell
class PackStoreListTableViewCell: UITableViewCell {

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
    @IBOutlet weak var packNameLabel: UILabel!
    @IBOutlet weak var locationLabel: UILabel!
    @IBOutlet weak var descriptionLabel: UILabel!
    @IBOutlet weak var creatorNameLabel: UILabel!
    @IBOutlet weak var gameTimeLabel: UILabel!

    // Called when the cell is loaded into the table
    override func awakeFromNib() {
        // Sets up the functionality of the cell
        super.awakeFromNib()
    }

    // Called when the cell is selected in the table
    override func setSelected(_ selected: Bool, animated: Bool) {
        // Changes the colour of the cell
        super.setSelected(selected, animated: animated)
    }
}
```

Pin Pack Store List Table View Controller Code

```
//  PackListTableViewController.swift
//  TownHunt App
//
//  This file defines the logic behind the pack store list table screen

import UIKit // UIKit constructs and manages the app's UI
// Class controls the pack store list table view
class PackStoreListTableViewController: UIViewController, UITableViewDelegate,
UITableViewDataSource {

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
    @IBOutlet weak var infoBarButton: UIButton! // Info bar just below the orange navigation
bar
    @IBOutlet var packListTable: UITableView! // The table itself

    // This flag determines if the local packs from the phone should be loaded
    public var loadLocalPacksFlag = false
    // Attribute stores the search string to post
    public var searchDataToPost = ""
    // The data source of the table - containing a list of packs and their respective details
    public var packListTableData = [[String: String]]()
    // Holds the selected pack's details
    public var selectedPackDetails = [String:Any]()

    // Called when the view controller first loads. This method sets up the view
```

```
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Sets up the leaderboard table
        packListTable.delegate = self // Gives this current class the ability to control the
table UI element
        packListTable.dataSource = self // The data source of the table is set as this
current class

        // Checks to see if there is a search string
        if !searchDataToPost.isEmpty{
            self.navigationItem.title = "Store Search Results" // Changes title
            searchDatabaseForPacks() // Sends the search string to the online database API
        // Checks if the loadLocalPacks flag is true
        } else if loadLocalPacksFlag == true{
            self.navigationItem.title = "List Of Local Packs" // Changes title
            loadLocalPacksIntoView() // Retrieves all of the pack on the phone
        }
    }

    // [--------------------------- Table Mechanics ---------------------------------------]

    // Sorts the pack list array and refreshed the table
    private func loadDataIntoTable(data: [[String: String]]){
        // Sorts the pack list in alphabetical order
        packListTableData = data.sorted{ ($0["PackName"])!.lowercased() <
($1["PackName"])!.lowercased() }
        packListTable.reloadData() // Refreshes the table
    }

    // TableView method that sets the number of rows in the table
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        // Number of rows in the table is the same as the number of packs in the
packListTableData array
        return packListTableData.count
    }

    // TableView method which defines what to display in each cell
    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        // Returns a reusable table-view cell object based on the prototype cell defined in
the UI storyboard
        let cell = tableView.dequeueReusableCell(withIdentifier: "packInfoCell", for:
indexPath) as! PackStoreListTableViewCell

        // Sets up the label in the cell with information about each pack
        let pack = packListTableData[indexPath.row]
        cell.packNameLabel?.text = pack["PackName"]!
        cell.locationLabel?.text = "Location: \(pack["Location"]!)"
        cell.descriptionLabel?.text = pack["Description"]!
        // Creator username is optional as it is not stored for local packs
        if let creatorUsername = pack["CreatorUsername"]{
            cell.creatorNameLabel?.text = "Made By: \(creatorUsername)"
        } else{
            cell.creatorNameLabel?.text = ""
        }
        cell.gameTimeLabel?.text = "Time Cap:\(pack["TimeLimit"]!)mins"

        return cell
    }
```

```
    // TableView method which is called when a row is selected
    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {

        // Retrieves the selected row's pack details
        selectedPackDetails = packListTableData[indexPath.row]
        // Checks the load local packs flag to determine what options to present the user
with
        if loadLocalPacksFlag == true{ // Local packs are therefore loaded into the table

            // Alert is presented to the user with the full pack details of the selected
pack, there is an option to delete the pack from the phone
            let alertCon = UIAlertController(title: selectedPackDetails["PackName"]! as?
String, message: "Location: \(selectedPackDetails["Location"]!)\n\nAbout:
\(selectedPackDetails["Description"]!)!\n\nTime Cap: \(selectedPackDetails["TimeLimit"]!)
mins\n\nVersion: \(selectedPackDetails["Version"]!)", preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Delete", style: .destructive, handler:
{action in
                // If the user taps delete then the pack details are sent to the
deleteSelectedLocalPack function
                self.deleteSelectedLocalPack(packName: self.selectedPackDetails["PackName"]!
as! String, packLocation: self.selectedPackDetails["Location"]! as! String, creatorID:
self.selectedPackDetails["CreatorID"]! as! String)
            }))
            alertCon.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: nil))
            self.present(alertCon, animated: true, completion: nil)

        } else{ // The packs loaded in the table are search results
            // Alert is presented to the user with the full pack details of the selected
pack, there is an option to download the pack to the phone
            let alertCon = UIAlertController(title: selectedPackDetails["PackName"]! as?
String, message: "Made By: \(selectedPackDetails["CreatorUsername"]!)\n\nLocation:
\(selectedPackDetails["Location"]!)\n\nAbout: \(selectedPackDetails["Description"]!)!\n\nTime
Cap: \(selectedPackDetails["TimeLimit"]!)mins\n\nVersion:
\(selectedPackDetails["Version"]!)", preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Download", style: .destructive, handler:
{action in
                // If the user taps download then the pack id is sent to the
downloadPackFromDB function
                self.downloadPackFromDB(packID: self.selectedPackDetails["PackID"]! as!
String)}))
            alertCon.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: nil))
            self.present(alertCon, animated: true, completion: nil)
        }
    }

    // [---------------------------- Local Storage Mechanics ------------------------------]

    // Generates (and returns) the pack file name and subdirectory
    private func getFileNameAndSubDirString(packName: String, creatorID: String) ->
[String:String]{
        let fileName = (packName).replacingOccurrences(of: " ", with: "_")
        let subDirect = "UserID-\(creatorID)-Packs"
        return ["fileName":fileName, "subDirectory": subDirect]
    }

    // Retrieves local pack data
    private func loadLocalPacksIntoView(){
        packListTableData = [[String: String]]() // Clears pack list table data
        let defaults = UserDefaults.standard
        // Retrieves the list of user ids whose packs are found in local storage
```

```
        if let listOfUsersOnPhone = defaults.array(forKey: "listOfLocalUserIDs"){
            // Iterates over every id and retrieves all packs on the phone
            for userID in listOfUsersOnPhone as! [String]{
                let userPackSubDirectory = "UserID-\(userID)-Packs"
                // Checks if the user pack dictionary exists
                if let displayNameFilenamePairs = defaults.dictionary(forKey:
userPackSubDirectory) {
                    // Appends all pack names and their creator user id to the allPacksDict
                    for filename in Array(displayNameFilenamePairs.values){
                        // Instantiates an instance of the LocalStorageHandler class
                        let storageHandler = LocalStorageHandler(fileName: filename as!
String, subDirectory: userPackSubDirectory, directory: .documentDirectory)
                        // Retrieves the pack info
                        var packOnPhone = storageHandler.retrieveJSONData() as! [String:Any]
                        packOnPhone.removeValue(forKey: "Pins") // Removes the pins as we are
only concerned with the pack details right now
                        packListTableData.append(packOnPhone as! [String:String]) // Appends
to the table data source
                    }
                }
            }
        } // Loads the data into the UI table
        loadDataIntoTable(data: packListTableData)
        // Checks if any packs were found
        if packListTableData.isEmpty{ // No packs were found
            updateInfoBarMessage(message: "0 Results Found")
            // Message is displayed to the user
            let alertCon = UIAlertController(title: "Error", message: "No packs found on the
device. Search for and download some packs!", preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Ok", style: .default, handler: {action
in
                // Pack store table view is dismissed
                self.dismiss(animated: true, completion: nil) }))
            self.present(alertCon, animated: true, completion: nil)
        }else{ // Packs were found
            // Number of results is generated and displayed
            updateInfoBarMessage(message: "\(packListTableData.count) Result(s) Found")
        }
    }

    // Prepares the pack data to be saved as a json file
    private func preparePackToSave(pins: [[String:String]], didContainPins: Bool){
        // Initiates JSON array
        var jsonToWrite = selectedPackDetails
        jsonToWrite.removeValue(forKey: "CreatorUsername")

        // Checks if there are pins in the pack
        if didContainPins{
            jsonToWrite["Pins"] = pins
        } else{ // Otherwise an empty array will be stored
            jsonToWrite["Pins"] = []
        }

        // Converts pack details into the correct filename and subdirectory
        let filePathDetails = getFileNameAndSubDirString(packName: jsonToWrite["PackName"]
as! String, creatorID: jsonToWrite["CreatorID"] as! String)

        // Instantiates a LocalStorageHandler instance
        let storageHandler = LocalStorageHandler(fileName: filePathDetails["fileName"]!,
subDirectory: filePathDetails["subDirectory"]!, directory: .documentDirectory)
```

```
        // Checks if the pack has already been downloaded and a file for the pack already
exists
        if storageHandler.getDoesFileExist() == true{
            // User is alerted about the file's existence and asked if s/he wants to override
it
            let alertCon = UIAlertController(title: "A Version Of The Pack Exists On The
Phone", message: "Do you want to overwrite the version on the phone with the downloaded
pack?", preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: nil))
            alertCon.addAction(UIAlertAction(title: "Yes", style: .destructive, handler:
{action in
                // If yes is selected then the file overrides the existing file
                self.savePackToLocalStorage(storageHandler: storageHandler, dataToWrite:
jsonToWrite as NSDictionary)}))
            self.present(alertCon, animated: true, completion: nil)
        } else{
            //Stores the pack details (JSON file) to local storage via the
LocalStorageHandler class
            self.savePackToLocalStorage(storageHandler: storageHandler, dataToWrite:
jsonToWrite as NSDictionary)
        }
    }

    // Saves packs (in JSON format) to the local storage
    private func savePackToLocalStorage(storageHandler: LocalStorageHandler, dataToWrite:
NSDictionary){
        // If there is an error with saving the json file, the user is presented with an
alert with details of the error
        if storageHandler.addNewPackToPhone(packData: dataToWrite as NSDictionary){
            displayAlertMessage(alertTitle: "Success", alertMessage:
"\(dataToWrite["PackName"]! as! String) Saved To Phone")
        } else{
            displayAlertMessage(alertTitle: "Error", alertMessage:
"\(dataToWrite["PackName"]! as! String) Wasn't 'Saved To Phone")
        }
    }

    // Deletes the passed pack file from the local storage
    private func deleteSelectedLocalPack(packName: String, packLocation: String, creatorID:
String){
        // Converts pack details into the correct filename and subdirectory
        let filePathDetails = getFileNameAndSubDirString(packName: packName, creatorID:
creatorID)
        // Instantiates a LocalStorageHandler instance
        let storageHandler = LocalStorageHandler(fileName: filePathDetails["fileName"]!,
subDirectory: filePathDetails["subDirectory"]!, directory: .documentDirectory)
        // Attempts to delete the file
        if storageHandler.deleteFile(packName: packName, packLocation: packLocation,
creatorID: creatorID) {
            // User is alerted about the successful deletion
            displayAlertMessage(alertTitle: "Success", alertMessage: "\(packName) was deleted
from the phone")
            loadLocalPacksIntoView() // Local pack list table is reloaded
        } else{
            // User is alerted about the unsuccessful deletion
            displayAlertMessage(alertTitle: "Error", alertMessage: "\(packName) could not be
deleted from the phone")
        }}

    //[ ------------------------- Online Database Mechanics ----------------------------]
```

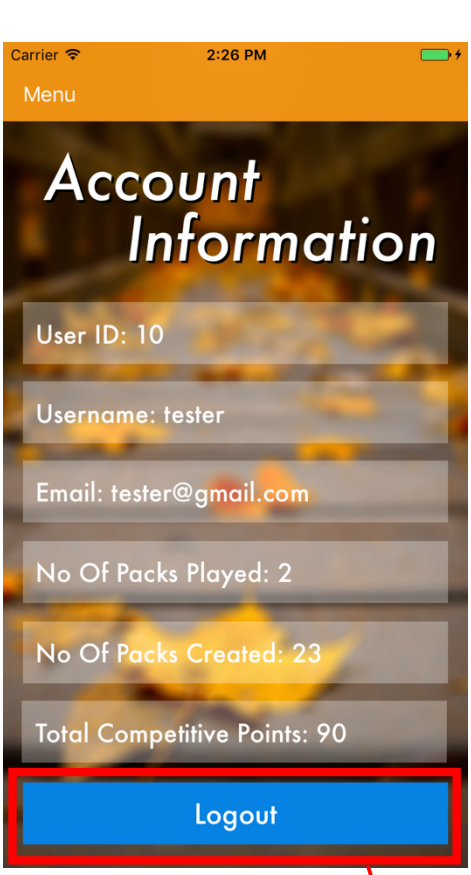
```
    // Sets up the database interaction to download a pack from the online database
    private func downloadPackFromDB(packID: String){
        // The "getPinsFromPack.php" API returns all of the pins found for a certain pack id
from the online database
        retrieveDataFromDatabase(api: "getPinsFromPack.php", postData: "packID=\(packID)")
    }

    // Sets up the database interaction to search for packs from the online database
    private func searchDatabaseForPacks(){
        // The "packSearch.php" API returns a list of pack details that is similar to the
search fragments input by the user
        retrieveDataFromDatabase(api: "packSearch.php", postData: searchDataToPost)
    }

    // Retrieves data from the online database
    private func retrieveDataFromDatabase(api: String, postData: String){
        // Initialises database interaction object
        let dbInteraction = DatabaseInteraction()
        // Tests for internet connectivity
        if dbInteraction.checkInternetAvaliability(){

            // A post string is posted to the online database API via the DatabaseInteraction
class on the background thread
            let responseJSON = dbInteraction.postToDatabase(apiName: api, postData:
postData){ (dbResponse: NSDictionary) in
                // Database response is interpreted

                // Default error variables initialised
                let isError = dbResponse["error"]! as! Bool
                var errorMessage = ""

                // Checks if there is an error, if there is then the error message is
retrieved
                if isError{
                    errorMessage = dbResponse["message"]! as! String
                }

                // Returns the code execution flow back to the main thread
                DispatchQueue.main.async(execute: {
                    // Displays a message to the user indicating the successful/unsuccessful
creation of a new pack
                    // Checks if there was an error with the database interaction
                    if isError{
                        // Alerts the user of the error
                        let alertCon = UIAlertController(title: "Error", message:
errorMessage, preferredStyle: .alert)
                        alertCon.addAction(UIAlertAction(title: "Ok", style: .default,
handler: {action in
                            // If the database interaction was a search request then the pack
store list table view is dismissed
                            if api == "packSearch.php"{ self.dismiss(animated: true,
completion: nil) }}))
                        self.present(alertCon, animated: true, completion: nil)
                    } else{ // No errors found
                        // Checks if the database interaction was a search request
                        if api == "packSearch.php"{
                            // The search result (list of packs) is retrieved
                            let searchResults = dbResponse["searchResult"] as! [[String:
String]]
                            self.loadDataIntoTable(data: searchResults) // Search result is
loaded into the table
```

```
                            self.updateInfoBarMessage(message: "\(searchResults.count)
Pack(s) Found") // Number of results is displayed
                        // Checks if the database interaction was to download the pins from a
pack
                        } else if api == "getPinsFromPack.php"{
                            // Pin contains flag is set
                            let packContainsPinsFlag = dbResponse["packContainsPinsFlag"]!
as! Bool
                            // Checks if the pack contains pins
                            if  packContainsPinsFlag { // Pins are saved
                                self.preparePackToSave(pins: dbResponse["Pins"] as! [[String:
String]], didContainPins: true)
                            } else{ // An empty array for the pins are saved
                                self.preparePackToSave(pins: [[:]], didContainPins: false)
                            }
                        }
                    }
                })
            }
        } else{ // If no internet connectivity, an error message is displayed asking the user
to connect to the internet
            let alertCon = UIAlertController(title: "Error: Couldn't Connect to Database",
message: "No internet connectivity found. Please check your internet connection",
preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Try Again", style: .default, handler: {
action in
                // Recursion is used to recall the retrieveDataFromDatabase function until
internet connectivity is restored
                self.retrieveDataFromDatabase(api: api, postData: postData)
            }))
            self.present(alertCon, animated: true, completion: nil)
        }
    }

    //[---------------------- System Buttons -------------------------------]

    // Changes the title (text) displayed in the number of results info bar
    private func updateInfoBarMessage(message: String){
        infoBarButton.setTitle(message, for: UIControlState())
    }

    // Connects the back button with the code
    // - Dismisses the view when tapped
    @IBAction func backNavBarButtonTapped(_ sender: Any) {
        self.dismiss(animated: true, completion: nil)
    }

    // Connects the help button with the code
    // - Displays an alert with information about how to interact with the table
    @IBAction func helpButtonTapped(_ sender: Any) {
        var alertMessage = "Tap a pack to"
        if loadLocalPacksFlag == true{
            alertMessage = alertMessage + " delete it from the phone"
        } else{
            alertMessage = alertMessage + " download it onto the phone"
        }
        displayAlertMessage(alertTitle: "Help", alertMessage: alertMessage)
    }
}
```

Account Information/Login Section

Overview of View Connections & Transitions in This Section

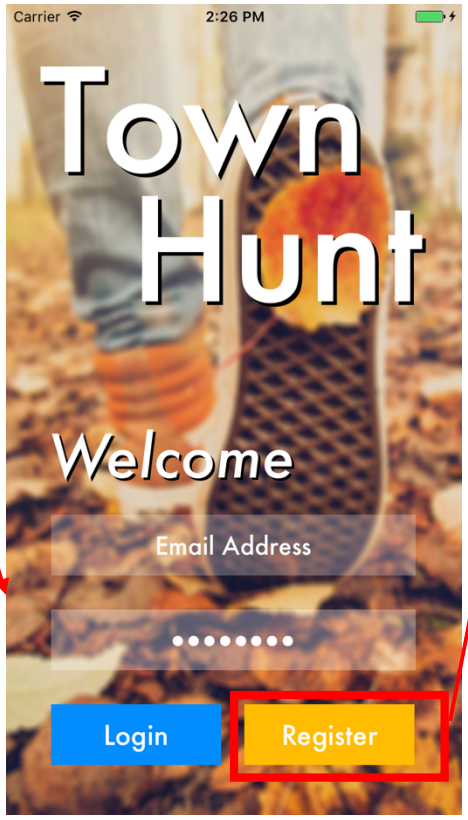
(Red arrows represents a transition (segue) caused by the button which is encapsulated by a red box. Segue identifier is in red text by the transition arrow. However, not all segues have identifiers)



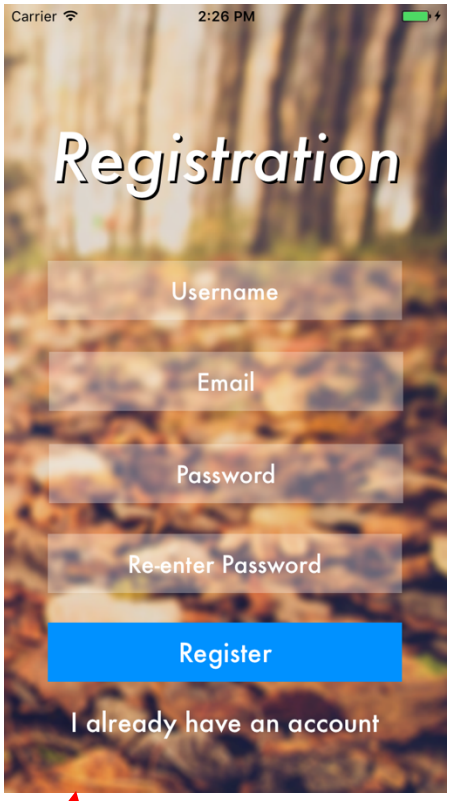
First view in this section i.e. when the Account Page is selected from the menu this view appears

Account Info Page View

LoginViewAfterLogout

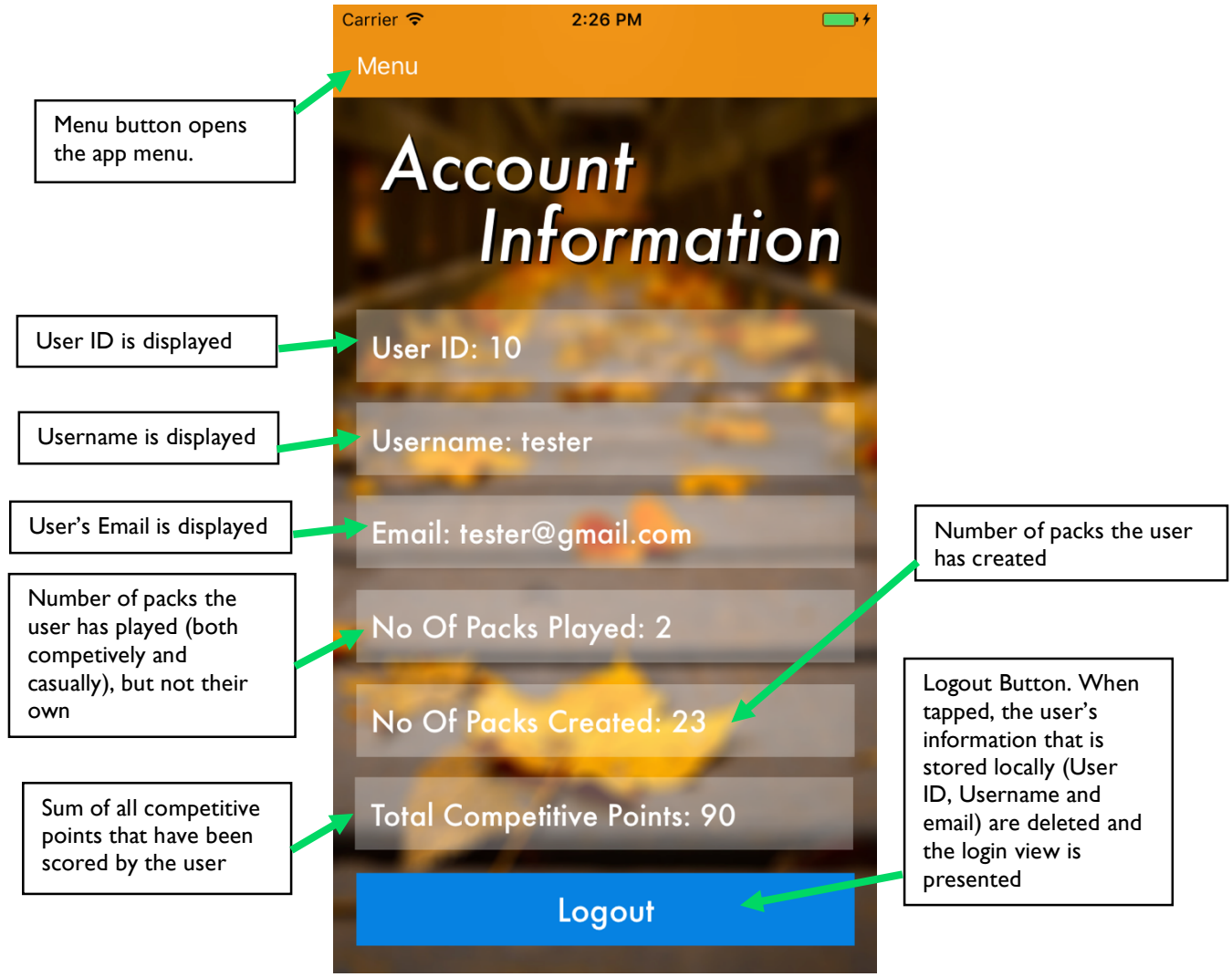


Login Page View



Registration Page View

Account Info Page View



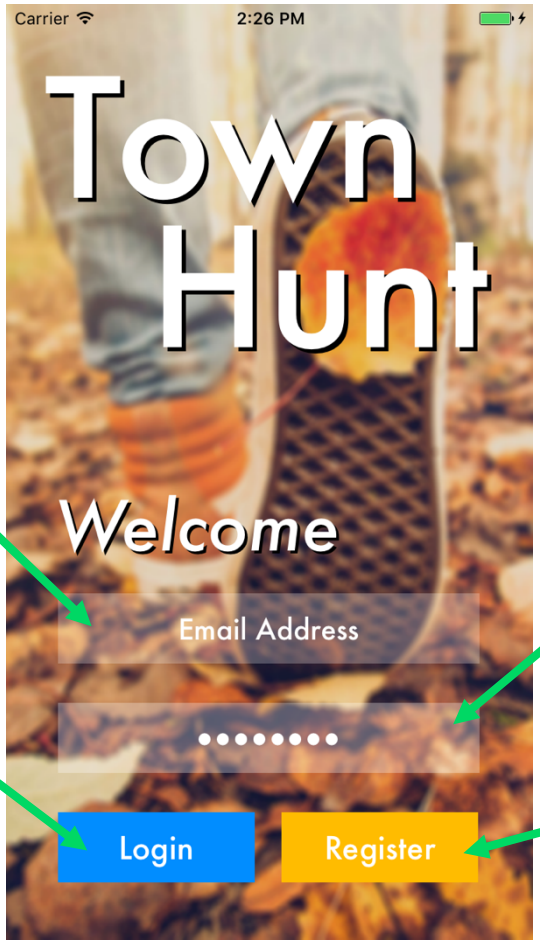
Account Info Page View Controller Code

