mizen

A Location-Aware Guidance App For Events

Colm Patrick Cahalane

Final-Year Project - BSc in Computer Science
Supervised by Prof. C.J Sreenan
April 2017

UNIVERSITY COLLEGE CORK
DEPARTMENT OF COMPUTER SCIENCE

Abstract

The Final Year Project (FYP) open day is an event attended by staff, students and industry professionals every year where around 75 students exhibit projects in different rooms in the Western Gateway Building. A booklet is produced every year listing the projects, but this is inefficient for many staff and visitors.

This project (*Mizen*) aimed to improve this experience by using a cross-platform mobile app that provides guidance for navigating the event with providing location-aware information, provided by Bluetooth beacons in each of the rooms. The information should be easy to search through, and provide the ability to bookmark projects or mark them as seen.

Mizen aimed also to explore the feasibility of creating similar applications for other uses, exploring the idea of beacon-driven applications for similar events such as trade shows and conferences, and the reliability of the beacon technology in applications such as the smart home.

Mizen also aims to explore the feasibility of developing an application for both major mobile platforms under the time constraint of being available before the FYP Open Day, using the React Native framework.

Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print

Signed: Date:

_____ _____

Acknowledgements

"And we'll drink to the gentle, and meek and the kind, and the funny little flaws in this earthly design."

– Conor O'Brien, *'My Lighthouse'* from the album {Awayland}

The design of this document is inspired by the works of Edward Tufte and Donald Knuth. That said, it's made in Microsoft Word, so I don't know if they'd think much of it.

The lighthouse logo provided by Flat Icon is used under the FlatIcon.com Basic Licence.

None of this work would be possible if not for the tireless effort of my parents and the value that they place on education. This work marks the end of *seventeen* consecutive years of keeping at least one of their sons supported through college; so hopefully here's to some rest at last. *Mizen* is named for a place not far from home.

I am also indebted to Prof. Cormac Sreenan, Prof. Ken Brown and Dr. Frank Boehme for supporting this project with their time, efforts and advice; and for their work as educators of the highest degree throughout my time at UCC. It's also important to credit many other lecturers in the department such as Dr. Derek Bridge, Prof. John Morrison, Prof. Michel Schellekens and Dr. John Vaughan for inspiring me to follow my interests in the subject and move towards greater things.

Jonathan O'Mahony, who introduced me to one of the core technologies within this project, most likely saved it.

Table of Contents

Table of Figures

1 Introduction

1.1 The Student

1.1.1 Motivations in accepting the project

The student accepted this project due to an interest in learning functional mobile development techniques with a cross platform focus. The student had no previous experience with developing for mobile, and wanted to gain this valuable knowledge. The time constraint and having a deliverable ready for the Open Day also proved appealing. Finally, the student's interest in building software that interacted with the physical world or networked devices drove him towards this project – the student's second choice of Final Year Project would've involved using smartphones as controllers for local multi-player video games.

1.1.2 The student's background

Colm Cahalane is a final-year UCC student, studying BSc Computer Science. Through the course, the student gained a development skillset focused on the web, as well as software development in Java. Through internships, the student further honed these skills, and expanded upon them, learning the principles of Site Reliability Engineering at Google, and the agile development methodology working at Teamwork.com. He's previously worked with UCC Netsoc both in an organisational role as society finance officer and chairperson, and as a developer and administrator. In other extra-curricular activities, the student worked mainly in media, such as taking on a role as art director for Motley Magazine and a radio host and producer with UCC 98.3FM.

1.2 Project Background

1.2.1 The Open Day

Every year the Final Year Project (FYP) Open Day gives students an opportunity to present their project to the academic staff, research staff, and industry visitors. Each person typically decides in advance which projects are most interesting to them, as it is not feasible to visit all projects. This year, 75 projects were arranged across three rooms – although not all students who were listed appeared to present on the day. A booklet is given to guests on registration listing the project abstracts and the rooms where they are located.

This however is inefficient for most visiting staff and industry visitors. It's not immediately clear where staff must go to see the projects that they are marking, while industry visitors are unable to screen projects ahead of time to find the technologies and skills that they are most interested in. This must be improved.