```
//
//  AccountInfoPageViewController.swift
//  TownHunt App
//
//  This file defines the behaviour of the Account Info Page

import UIKit // UIKit constructs and manages the app's UI

// This class controls the logic behind the account info page view. Inherits
UIViewController, ModalTransitionListener
class AccountInfoPageViewController: UIViewController, ModalTransitionListener {

    // Outlets connect UI elements to the code, thus making UI elements accessible
programmatically
    @IBOutlet weak var menuOpenNavBarButton: UIBarButtonItem! // Menu button on the nav bar
    @IBOutlet weak var userIDInfoLabel: UILabel!
    @IBOutlet weak var usernameInfoLabel: UILabel!
    @IBOutlet weak var emailInfoLabel: UILabel!
```

```
    @IBOutlet weak var noPacksPlayedInfoLabel: UILabel! // Number of packs played by the user
    @IBOutlet weak var noPacksCreatedInfoLabel: UILabel! // Number of packs created by the
user
    @IBOutlet weak var totCompPointsLabel: UILabel! // Total number of 'competitive' points
scored by the user

    // Initialises attributes relating to the stats of the logged in user
    private var noPacksPlayed = 0
    private var noPacksCreated = 0
    private var totCompPoints = 0

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Singleton which sets up an event listener (listening for the login page to be
dismissed) in this instance of the class
        ModalTransitionMediator.instance.setListener(listener: self)

        // Connects the menu button to the menu screen
        menuOpenNavBarButton.target = self.revealViewController()
        menuOpenNavBarButton.action = #selector(SWRevealViewController.revealToggle(_:))

        // Sets the background image
        setBackgroundImage(imageName: "accountInfoBackgroundImage")

        // Initiates the process of retrieving the user stats from the database
        loadAccountDetails()
    }

    // [------------------ Retrieving and Displaying the Account Info ----------------------]

    // Retrieves the user stats from the database via the API
    private func getDBAccountStats(userID: String){
        // Initialises database interaction object
        let dbInteraction = DatabaseInteraction()
        // Tests for internet connectivity
        if dbInteraction.checkInternetAvaliability(){
            // A post string is posted to the online database API via the DatabaseInteraction
class on the background thread
            // The "getAccountStats.php" API queries the online database for the user stats
and returns the information
            let responseJSON = dbInteraction.postToDatabase(apiName: "getAccountStats.php",
postData: "userID=\(userID)"){ (dbResponse: NSDictionary) in

                // The local user stats variables are updated
                self.noPacksPlayed = Int(dbResponse["totalNumPacksPlayed"]! as! String)!
                self.noPacksCreated = Int(dbResponse["totalNumPacksCreated"]! as! String)!
                self.totCompPoints = Int(dbResponse["totalNumCompPoints"]! as! String)!

                // Returns the execution flow to the main thread
                DispatchQueue.main.async(execute: {
                    // The UI labels are updated to show the retrieved stats
                    self.noPacksPlayedInfoLabel.text = "No Of Packs Played:
\(self.noPacksPlayed)"
                    self.noPacksCreatedInfoLabel.text = "No Of Packs Created:
\(self.noPacksCreated)"
                    self.totCompPointsLabel.text = "Total Competitive Points:
\(self.totCompPoints)"
                })
```

```
            }
        }else{ // If no internet connectivity, an error message is diplayed asking the user
to connect to the internet
            let alertCon = UIAlertController(title: "Error: Couldn't Connect to Database",
message: "No internet connectivity found. Please check your internet connection",
preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Try Again", style: .default, handler: {
action in
                // Recursion is used to recall the getDBAccountStats function until internet
connectivity is restored
                self.getDBAccountStats(userID: userID)
            }))
            self.present(alertCon, animated: true, completion: nil)
        }
    }

    // Method which loads the user details
    private func loadAccountDetails(){

        // Retreives the user details stored locally
        let userID = UserDefaults.standard.string(forKey: "UserID")!
        let username = UserDefaults.standard.string(forKey: "Username")!
        let userEmail = UserDefaults.standard.string(forKey: "UserEmail")!
        // Updates the UI labels
        userIDInfoLabel.text = "User ID: \(userID)"
        usernameInfoLabel.text = "Username: \(username)"
        emailInfoLabel.text = "Email: \(userEmail)"
        // Initiates the retrieval of the user stats found on the database
        getDBAccountStats(userID: userID) // Passes the user id
    }

    // Connects the logout button to the code
    // - Carries out the logout process i.e. Removes all information about the logged in user
    @IBAction func LogoutButtonTapped(_ sender: Any) {
        UserDefaults.standard.set(false, forKey: "isUserLoggedIn") // Changes flag
        // Removes account information
        UserDefaults.standard.removeObject(forKey: "UserID")
        UserDefaults.standard.removeObject(forKey: "Username")
        UserDefaults.standard.removeObject(forKey: "UserEmail")
        UserDefaults.standard.synchronize() // Commits the changes to the userdefaults
database
        // Transitions from the account info page to the login page
        self.performSegue(withIdentifier: "loginViewAfterLogout", sender: self)

    }

    // ModalTransitionalListener protocol function which is called when the presented view
(login page) is dismissed
    func modalViewDismissed(){
        self.navigationController?.dismiss(animated: true, completion: nil)
        self.loadAccountDetails() // Reloads the UI labels with information about the newly
logged in user
    }
}
```

Login Page View



Email Address textfield. Users can enter the email address used to create their account. When the login is tapped, an error message is presented if email isn't valid or the field is empty.

Password textfield. Users enter the password to their account. The password is hidden with circles for security reasons. When the login is tapped, an error message is presented if the field is empty

Login Button. When tapped the email address and password entered are retrieved and sent to the database. If an account is found then the user's ID, email address and username are stored in UserDefaults

Register Button causes a segue to the register new account form where a user can register an account

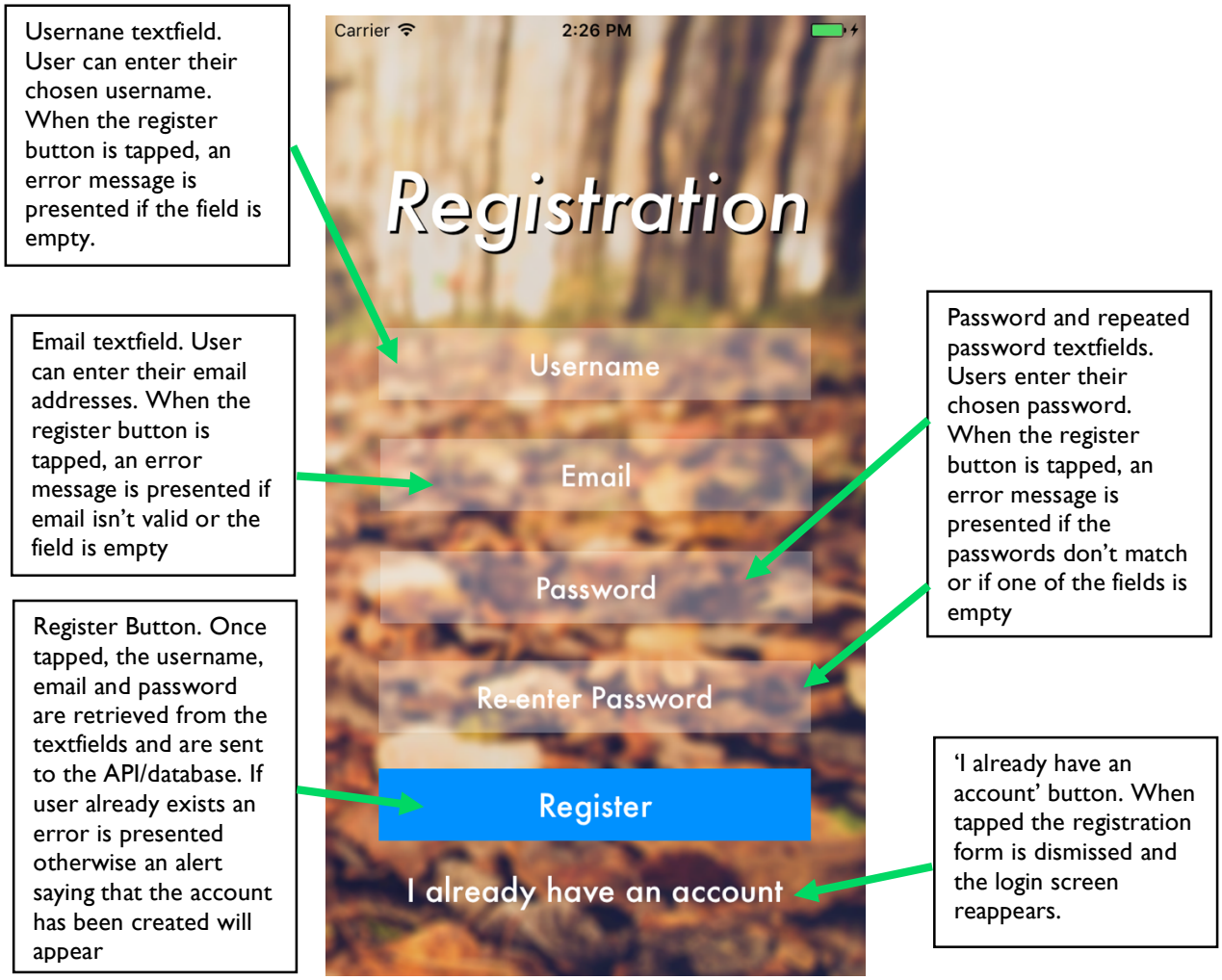
Login Page View Controller Code

```
//
//  LoginPageViewController.swift
//  TownHunt App
//
//  This file defines the behaviour of the login page

import UIKit // UIKit constructs and manages the app's UI

// Class defines the logic of the login view. Inherits FormTemplateExtensionOfViewController
class LoginPageViewController: FormTemplateExtensionOfViewController {

    // Outlets connect UI elements to the code, thus making UI login textfield elements
accessible programmatically
    @IBOutlet weak var userEmailTextField: UITextField!
    @IBOutlet weak var userPasswordTextField: UITextField!

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Sets the background image
```

```
        setBackgroundImage(imageName: "loginBackgroundImage")

        // Covers the password, entered on screen, with circles
        userPasswordTextField.isSecureTextEntry = true

    }

    // Connects the login button to the code
    // - Checks if the user and password exists in the database. If the user is found the
user details are stored locally
    @IBAction func loginButtonTapped(_ sender: Any) {
        // Initialises database interaction object
        let dbInteraction = DatabaseInteraction()
        // Tests for internet connectivity
        if dbInteraction.checkInternetAvaliability(){

            // Retrieves user input for the email and password
            let userEmail = userEmailTextField.text
            let userPassword = userPasswordTextField.text

            // Checks that the user has entered an email and the password
            if((userEmail?.isEmpty)! || (userPassword?.isEmpty)!) {
                // Display error message
                displayAlertMessage(alertTitle: "Data Entry Error", alertMessage: "All fields
must be complete")

            // Checks if the email address entered is a valid email address
            } else if !isEmailValid(testStr: userEmail!){
                // Display error message
                displayAlertMessage(alertTitle: "Email Invalid", alertMessage: "Please enter
a valid email")

            } else {
                // A post string is posted to the online database API via the
DatabaseInteraction class on the background thread
                // The "loginUser.php" API checks if the user and password exists in the
online database
                let responseJSON = dbInteraction.postToDatabase(apiName: "loginUser.php",
postData: "userEmail=\(userEmail!)&userPassword=\(userPassword!)"){ (dbResponse:
NSDictionary) in

                    // Initialises the error message variables and isAccountFound flag
                    var alertTitle = "ERROR"
                    var alertMessage = "JSON File Invalid"
                    var isAccountFound = false

                    // Checks if there is a database error indicating that the username and
or password was incorrect
                    if dbResponse["error"]! as! Bool{
                        // Prepares 'unsuccessful' alert title and message
                        alertTitle = "ERROR"
                        alertMessage = dbResponse["message"]! as! String
                    // Checks if there were no errors indicating that the username and
password combination was correct
                    } else if !(dbResponse["error"]! as! Bool){
                        // Prepares 'successful' alert title and message
                        alertTitle = "Thank You"
                        alertMessage = "Successfully Logged In"

                        // Sets the isAccountFound flag to true
                        isAccountFound = true
```

```
                        // Retrieves the account information from the database response
                        let accountDetails = dbResponse["accountInfo"]! as! NSDictionary
                        // Updates local logged in user info
                        UserDefaults.standard.set(true, forKey: "isUserLoggedIn") // Is a
user logged in flag is set to true
                        UserDefaults.standard.set(accountDetails["UserID"]! as! String,
forKey: "UserID") // Logged in user ID is stored
                        UserDefaults.standard.set(accountDetails["Username"]! as! String,
forKey: "Username") // Logged in username is stored
                        UserDefaults.standard.set(accountDetails["Email"]! as! String,
forKey: "UserEmail") // Logged in user email is stored
                        UserDefaults.standard.synchronize() // Commits the user information
to the local userdefaults database
                    }

                    // Returns the execution flow to the main thread
                    DispatchQueue.main.async(execute: {
                        // Displays an alert to the user about the attempted log in
                        let alertCon = UIAlertController(title: alertTitle, message:
alertMessage, preferredStyle: .alert)
                        alertCon.addAction(UIAlertAction(title: "Ok", style: .default,
handler: { action in
                            // Checks if the user account was found
                            if isAccountFound{
                                // If the account is logged in then the login view is
dismissed
                                self.dismiss(animated: true, completion: nil)
                                // The presenting view is notified that the login view has
been dismissed
                                ModalTransitionMediator.instance.sendModalViewDismissed(model
Changed: true)
                            }}))
                        self.present(alertCon, animated: true, completion: nil)
                    })
                }
            }
        }else{ // If no internet connectivity, an error message is displayed asking the user
to connect to the internet
            let alertCon = UIAlertController(title: "Error: Couldn't Connect to Database",
message: "No internet connectivity found. Please check your internet connection",
preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Try Again", style: .default, handler: {
action in
                // Recursion is used to recall the loginButtonTapped function until internet
connectivity is restored
                self.loginButtonTapped(Any.self)
            }))
            self.present(alertCon, animated: true, completion: nil)
        }

    }

}
```

Registration Page View



Registration Page View Controller Code

```
//  RegistrationPageViewController.swift
//  TownHunt App
//
//  This file defines the behaviour of the new account registration form

import UIKit // UIKit constructs and manages the app's UI
// Class controls the functionality of the new account registration view. Inherits from
FormTemplateExtensionOfViewController
class RegistrationPageViewController: FormTemplateExtensionOfViewController {

    // Outlets connect UI elements to the code, thus making UI login textfield elements
accessible programmatically
    @IBOutlet weak var usernameTextField: UITextField!
    @IBOutlet weak var userEmailTextField: UITextField!
    @IBOutlet weak var userPasswordTextField: UITextField!
    @IBOutlet weak var userRepeatPassWordTextField: UITextField!

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
```

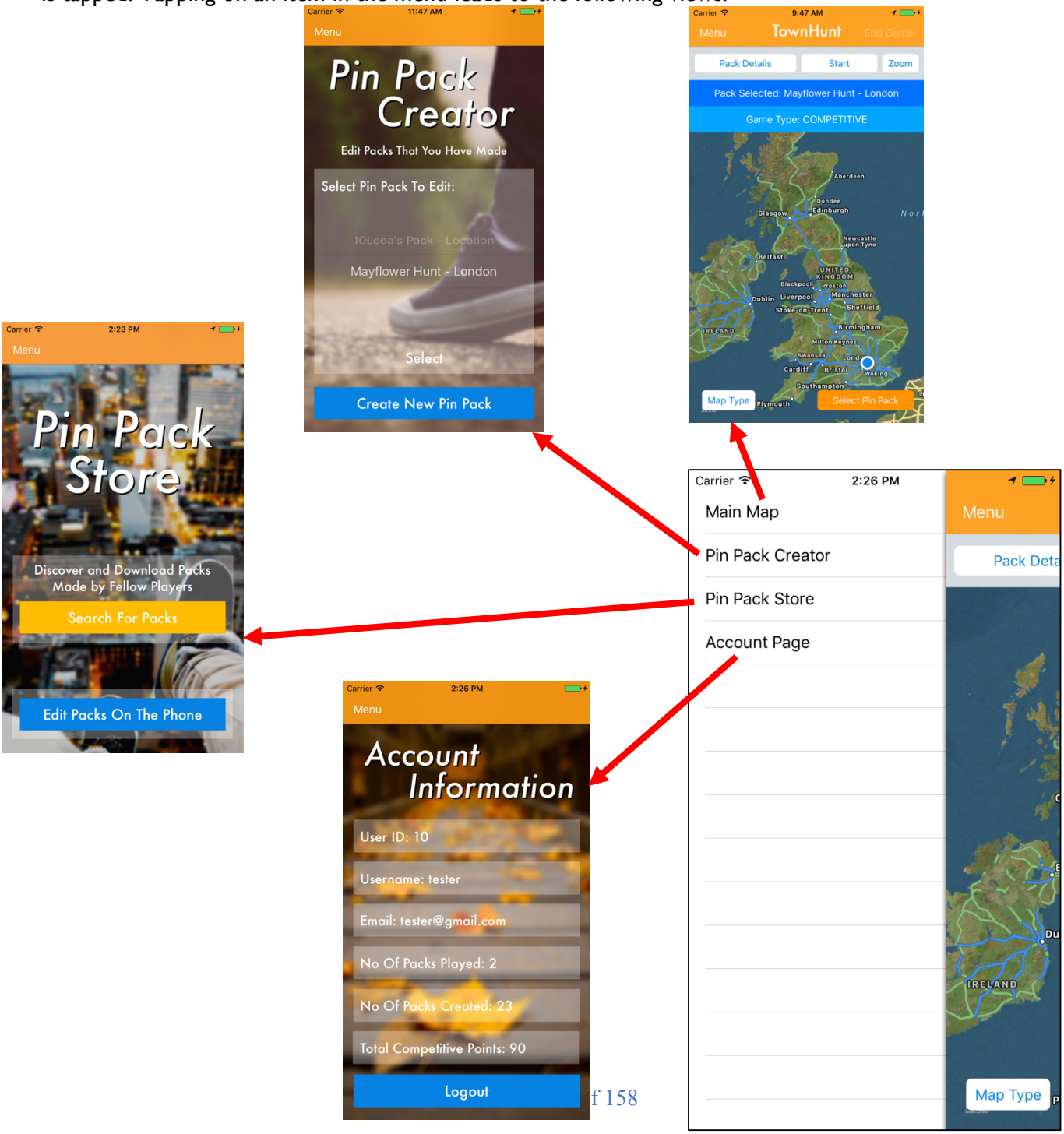
```
        super.viewDidLoad()
        // Sets the background image
        setBackgroundImage(imageName: "registrationBackgroundImage")
    }

    // Connects the register button to the code
    // - Validates the new account details and sends it to the database API
    @IBAction func registerButtonTapped(_ sender: Any) {
        // Initialises database interaction object
        let dbInteraction = DatabaseInteraction()
        // Tests for internet connectivity
        if dbInteraction.checkInternetAvaliability(){

            // Retrieves the user input
            let username = usernameTextField.text
            let userEmail = userEmailTextField.text
            let userPassword = userPasswordTextField.text
            let userRepeatPassword = userRepeatPassWordTextField.text

            // Check for empty fields
            if((username?.isEmpty)! || (userEmail?.isEmpty)! || (userPassword?.isEmpty)! ||
(userRepeatPassword?.isEmpty)!) {
                // Display data entry error message
                displayAlertMessage(alertTitle: "Data Entry Error", alertMessage: "All fields
must be complete")

            // Checks if email is a valid email address
            } else if !isEmailValid(testStr: userEmail!){
                // Display email invalid error message
                displayAlertMessage(alertTitle: "Email Invalid", alertMessage: "Please enter
a valid email")

            // Checks if username only contains alphanumeric characters
            } else if !isAlphanumeric(testStr: username!){
                // Display username invalid error message
                displayAlertMessage(alertTitle: "Username Invalid", alertMessage: "Username
must only contain alphanumeric characters")

            // Checks if username character length is greater than 20
            } else if((username?.characters.count)! > 20){
                //Displays username character length error message
                displayAlertMessage(alertTitle: "Username is Greater Than 20 Characters",
alertMessage: "Please enter a username which is less than or equal to 20 characters")

            // Checks if email character length is greater than 100
            } else if((username?.characters.count)! > 100){
                //Displays email character length error message
                displayAlertMessage(alertTitle: "Email address is Greater Than 100
Characters", alertMessage: "Please enter an email address which is less than or equal to 100
characters")

            // Checks if passwords are the same
            } else if(userPassword != userRepeatPassword){
                //Displays non-matching passwords error message
                displayAlertMessage(alertTitle: "Passwords Do Not Match", alertMessage:
"Please enter matching passwords")

            } else { // User input meets the standards
                // A post string is posted to the online database API via the
DatabaseInteraction class on the background thread
```

```
                // The "registerUser.php" API checks if the user already exists in the online
database, if not then the user is added to the database
                let responseJSON = dbInteraction.postToDatabase(apiName: "registerUser.php",
postData: "username=\(username!)&userEmail=\(userEmail!)&userPassword=\(userPassword!)"){
(dbResponse: NSDictionary) in

                    // Initialises the error message variables and isUserRegistered flag
                    var alertTitle = "ERROR"
                    var alertMessage = "JSON File Invalid"
                    var isUserRegistered = false

                    // Checks if there is a database error indicating that the new user
couldn't be added to the database
                    if dbResponse["error"]! as! Bool{
                        // Prepares 'unsuccessful' alert title and message
                        alertTitle = "ERROR"
                        alertMessage = dbResponse["message"]! as! String
                    // Checks if there wasn't a database error indicating that the new user
was added to the database
                    } else if !(dbResponse["error"]! as! Bool){
                        // Prepares 'successful' alert title and message
                        alertTitle = "Thank You"
                        alertMessage = dbResponse["message"]! as! String
                        isUserRegistered = true // Changes isUserRegistered flag
                    }

                    // Returns the execution flow to the main thread
                    DispatchQueue.main.async(execute: {
                        // Displays an alert to the user about the attempted user
registration
                        let alertCon = UIAlertController(title: alertTitle, message:
alertMessage, preferredStyle: .alert)
                        alertCon.addAction(UIAlertAction(title: "Ok", style: .default,
handler: { action in
                            // Checks if the user is registered flag
                            if isUserRegistered{
                                // Registration form is dismisses/exited
                                self.dismiss(animated: true, completion: nil)
                            }}))
                        self.present(alertCon, animated: true, completion: nil)
                    })
                }
            }
        }else{ // If no internet connectivity, an error message is displayed asking the user
to connect to the internet
            let alertCon = UIAlertController(title: "Error: Couldn't Connect to Database",
message: "No internet connectivity found. Please check your internet connection",
preferredStyle: .alert)
            alertCon.addAction(UIAlertAction(title: "Try Again", style: .default, handler: {
action in
                // Recursion is used to recall the registerButtonTapped function until
internet connectivity is restored
                self.registerButtonTapped(Any.self)
            }))
            self.present(alertCon, animated: true, completion: nil)
        }
    }

    // Connects the 'I Already Have An Account' button to the code
    // - Dismisses the view when tapped
    @IBAction func alreadyHaveAccountButtonTapped(_ sender: Any) {
```

```
        self.dismiss(animated: true, completion: nil)
    }
}
```

Menu

For the slide out menu I made use of a commonly used repository SWRevealViewController (for link to repository see Bibiography/References). It is open source and provides a UIViewController subclass which reveals a rear view controller behind a front view controller. In my case the rear view controller is a UITableView (the menu) is revealed from behind the current front view controller when the menu button is tapped. Tapping on an item in the menu leads to the following views:



Menu Code

```
//
//  BackTableVC.swift
//  TownHunt App
//
//  This file defines the app's menu back table

import UIKit // UIKit constructs and manages the app's UI

// Class controls the app's menu
class BackTableVC: UITableViewController{

    // List of options in the app
    var menuOptions: [String] = ["Main Map", "Pin Pack Creator", "Pin Pack Store", "Account Page"]

    // TableView method that sets the number of rows in the table
    override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        // Number of rows in the table is the same as the number of menu options
        return menuOptions.count
    }

    // TableView method which defines what to display in each cell
    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        // Returns a reusable table-view cell object based on the prototype cell defined in the UI
storyboard
        let cell = tableView.dequeueReusableCell(withIdentifier: menuOptions[(indexPath as
NSIndexPath).row], for: indexPath) as UITableViewCell
        // Sets the cell label as the corresponding menu option
        cell.textLabel?.text = menuOptions[(indexPath as NSIndexPath).row]
        return cell
    }
}
```

Supporting App Files

UIViewController Extension

The majority of my classes inherit from UIViewController. Setting a background image and displaying a generic alert to the user with only an OK button to respond were two functions which appeared in most of my classes. I therefore decided to extend the UIViewController class with these two functions.

```
//
//  UIViewControllerExtension.swift
//  TownHunt App
//
//  This file extends the UIViewController class with an additional two methods

extension UIViewController {

    // Method sets the background image of the view.
    // - The name of the image (found in the 'Assets' folder) is passed into this function
    func setBackgroundImage(imageName: String){
        // Creates a temp image frame/context - the size of the image matches the size of the
view/screen
        UIGraphicsBeginImageContext(self.view.frame.size)
        // The image is retrieved from the 'Assets' folder and is loaded into the temp image frame
        UIImage(named: imageName)?.draw(in: self.view.bounds)
        // Copies the temp image frame into the constant 'backgroundImage'
        let backgroundImage: UIImage = UIGraphicsGetImageFromCurrentImageContext()!
        // Deletes the temp image frame
        UIGraphicsEndImageContext()
        // Sets the view background image
        self.view.backgroundColor = UIColor(patternImage: backgroundImage)
    }
```

```
    // Function displays a generic alert with an OK button that just dismisses the alert
    // - Takes in an alert title and message that will be displayed
    func displayAlertMessage(alertTitle: String, alertMessage: String){
        let alertCon = UIAlertController(title: alertTitle, message: alertMessage, preferredStyle:
.alert)
        alertCon.addAction(UIAlertAction(title: "Ok", style: .default, handler: nil))
        self.present(alertCon, animated: true, completion: nil)
    }

}
```

PinPackMapViewController

This class is the parent to the MainGameScreenViewController and the PinPackEditorViewController. Both of these two classes contain maps and require pins to be loaded from a file and for those pins to be converted into PinLocation objects (to display on the map). This parent class defines functions which does this.

```
//
//  PinPackMapViewController.swift
//  TownHunt App
//
//  This file defines the pin pack map view parent class

import MapKit // UIKit constructs and manages the app's UI

// Class defines two methods which are used by both map view screens. Inherits from
UIViewController
class PinPackMapViewController: UIViewController {

    // Returns an array of Pin Location objects from a list of pin data
    func getListOfPinLocations(packData: [String: Any]) -> [PinLocation] {
        // Extracts the pins from the pack data
        let packDetailPinList = packData["Pins"] as! [[String:String]]
        // Initiates PinLocation array
        var gamePins: [PinLocation] = []
        // Checks if there are any pins in the pack
        if packDetailPinList.isEmpty {
            print(packDetailPinList)
            print("No Pins in pack")
        } else{ // If there are pins then each pin in the pack is converted to a PinLocation and
appended to the gamePins array
            print("There are pins in the loaded pack")
            for pin in packDetailPinList{
                let pinToAdd = PinLocation(title: pin["Title"]!, hint: pin["Hint"]!, codeword:
pin["Codeword"]!, coordinate: CLLocationCoordinate2D(latitude: Double(pin["CoordLatitude"]!)!,
longitude: Double(pin["CoordLongitude"]!)!), pointVal: Int(pin["PointValue"]!)!)
                gamePins.append(pinToAdd)
            }
        } // List of PinLocations are returned
        return gamePins
    }

    // Returns all of the pack data loaded from local storage
    func loadPackFromFile(filename: String, userPackDictName: String, selectedPackKey: String,
userID: String) -> [String : AnyObject]{
        // Initialises pack data array
        var packData: [String : AnyObject] = [:]
        // Initialises a localStorageHandler object
        let localStorageHandler = LocalStorageHandler(fileName: filename, subDirectory: "UserID-
\(userID)-Packs", directory: .documentDirectory)
        // Retrieves JSON data
        let retrievedJSON = localStorageHandler.retrieveJSONData()
```

```
        // Converts JSON data into an editable dictionary
        packData = retrievedJSON as! [String : AnyObject]
        return packData // JSON data returned
    }
}
```

FormTemplateExtensionOfViewController Class

This class is the parent class to the view controllers where data entry is required. It provides methods that are triggered by keyboard interactions as well regex testing methods (to check email validity and check that a string only contains alphanumeric character)

```
//
//  FormTemplateExtensionOfViewController.swift
//  TownHunt App
//
//  This file defines the form view parent class

import UIKit // UIKit constructs and manages the app's UI

// Class provides additional methods relating to view with forms
class FormTemplateExtensionOfViewController: UIViewController {

    // Called when the view controller first loads. This method sets up the view
    override func viewDidLoad() {
        // Creates the view
        super.viewDidLoad()

        // Adds an event listener to the view that will shift the view upwards when the keyboard
is displayed on screen
        NotificationCenter.default.addObserver(self, selector:
#selector(self.keyboardWillShow(sender:)), name: NSNotification.Name.UIKeyboardWillShow, object:
nil)

        // Adds an event listener to the view which looks out for taps on the view. If the user
taps the view (not the keyboard)
        // and an onscreen keyboard is displayed, the onscreen keyboard will disappear
        let tap: UITapGestureRecognizer = UITapGestureRecognizer(target: self, action:
#selector(self.dismissKeyboard))
        view.addGestureRecognizer(tap)

        // Adds an event listener to the view that will re-centre the view when the keyboard is
dismissed from the screen screen
        NotificationCenter.default.addObserver(self, selector:
#selector(self.keyboardWillHide(sender:)), name: NSNotification.Name.UIKeyboardWillHide, object:
nil)
    }

    // Method which dismisses the keyboard from the screen
    func dismissKeyboard() {
        view.endEditing(true)
    }

    // Method which shifts the view upwards by 150 pixels
    func keyboardWillShow(sender: NSNotification){
        self.view.frame.origin.y = -150
    }

    // Method which sets the view in its original position
    func keyboardWillHide(sender: NSNotification){
        self.view.frame.origin.y = 0
    }

    // Method which sees if a test string matches a regular expression pattern
    private func stringTester(regExPattern: String, testString: String) -> Bool {
```

```
        let stringTester = NSPredicate(format:"SELF MATCHES %@", regExPattern)
        return stringTester.evaluate(with: testString)
    }

    // Method which returns whether the test string is a valid email address
    func isEmailValid(testStr:String) -> Bool {
        let emailRegExPattern = "^[A-Za-z0-9._-]+@[A-Za-z0-9._-]+\\.[A-Za-z]{2,}"
        return stringTester(regExPattern: emailRegExPattern, testString: testStr)
    }

    // Method which returns whether the test string only contains alphanumeric characters
    func isAlphanumeric(testStr:String) -> Bool {
        let alphanumericRegExPattern = "^[A-Za-z0-9]+$"
        return stringTester(regExPattern: alphanumericRegExPattern, testString: testStr)
    }
}
```

First Load Setup Class

This class is run on the first launch of the app. It sets up the initial listOfLocalUserIDs array.

```
//
//  FirstLoadSetup.swift
//  TownHunt App
//
//  This file defines the initial app setup when the app launches for the first time

import UIKit // UIKit constructs and manages the app's UI

class FirstLoadSetup {

    // Method is called when the app launches for the first time
    func initialSetup(){

        // Initialises the list of IDs of all the users whose packs are found on the device
        UserDefaults.standard.setValue([String](), forKey:"listOfLocalUserIDs")
        // Changes the need initial setup flag
        UserDefaults.standard.set(false, forKey: "isInitialSetupRequired")
        // Commits the changes, to the pack linked list information and isInitialSetupRequired, to
the local userdefaults database
        UserDefaults.standard.synchronize()

    }
}
```

PinLocation Class

This class is the object that is placed on the map.

```
//
//  PinLocation.swift
//  TownHunt App
//
// This file defines the PinLocation class

import MapKit // MapKit constructs and manages the map and annotations
import UIKit // UIKit constructs and manages the app's UI

// PinLocation class is a child class of NSObject and MKAnnotations
// - It stores information about a pin from a pack
class PinLocation: NSObject, MKAnnotation{

    // Attributes about the pin
    let title: String?
```

```
    let hint: String
    let codeword: String
    let coordinate: CLLocationCoordinate2D
    let pointVal: Int
    var isFound = false

    // Attributes are set on instantiation
    init(title: String, hint:String, codeword: String, coordinate: CLLocationCoordinate2D,
pointVal: Int){
        self.title = title
        self.hint = hint
        self.codeword = codeword
        self.coordinate = coordinate
        self.pointVal = pointVal
        super.init()
    }

    // The annotation subtitle is set as how many points the pin is worth
    var subtitle: String? {
        return String("\(pointVal) Points")
    }

    // Returns a dictionary containing all of the information about the pin
    func getDictOfPinInfo() -> [String : String] {
        let dictOfPinInfo = ["Title": self.title!, "Hint": self.hint, "Codeword": self.codeword,
"CoordLatitude": String(self.coordinate.latitude), "CoordLongitude":
String(self.coordinate.longitude), "PointValue": String(self.pointVal)]
        return dictOfPinInfo
    }
}
```

DatabaseInteraction Class

```
//
//  DatabaseInteraction.swift
//  TownHunt App
//
//  This file defines the DatabaseInteraction class which checks the internet avaliability of the
device and sends HTTP POST requests

import UIKit // UIKit constructs and manages the app's UI
import SystemConfiguration // Grants, the app, access a device’s network configuration settings.

// Class acts as interface between the app and the database API
class DatabaseInteraction: NSObject {

    // URL stem (online location) of the TownHunt api
    let mainSQLServerURLStem = "http://alvinlee.london/TownHunt/api"

    // Checks to see if the phone is connected to the internet, returns true if there is internet
connectivity and false if there isn't
    func checkInternetAvaliability() -> Bool {

        // Creates the socket address pointer that access to will be tested
        var zeroTestAddress = sockaddr_in()
        zeroTestAddress.sin_len = UInt8(MemoryLayout<sockaddr_in>.size) // Defines the size of the
socket as the number of bytes allocated to the socket
        // Sets the type of addresses that the socket can communicate with, in this case IPv4
addresses (AF_INET) - the safest option
        zeroTestAddress.sin_family = sa_family_t(AF_INET)

        // Uses predefined 'Reachability' function to attempt to reach (connect) to the test
socket
        guard let defaultRouteReachability = withUnsafePointer(to: &zeroTestAddress, {
            $0.withMemoryRebound(to: sockaddr.self, capacity: 1) {
```

```
                SCNetworkReachabilityCreateWithAddress(nil, $0)
            }
        }) else { // If no connection can be established then false is returned
            return false
        }

        // Stores information (info flags) about the test connection
        var connectionTestflags: SCNetworkReachabilityFlags = []
        // Attempts to retrieve the flags information about the devices' connectivity
        if !SCNetworkReachabilityGetFlags(defaultRouteReachability, &connectionTestflags) {
            // If no flags can be retrieved then there is no internet connection and false is
returned
            return false
        }

        // Retrieves the two internet connection flags
        let isReachable = connectionTestflags.contains(.reachable) // Are web addresses
reachable/connectable? flag
        let deviceNeedsConnection = connectionTestflags.contains(.connectionRequired) // Does the
device have an internet connection? flag

        // Returns the AND of the two internet connection flags.
        // If there is an internet connection and ip addresses are reachable then true is returned
        return (isReachable && !deviceNeedsConnection)
    }

    // Posts data to a specified TownHunt API to the online database and returns a dictionary
containing the API's/Database's response
    func postToDatabase(apiName :String, postData: String, completion: @escaping (NSDictionary)-
>Void){
        // Prepares the URL location of the specific API file
        let apiURL = URL(string: mainSQLServerURLStem + "/" + apiName + "/")
        // Initialises the URL (HTTP) request
        var request = URLRequest(url: apiURL!)
        // HTTP request method is set to POST as the app will be POSTing the data
        request.httpMethod = "POST"
        // The data (POST parameters) to be POSTed is set as the URL request message body
        request.httpBody = postData.data(using: String.Encoding.utf8) // postData is encoded into
UTF-8 which is used by the online API

        // A URLSession object is instantiated to handle the request on a background thread.
        let task = URLSession.shared.dataTask(with: request) {
            // Completion handler defines - once the request is completed, the info retrieved from
the API/Database is interpreted
            // Gets passed the data, response and error (All of which are optional)
            (data: Data?, response: URLResponse?, error: Error?) in

            // Checks for errors
            if error != nil{
                print("error=\(error)")
                return
            }
            // Prints the full URL response
            print("response = \(response)")

            // Attempts to convert the data received into a dictionary
            do {
                // Tries to serialise the data into a json dictionary
                let jsonDicationary = try JSONSerialization.jsonObject(with: data!, options:
.mutableContainers) as? NSDictionary
                // Completes/ends the task and returns the json dictionary
                completion(jsonDicationary!)
            } catch {
                print(error)
            }
```

```
        } // Since all tasks start in a suspended state by default, 'resume' starts the task
        task.resume()
    }
}
```

Local Storage Handler Class

```
//
//  LocalStorageHandler.swift
//  TownHunt App
//
//  This file acts as an interface between the local device storage and the application

import UIKit // UIKit constructs and manages the app's UI

// Class grants access to specified file
class LocalStorageHandler {

    // Instantiates a default file manager object
    private let fileManager = FileManager.default

    // Attributes storing information about the file
    private let directory: FileManager.SearchPathDirectory
    private let directoryPath: String
    private let subDirectory: String
    private let pathToSubDir: String
    private let fileName: String
    private let fullPathToFile: String

    // File and sub directory existence flags
    private let doesFileExist: Bool
    private var doesSubDirectoryExist: ObjCBool = false

    // Stores the local storage response
    private var response = [String:String]()

    // Initialises class attributes
    init(fileName: String, subDirectory: String, directory: FileManager.SearchPathDirectory){
        self.fileName = fileName
        self.subDirectory = "/\(subDirectory)"
        self.directory = directory
        // Gets directory path specific to the device
        self.directoryPath = NSSearchPathForDirectoriesInDomains(directory, .userDomainMask,
true)[0]
        self.pathToSubDir = directoryPath + self.subDirectory
        self.fullPathToFile = "\(self.pathToSubDir)/\(self.fileName).json"
        // Updates file/subdirectory flags
        self.doesFileExist = self.fileManager.fileExists(atPath: fullPathToFile) // Checks if the
file exists
        self.fileManager.fileExists(atPath: pathToSubDir, isDirectory: &doesSubDirectoryExist) //
Checks if sub directory exists

        // Calls a function that creates the subdirectory if needed
        createDirectory()

    }

    // Method which creates a directory if directory doesn't exist
    private func createDirectory(){
        // Checks if subdirectory doesn't exists
        if !(doesSubDirectoryExist.boolValue){
            do{ // Attempts to create the subdirectory
                try self.fileManager.createDirectory(atPath: pathToSubDir,
withIntermediateDirectories: false, attributes: nil)
            } catch { // If not an error is raised
```

```
                print("Error: \(error.localizedDescription)")
            }
        }
    }

    // Method which returns a bool indicating whether a file exists
    public func getDoesFileExist() -> Bool{
        return doesFileExist
    }

    // [------------------------- JSON Conversion & Retrieval---------------------------------]

    // Method which converts JSON objects into JSON strings
    public func jsonToString(jsonData: Any) -> String{
        // Checks if jsonData passed is a valid JSON object
        if JSONSerialization.isValidJSONObject(jsonData) {
            do{
                // Tries to serialise the JSON object
                let data = try JSONSerialization.data(withJSONObject: jsonData)
                // Attempts to convert the JSON object into a string
                if let string = NSString(data: data, encoding: String.Encoding.utf8.rawValue) {
                    return string as String
                }
            }catch { // Prints error
                print("Couldn't Convert JSON object into string")
            }
        }
        // If the JSON object is not converted, a null string is returned
        return ""
    }

    // Reads JSON Files and retrieves the JSON data as a dictionary
    public func retrieveJSONData() -> NSDictionary{
        // Checks that the file exists
        if doesFileExist{
            do{ // Attempts to retrieve data from the file
                let fileData = try NSData(contentsOfFile: fullPathToFile, options:
NSData.ReadingOptions.mappedIfSafe)
                // Attempts to serialise the JSON object (convert the JSON data into a Swift-
readable dictionary)
                let jsonDicationary = try JSONSerialization.jsonObject(with: fileData as Data,
options: .allowFragments) as! NSDictionary
                return jsonDicationary
            } catch{ // If the JSON file couldn't be retrieved or serialised then an error is
returned
                self.response["Error"] = "true"
                self.response["Message"] = "Couldn't retrieve JSON file contents"
                return self.response as NSDictionary
            }
        } else { // If the JSON file doesn't exist then an error is returned
            self.response["Error"] = "true"
            self.response["Message"] = "File doesn't exist"
            return self.response as NSDictionary
        }
    }

    // [-------------------------Saving Packs & JSON Files---------------------------------]

    // Method which saves JSON files to the full path. If the file is successfully saved 'true' is
returned
    public func saveJSONFile(dataToWrite: Any) -> Bool {
        // Initialises convertedJSONToSave as an empty data object
        var convertedJSONToSave: Data?