1.2.2 Managing FYPs

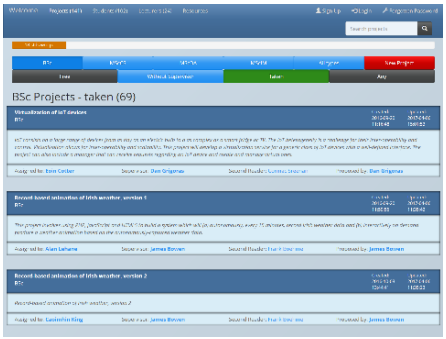


Figure 1: Screenshot of FPM, the FYP project manager developed by Dr. Boehme

Colm Cahalane Supervisor: Prof. Cormac Sreenan	Title: Mizen – A Location-Aware Guidance App For Events Abstract: The FYP open day is an event attended by staff, students and industry professionals every year where around 75 students exhibit projects in different rooms in the Western Gateway Building. A booklet is produced every year listing the projects. Still, it is often difficult to find the projects you need to see based on their technologies, location in the building, or if you're required to grade them. This project (Mizen) aimed to improve this experience by using a cross-platform mobile app that provides guidance for navigating the event with providing location-aware information, provided by Bluetooth beacons in each of the rooms. The information should be easy to search through, and provide the ability to bookmark projects or mark them as seen. Mizen aimed also to explore the feasibility of creating similar applications for other uses, exploring the idea of beacon-driven applications for similar events such as trade shows and conferences, and the reliability of the beacon technology in applications such as the smart home. Mizen also aims to explore the feasibility of developing an application for both major mobile platforms under the time constraint of being available before the FYP Open Day. Keywords: location-aware, mobile, cross-platform, ios, android, Technologies: React Native, Bluetooth Low Energy, NPM, Estimote

Figure 2: An example of a FYP abstract submitted by a student in DOCX format.

The process of allocating students to FYPs has been done using a Laravel-based web application called *Frank's Project Manager* (FPM; project.ucc.ie) developed by Dr. Frank Boehme, a lecturer in the department[1]. FPM is primarily used at the start of the year, to track the initial assignment of projects to students, and at the end of the year, to track the assignment of second readers to projects. Much of the project information never reaches FPM.

For the creation of the booklet and room layout maps, information is gathered manually using Microsoft Word and Microsoft Excel files. In generating the booklets, students provide their project abstracts in Microsoft Word .docx files to a Moodle module, and these are then gathered and stitched together, in a roughly-alphabetical order. The use of proprietary formats meant it would not be feasible to create a parser.

Alphabetical order is often used to assign students to rooms as well, but there are some exceptions: the VR lab is necessary for some 3D and VR projects that require specialist hardware, and students who have gained dedicated development machines in Room 1.09 will be given these machines to use for development. In a previous year, a project had to be hosted outdoors, as it involved a drone.

1.3 Improving the FYP Open Day – This Project

The original project description outlined a plan to improve the Final Year Project Open Day by implementing "a smartphone app (for both Android and iOS)" that "will likely involve the use of Bluetooth beacons" to "offer a tailored guide for each person". [1]

This report discusses the implementation of a cross-platform, location-aware guidance application (*"Mizen"*) that aims to improve the FYP open day by providing location-aware information, a searchable project list, the ability to mark and save projects, and view external sources of additional information.

[1] Notably, the tools used to manage Final Year Projects with a single web application and data source was the aim of another final year project this year: Fatima Zhara, supervised by Dr. Kieran Herley.

2 Analysis & Goals

2.1 Technical Background

2.1.1 Bluetooth Beacons

Figure 3: Three Estimote branded BLE beacons, with iPhone for scale [5]

A "Bluetooth Beacon" is a Bluetooth Low-Energy (BLE) device that emits identifying information depending on its broadcasting standard. BLE allows beacons to emit small data-frames while only using a fraction of the power required by normal Bluetooth, ranging from 20% to as low as 1% of the power of standard Bluetooth within the beacon, and 3% to 1% in the end-user's device. [2]

iBeacon, devised by Apple, is one example of a standard for beacons. It is tightly integrated with iOS' own CoreLocation framework, but is widely implemented in libraries for other operating systems. It specifies that each beacon emits packets containing these fields [3]:

- A UUID (Universally Unique Identifier) that specifies the application in use.
- A "major number", and a "minor number" that can be given app-specific meanings.

Figure 4: Exploded diagram of an Estimote beacon, showing its small interior board and CR 2450 battery

An example is a chain department store that would use a UUID to represent the company, major number to denote individual stores, and a minor number to denote departments. Other beacon standards include Google's Eddystone and AltBeacon, which broadcast more complex data. For example, Eddystone-URL links beacons to URLs, a concept known as "the Physical Web" [4].

The CS Department in UCC owns several Estimote-branded beacons. Estimote manufacture beacons that are compatible with both iBeacon and Eddystone [5], however some of UCC's stock are early developer units which only support iBeacon. As the functionality of iBeacon was sufficient for this small-scale deployment (spanning six rooms on the FYP Open Day) the student decided to use it for this project.

2.1.1.1 Working with iBeacons

Apps detect and interact with iBeacons in two ways: *ranging* and *monitoring*. [5]

- Monitoring: actions are triggered on entering/exiting region's range.
- Ranging: actions are triggered based on proximity to a beacon; that is, a method is given a list of beacons in range, with an estimated proximity to each of them

Notably, monitoring can happen in the background, but ranging requires the app to be open.

2.1.2 Mobile Development & the State of Cross-Platform

There are two major mobile operating systems, Android and iOS. Android recently became the most common operating system among internet users. iOS powers Apple's successful, high-end iPhone range. Developing applications for these platforms traditionally had some major restrictions. Android apps must be written in Java, and iOS applications must be written in either Swift or Objective-C. Furthermore, there are completely different libraries and conventions for UI design and implementation, and dealing with user interaction.

Figure 5: Example of Cordova using HTML for layout. [32]

2.1.2.1 Cordova – Using Web Technologies to Unify App Development

Cross-platform development traditionally relied on styling web applications written using HTML, CSS and JavaScript to look like native applications, and wrapping them as apps for both devices. A tool known as Cordova originally appeared in 2009 allowing for this style of development. The open-source core of Cordova has been succeeded by Adobe PhoneGap and Ionic. [6] Here, the term "Cordova" is used to refer to all implementations of this style of development.

An argument against Cordova-driven development is the interface lag associated with HTML\CSS\JavaScript leading to a poor user experience. The lack of a "native feeling" to these apps makes them undesirable: a native app will be faster and more pleasing to use.

Another argument against the use of Cordova is that it is difficult to bridge to native APIs and features; though it offers a plugin framework to write functions that interface with native APIs and features, it is unwieldy. There is a community associated with PhoneGap that provides many of these plugins, but it is still rather imperfect.

It was decided early in the project development stage to avoid using Cordova.

2.1.2.2 React Native: A Modern Take on Unified App Development

A more recent development in cross-platform application development is React Native, pioneered by Facebook.

```
return (
	<View style={styles.splashScreen}>
		<Spacing />
		<Image source={require('./splashimg.png')}
			style={{width: 400, height:400}} />
		<ActivityIndicator
				animating={true}
				style={{height: 80}}
				size="large"
				color="white"
		/>
	</View>
);
```

Figure 6: Demonstrating JSX: A React Native method returning a layout specified in XML

React Native is built upon Facebook's web MVC framework, React.js, which introduced the concept of JSX – a pre-processor step allowing the developer to write XML within JavaScript files to specify the layout of components. [7]

React Native is like Cordova in its use of JavaScript, and like React.js in its use of JSX. The primary difference is that it doesn't use the HTML/CSS Document Object Model. Instead, XML is used to specify app layouts that are then rendered using native UI Components on both platforms. This allows for the user interface to work at 60fps and achieve a native look and feel. [8]

React Native has a strong, active community as it is tightly linked to NPM, the Node Package Manger. NPM is the package ecosystem for any programming language, well over twice the size of Java's Maven Central [9]; as such it is easy to import libraries developed by the community.

Although React Native is powered by modern JavaScript (ES7) and uses tools like NPM and Babel, it is not an implementation of Node.js for mobile. It uses the OS' own JavaScriptCore.

Originally, the student had aimed to develop two separate native applications. The pivot to React Native happened at a later stage: upon realising that the workload of learning and implementing the two radically different styles of application would be unfeasible within the time requirements, the decision was made to use React Native to unify the development process.

2.2 Existing uses of beacon technology

Beacon manufacturer Kontakt.io has written a few whitepapers on different uses of beacons, exploring the use cases of beacons at events and museums.

- The event whitepaper [10] deals with a deployment of the enterprise event app framework *MOCA* at the Mobile World Congress (MWC) in Barcelona. It concludes that using proximity data and beacons at the MWC led to 235% higher engagement than a standard event app would have given, and using beacons rather

than GPS to trigger notifications allowed them to reach 17x the number of people that would otherwise be possible. Here are some of the features that the beacon-driven event app permitted:
    - It allowed attendees of the MWC to register for the event at the airport: the app would notify a registration desk if an attendee was in the airport and their badge would be pre-printed there.
    - The app would notify users when they were near points of interest at the event.
    - The app would display transport related information as it noticed the user was leaving the event.
- The museum whitepaper [11] deals with a deployment of a beacon-driven audio tour app *Muzze* at 40 museums, primarily in the Netherlands. It notes that deploying a beacon-based audio tour in a museum can cost as low as 3% of the cost of traditional tech for this purpose, and that it has a 240% higher engagement rate compared to other audio tour applications.

For local examples, an Irish start-up known as Pulsate provides beacon solutions to Irish retailers such as Brown Thomas, BoyleSports and Topaz to power loyalty programs and location-driven marketing. [12]

The findings from the whitepapers validated the use case of beacons for the Final Year Project Open Day. The FYP Open Day shares elements in common with both the event and exhibition use cases described, despite existing on a smaller scale than the enterprise deployments discussed in those papers.