        // Attempts to convert given dictionary into JSON file
```

```
        do{
            convertedJSONToSave = try JSONSerialization.data(withJSONObject: dataToWrite, options:
.prettyPrinted)
        } catch{ // If the JSON file couldn't be serialised then false is returned
            print("Couldn't serialise JSON file")
            return false
        }

        // Attempts to write JSON File to storage
        if fileManager.createFile(atPath: fullPathToFile, contents: convertedJSONToSave,
attributes: nil){
            print("File Saved")
            return true
        } else { // Error has occurred and false is returned
            print("File could not be saved")
            return false
        }
    }

    // Method for saving the edited version of a pack, returns the pack details dictionary
    public func saveEditedPack(packData: [String: Any]) -> [String:AnyObject] {
        // Initialises data to write array
        var dataToWrite = packData
        // Retrieves the original pack
        let originalPackData = retrieveJSONData()

        // Compares the original pack to the one that was passed
        if !(originalPackData.isEqual(to: packData)){
            print("Packs Are not the same")
            // If a change has been detected the version number is incremented by one
            let currentVersionNum = Int(packData["Version"]! as! String)!
            dataToWrite["Version"] = String(currentVersionNum + 1)

            // Attempts to save the pack
            if saveJSONFile(dataToWrite: dataToWrite){
                // If saving the file was successful then the pack data is passed back and the
error flag is set to false
                return ["error": false as AnyObject, "data": dataToWrite as AnyObject]
            } else{ // If the file couldn't be saved then the error flag is set to true and an
error message is passed back
                print("Couldn't save edited pack")
                return ["error": true as AnyObject, "message": "Couldn't save edited pack" as
AnyObject]
            }
        } // If no changes to the file was made then an error message is returned
        print("Packs are the same")
        return ["error": true as AnyObject, "message": "No changes to the pack were made" as
AnyObject]

    }

    // Method that adds the new pack to the device - the pack is saved as a json file and the
linked lists are updated to reflect the new pack addition
    public func addNewPackToPhone(packData: NSDictionary) -> Bool {
        // Saves data as a JSON file in the location specified by the 'fullPathToFile'
        if saveJSONFile(dataToWrite: packData as NSDictionary){
            // Retrieves the pack details needed to update the linked lists
            let packName = packData["PackName"]! as! String
            let creatorID = packData["CreatorID"]! as! String
            let packLocation = packData["Location"]! as! String

            let defaults = UserDefaults.standard

            // Checks if listOfLocalUserIDs exists
            if var listOfUserIDs = defaults.array(forKey: "listOfLocalUserIDs") {
```

```
                // Converts the list of local user IDs array into an array of strings
                listOfUserIDs = listOfUserIDs as! [String]

                // Checks if creator user ID of the pack is part of the listOfLocalUserIDs array
                if !(listOfUserIDs.contains(where: {$0 as! String == creatorID})) {
                    // If new creator user ID not found, the new ID is appended to the
listOfLocalUserIDs
                    listOfUserIDs.append(creatorID)
                    defaults.set(listOfUserIDs, forKey: "listOfLocalUserIDs")
                }

                // Array containing all of a specific user's created packs is instantiated
                let creatorDictKey = "UserID-\(creatorID)-Packs"
                var creatorLocalPackDict = [String:String]()

                // Checks if specific user ID dictionary exists. If exists, the dictionary is
retrieved
                if let dictionary = defaults.dictionary(forKey: creatorDictKey){
                    creatorLocalPackDict = dictionary as! [String : String]
                }

                // Pack identifier/display name (name-location key) is appended to the dictionary
along with the corresponding filename
                let packKey = "\(packName) - \(packLocation)"
                creatorLocalPackDict[packKey] = self.fileName
                defaults.set(creatorLocalPackDict, forKey: creatorDictKey)

                // Commits the pack linked list information to the local userdefaults database
                defaults.synchronize()
                return true
            } else{ // If the local list of creator IDs couldn't be retrieved then an error is
printed and 'false' is returned
                print("Error loading array of local creator IDs")
                return false
            }
        } else{ // If the file couldn't be saved then an error is printed and 'false' is returned
            print("Error saving JSON file")
            return false
        }
    }

    //[-------------------Pack Deletion-------------------------]

    // Method which deletes a pack (file) and updates the pack linked lists to reflect the file
deletion
    public func deleteFile(packName: String, packLocation:String, creatorID: String) -> Bool{
        do { // Attempts to delete the file
            try fileManager.removeItem(atPath: self.fullPathToFile)

            // Generates the name of the array containing all of a specific user's created packs
            let creatorDictKey = "UserID-\(creatorID)-Packs"
            let defaults = UserDefaults.standard
            // Retrieves the creator's local packs dictionary
            var creatorLocalPackDict = defaults.dictionary(forKey: creatorDictKey) as! [String :
String]

            // Pack identifier/display name (name-location key) is removed from the dictionary
along with the corresponding filename
            let packKey = "\(packName) - \(packLocation)"
            creatorLocalPackDict.removeValue(forKey: packKey)

            // Checks if there are any local packs, made by the creator user ID passed, remaining
            if creatorLocalPackDict.isEmpty{
                // If no more packs made by that creator exists on the device then the creator
dictionary is deleted
```

```
                defaults.removeObject(forKey: creatorDictKey)
                // The list of local user IDs is updated with the creator being removed
                var listOfUserIDs = defaults.array(forKey: "listOfLocalUserIDs") as! [String]
                if let index = listOfUserIDs.index(of:creatorID) {
                    listOfUserIDs.remove(at: index)
                } // Saves the new list of user IDs
                defaults.set(listOfUserIDs, forKey: "listOfLocalUserIDs")

                do{ // Attempts to delete the creator's subdirectory
                    try fileManager.removeItem(atPath: self.pathToSubDir)
                } catch{ // Error is printed and false is returned if the sub directory can't be
deleted
                    print("Couldn't delete user subdirectory")
                    return false
                }
            } else { // If there are still packs made by the creator on the device, the new
creator's pack list is saved
                defaults.set(creatorLocalPackDict, forKey: creatorDictKey)
            }
            // Commits the pack linked list information to the local userdefaults database
            defaults.synchronize()
            return true
        } catch { // Error is printed and false is returned if the file can't be deleted
            print("Couldn't delete file")
            return false
        }
    }

}
```

AppDelegation Class

```
//
//  AppDelegate.swift
//  TownHunt App
//
//  Created by Alvin Lee.
//  Copyright © 2017 Alvin Lee. All rights reserved.
//
//  This file is the main delegate of the whole app. It is automatically generated when the Xcode
project is initialised

import UIKit // UIKit constructs and manages the app's UI
import CoreLocation // Allows interactivity with the user's GPS location

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?
    var locationManager: CLLocationManager?

    // Standard ApplicationDelegate method which is automatically called when the application is
launched
    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // Asks for permission to gain access to the user's GPS location
        locationManager = CLLocationManager()
        locationManager?.requestWhenInUseAuthorization()
        return true
    }
```

Sound Class

```
//
//  Sound.swift
```

```
//  TownHunt App
//
//  Allows mp3 files to be played from the app

import AVFoundation // AVFoundation contains the sound management interface
import UIKit // UIKit constructs and manages the app's UI

// Equips the app with an audio player
var audioPlayer = AVAudioPlayer()

// Class which holds and plays a sound file
class Sound{

    // Method which plays a specified sound file from the sound resource files
    func playSound(_ soundName: String){
        // Retrieves the sound file
        let soundFile = URL(fileURLWithPath: Bundle.main.path(forResource: soundName, ofType:
"mp3")!)
        do{ // Attempts to load the sound file into the sound player
            audioPlayer = try AVAudioPlayer(contentsOf: soundFile)
            // Plays the sound
            audioPlayer.play()
        } catch{ // If sound couldn't be loaded then an error is printed
            print("Error getting the audio file")
        }
    }
}
```

TownHunt API Code

DBConnection Class

```
<?php

/*
 *  DBConnection.php
 *  TownHunt API
 *
 *  This file creates the connection to the TownHunt MySQL server
 */

// Class which allows connection to the database
class DBConnection{

   // Stores the connection to the MySQL server
   private $connection;

   // Function which creates the connection to the server
   function connectToServer()
   {
      // Information about the MySQL server (Info predefined by the host)
      $host_name  = "db648556307.db.1and1.com";
      $database   = "db648556307"; // TownHunt database
         // Access details
      $user_name  = "dbo648556307";
      $password   = "P4SSw0rd";

      // Opens the connection to the TownHunt MySQL server
```

```
27        $this->connection = new mysqli($host_name, $user_name, $password, $database);
28
29        // Checks if there was an error in establishing a connection with the server
30        if(mysqli_connect_errno())
31        {
32            // If there was an error, it is echoed
33            echo 'Failed to connect to MySQL Server'.mysqli_connect_error().'</p>';
34        }
35
36        // Returns an object that represents the connection to the MySQL server
37        return $this->connection;
38    }
39 }
40
```

DBOperation Class

```
 1 <?php
 2
 3 /*
 4  *  DBOperation.php
 5  *  TownHunt API
 6  *
 7  *  This file defines the database operations (SQL Queries)
 8  */
 9
10 // Class contains all of the database operations (SQL Queries)
11 class DBOperation
12 {
13     // Stores the connection to the MySQL server
14     private $connection;
15
16     // Initiates the database connection when the class is instantiated
17     function __construct() {
18         // Temporarily imports the DBConnection class
19         require_once dirname(__FILE__) . '/DBConnection.php';
20         // Creates a database connection
21         $database = new DBConnection();
22         $this->connection = $database->connectToServer();
23     }
24
25     // Executes queries and fetches/returns a single array of data
26     // - To be used when query result set is expected to contain only one element
27     private function execQueryFetchSingleRow($query)
28     {
29         // Stores the result in an associative array
30         $result = array();
31         // Executes the passed query (prepared statement) on the database
32         $stmt = $this->connection->query($query);
33         // Checks that there is at least one row in the query result set
34         // and the executing (prepared) statement was not empty
35         if ((mysqli_num_fields($stmt)>=1) && $stmt != null)
36         {
37             // Retreives the query result as an associative array
38             $row = $stmt->fetch_array(MYSQLI_ASSOC);
```

```
39          // Checks if the result row isn't empty
40          if (!empty($row))
41          {
42              // If not empty, the result row is set as $result
43              $result = $row;
44          }
45          // Closes (clears) the prepared statement
46          $stmt->close();
47      }
48      // Result row is returned
49      return $result;
50  }
51
52  // Executes queries and fetches/returns a multiple arrays of data
53  // - To be used when query result set is expected to contain multiple elements
54  private function execQueryFetchMultiRows($query)
55  {
56      // Stores the result in an associative array
57      $result = array();
58      // Executes the passed query (prepared statement) on the database
59      $stmt = $this->connection->query($query);
60      // Checks that there is at least one row in the query result set
61      // and the executing (prepared) statement was not empty
62      if ((mysqli_num_fields($stmt)>=1) && $stmt != null)
63      {
64          // Retreives all rows and columns in the query result set
65          $result = mysqli_fetch_all($stmt, MYSQLI_ASSOC);
66          // Closes (clears) the prepared statement
67          $stmt->close();
68      }
69      // Query result is returned
70      return $result;
71  }
72
73  //[---------------- User Account Queries ---------------------------------]
74
75  // Retreives, from the Users table, the user details matching a certain username and email
76  public function searchForUsers($username, $userEmail)
77  {
78      // Parameterised SQL Query to retreive all of the user details
79      $query = "SELECT * FROM `db648556307`.`Users` WHERE `Username` = '".$username."' OR `Email` =
'".$userEmail."'";
80      return $this->execQueryFetchSingleRow($query);
81  }
82
83  // Searches for a record, in the Users table, which matches the passed login credentials to an account
84  // If an account is found, the main user details (user's id, username and email) are returned
85  public function checkLoginCredGetUserDetails($userEmail, $userPassword)
86  {
87      // Parameterised SQL Query
88      $query = "SELECT `UserID`, `Username`, `Email` FROM `db648556307`.`Users` WHERE `Email` =
'".$userEmail."' AND `Password` = '".$userPassword."'";
89      return $this->execQueryFetchSingleRow($query);
90  }
91
92  // Inserts a the passed user details as a new record in the Users table
```

```
 93    public function addUser($username, $userEmail, $userPassword)
 94    {
 95        // Initialises the prepared INSERT statement
 96        $stmt = $this->connection->prepare("INSERT INTO `db648556307`.`Users` (`UserID`, `Username`, `Email`,
`Password`) VALUES (NULL, ?, ?, ?);");
 97        // Binds (three string type) parameters -the user details- to the prepared statement (SQL Query)
 98        $stmt->bind_param("sss", $username, $userEmail, $userPassword);
 99        // Executes the prepared statement (SQL Query) - True(successful execution)/False(unsuccessful execution) is
returned
100        $result = $stmt->execute();
101        // Closes (clears) the prepared statement
102        $stmt->close();
103        // Returns the outcome status of the query execution
104        return $result;
105    }
106
107    //[---------------Pin Pack Queries-----------------------------------]
108
109    // Receives an array of pins and adds each pin to the Pins table in the database
110    public function addPins($pinsToAdd, $packID)
111    {
112        // Initialises the prepared INSERT statement
113        $stmt = $this->connection->prepare("INSERT INTO `db648556307`.`Pins` (`PinID`, `Title`, `Hint`, `Codeword`,
`CoordLongitude`, `CoordLatitude`, `PointValue`, `PackID`) VALUES (NULL, ?, ?, ?, ?, ?, ?, ?)");
114        // Binds (three string, two decimal and two integer type) parameters -the pin details- to the prepared statement (SQL
Query)
115        $stmt->bind_param("sssddii", $pinTitle, $pinHint, $pinCodeword, $pinCoordLong, $pinCoordLat, $pinPointVal,
$packID);
116
117        // Initialses an array which will store the pin titles of pins that couldn't be added to the database
118        $pinsNotAddedList = array();
119
120        // Iterates over every pin
121        foreach ($pinsToAdd as $pin)
122        {
123            // Retreives the pin details from the pin associative array
124            $pinTitle = $pin->{"Title"};
125            $pinHint = $pin->{"Hint"};
126            $pinCodeword = $pin->{"Codeword"};
127            $pinCoordLong = $pin->{"CoordLongitude"};
128            $pinCoordLat = $pin->{"CoordLatitude"};
129            $pinPointVal = $pin->{"PointValue"};
130
131            // Attempts to add pin to database
132            if (!$stmt->execute())
133            {
134                // If not added, the pin's title is appended to 'pinsNotAddedList'
135                $pinsNotAddedList[] = $pinTitle;
136            }
137        }
138        // Closes (clears) the prepared statement
139        $stmt->close();
140        // Returns the list of pins that couldn't be added to the database
141        return $pinsNotAddedList;
142    }
143
```

```
144     // Adds a pin pack record to the PinPacks table
145     public function addPinPack($packName, $packDescrip, $creatorID, $packLocation, $gameTime, $version)
146     {
147         // Initialises the prepared INSERT statement
148         $stmt = $this->connection->prepare("INSERT INTO `db648556307`.`PinPacks` (`PackID`, `PackName`,
`PackDescription`, `Creator_UserID`, `Location`, `GameTime`, `Version`) VALUES (NULL, ?, ?, ?, ?, ?, ?)");
149         // Binds (three string and three integer type) parameters -the pack details- to the prepared statement (SQL Query)
150         $stmt->bind_param("ssisii", $packName, $packDescrip, $creatorID, $packLocation, $gameTime, $version);
151         // Executes the prepared statement (SQL Query) - True(successful execution)/False(unsuccessful execution) is
returned
152         $result = $stmt->execute();
153         // Closes (clears) the prepared statement
154         $stmt->close();
155         // Returns the outcome status of the query execution
156         return $result;
157     }
158
159     // Searches the PinPacks table for a record -matching the passed pack name, creator ID and the pack location-
160     // and retreives the record's pack ID
161     public function getPinPackID($packName, $creatorID, $packLocation)
162     {
163         // Parameterised SQL Query
164         $query = "SELECT `PackID` FROM `db648556307`.`PinPacks` WHERE `PackName` = '".$packName."' AND
`Creator_UserID` = '".$creatorID."' AND `Location` = '".$packLocation."'";
165         return $this->execQueryFetchSingleRow($query);
166     }
167
168     // Searches (and retreives from) the Pins table for all of the pins which have a given pack id
169     public function getPinsFromPack($packID)
170     {
171         // Parameterised SQL Query
172         $query = "SELECT `Title`, `Hint`, `Codeword`, `CoordLongitude`, `CoordLatitude`, `PointValue` FROM
`db648556307`.`Pins` WHERE `PackID` = '".$packID."'";
173         return $this->execQueryFetchMultiRows($query);
174     }
175
176     // Updates the pack description, game time and version number of a record in the PinPacks table
177     // where the pack id and creator id matches the one passed
178     public function alterPinPackRecord($packDescrip, $gameTime, $version, $packID, $creatorID)
179     {
180         // Initialises the prepared UPDATE statement
181         $stmt = $this->connection->prepare("UPDATE `db648556307`.`PinPacks` SET `PackDescription` = ?,
`GameTime` = ?, `Version` = ? WHERE `PinPacks`.`PackID` = ? AND `PinPacks`.`Creator_UserID` = ?;");
182         // Binds (one string and four integer type) parameters -the updated pack details- to the prepared statement (SQL
Query)
183         $stmt->bind_param("siiii", $packDescrip, $gameTime, $version, $packID, $creatorID);
184         // Executes the prepared statement (SQL Query) - True(successful execution)/False(unsuccessful execution) is
returned
185         $result = $stmt->execute();
186         // Closes (clears) the prepared statement
187         $stmt->close();
188         // Returns the outcome status of the query execution
189         return $result;
190     }
191
192     // Deletes a pin record from the Pins table where the pin details matches the details passed
```

```
193     public function deletePin($pinTitle, $pinHint, $pinCodeword, $pinCoordLong, $pinCoordLat, $pinPointVal,
$packID)
194     {
195         // Initialises the prepared DELETE statement
196         $stmt = $this->connection->prepare("DELETE FROM `db648556307`.`Pins` WHERE `Title` = ? AND `Hint` = ?
AND `Codeword` = ? AND `CoordLongitude` = ? AND `CoordLatitude` = ? AND `PointValue` = ? AND `PackID` = ?;");
197         // Binds (three string, two decimal and four integer type) parameters -the pin details- to the prepared statement (SQL
Query)
198         $stmt->bind_param("sssddii", $pinTitle, $pinHint, $pinCodeword, $pinCoordLong, $pinCoordLat, $pinPointVal,
$packID);
199         // Executes the prepared statement (SQL Query) - True(successful execution)/False(unsuccessful execution) is
returned
200         $result = $stmt->execute();
201         // Closes (clears) the prepared statement
202         $stmt->close();
203         // Returns the outcome status of the query execution
204         return $result;
205     }
206
207     // Deletes all pin records, with the passed pack id, from the Pins table
208     public function deleteAllPinsFromPack($packID)
209     {
210         // Initialises the prepared DELETE statement
211         $stmt = $this->connection->prepare("DELETE FROM `db648556307`.`Pins` WHERE `Pins`.`PackID` = ?;");
212         // Binds (an integer type) parameter -the pin's pack id- to the prepared statement (SQL Query)
213         $stmt->bind_param("i", $packID);
214         // Executes the prepared statement (SQL Query) - True(successful execution)/False(unsuccessful execution) is
returned
215         $result = $stmt->execute();
216         // Closes (clears) the prepared statement
217         $stmt->close();
218         // Returns the outcome status of the query execution
219         return $result;
220     }
221
222     //[------------------Pack Table Search---------------------------------]
223
224     // Retrieves all of the pin pack records (from the PinPacks table) which have similar creator username,
225     //  pack name or pack location to the one that is passed
226     public function getPackDetailsFromSearch($usernameFragment, $packNameFragment, $locationFragment)
227     {
228         // Parameterised Multi-table SQL Query
229         $query = "
230             /* Selects the Pin Pack details and alias the column names to match the key names used when the pack is stored
on the phone */
231             SELECT pinPackTable.`PackID`, pinPackTable.`PackName`, pinPackTable.`PackDescription` AS Description,
pinPackTable.`Location`, pinPackTable.`GameTime` AS TimeLimit, pinPackTable.`Version`,
possibleCreatorTable.`Username` as CreatorUsername, possibleCreatorTable.`UserID` AS CreatorID
232             /* Sets the source of the search query as the PinPacks table and a generated possibleCreatorTable */
233             FROM `db648556307`.`PinPacks` AS pinPackTable,
234                 /* possibleCreatorTable is created by searching the Users table for records with usernames similar to the one
passed*/
235                 (SELECT `UserID`, `Username`
236                 FROM `db648556307`.`Users`
237                 WHERE `Username` LIKE '%".$usernameFragment."%') AS possibleCreatorTable
```

```
238         /* Retreived results set are records which have similar pack names and locations to the values passed. The records
in both tables are linked via the user IDs */
239         WHERE pinPackTable.`PackName` LIKE '%".$packNameFragment."%' AND pinPackTable.`Location` LIKE
'%".$locationFragment."%' AND pinPackTable.`Creator_UserID` =  possibleCreatorTable.`UserID`
240         /* The results set are ordered alphabetically */
241         ORDER BY pinPackTable.`PackName` ASC";
242       return $this->execQueryFetchMultiRows($query);
243   }
244
245   //[---------------Leaderboard Queries-----------------------------------]
246
247   // Retrieves a score record from the PlayedPacks table where the pack id and player user
248   // id matched the values that were passed
249   public function getScoreRecord($packID, $playerUserID)
250   {
251       // Parameterised SQL Query
252       $query = "SELECT * FROM `db648556307`.`PlayedPacks` WHERE `PackID` = '".$packID."' AND
`Player_UserID` = '".$playerUserID."'";
253       return $this->execQueryFetchSingleRow($query);
254   }
255
256   // Returns the count of the total number of packs played (competitively and casually) by the user
257   public function getTotalNumPacksPlayed($userID)
258   {
259       // Parameterised SQL Query - Uses the COUNT aggregate function to
260       // count the number of pack records with the user id passed from the PlayedPacks table.
261       $query = "SELECT COUNT(`Player_UserID`) FROM `db648556307`.`PlayedPacks` WHERE `Player_UserID` =
'".$userID."'";
262       return $this->execQueryFetchSingleRow($query);
263   }
264
265   // Returns the count of the total number of packs created by the user
266   public function getTotalNumPacksCreated($userID)
267   {
268       // Parameterised SQL Query - Uses the COUNT aggregate function to
269       // count the number of pack records with the user id passed from the PinPacks table
270       $query = "SELECT COUNT(`Creator_UserID`) FROM `db648556307`.`PinPacks` WHERE `Creator_UserID` =
'".$userID."'";
271       return $this->execQueryFetchSingleRow($query);
272   }
273
274   // Returns the sum of all of the score values, scored by the specified user
275   public function getTotalNumPoints($userID, $gameType)
276   {
277       // Parameterised SQL Query - Uses the SUM aggregate function to add up all of the
278       // score values from the PlayedPacks records where the user id and gametype matches the values passed
279       $query = "SELECT SUM(`Score`) FROM `db648556307`.`PlayedPacks` WHERE `Player_UserID` = '".$userID."'
AND `GameType` = '".$gameType."'";
280       return $this->execQueryFetchSingleRow($query);
281
282   }
283
284   // Returns the total number of players who have played a specified pack with a specified game type
285   public function getTotalNumOfPlayersOfPlayedPack($packID, $gameType)
286   {
287       // Parameterised SQL Query - Uses the COUNT aggregate function to
```

```
288         // count the number of distinct score records with the pack id and gametype  passed from the PlayedPacks table.
289         $query = "SELECT COUNT(DISTINCT `Player_UserID`) FROM `db648556307`.`PlayedPacks` WHERE
`PackID` = ".$packID." AND `GameType` = '".$gameType."'";
290         return $this->execQueryFetchSingleRow($query);
291     }
292 
293     // Returns the average score of all of the score values from records which have a specified pack id and game type
294     public function getAverageScoreOfPlayedPack($packID, $gameType)
295     {
296         // Parameterised SQL Query - Users the AVG (average) aggregate function to
297         // calculate the average score of selected records from the PlayedPacks table
298         $query = "SELECT AVG(`Score`) FROM `db648556307`.`PlayedPacks` WHERE `PackID` = ".$packID." AND
`GameType` = '".$gameType."'";
299         return $this->execQueryFetchSingleRow($query);
300     }
301 
302     // Retreives the score and ranking of a specifed user for a specific pack leaderboard
303     public function getUserPackScoreAndRank($packID, $gameType, $userID)
304     {
305         // Parameterised SQL Query
306         $query = "
307             /* Retreives the user's score and rank */
308             SELECT `Score`, PlayerRank
309             /* Sets the main query's source as a new table generated form a subquery */
310                 /* 'rank' is incremented by one every row generated */
311             FROM (SELECT `Player_UserID`, `Score`, (@rank := @rank + 1) as PlayerRank
312                 /* New table is created where each record in PlayedPacks has an initial rank value of 0 */
313                 FROM `db648556307`.`PlayedPacks` CROSS JOIN (SELECT @rank := 0) CONST /* 'rank' variable is
initialised to 0 */
314                 /* Retreives records with data values which match the passed variables */
315                 WHERE `PackID` = ".$packID." AND `GameType` = '".$gameType."'
316                 /* The results set is ordered by score with the highest score appearing first */
317                 ORDER BY `Score` DESC
318                 ) AS RankedScoreRecords /* New table is aliased as 'RankedScoreRecords' */
319             /* Retreives the record containing the score and rank of the user whose ID was passed */
320             WHERE `Player_UserID` = ".$userID."";
321         return $this->execQueryFetchSingleRow($query);
322     }
323 
324     // Retrieves the highest scoring records for a specified pack and game type
325     public function getTopPackScoreRecords($packID, $gameType, $numRecords)
326     {
327         // Parameterised Multi-table SQL Query
328         $query = "
329             /* Retreives the score and Username for the highest scoring records */
330             SELECT playedPacksTable.`Score`, userTable.`Username`
331             /* Sets the data source of the query as the PlayedPacks and Users table */
332             FROM `db648556307`.`PlayedPacks` AS playedPacksTable,  `db648556307`.`Users` AS userTable
333             /* Defines the query parameters - the pack ID and game type have to match the values passed. The records in
both tables are linked via the user IDs */
334             WHERE playedPacksTable.`PackID` = ".$packID." AND playedPacksTable.`Player_UserID` =
userTable.`UserID` AND playedPacksTable.`GameType` = '".$gameType."'
335             /* Result set is ordered by score with the highest score appearing first  */
336             ORDER BY playedPacksTable.`Score` DESC LIMIT ".$numRecords."";
337         return $this->execQueryFetchMultiRows($query);
338     }
```

```
339
340     // Inserts a score record into the PlayedPacks table
341     public function addScoreRecord($packID, $playerUserID, $score, $gameType){
342         // Initialises the prepared INSERT statement
343         $stmt = $this->connection->prepare("INSERT INTO `db648556307`.`PlayedPacks` (`PackID`, `Player_UserID`,
`Score`, `GameType`) VALUES (?, ?, ?, ?);");
344         // Binds (three integer and one string type) parameter -the score record details- to the prepared statement (SQL
Query)
345         $stmt->bind_param("iiis", $packID, $playerUserID, $score, $gameType);
346         // Executes the prepared statement (SQL Query) - True(successful execution)/False(unsuccessful execution) is
returned
347         $result = $stmt->execute();
348         // Closes (clears) the prepared statement
349         $stmt->close();
350         // Returns the outcome status of the query execution
351         return $result;
352     }
353
354 }
```

loginUser.php

```
1 <?php
2
3 /*
4  *  loginUser.php
5  *  TownHunt API
6  *
7  *  This file determines if a user and associated password is found in the database
8  */
9
10 // The API/Database response is stored as an associative array
11 $response = array();
12
13 // Checks whether the server request method was POST
14 // The API file only allows POSTed data to be processed
15 if ($_SERVER['REQUEST_METHOD'] === 'POST')
16 {
17     // Retrieves the POSTed (passed) variables
18     // (htmlentities function converts all applicable "special" characters to HTML entities)
19     // This removes characters that could potentially interfere with the SQL query
20     $userEmail = htmlentities($_POST["userEmail"]);
21     $userPassword = htmlentities($_POST["userPassword"]);
22
23     // Checks to see if any required variable was not passed i.e. variables are empty
24     if(empty($userEmail) || empty($userPassword))
25     {
26         // Error flag is set to true and 'missing field(s)' error message is appended to the response array
27         $response["error"] = true;
28         $response["message"] = "One or more fields are empty";
29     }
30     else
31     {
32         // If all required values were passed then
33
```

```
34      // Temporarily imports the DBOperation class file
35      require_once '../includes/DBOperation.php';
36      // Instantiates a new DBOperation object
37      $database = new DBOperation();
38
39      // The user's password is hashed for security reasons as the database only stores a hashed version
40      // of the passwords so malicious users who gain access to the database do not know what the unhashed passwords are
41      $secureUserPassword = hash("ripemd128", $userPassword);
42      // Searches the database for the email address and the associated hashed password
43      $userDetails = $database->checkLoginCredGetUserDetails($userEmail, $secureUserPassword);
44      // If userDetails is not empty and a match was found then the email-password passed matched a user account
45      if(!empty($userDetails))
46      {
47          // Login success response generated and the user details are added to the response array
48          $response['error'] = false;
49          $response['message'] = 'User is registered';
50          $response['accountInfo'] = $userDetails;
51      }
52      else
53      {
54          // If no record was found that means that the entered details do not match any user account
55          // Login unsuccessful response generated
56          $response['error'] = true;
57          $response['message'] = 'Account not found. The email and or the password is incorrect';
58      }
59  }
60 }
61 else
62 {
63      // If the request method isn't POST then an error message is returned
64      $response['error'] = true;
65      $response['message'] = 'You are not authorised.';
66 }
67 // The Database/API response is encoded in a JSON format and echoed (sent) to the requester
68 echo json_encode($response);
```

registerUser.php

```
1 <?php
2
3 /*
4  *  registerUser.php
5  *  TownHunt API
6  *
7  *  This file registers new users and appends the details to the database
8  */
9
10 // The API/Database response is stored as an associative array
11 $response = array();
12
13 // Checks whether the server request method was POST
14 // The API file only allows POSTed data to be processed
15 if ($_SERVER['REQUEST_METHOD'] === 'POST')
16 {
17      // Retreives the POSTed (passed) variables
```

```
18  // (htmlentities function converts all applicable "special" characters to HTML entities)
19  // This removes characters that could potentially interfere with the SQL query
20  $username = htmlentities($_POST["username"], ENT_QUOTES);
21  $userEmail = htmlentities($_POST["userEmail"]);
22  $userPassword = htmlentities($_POST["userPassword"]);
23
24  // Checks to see if any required variable was not passed i.e. variables are empty
25  if(empty($username) || empty($userEmail) || empty($userPassword))
26  {
27      // Error flag is set to true and 'missing field(s)' error message is appended to the response array
28      $response["error"] = true;
29      $response["message"] = "One or more fields are empty";
30  }
31  else
32  {
33      // If all required values were passed then
34
35      // Temporarily imports the DBOperation class file
36      require_once '../includes/DBOperation.php';
37      // Instantiates a new DBOperation object
38      $database = new DBOperation();
39
40      // Searches the database for record with the passed username and or email. If any records
41      // are found, then another account with the same username and or email already exists
42      $userDetails = $database->searchForUsers($username, $userEmail);
43
44      // Checks to see fi any records were returned
45      if(empty($userDetails))
46      {
47          // If no existing accounts were found then the new account is registered
48
49          // The user's password is hashed for security reasons as the database only stores a hashed version
50          // of the passwords so malicious users who gain access to the database do not know what the unhashed passwords
are
51          $secureUserPassword = hash("ripemd128", $userPassword);
52
53          // Attempts to add the new user to the database
54          if ($database->addUser($username, $userEmail, $secureUserPassword))
55          {
56              // Successful response genereated to indicate that the account was added to the database
57              $response['error'] = false;
58              $response['message'] = 'User added sucessfully';
59          }
60          else
61          {
62              // Unsuccessful response genereated to indicate that the account was not added to the database
63              $response['error'] = true;
64              $response['message'] = 'Could not add user';
65          }
66      }
67      else
68      {
69          // 'Account already exists' Error response generated
70          $response['error'] = true;
71          $response['message'] = 'Account already exists. Please enter a different email and or username';
72      }
```

```
73     }
74 }
75 else
76 {
77     // If the request method isn't POST then an error message is returned
78     $response['error'] = true;
79     $response['message'] = 'You are not authorised.';
80 }
81 // The Database/API response is encoded in a JSON format and echoed (sent) to the requester
82 echo json_encode($response);
```

addPlayedPackRecord.php

```
1 <?php
2
3 /*
4  *  addPlayedPackRecord.php
5  *  TownHunt API
6  *
7  *  This file adds attempts to add a pack score record to the database
8  */
9
10 // The API/Database response is stored as an associative array
11 $response = array();
12
13 // Checks whether the server request method was POST
14 // The API file only allows POSTed data to be processed
15 if ($_SERVER['REQUEST_METHOD'] === 'POST')
16 {
17     // Retrieves the POSTed (passed) variables
18     // (htmlentities function converts all applicable "special" characters to HTML entities)
19     // This removes characters that could potentially interfere with the SQL query
20     $packID = htmlentities($_POST["PackID"]);
21     $playerUserID = htmlentities($_POST["PlayerUserID"]);
22     $score = htmlentities($_POST["Score"]);
23     $gameType = htmlentities($_POST["GameType"]);
24
25     // Checks to see if any required variable was not passed i.e. variables are empty
26     if(empty($packID) || empty($playerUserID) || !isset($score) || empty($gameType))
27     {
28         // If a variable is missing then the error flag is set to true and an "missing field" error message
29         // is appended to the response array
30         $response["error"] = true;
31         $response["message"] = "One or more fields are empty";
32     }
33     else
34     {
35         // If all required values were passed then
36
37         // Temporarily imports the DBOperation class file
38         require_once '../includes/DBOperation.php';
39         // Instantiates a new DBOperation object
40         $database = new DBOperation();
41
42         // Checks to see if there is already a score for the specified pack and player
```

```
43          // Searches for and retrieves any score record with the passed pack id and player user id
44      $userDetails = $database->getScoreRecord($packID, $playerUserID);
45      // If none was found then the passed score is elegible to be added to the leaderboard
46      if(empty($userDetails))
47      {
48          // Attempts to add the score record to the database via the DBOperation object
49          if ($database->addScoreRecord($packID, $playerUserID, $score, $gameType))
50          {
51              // Prepares the success message if the score was added successfully
52              $response['error'] = false;
53              $response['message'] = 'Record Added to PlayedPack Table';
54          }
55          else
56          {
57              // Prepares the unsuccessful message if the score couldn't be added
58              $response['error'] = true;
59              $response['message'] = 'Could not add record';
60          }
61      }
62      else
63      {
64          // If the user has already played the pack then this is conveyed in the response message
65          $response['error'] = false;
66          $response['message'] = 'Records for first play through of pack already exists';
67      }
68  }
69 }
70 else
71 {
72      // If the request method isn't POST then an error message is returned
73      $response['error'] = true;
74      $response['message'] = 'You are not authorised.';
75 }
76 // The Database/API response is encoded in a JSON format and echoed (sent) to the requester
77 echo json_encode($response);
```

registerNewPinPack.php

```
1 <?php
2
3 /*
4  *  registerNewPinPack.php
5  *  TownHunt API
6  *
7  *  This file registers (adds to the database) new pin packs
8  */
9
10 // The API/Database response is stored as an associative array
11 $response = array();
12
13 // Checks whether the server request method was POST
14 // The API file only allows POSTed data to be processed
15 if ($_SERVER['REQUEST_METHOD'] === 'POST')
16 {
17      // Retrieves the POSTed (passed) variables
```

```
18    // (htmlentities function converts all applicable "special" characters to HTML entities)
19    // This removes characters that could potentially interfere with the SQL query
20    $packName = htmlentities($_POST["packName"], ENT_QUOTES);
21    $packDescrip = htmlentities($_POST["description"], ENT_QUOTES);
22    $creatorID = htmlentities($_POST["creatorID"]);
23    $packLocation = htmlentities($_POST["location"], ENT_QUOTES);
24    $gameTime = htmlentities($_POST["gameTime"]);
25    // Initial pack version is set to 0
26    $version = 0;
27
28    // Checks to see if any required variable was not passed i.e. variables are empty
29    if(empty($packName) || empty($packDescrip) || empty($creatorID) || empty($packLocation) || empty($gameTime))
30    {
31       // Error flag is set to true and 'missing field(s)' error message is appended to the response array
32       $response["error"] = true;
33       $response["message"] = "One or more fields are empty";
34    }
35    else
36    {
37       // If all required values were passed then
38
39       // Temporarily imports the DBOperation class file
40       require_once '../includes/DBOperation.php';
41       // Instantiates a new DBOperation object
42       $database = new DBOperation();
43
44       // Checks the database to see if the pack already exists
45       $doesPackExist = !empty($database->getPinPackID($packName, $creatorID, $packLocation));
46       if (!$doesPackExist)
47       {
48          // If the pack doesn't already exists then an attempt to add the new pack to the database occurs
49          if ($database->addPinPack($packName, $packDescrip, $creatorID, $packLocation, $gameTime, $version))
50          {
51             // New pack was registered successfully - Success response generated
52             $response['error'] = false;
53             $response['message'] = 'Pack added sucessfully';
54             // New pack's id is retreived and appended to the response array
55             $response['packData'] = $database->getPinPackID($packName, $creatorID, $packLocation);
56          }
57          else
58          {
59             // Unsuccessful registration of the pack error response is generated
60             $response['error'] = true;
61             $response['message'] = 'Could not add pack';
62          }
63       }
64       else
65       {
66          // 'Pack Already Exists' error response is generated
67          $response['error'] = true;
68          $response['message'] = 'Pack Already Exists';
69       }
70    }
71 }
72 else
73 {
```

```
74     // If the request method isn't POST then an error message is returned
75     $response['error'] = true;
76     $response['message'] = 'You are not authorised.';
77 }
78 // The Database/API response is encoded in a JSON format and echoed (sent) to the requester
79 echo json_encode($response);
```

updatePinPack.php

```
 1 <?php
 2
 3 /*
 4  *  updatePinPack.php
 5  *  TownHunt API
 6  *
 7  *  This file updates the database records to reflect the changes in the pin pack
 8  */
 9
10 // The API/Database response is stored as an associative array
11 $response = array();
12
13 // Checks whether the server request method was POST
14 // The API file only allows POSTed data to be processed
15 if ($_SERVER['REQUEST_METHOD'] === 'POST')
16 {
17     // Retrieves the POSTed data (JSON string)
18     $jsonData = $_POST["data"];
19     // Decodes the passed JSON string
20     $packData = json_decode($jsonData);
21
22     // Checks if json data was sent and received
23     if(empty($jsonData))
24     {
25         $response["error"] = true;
26         $response["message"] = "No JSON data was sent";
27     }
28     // Checks if JSON data was decoded sucessfully
29     elseif (is_null($packData))
30     {
31         // If $packData is null then the JSON string wasn't decoded
32         // "JSON couldn't be decoded" error response is generated
33         $response["error"] = true;
34         $response["message"] = "JSON data could not be decoded";
35     }
36     else
37     {
38         // Pack details are retreived from the passed data
39         $packDescrip = $packData->{"Description"};
40         $gameTime = $packData->{"TimeLimit"};
41         $version = $packData->{"Version"};
42         $packID = $packData->{"PackID"};
43         $creatorID = $packData->{"CreatorID"};
44
45         // Temporarily imports the DBOperation class file
46         require_once '../includes/DBOperation.php';
```

```
47        // Instantiates a new DBOperation object
48        $database = new DBOperation();
49
50        // Attempts to update pack details
51        if ($database->alterPinPackRecord($packDescrip, $gameTime, $version, $packID, $creatorID))
52        {
53            // Attempts to delete old existing pack pins
54            if ($database->deleteAllPinsFromPack($packID))
55            {
56                // Retrieves iterable pack pin array
57                $packPins = $packData->{"Pins"};
58
59                // Sends all the pack pins to be added to the database
60                // the DBOperation function addPins returns an array of pins that
61                // couldn't be added to the database
62                $response['pinsNotAddedList'] = $database->addPins($packPins, $packID);
63
64                // If 'pinsNotAddedList' is empty all pins were sucessfully added to DB
65                if (empty($response['pinsNotAddedList']))
66                {
67                    // Successfully updated pack response generated
68                    $response['error'] = false;
69                    $response['message'] = 'Pack Updated';
70                }
71                else
72                {
73                    // Error response generated
74                    $response['error'] = true;
75                    $response['message'] = 'Could not add pin(s)';
76                }
77            }
78            else
79            {
80                // Error response generated
81                $response["error"] = true;
82                $response["message"] = "Old pin data could not be deleted";
83            }
84        }
85        else
86        {
87            // Error response generated
88            $response["error"] = true;
89            $response["message"] = "Pack details could not be altered";
90        }
91    }
92 }
93 else
94 {
95    // If the request method isn't POST then an error message is returned
96    $response['error'] = true;
97    $response['message'] = 'You are not authorised.';
98 }
99 // The Database/API response is encoded in a JSON format and echoed (sent) to the requester
100 echo json_encode($response);
```

packSearch.php

```php
<?php

/*
 *  packSearch.php
 *  TownHunt API
 *
 *  This file retrieves details of packs found from searching the database with
 *  the passed search criterion
 */

// Function decodes a passed string (HTML Entity) back to standard characters
function decode($stringToDecode) {
    return html_entity_decode($stringToDecode, ENT_QUOTES);
}

// The API/Database response is stored as an associative array
$response = array();

// Checks whether the server request method was POST
// The API file only allows POSTed data to be processed
if ($_SERVER['REQUEST_METHOD'] === 'POST')
{
    // Retrieves the POSTed (passed) variables
    // (htmlentities function converts all applicable "special" characters to HTML entities)
    // This removes characters that could potentially interfere with the SQL query
    $usernameFragment = htmlentities($_POST["usernameFragment"], ENT_QUOTES);
    $packNameFragment = htmlentities($_POST["packNameFragment"], ENT_QUOTES);
    $locationFragment = htmlentities($_POST["locationFragment"], ENT_QUOTES);

    // Temporarily imports the DBOperation class file
    require_once '../includes/DBOperation.php';
    // Instantiates a new DBOperation object
    $database = new DBOperation();

    // Searches the database for packs with details similar to the criterion (fragments) passed
    $searchResult = $database->getPackDetailsFromSearch($usernameFragment, $packNameFragment,
$locationFragment);
    // Checks if search results (packs) were returned
    if (!empty($searchResult))
    {
        // Packs Were Found

        // Error flag is set to false
        $response['error'] = false;

        // Initialises the decoded pack detail array
        $decodedSearchResult = [];
        // Iterates through the search results and decodes all of the pack details
        foreach ($searchResult as $pack)
        {
          // Decoded pack results are appended to the decoded pack details array
          $decodedSearchResult[] = array_map("decode",$pack);
        }
        // Decoded pack details array is appended to the response array
```

```
54        $response['searchResult'] = $decodedSearchResult;
55    }
56    else
57    {
58        // 'No Matches Found' error generated
59        $response['error'] = true;
60        $response['message'] = 'Could not find any matches in the database';
61    }
62 }
63 else
64 {
65    // If the request method isn't POST then an error message is returned
66    $response['error'] = true;
67    $response['message'] = 'You are not authorised.';
68 }
69 // The Database/API response is encoded in a JSON format and echoed (sent) to the requester
70 echo json_encode($response);
```

getPinsFromPack.php

```
1 <?php
2
3 /*
4  *  getPinsFromPack.php
5  *  TownHunt API
6  *
7  *  This file retrieves all of the pins with a specified pack ID
8  */
9
10 // Function decodes a passed string (HTML Entity) back to standard characters
11 function decode($stringToDecode) {
12    return html_entity_decode($stringToDecode, ENT_QUOTES);
13 }
14
15 // The API/Database response is stored as an associative array
16 $response = array();
17
18 // Checks whether the server request method was POST
19 // The API file only allows POSTed data to be processed
20 if ($_SERVER['REQUEST_METHOD'] === 'POST')
21 {
22    // Retrieves the POSTed user ID
23    // (htmlentities function converts all applicable "special" characters to HTML entities)
24    // This removes characters that could potentially interfere with the SQL query
25    $packID = htmlentities($_POST["packID"]);
26
27    // Checks to see if the user ID was passed
28    if(empty($packID))
29    {
30        // If a variable is missing then the error flag is set to true and an "missing user ID" error message
31        // is appended to the response array
32        $response["error"] = true;
33        $response["message"] = "No Pack ID was passed";
34    }
35    else
```

```
36    {
37        // If all required values were passed then
38
39        // Temporarily imports the DBOperation class file
40        require_once '../includes/DBOperation.php';
41        // Instantiates a new DBOperation object
42        $database = new DBOperation();
43
44        // Retrieves all pins from with a specific Pack ID from the database
45        $pins = $database->getPinsFromPack($packID);
46
47        // Error flag set to false
48        $response['error'] = false;
49        // Checks to see if any pins were found
50        if (!empty($pins))
51        {
52            // If pins were found then they are decoded and appended to the response array
53            // Pins found flag is set to true
54            $response['packContainsPinsFlag'] = true;
55
56            // Initialises the array holding the decoded pins
57            $decodedPins = [];
58            // Decodes each pin (and associated details) and appends it to the decoded pins array
59            foreach ($pins as $pin)
60            {
61              $decodedPins[] = array_map("decode",$pin);
62            }
63            // Decoded pin array added to the response array
64            $response['Pins'] = $decodedPins;
65        }
66        else
67        {
68            // Pins found flag is set to false
69            $response['packContainsPinsFlag'] = false;
70        }
71    }
72 }
73 else
74 {
75    // If the request method isn't POST then an error message is returned
76    $response['error'] = true;
77    $response['message'] = 'You are not authorised.';
78 }
79 // The Database/API response is encoded in a JSON format and echoed (sent) to the requester
80 echo json_encode($response);
```

getPackLeaderboardInfo.php

```
1 <?php
2
3 /*
4  *  getPackLeaderboardInfo.php
5  *  TownHunt API
6  *
7  *  This file retrieves the leaderboard information for a specific pack and the
```

```
 8  *  performance of a specific player for the pack
 9  */
10
11 // The API/Database response is stored as an associative array
12 $response = array();
13
14 // Checks whether the server request method was POST
15 // The API file only allows POSTed data to be processed
16 if ($_SERVER['REQUEST_METHOD'] === 'POST')
17 {
18    // Retrieves the POSTed (passed) variables
19    // (htmlentities function converts all applicable "special" characters to HTML entities)
20    // This removes characters that could potentially interfere with the SQL query
21    $packID = htmlentities($_POST["packID"]);
22    $userID = htmlentities($_POST["userID"]);
23    // Game type variable is set to "competitive" and the number of records to retrieve is set to 10
24    // as in this version of the TownHunt App/API the leaderboard/account stats are only concerned
25    // with packs that were played competitively and the top ten scores
26    $gameType = "competitive";
27    $numRecords = 10;
28
29    // Checks to see if any required variable was not passed i.e. variables are empty
30    if(empty($packID) || empty($userID))
31    {
32       // Error flag is set to true and 'missing id(s)' error message is appended to the response array
33       $response["error"] = true;
34       $response["message"] = "No User ID  and or Pack ID passed";
35    }
36    else
37    {
38       // If all required values were passed then
39
40       // Temporarily imports the DBOperation class file
41       require_once '../includes/DBOperation.php';
42       // Instantiates a new DBOperation object
43       $database = new DBOperation();
44
45       // The top 10 scores records and their details are attempted to be retrieved from the database
46       $topScoreRecords = $database->getTopPackScoreRecords($packID, $gameType, $numRecords);
47       // Checks if any score were found
48       if (!empty($topScoreRecords))
49       {
50          // If scores were found it means the pack has been played competitively
51          // Error flag set to false by default
52          $response['error'] = false;
53          // The score records are appended to the response array
54          $response['topScoreRecords'] = $topScoreRecords;
55
56          // The total number of players who have played the pack is retrieved
57          $numOfPlayersOfPack = $database->getTotalNumOfPlayersOfPlayedPack($packID,
$gameType)["COUNT(DISTINCT `Player_UserID`)"];
58          // Checks to see if a value for $numOfPlayersOfPack was retrieved
59          if (isset($numOfPlayersOfPack))
60          {
61             // If a value was set then the number of players is appended to the array
62             $response['numOfPlayersOfPack'] = $numOfPlayersOfPack;
```

```
63          }
64          else
65          {
66              // Otherwise an the error flag is set to true and an error message is generated
67              $response['error'] = true;
68              $response['message'][] = 'Could not get the number of players who have played the pack';
69          }
70
71          // Retreives the average competitive score for the pack
72          $averageScore = $database->getAverageScoreOfPlayedPack($packID, $gameType)["AVG(`Score`)"];
73          // Checks to see if a value for $averageScore was retrieved
74          if (isset($averageScore))
75          {
76              // If a value was set then the average score (rounded to 1 decimal place) is appended to the array
77               $response['averageScore'] = round($averageScore, 1);
78          }
79          else
80          {
81              // Otherwise an the error flag is set to true and an error message is generated
82              $response['error'] = true;
83              $response['message'][] = 'Could not get average score of everyone who has played the pack';
84          }
85
86          // Retreives the score and ranking of the specified (logged in) user
87          $userScoreAndRank = $database->getUserPackScoreAndRank($packID, $gameType, $userID);
88          $score = $userScoreAndRank["Score"];
89          $rank = $userScoreAndRank["PlayerRank"];
90          // Checks to see if values were set for the user's score and ranking
91          if (isset($score) && isset($rank))
92          {
93              // If a values were set then they are appended to the array
94              $response['userScore'] = $score;
95              $response['userRank'] = $rank;
96
97          }
98          else
99          {
100              // User score and rank is set to N/A as the user hasn't played the pack competitively
101              $response['userScore'] = "N/A ";
102              $response['userRank'] = "N/A";
103              $response['message'][] = 'Could not get user score and rank';
104          }
105      }
106      else
107      {
108          // No user has played the pack competitively
109          // Error message is generated
110          $response['error'] = true;
111          $response['message'][] = "Pack Hasn't Been Played Competitively";
112      }
113  }
114 }
115 else
116 {
117     // If the request method isn't POST then an error message is returned
118     $response['error'] = true;
```

```
119     $response['message'] = 'You are not authorised.';
120 }
121 // The Database/API response is encoded in a JSON format and echoed (sent) to the requester
122 echo json_encode($response);
```

getAccountStats.php

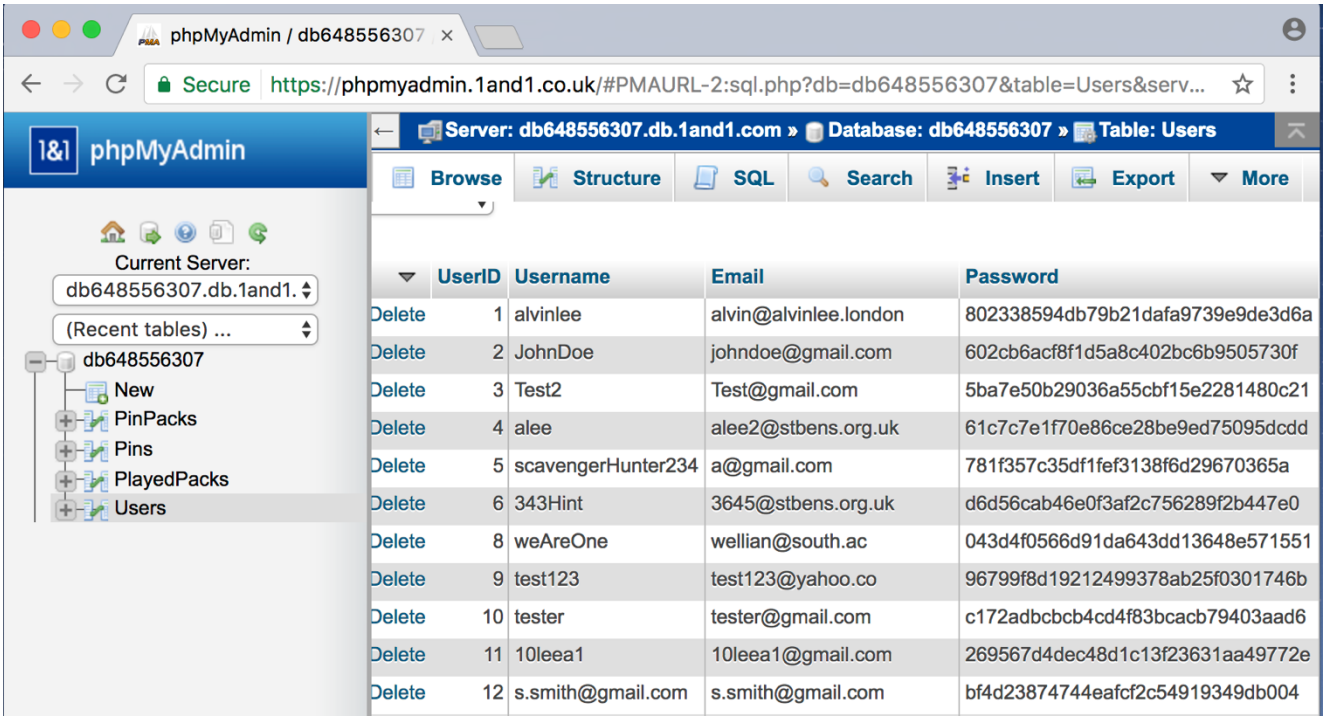
```
1 <?php
 2
 3 /*
 4  *  getAccountStats.php
 5  *  TownHunt API
 6  *
 7  *  This file retrieves information/stats about a specified account
 8  */
 9
10 // The API/Database response is stored as an associative array
11 $response = array();
12
13 // Checks whether the server request method was POST
14 // The API file only allows POSTed data to be processed
15 if ($_SERVER['REQUEST_METHOD'] === 'POST')
16 {
17     // Retrieves the POSTed user ID
18     // (htmlentities function converts all applicable "special" characters to HTML entities)
19     // This removes characters that could potentially interfere with the SQL query
20     $userID = htmlentities($_POST["userID"]);
21     // Game type variable is set to "competitive" as in this version of the TownHunt App/API
22     // The leaderboard/account stats are only concerned with packs that were played competitively
23     $gameType = "competitive";
24
25     // Checks to see if the user ID was passed
26     if(empty($userID))
27     {
28         // If a variable is missing then the error flag is set to true and an "missing user ID" error message
29         // is appended to the response array
30         $response["error"] = true;
31         $response["message"] = "No User ID passed";
32     }
33     else
34     {
35         // If all required values were passed then
36
37         // Temporarily imports the DBOperation class file
38         require_once '../includes/DBOperation.php';
39         // Instantiates a new DBOperation object
40         $database = new DBOperation();
41
42         // The total number of packs created by the specified user is retrieved from the database
43         $totNumCreated = $database->getTotalNumPacksCreated($userID);
44         // Checks to see if a value has been set for $totNumCreated and therefore if the database query was successful
45         if (isset($totNumCreated))
46         {
47             // If a value has been set, then the error flag is set to false and the total number
48             // of packs created by the user is added to the response array
```

```
49          $response['error'] = false;
50          $response['totalNumPacksCreated'] = $totNumCreated["COUNT(`Creator_UserID`)"];
51      }
52      else
53      {
54          // Error flag is set to true and an error message is appended to the response message array
55          $response['error'] = true;
56          $response['message'][] = 'Could not retrieve number of packs created';
57      }
58      // The total number of packs played by the specified user is retrieved from the database
59      $totNumPlayed = $database->getTotalNumPacksPlayed($userID)["COUNT(`Player_UserID`)"];
60      // Checks to see if a value has been set for $totNumPlayed and therefore if the database query was successful
61      if (isset($totNumPlayed))
62      {
63          // If a value has been set, then the error flag is set to false if there wasn't an error in an earlier query
64          // and the total number of packs played by the user is added to the response array
65          $response['error'] = false || $response['error'];
66          $response['totalNumPacksPlayed'] = $totNumPlayed;
67      }
68      else
69      {
70          // Error flag is set to true and an error message is appended to the response message array
71          $response['error'] = true;
72          $response['message'][] = 'Could not retrieve number of packs played';
73      }
74
75      // The total number of points scored competitively by the specified user is retrieved from the database
76      $totNumCompPoints = $database->getTotalNumPoints($userID, $gameType)["SUM(`Score`)"];
77      // The query either results in nul (if the user hasn't played any packs yet) or the number of points
78      // scored therefore the error is set to false if there wasn't an error in an earlier query
79      $response['error'] = false || $response['error'];
80      // Checks if the value retrieved was null
81      if (is_null($totNumCompPoints))
82      {
83          // If so, the total number of points is set to 0 and appended to the array
84          $response['totalNumCompPoints'] = "0";
85      }
86      else
87      {
88          // the total number of points is set by the value retrieved and appended to the array
89          $response['totalNumCompPoints'] = $totNumCompPoints;
90      }
91  }
92 }
93 else
94 {
95     // If the request method isn't POST then an error message is returned
96     $response['error'] = true;
97     $response['message'] = 'You are not authorised.';
98 }
99 // The Database/API response is encoded in a JSON format and echoed (sent) to the requester
100 echo json_encode($response);
```

Testing

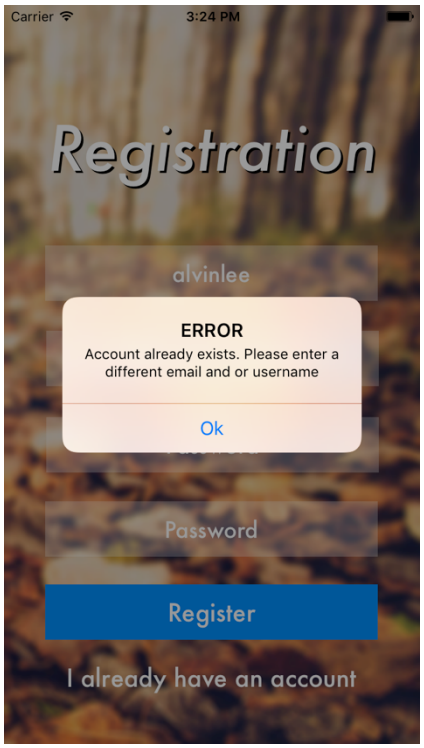
Due to the large system, testing documented here will focus on the main parts of the system.

Registration Page



	UserID	Username	Email	Password
Delete	1	alvinlee	alvin@alvinlee.london	802338594db79b21dafa9739e9de3d6a
Delete	2	JohnDoe	johndoe@gmail.com	602cb6acf8f1d5a8c402bc6b9505730f
Delete	3	Test2	Test@gmail.com	5ba7e50b29036a55cbf15e2281480c21
Delete	4	alee	alee2@stbens.org.uk	61c7c7e1f70e86ce28be9ed75095dcdd
Delete	5	scavengerHunter234	a@gmail.com	781f357c35df1fef3138f6d29670365a
Delete	6	343Hint	3645@stbens.org.uk	d6d56cab46e0f3af2c756289f2b447e0
Delete	8	weAreOne	wellian@south.ac	043d4f0566d91da643dd13648e571551
Delete	9	test123	test123@yahoo.co	96799f8d19212499378ab25f0301746b
Delete	10	tester	tester@gmail.com	c172adbcbcb4cd4f83bcacb79403aad6
Delete	11	10leea1	10leea1@gmail.com	269567d4dec48d1c13f23631aa49772e
Delete	12	s.smith@gmail.com	s.smith@gmail.com	bf4d23874744eafcf2c54919349db004

This is the current Users table in the online database. It contains all the information about users who have registered accounts on the app.



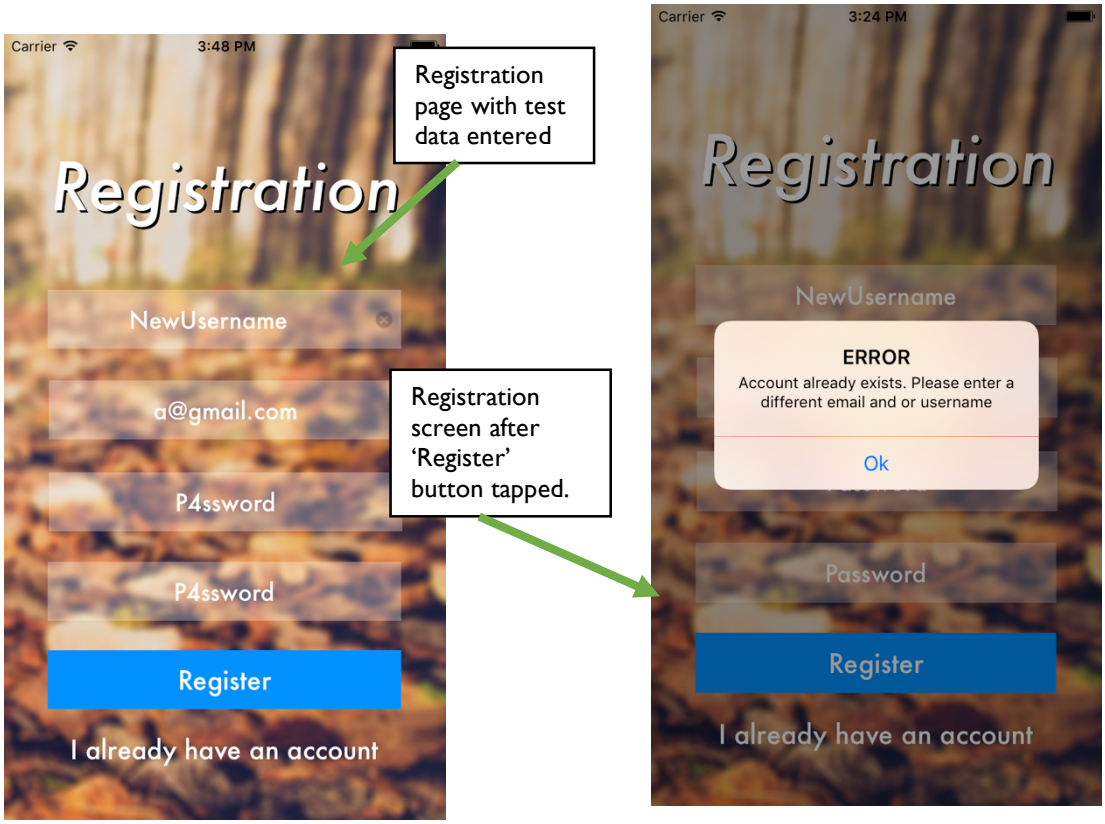
<u>Test 1:</u>
Test purpose: Check that the system doesn't allow a new account to be added to the database if the username is already taken by another account.

Test data: The following was entered into the registration form: username = 'alvinlee', email = 'NewUsername@gmail.com', password = 'Password' and re-entered password = 'Password'.

Expected outcome: An account with the username 'alvinlee' already exists in the Users table (user ID 4). Therefore the system should not allow the account to be created and an error should be displayed.

Actual outcome: PASS Account registration did not occur (see image to the left).

Test 2:



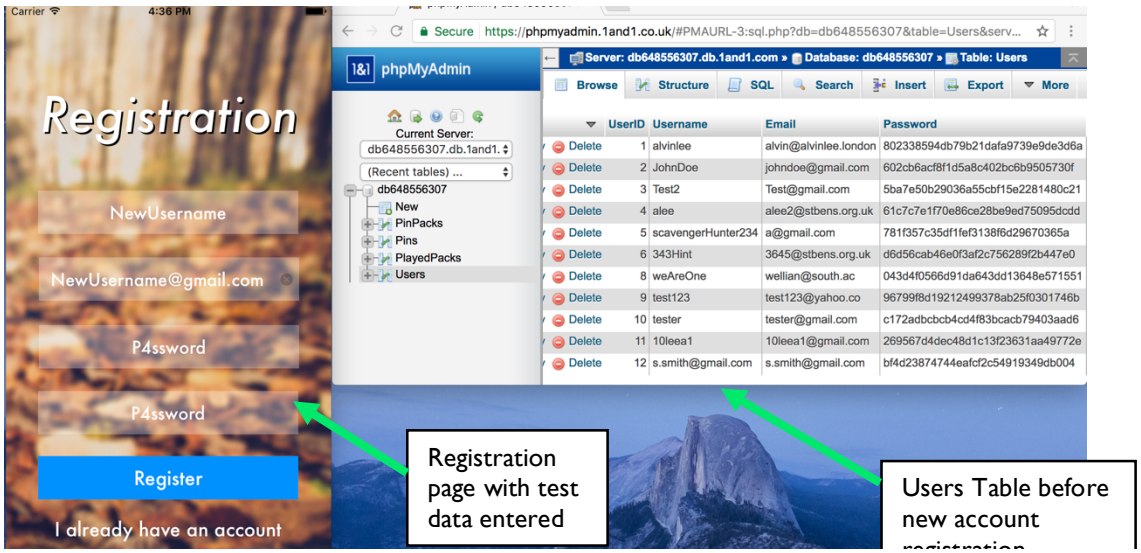
Test purpose: Check that the system doesn't allow a new account to be added to the database if the email address is already taken by another account.

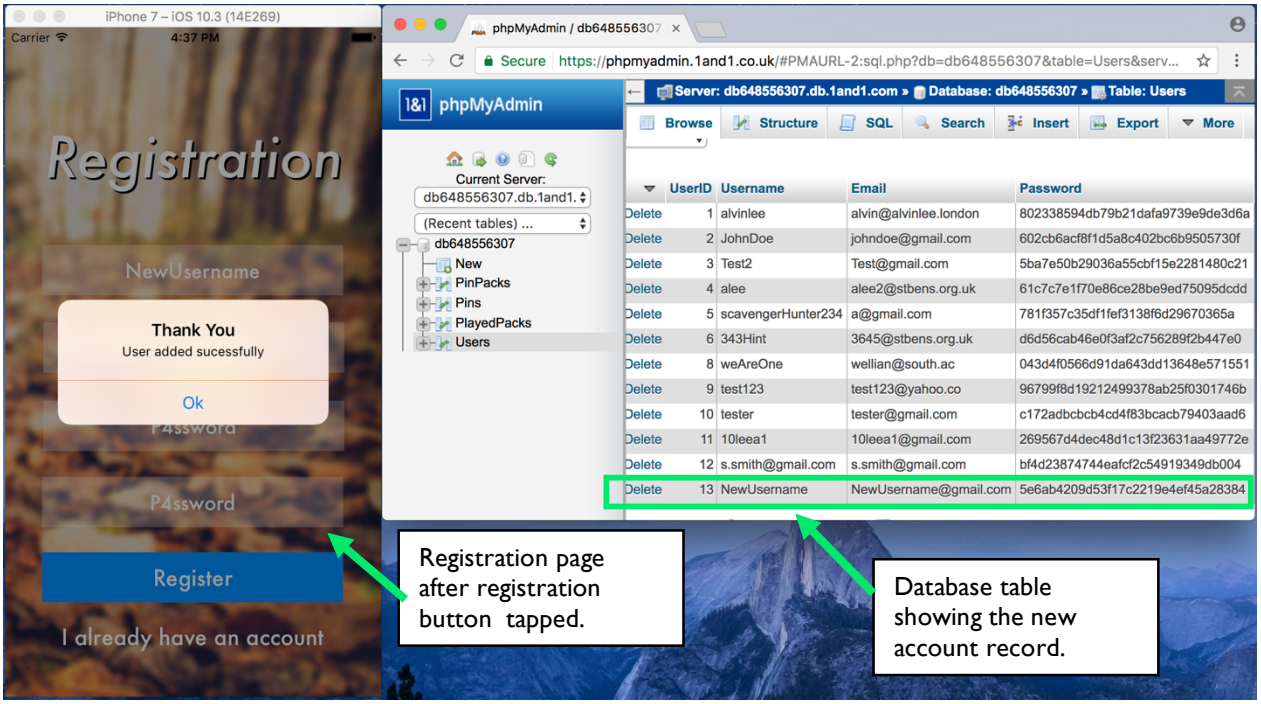
Test data: The following was entered into the registration form: username = 'NewUsername, email = 'a@gmail.com', password = 'P4ssword' and re-entered password = 'P4ssword'.

Expected outcome: An account with the email address 'a@gmail.com" already exists in the Users table (user ID 5). Therefore the system should not allow the account to be created and an error should be displayed.

Actual outcome: PASS Account registration did not occur.

Test 3:





Test purpose: Check that the system allows a new account to be added to the database if the email address and username are not already taken by another account.

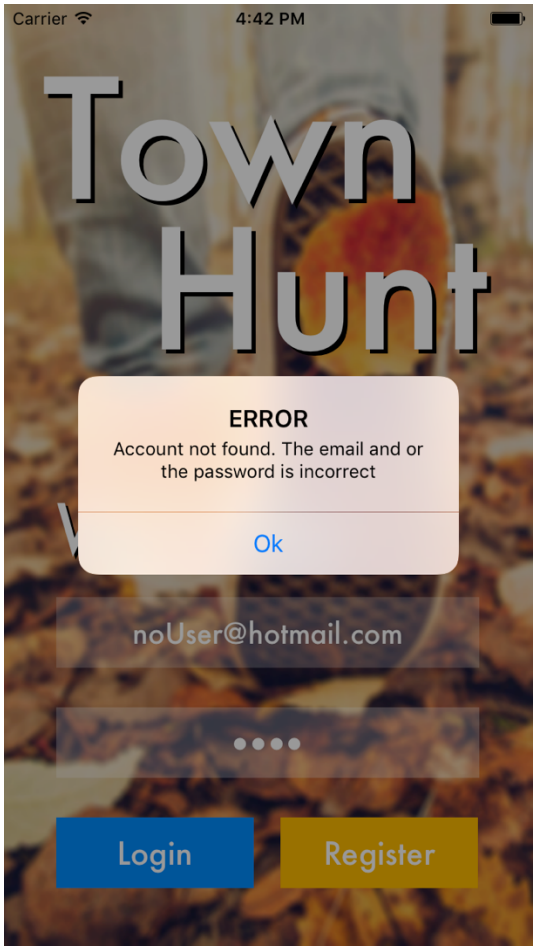
Test data: The following was entered into the registration form: username = 'NewUsername, email = 'NewUsername@gmail.com', password = 'P4ssword' and re-entered password = 'P4ssword'.

Expected outcome: No account with the email/username exists in the Users table. Therefore the new account should be registered by the system.

Actual outcome: PASS Account registration did occur. The new account details were successfully passed from the phone to the database API via a HTTP POST request. The details were then successfully inserted in to the database.

(Email validity checks use the same regex pattern matching functions as shown being tested in the documented design)

Login Page



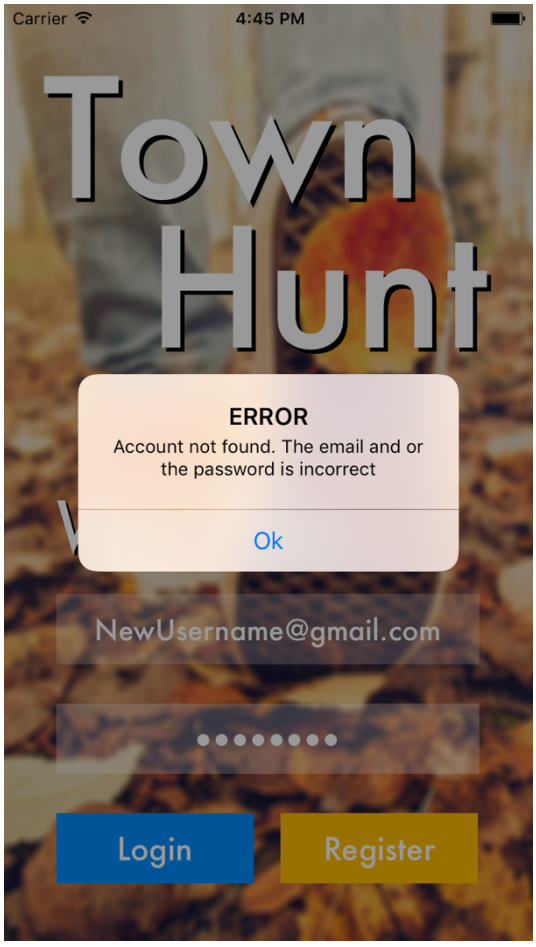
Test 4:

Test purpose: Check that the system doesn't allow an account to be logged in if the email address entered doesn't match an account found in the database.

Test data: The following was entered into the login form: email = 'NoUser@hotmail.com' and password = 'P444'.

Expected outcome: No account with the email exists in the Users database. Therefore, no account should be logged into the phone.

Actual outcome (see image to right): PASS Account login did not occur as the email could not be matched to any record in the Users table.



Test 5:

Test purpose: Check that the system doesn't allow an account to be logged in, if the password entered doesn't match the password for the account (specified by the email) in the database.

Test data: The following was entered into the login form: email = 'NewUsername@gmail.com' and password = 'P444'.

Expected outcome: The password entered doesn't match the password stored for this account ("P444" ≠ "P4ssword").

Actual outcome (see image to left): PASS Account login did not occur as the password did not match the password stored in the record (with the email 'NewUsername@gmail.com')

Test 6:
Test purpose: Check that the system allows an account to be logged in, if the password entered matches the password for an existing account (specified by the email). The phone app should also store details about the user in the app.

Test data: The following was entered into the login form: email = 'NewUsername@gmail.com' and password = 'P4ssword'.

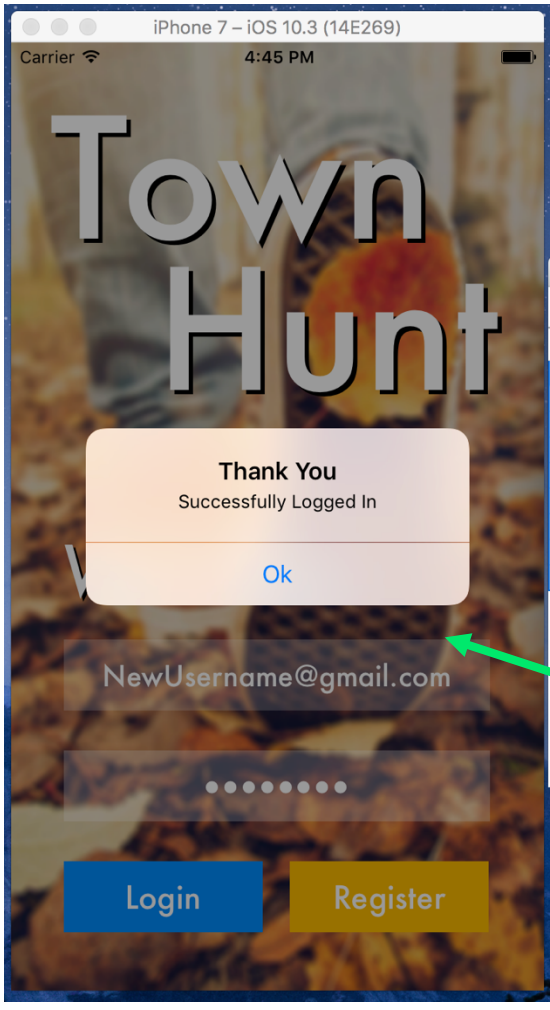
Expected outcome: The password entered matches the password stored for the existing 'NewUsername@gmail.com' account. The account should be allowed to login to the system and the user ID, username and email of the account should be stored locally on the phone.

Actual outcome (see image to left): PASS Account login did occur. The logged in user's details are stored locally in persistent storage. When the user logs out the user's info is deleted from storage.

UserDefaults persistent data table before account is logged into. No account information is found.

london.alvinlee.ALee.TownHunt.plist 〉 No Selection

Key	Type	Value
▼ Root	Dictionary	(4 items)
▶ listOfLocalUserIDs	Array	(0 items)
isAppAlreadyLaunchedOnce	Boolean	YES
isInitialSetupRequired	Boolean	NO
isUserLoggedIn	Boolean	NO



UserDefaults persistent data table after account is logged into. The logged in user's ID, username and email is stored locally. The isUserLoggedIn flag is set to true ("YES")

london.alvinlee.ALee.TownHunt.plist 〉 No Selection

Key	Type	Value
▼ Root	Dictionary	(8 items)
UserID	String	13
Username	String	NewUsername
▶ listOfLocalUserIDs	Array	(1 item)
▶ UserID-10-Packs	Dictionary	(2 items)
isUserLoggedIn	Boolean	YES
UserEmail	String	NewUsername@gmail.com
isInitialSetupRequired	Boolean	NO
isAppAlreadyLaunchedOnce	Boolean	YES

Login screen after user has successfully logged in

After user has logged out the user information is deleted.

london.alvinlee.ALee.TownHunt.plist 〉 No Selection

Key	Type	Value
▼ Root	Dictionary	(4 items)
▶ listOfLocalUserIDs	Array	(0 items)
isAppAlreadyLaunchedOnce	Boolean	YES
isInitialSetupRequired	Boolean	NO

Video Testing

For the rest of the testing of the TownHunt system, I have produced a video which demonstrates the rest of the system. The video can be found on YouTube through the following link:

https://youtu.be/AFH8NqMzfH8

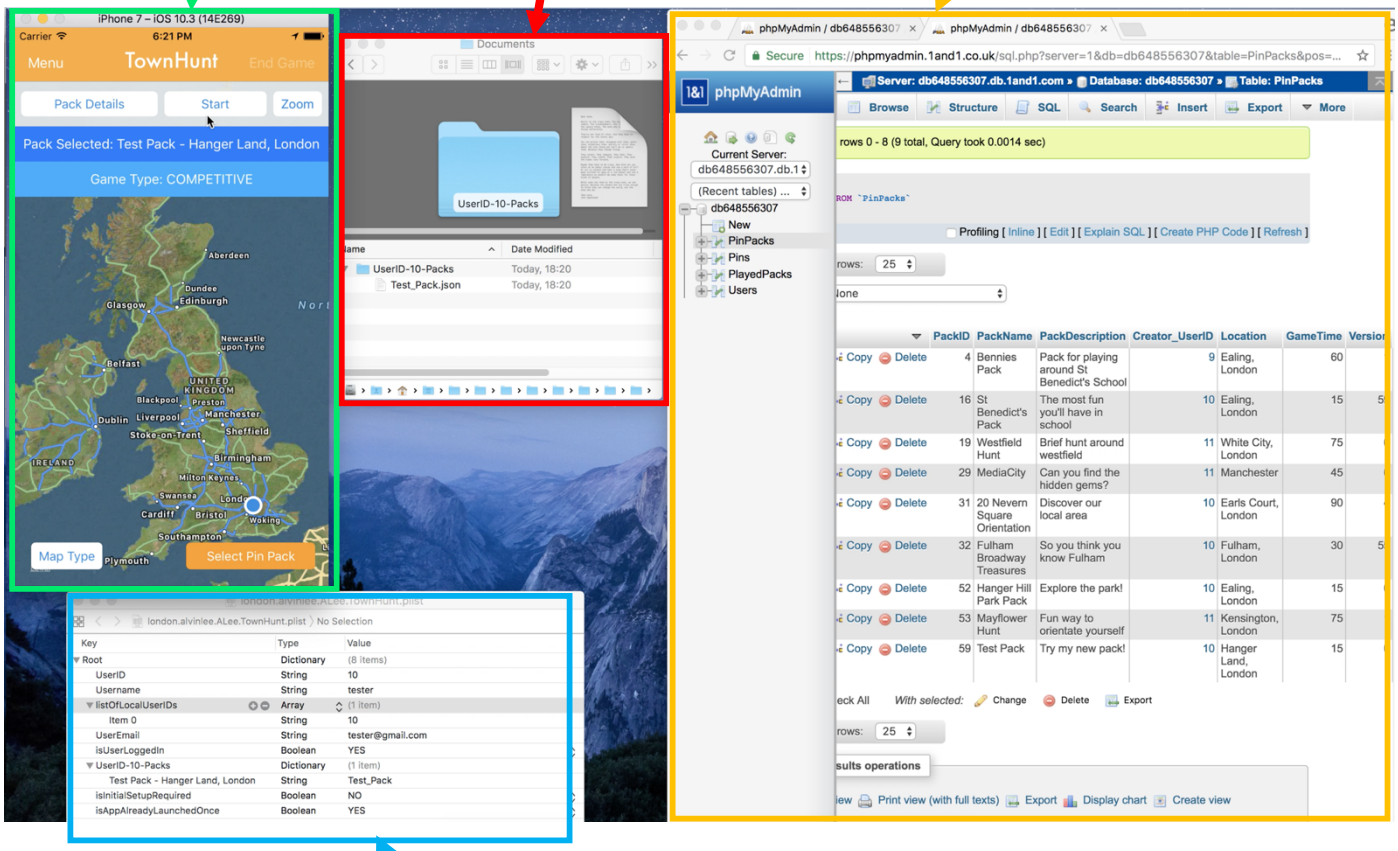
This document will contain commentary about what is being tested in the video.

Explanation of the Windows Seen in the Video

Simulation of the TownHunt app. Running on a virtual iPhone

Contents of the Documents folder, on the virtual iPhone, where the pack JSON files are stored.

Online database window. This shows the contents of a specified table (one of Users, Pins, PinPacks or PlayedPacks) in the online MySQL database. The database is set up so that record IDs (e.g. PackID) are incremented automatically. The IDs in some tables are not sequential as records have been deleted by me during the development process/testing.



UserDefaults data table window. This shows the contents of the cached persistent data stored on the phone. The pack's linked lists, logged in user info and important system flags are stored here.

Test Number	Test Purpose	Expected Outcome (& Actual Outcome Discussion)	Pass /Fail	Video Time
7	Ensure that the app can deal with no packs being on the phone.	There are no packs (0:00) on the phone (as can been seen by the lack of files inside the Documents folder and empty linked lists). When the 'Edit New Packs' button is tapped an error message should be presented to the user indicating that there are no packs on the phone (0:06). Likewise when the pin pack creator screen is opened 'No Packs Found' should be displayed in the pack selector (0:10)	PASS	0:00 to 0:10
8	Ensure a new pack can be created with its details being stored both locally on the device and in the PinPacks table inside the online database.	The new pin pack registration form will appear (0:11). The user should be allowed to enter details about the new pack (0:18 test data entered can be seen in the form – pack name = 'Test Pack'). The game time should be changeable –incremented/ decremented by 15 mins - via the stepper (0:42). Minimum game time allowed should be 15 mins and maximum 300 mins. Once the create pack button is tapped (0:55) the pack file should be saved locally as a JSON file (filename as 'Test_Pack.json') in the Documents directory inside the subfolder relating to the creator user ID (in this case 'Users-10-Packs'). The pack linked lists should also be updated. 'listOfLocalUserIDs' should be appended with the user ID 10. A new array should be created called 'UserID-10-Packs' which contains the display pack name-location pair as the key with the associated value being the filename ('UserID-10-Packs' list can be seen in the video with the key 'Test Pack – Hanger Land, London' having the associated value 'Test_Pack'). The PinPacks database should be updated with the new pack's details (1:06 the table can be seen to be updated with the new pack record).	PASS	0:11 to 1:15
9	Ensure that packs with less than 5 pins are not playable.	The newly created pack should be selectable in the pack selector of the main game screen (1:28). When the pack is selected and attempted to be played, an error should appear informing the user that the pack is not playable (1:33) due to 'Too Few Pins'.	PASS	1:23 to 1:33
10	Ensure that the user can select his/her own pack to edit in the pack creator	The user should be able to see his/her created packs inside the pack creator view. A selected pack should be allowed to be edited.	PASS	1:41

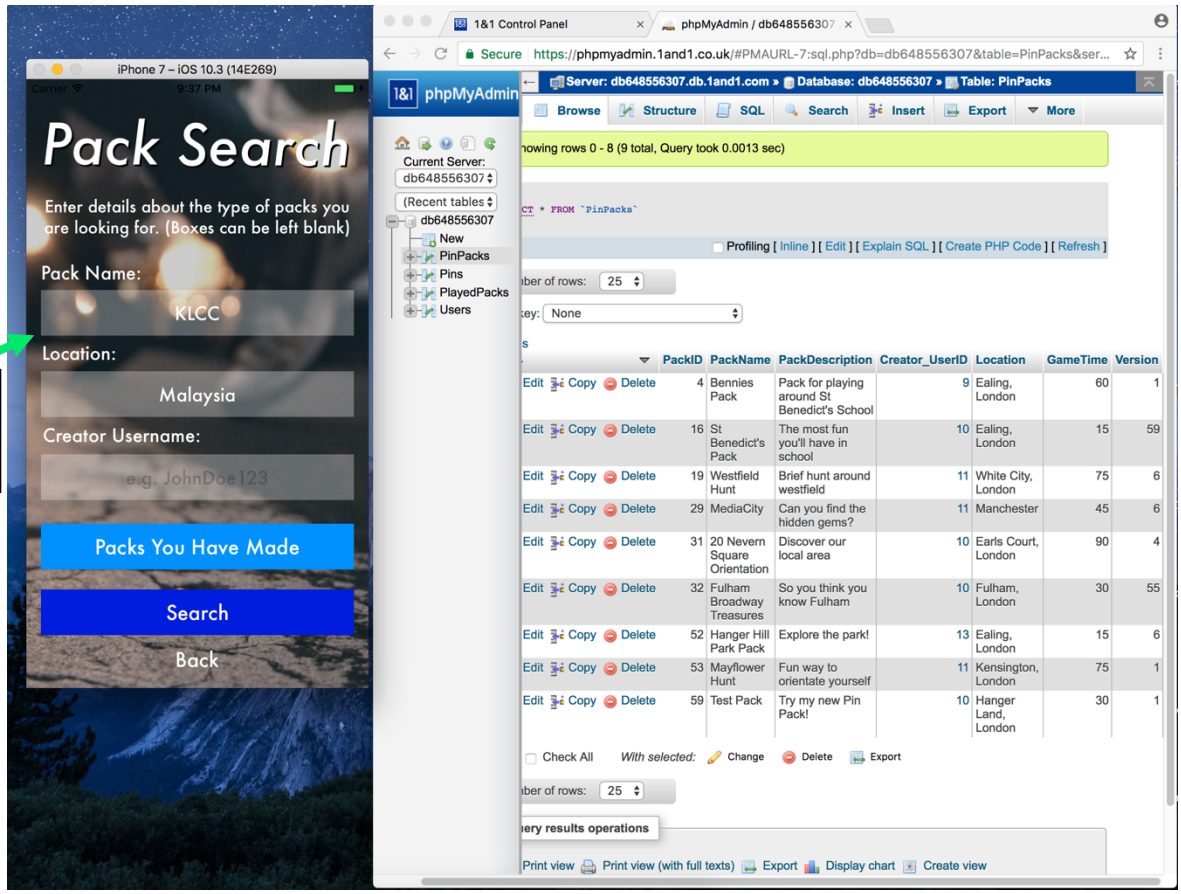
11	Ensure that a new pin, and its details, can be added to the map inside the pin pack editor.	Long tapping the screen should add a generic empty PinLocation object to the map (1:53). Tapping 'Add Details' button should centre the map around the new pin and a form to enter the pin's details should appear (1:58). These details should be able to be saved to the PinLocation object (2:22 –details of the PinLocation object can be seen in the pin list view (2:33)). When the new pin is added to the map the 'Total Pins' display should be incremented by one and the 'Max Points' display should change to reflect the new 'Max Points' value.	PASS	1:50 to 2:26
12	Ensure that the Pin List lists all of the pins in the pack and pins are deletable.	Tapping the Pin List button should present a list displaying all the pins in the pack with their respective details (3:08 also demonstrates this). When a pin row is tapped, the full pin details should be displayed with an option to delete the pin. If the delete option is selected the selected pin should be removed from the editor (in the video the deleted pin can be seen being disappearing from the table and the map).	PASS	2:29 to 2:43,
13	Ensure that created packs details are editable	Tapping the Pack Details button should present a form with the pack's details. Then tapping the Edit Details button should make the game time and brief description editable.	PASS	2:49 to 3:08
14	Ensure that changes to the pack are updated both locally on the device and in the online database	(Original Test_Pack.json file contents can be seen on the screen 2:44). When the user exits the pack creator and agrees to save the changes to the pack, the JSON file should be updated to reflect the changes (3:25 – New pins, updated game time and description can be seen in the updated JSON file compared to the original). The Pins table in the database should be updated to reflect the Pin changes in the pack (3:36 – the new pack pins are added to the Pins table). The PinPacks table should also be updated with any changes to the pack's description or game time (3:53 – the PinPack table's record 'Test Pack' has had its description and game time updated to reflect the changes)	PASS	3:15 to 4:00
15	Ensure that packs which haven't been played competitively don't display a leaderboard	When a user tries to access a leaderboard with no records an error informing that the pack hasn't been played competitively yet will be displayed. (In the video the PlayedPacks table can be seen. Since no records with the selected pack's ID exist the 'Pack Hasn't Been Played Competitively' alert is displayed)	PASS	4:08 to 4:12

16	Ensures that scores achieved by the creator of the pack are not inserted to the database	When the creator a pack attempts to play his/her own pack an alert is presented informing the user that no score for achieved by them will be added to the PlayedPacks table – table that store score records. (In the video after the game is 'played' the PlayedPacks table is refreshed and no additional record has been added. Therefore no score from the creator of the pack was inserted into the database)	PASS	4:15 to 4:31
17	Ensures that Pin Pack Store Search can search the database for packs and return the pack details of the packs found.	The user can define the search criteria. Once the Search button is tapped, a list of all packs which are similar to the search criteria should be presented with the associated details. (In the video, the PinPack table is displayed with the respective details. The expected search result can be determined by looking at the details in the PinPack table e.g. In the first test 'ben' is entered into as the pack name criteria. The expected resulting packs are "St Benedict's Pack" and "Bennies Pack". In all of the tests all of the expected results appeared as the actual search results. NB 'tester' has the User ID of 10. From testing, I have discovered that the 'Packs You Have Made' button search mechanism also returns packs made by users with similar usernames to the person logged in if any exist, e.g. if 'tester' was logged in, packs made by 'tester123' would also be shown. However this is a minor error.)	PASS	4:32 to 6:00
18	To check that Packs can be downloaded from the database and stored locally	When the user taps on a pack from the search results table, the user should be presented with the option to download the pack. If the user chooses to download the pack, the pack and its corresponding details should be downloaded from the database and stored locally. The pack linked lists should be maintained. (6:42 – the downloaded JSON file's pins can be seen to match the pins –with the pack ID of 52- in the database table)	PASS	6:03 to 7:36
19	To ensure that downloaded packs that were made by the logged in user can be edited in the pack editor.	"Fulham Broadway Treasures", "St Benedict's Pack" and "Hanger Hill Park Pack" were all downloaded from the database. However only the first two packs were made by the logged in user. Only these two should appear alongside 'Test Pack' in the pack creator selector screen. These packs can then be loaded into the editor and edited.	PASS	7:38 to 8:15
20	Ensure that packs can be played with	The 'Hanger Hill Pack' is selected and the game type is set to 'Competitive'. When the start button is	PASS	8:48 to

	the actual main game mechanics working	tapped, the 15 min timer should start and five pins should initially be added to the game. New pins should then be added randomly to the game (a new pin can be seen/heard added to the game at 9:00 just below the mouse). The pins can be tapped and the associated codeword can be entered into the presented text boxes. The user is then told if the answer entered was correct/incorrect. If correct, the pin's points should be added to the score and the pin should be removed from the game. After the timer reaches to 0 the game should end with the final score being displayed to the user.		10:19
21	Ensure that packs can be played in casual mode.	Casual mode should be able to be chosen inside the pack selector. When the Start button is tapped then the an info alert should notify the user that casual mode has been selected. All of the pins from the pack should be added the map. The game mechanics system for handling the entry of the pin's codeword (shown in test 20) should still work. The game should end as the user taps the 'End Game' button.	PASS	11:14 to 11:49
22	Ensure that the leaderboard mechanics work.	A score obtained by the player for a pack should only be stored in the leaderboard if there isn't a score record, for a specific user/pack combo, already in the PlayedPacks table. This ensures that scores are only stored for first time playthroughs of a pack. The leaderboard should display the statistics and only the top 10 records of 'competitive' game scores. If no score for the logged in user is found then 'N/A' should be displayed for the User's rank and score. (Explanation of video: At 8:33 the original leaderboard screen for the 'Hanger Hill Park Pack' – pack 52- can be seen as well as the PlayedPacks table. The pack statistics (total number of players and average score) and the leaderboard records are displayed as expected. Since the user –ID 10- hasn't played the pack yet, the ranking/score is indeed 'N/A'. At 10:20, when the game ends, the PlayedPacks table is refreshed and the new score record can be seen. When the game is played competitively again (10:29) and ended, a new record for user 10 is not added to the table. At 10:59, the leaderboard for 'Hanger Hill Park Pack' is displayed again and now the pack statistics, user rank/score and top 10 records reflect the addition of a new score record. (At 11:09 I deleted the score record for user 10 for testing purposes.) At 11:14 the pack is played again in 'casual' mode. At 11:48 the casual score can be seen to be added to the PlayedPacks table. At 12:03 the leaderboard is shown again with	PASS	10:38 to 12:29

		the new 'casual' score record not being used in the leaderboard. At 12:09 the pack is played again in 'competitive' mode and the score obtained is not added to the PlayedPacks table. This system ensures that users can't cheat by playing the pack in 'casual' mode first and then replaying in 'competitive' mode.)		
23	Ensure that when packs are deleted from the device, the linked lists and Documents folder are updated.	When packs are chosen by the user be deleted, the JSON file containing the pack should be deleted from the device and linked lists updated to remove reference to the deleted pack. If no more packs made by the creator are found then the creator's subdirectory and linked list should be removed.	PASS	12:29 to 13:14
24	Ensure that the device checks for an internet connection before any HTTP request is made.	If the internet connection is turned off and the user attempts to run a process, which needs to send a HTTP request, an error message should be presented informing the user that no internet connection is found. The user should then be forced to connect to the internet. Once internet connectivity is found the process called should run normally. (In the video the user attempts to access the leaderboard. However, due to the lack of internet connectivity, an error is raised. When the internet is turned back on the leaderboard is loaded)	PASS	13:18 to 13:39
25	Ensure that the app can handle the lack of user's GPS location.	When no GPS coordinate for the user's location is found, i.e. not shown on the map, and the Zoom button is tapped, an error should be shown. (In the video, the GPS location is initially turned off and an error alert is shown when the user taps the Zoom button. At 13:47 the simulated GPS location is turned on and the Zoom button's functionality is demonstrated)	PASS	13:39 to 13:56
26	Ensure that the map type can be changed	When the Map Type button is tapped the hybrid satellite map should change to a standard flat map type and vice versa.	PASS	13:57 to14:01
27	Ensure that the logged in user's details are deleted from persistent storage after the user chooses to logout	When the user chooses to logout from the device the user's ID, email address and username should be deleted from persistent storage. The isUserLoggedIn flag should also be set to false ("NO"). The login screen should then be shown on the screen.	PASS	14:03 to 14:08

Test 28:



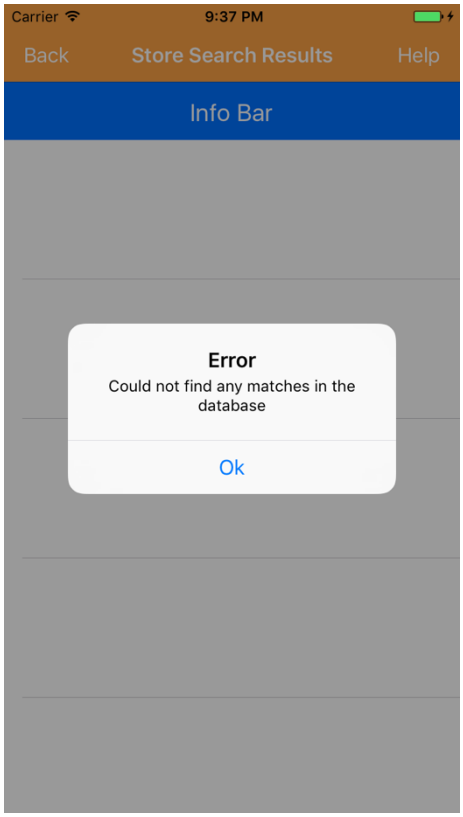
Pack Search Form with test data entered.

Test purpose: Check that the user is alerted with an error message when no packs are found to match the search criteria entered in the Pack Store Search.

Test data: The following was entered into the search form: Pack Name = 'KLCC' and Location = 'Malaysia'.

Expected outcome: Since no pack records have the Pack Name of 'KLCC' and the location of 'Malaysia'. An error message should be displayed informing the user that no matches were found.

Actual outcome (see image to right): PASS



Test 29:

Test purpose:Check that the account statistics displayed in the 'Account Information' page are correct.

Test data: 'tester' account was logged into on the phone.

Expected outcome: Logged in user's ID = 10, username = 'tester', email = 'tester@gmail.com', number of packs created = 2, numbers of packs played = 4 and total number of competitive points scored is 145. This information, calculated from info in the database tables, should be displayed to the user

Actual outcome (see images below): PASS Correct account statistics were displayed.

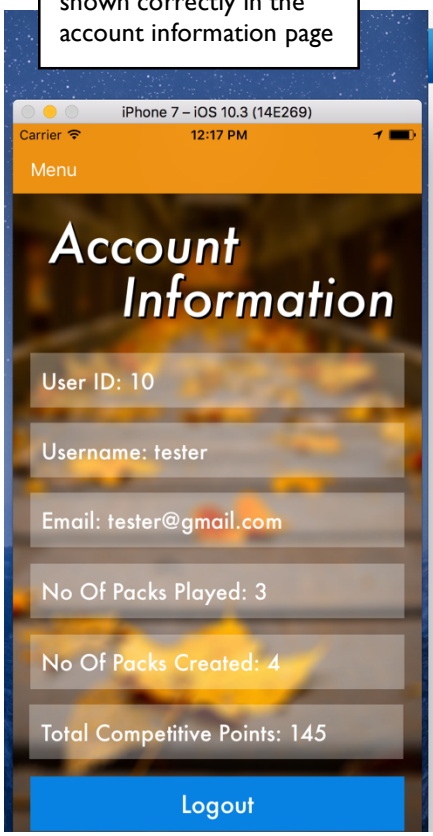
User has created four packs (records where Creator_UserID = 10)

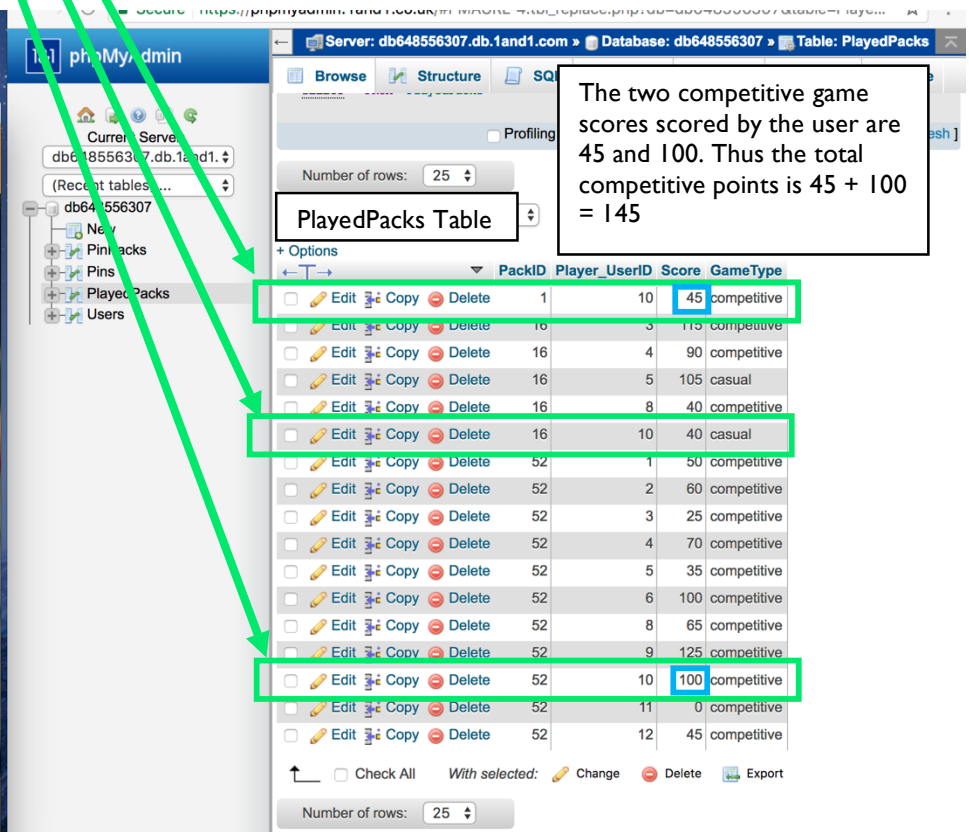
PinPacks Table

PackID	PackName	PackDescription	Creator_UserID	Location	GameTime	Version
4	Bennies Pack	Pack for playing around St Benedict's School	9	Ealing, London	60	1
16	St Benedict's Pack	The most fun you'll have in school	10	Ealing, London	15	59
19	Westfield Hunt	Brief hunt around westfield	11	White City, London	75	6
29	MediaCity	Can you find the hidden gems?	11	Manchester	45	6
31	20 Nevern Square Orientation	Discover our local area	10	Earls Court, London	90	4
32	Fulham Broadway Treasures	So you think you know Fulham	10	Fulham, London	30	55
52	Hanger Hill Park Pack	Explore the park!	13	Ealing, London	15	6
53	Mayflower Hunt	Fun way to orientate yourself	11	Kensington, London	75	1
59	Test Pack	Try my new Pin Pack!	10	Hanger Land, London	30	1

User has played three packs. (Both casual and competitive games are counted).

The account statistics are shown correctly in the account information page





The two competitive game scores scored by the user are 45 and 100. Thus the total competitive points is 45 + 100 = 145

Evaluation

Comparison of Technical Solution with Objectives (in italics)

1. *Storage Systems*
    a. *Local Storage*
        i. *Pin Packs, and associated details, should be able to be stored locally on the device as JSON files in an organised manner. The app should be able to read and write to these files*
        ii. *Linked lists should be used to store the file access information. These linked lists should be maintained and updated when new files are saved or old files are deleted*

This objective has been met as pin packs and their details are stored locally in JSON files. Each pack file is stored in its respective creator's subdirectory (e.g. User-10-Packs) within the Documents directory. Using the user defined Local Storage Handler Class (code on page 109), the app can extract and save pack data to the JSON files. This class also maintains the linked lists containing the pack's filename/display name information. Through using linked lists packs are accessible via direct access.

    b. *Online Storage*
        i. *MySQL database should be created to store the following TownHunt information: User account details, Pin Pack details, Pin details and leaderboard data (scores scored for the first play-through of a pack). This information should be stored as four interlinked tables.*

This objective has been met. Four interlinked tables have been created in the MySQL server. The 'Users' table stores all of the information about the account details. The 'PinPacks' table stores all of the pin pack details (without the pins). The 'Pins' table stores all of the pins and their associated details. The 'PlayedPacks' table stores all of the scores for a specific pack played by a user. Only the score scored for the first play through of a pack is stored.

2. *PHP API Bridge*
    a. *PHP API Bridge must allow data transfer between the App and the database. Data should be both retrieved and inserted into the online database via (multi-table) parameterised SQL queries. Statistics about tables should be retrieved via SQL aggregate functions. Data should be sent back to the app in the form of JSON strings.*

This objective has been met. Data is successfully transferred between the app and the database via a parameterised web service API (the TownHunt API that I made). The app first uses the Database Interaction class (code on page 107) to send HTTP POST requests with data to the API. Various API scripts then interpret the received request and execute functions inside the DBOperation class (code on page 115). The functions query the SQL database with parameterised SQL queries e.g. aggregate SQL functions are used to retrieve the statistics (like average scores) about the pack leaderboard and a cross-tabled parameterised query to search for packs which meet the user specified search criterion.

3. *Account System*
    a. *Users should be allowed to sign up for an account.*
    b. *Users should be allowed to login to an account with their login details stored locally on the device.*
    c. *Statistics about the user -total number of packs played/created and total number of competitive points scored- should be retrieved from the database and presented to the user.*

This objective has been met. Users are able to register for a new account and there are validation checks in place - regex pattern matching (code on page 106) - to ensure that the email entered is valid. The registered account details are stored in the online database. The app also allows users to login to an existing account. Once the account is verified by the API then the username, ID and email are stored locally on the device. This information is displayed in the Account Information screen along with statistics about the user (the total number of packs played/created and total number of competitive points scored).

4. *Main Game*
    a. *Users should be able to select a pack from local storage that they want to play. This pack should be then loaded into the game and be playable.*
    b. *Users should be able to select which game type (competitive or casual) and the game play will change according to the selection.*

This objective has been met. Users can select a pack from a list of local packs inside the Pack Selector. Inside the Pack Selector one of two modes can be selected: 'Competitive' or 'Casual'. The pack's details (such as pins and game time) are then loaded into the main game screen where the pack can be played.

    c. *Users should be able to start and play a scavenger-hunt game.*
        i. *A map should be displayed. Icons, representing the pins, should be shown on the map. The user's location should also be displayed.*
        ii. *Users should be able to tap on a pin and enter the codeword. The user should be alerted if the codeword entered was correct or incorrect. If the codeword was correct points should be added to a running score.*
        iii. *If casual mode was selected, all the pins in the pack should be added to the map.*
        iv. *If competitive mode was selected then five pins should be initially added to the map, the rest of the pins should be added at random times to the map. A timer counting down should be displayed and the game should be ended when the timer reaches 0.*
        v. *When the game ends, the final scores should be sent to the database and appended to the leaderboard table if it this was the first play-through of the pack.*

This objective has been met. The MainGameScreenViewController.swift (page 35) class deals with the main game screen. Inside the main game screen a map is displayed with the user's location. After the start button is tapped, pins from the loaded pack are added to the map as generated PinLocation objects (annotations) during the game; either all at once (casual mode) or starting with five pins and then the rest randomly added over the duration of the game (competitive mode). Pins can be tapped and its hint/question is posed

to the user. The user can then enter their answer/codeword in a textfield. The system would then alert the user if the answer was correct or not via an alert and a sound. If the answer entered matches the codeword stored for the pin then the point value of the pin is added to the running score total. If competitive mode was selected a timer (specified by the pack's game time details) will countdown until it reaches 0 and ends the game. Tapping the End Game button in either mode will also end the game. At the end of the game the final score is presented to the user and the score is sent to be added to the database. The API checks if there is already a score for that specific pack and player combination and only adds the score record if none was found.

d. *A pack-specific leaderboard should be accessible and display the top 10 scores for the pack as well as the score and ranking of the logged in user (if they have played the pack and aren't the creator).*

This objective has been met. There is a 'View Pack Leaderboard' inside the pack selector which will lead to the leaderboard for the selected pack. The leaderboard data is obtained via the API querying the database. The top 10 scores are obtained along with the player details of the users who scored them. These scores are presented in a scrollable TableView. The score/rank of the logged in user is calculated by the API and presented to the user along with the average score and total number of players who have played the pack. The parameterised SQL queries (some are cross-table) used to obtain the leaderboard data can be seen on page 120-122.

e. *The user should be able to zoom into their location on the map.*

This objective has been met. The player is able to tap a Zoom button which centres the map around the GPS location of the user. If there is no GPS signal and the Zoom button was tapped then an alert is presented to the user.

5. *Pack Creator*
    a. *Users should be able to create their own pin packs.*
        i. *A map should be displayed. Icons, representing the pins, should be shown on the map. The user's location should also be displayed.*
        ii. *Users should be allowed to add/delete pins (and respective details) to their pack.*
        iii. *The user should be able to save their packs. These packs should then be uploaded the online database as well as stored locally.*

This objective has been met. Users are able to register new packs (see page 62). This new pack's details are stored locally as a JSON file and inserted into the PinPacks table in the database. Packs made by the user can be loaded into the pack editor and edited (see page 66). The main editor screen has a map where the user can add a new pin by long tapping on the screen which adds a generic empty PinLocation object. The details about the new pin can then be added. Pins can also be deleted from the pack and the user is able to change the game time and the pack description. When the user saves their pack, the changes are reflected both locally in the pack's JSON file and in the database tables.

b. *The user should be able to zoom into their location on the map.*

This objective has been met. See 4.e.

6. *Pack Store*
    a. *Users should be able to search for packs (based on the pack's name, the pack's location and or the user name of the pack's creator) from the online database. A results list of packs should be presented to the user.*

This objective has been met. There is a search page where the user can enter in search criteria to search the database for packs which have values similar to the criteria entered. The search criteria could be the pack's name, pack's location or the creator of the pack's username. A result list of packs and associated pack details are returned from the database and presented to the user in a table of results.

    b. *The user should be able to select a pack and download it to the phone. This pack should be stored as a JSON file.*

This objective has been met. When the user taps on a pack from the search results table, the user is presented with the option to download the pack. If the user chooses to download the pack, the pack and its corresponding details are downloaded from the database and stored locally as a JSON file. The pack linked lists are then updated.

    c. *Users should be able to edit and delete pack files stored locally on the phone.*

This objective has been met. In the 'Edit Packs On Phone' table, packs can be selected and the user is given the option to delete the pack from the phone. If a pack is deleted then the linked lists are updated. The downloaded packs can be edited in the pack creator/editor if the pack was originally made by the logged in user.

User Feedback

I loaded the TownHunt app into my iPhone and gave it to Juliana Boudville. She then made a Pin Pack around 20 Nevern Square hotel (one of the Mayflower Collection hotels located in Earls Court London). Jacqueline Chua, Juliana Boudville and I then played the pack after Juliana had finished making the pack. I then re-interviewed them to obtain their feedback.

Interview with Juliana Boudville

Did TownHunt meet your expectations?

A) Yes, the whole TownHunt app does everything we talked about at our initial meeting. The final product you have made is very professional looking; I really like the general aesthetics of the app.

Pin Packs can be created for each of the hotels, downloaded onto guest's phones and played. Guests are able to see all of the pins in the pack by playing in 'casual' mode.

How did you find the pack creator?

A) Registering a new pack was very easy with the simple registration form. However, it would be nice to be able to add more than 30 characters in the brief description. In the actual editor, initially I wasn't sure what to do. I instinctively tapped the 'Add Pin Details' and was prompted on how to add pins. This along with the 'Help' button inside the list of pin table was very useful to show me what to do. The process of adding and removing new pins was quite intuitive and I can see my staff easily updating/creating new hotel packs through the TownHunt editor.

Would you still be interested in using TownHunt to help orientate guests?

A) Of course! When I was making the app, I showed it to one of the hotel guests the app and she commented that it would be a nice way to "quickly find out about the hotel's local area". It will also be able to replace our current system of physical maps. I could also for see guests downloading other packs based around certain tourist sites (like Trafalgar Square). Whilst playing in 'casual' mode they will be, in a way, guided around the area. Thus, making TownHunt a fun way for tourists to visit all the points of interest at a location.

Do you have suggestions on how to improve the app?

A) I only really have minor suggestions. As discussed earlier, probably extending the limit of the pack description. In certain screens displaying information, such as the leaderboard, the full title of the pack isn't shown. However, it is easy to infer what pack the information is related to as the beginning and the end of the pack name is shown.

Interview with Jacqueline Chua

Did TownHunt meet your expectations?

A) Yes, it did. I liked how the whole app was integrated together and easy to use. The leaderboard works as expected and adds that little competitive element that makes playing packs so fun to do. I especially liked the addition of the average score within the leaderboard as it created a target score that people should aim for. This is the same for the account statistics. Through displaying the total number of packs created/played, you have essentially 'gamified' the pack creation/playing system. Thus, more people will be encouraged to play/create more packs to increase their totals which in turn produces more content [packs] for the app. Although no official multiplayer option is in the app, it is easy enough for a group of friends all to have the app open and start playing the same pack at the same time.

Was it easy to use the store?

A) The store was, like the majority of the app, easy to use and navigate. I managed to find the Nevern Square Hotel pack very quickly by defining the search parameters. I also managed to find some packs that you have made and played them as well.

Did you enjoy the game play?

A) Yes, I really enjoyed playing the packs in both modes. The gameplay takes all the entertaining elements of scavenger hunts. I have also discovered new locations around Earls Court, despite

having been to that area many times. One thing I learnt was that a famous British bacteriologist/immunologist, Sir Almroth Wright, lived in Pembroke Square not too far from the hotel. I'm sure that many more new things will be learnt as I play more packs.

Do you have suggestions on how to improve the app?

A) I have a few minor suggestions. Firstly, a preinstalled 'Tutorial Pack' might be useful for new users to understand how to play the game. This could show how the gameplay mechanics work. Secondly, packs with less than 5 pins can still be downloaded from the database. These packs should not be downloadable unless it was made by the logged in user. Lastly, a button in the pack store which automatically updates downloaded packs if there is a new version would be a nice addition to the app.

Analysis of User Feedback + Possible Improvements

The app was very well received by the users. This is encouraging and reaffirming that my initial goals of the project were indeed met. The functionality of the main game, pack creator, pack store and account information all work as planned. It's also positive to hear that the user interface made the different features of the app easy to use. However, with all applications and projects there are always ways to improve if the project was revisited.

Juliana suggested that the pack description's limit should be longer than 30 characters as well as certain text fields should display the full pack name-location key. This is can easily be fixed by changing the single-lined TextFields to multiline TextViews and giving the text more space on screen. However, this does sacrifice screen real estate for other objects such as the map or the leaderboard records. The text displays which sometimes ends up with strings being truncated are when pack names and locations are being displayed. I could change these so that only the pack names are displayed.

Jacqueline suggested preinstalled tutorial packs to teach new users about how to play the game. This could be done by creating a pack which generates several pins around the user's location and then the app guides the user through the gameplay. The user's GPS coordinate location could be obtained then 'tutorial pins' could be placed around the user. This could be done by setting the tutorial pin's coordinates as slight variations of the user's GPS coordinate. The tutorial pins would contain information on how to play e.g. the title would be 'Tap Me', hint would be 'Once you have reached the pin's location, look for the answer to the question. Your first question is "what is the name of this app" Enter your answer below." This will help familiarise the user with the gameplay mechanics.

Jacqueline also suggested that packs with less than 5 packs should not be downloadable unless it was made by the user. My original reasoning behind allowing all packs, regardless of pin number, to be downloaded from the store was so that users would always be able to download their own created pack and continue working on the pack on another device. However, I do understand the criticism and would change this if the project was revisited. Currently the search mechanism for retrieving all of the packs that the logged in user has made and the mechanism for retrieving all of the packs that a certain creator (username) has made are the same. By separating the two processes, the general search form/criteria system could be changed to only display packs that have at least 5 pins. This additional pin count check would be done in the SQL query that searches the database. When retrieving all of the logged in user's packs, the app would connect to another API script that would download all of the packs with a certain creator user ID, regardless of

number of pins the pack has. This system would also rectify the 'Packs You Have Made' button error discussed in test 17 as unique user ID would be used in the query instead of usernames.

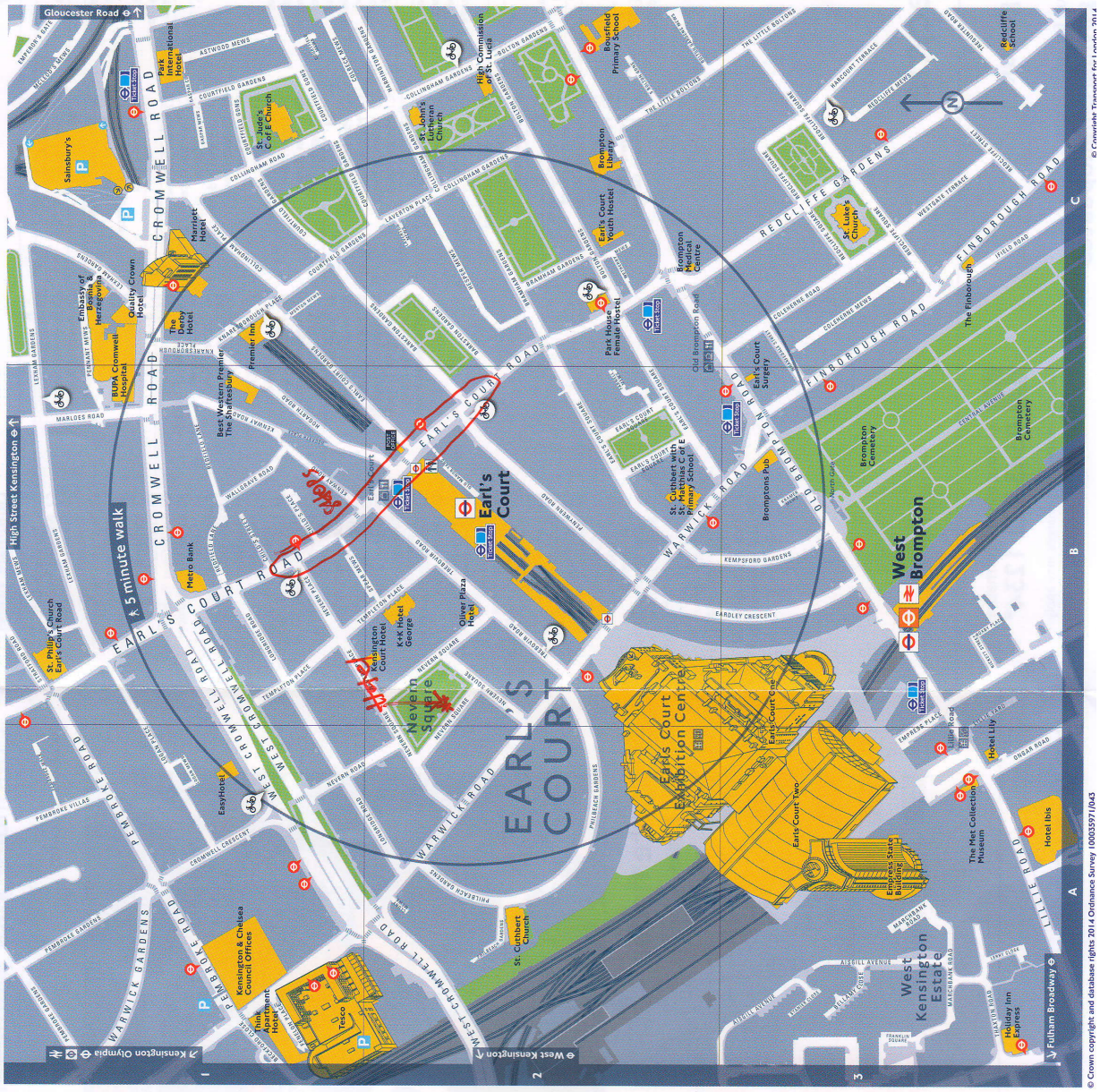
Another suggestion by Jacqueline was an 'Update All Local Packs' button. This is a good idea and is something that I would implement if the problem was to be revisited. Pack version numbers are kept on the phone within the pack's JSON file. The system could iterate through each local pack and compare its version number to the one stored in the PinPacks database table for the same pack. If the version number in the PinPacks table is greater than the one found in the pack's file then the new (updated) pack information would be downloaded and replace the old pack information.

One feature that I wanted to add to the app, if the problem was revisited, is an option to display the packs found in any pack table (e.g. list of local packs and pack store search results) on a map. This would make it obvious where the pack takes place in. This could be implemented by retrieving one pin's coordinates from each pack and displaying a pin (with the pack's name) on the map. Pins would be touchable and an option to download the pin would be shown.

Bibiologaphy/References

1. Scavenger Hunt Used in University of Manchester Orientation day
   http://documents.manchester.ac.uk/display.aspx?DocID=26335

2. TFL Map with Annotations by the Receptionist
   - Used by Mayflower Collection of Hotels



3. SWRevealController By John Lluch
https://github.com/John-Lluch/SWRevealViewController

4. Apple (Swift) Developer Documentation
https://developer.apple.com/reference