2.3 Requirements Analysis

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this analysis are to be interpreted as described in RFC 2119. [13]

2.3.1 Functional Requirements

These requirements that

1. The user MUST be able to see at a glance what projects are in the room they are in.
2. The user MUST be able to search and filter through a list of all projects – all fields of the project SHOULD be searchable.
    a. Title, Description, Student Name, Supervisor Name and Second Reader Name MUST be searchable.
3. The user MUST be able to bookmark projects to view later, and to mark them as done/seen.
4. The user MAY be able to separately "like" a project.
5. The user MAY be provided suggested projects based on the projects they have previously shown an interest in.
6. The user MUST be able to see at a glance what other rooms are nearby.
7. The user MUST be able to see additional information about the project such as its full abstract, student and supervising lecturer.
8. The user SHOULD be able to follow links to additional content such as the student's online presence or related documents to the project.

2.3.2 Non-Functional Requirements

These requirements must be fulfilled for the functional requirements to be fulfilled:

1. There MUST be a web backend and single data source for all information relating to projects.
2. This MUST allow for projects to be assigned to rooms.
    a. This MAY be an extension of the existing department tool FPM.
    b. This MAY be a wholly new application.
3. This backend MUST have an endpoint for revealing project data as JSON.
    a. This backend SHOULD have a caching layer on the JSON endpoint.
4. The app MUST be able to retrieve and interpret this JSON data.
5. The app MUST be able to store some of its state permanently, so that saved/marked projects persist as the app is used over time.

3 Design & Implementation

Within this section, code samples will be used to illustrate concepts and their implementation. These have been simplified and adapted for use in the report and do not necessarily reflect the final state of the project.

3.1 Developing the Backend

3.1.1 Projector

A very simple web backend called *Projector* was developed over two weeks in October to be used in the proof of concept, as Dr. Boehme initially did not respond to a request to make the source code for FPM available. Projector was a simple Laravel application that allowed students to register accounts and upload information about their project, and provided an endpoint for project information as JSON. Using this backend, the first proof of concept Android application was developed.

3.1.2 Extending FPM

After Dr. Boehme provided the student with a copy of the source code for FPM and a copy of the database, Projector was scrapped. In its place, FPM was extended. FPM is a web application written using the Laravel framework for PHP, which the student was already familiar with. Laravel is an MVC framework which allows for easy modification and extension of its models and controllers. The original version of FPM had these classes:

Project

id	int(11)
title	varchar(255)
abstract	text
description	text
type_id	tinyint(4)
proposer	int(11)
supervisor	int(11)
sndreader	int(11)

role()
is_lecturer()
is_student()
is_admin()
fullname()
assigned_project()
proposed_projects()

//must be lecturer
co_supervised_projects()
supervised_projects()
supervised_students()

User

id	int(11)
firstname	varchar(255)
lastname	varchar(255)
email	varchar(255)
password	varchar(60)
interest	text
multirole	int(11) (project ID or other role)
signature	text

type() // degree program
link() // URL
proposer_string()
supervisor_string()
sndreader_string()
view_type()
owner()
is_public()
is_visible()
is_free() // true if no students assigned
takers() // students assigned
url()

3.1.2.1 *Extending Models*

The first step of updating the application was to add another class:

Room

id	int(11)
name	varchar(255)
minor_number	int(11)

projects()

A *room_id* attribute and room() method was added to the *project* class to support the one-to-many relationship of Rooms and Projects. This was easily done with Laravel:

```
class Project extends Model
{
    public function room()
    {
        return $this->belongsTo('App\Room');
    }
}

class Room extends Model
{
    public function projects()
    {
        return $this->hasMany('App\Project');
    }
}
```

Figure 7: Laravel's Eloquent ORM in action to define a One-To-Many relationship.

This makes use of Laravel's "Eloquent ORM", which allows for a database-agnostic approach to querying for individual instances of classes and defining relationships, as well as define accessor methods.

Migrations were also used to extend the Project class, adding the attributes *image* (a URL for a project's featured image in the app), *linkedin* (an external link to the student's linkedin page) and *misc*, a text field for information that's useful for filtering but not intended to be user facing, like a keyword list.

3.1.2.2 *Creating a JSON Endpoint*

```
Route::get('/projects.json', function() {
        $ret = [];

        $assignedProjects = \App\User::where('multirole', '>', '0')
                                                        ->lists('multirole');
        $projects = \App\Project::whereIn('id', $assignedProjects)
                                                ->get();

        $ret["projects"] = $projects;
        $ret["rooms"] = \App\Room::all();
        return $ret;
});
```

Figure 8: A simple route that returns the information required as JSON

A new route was created that returned an array of all projects that are assigned to students and an array of all rooms encapsulated as a JSON object. This was easily done in Laravel: a controller method that doesn't return a View or Response object will instead return its output as JSON. As a result, the JSON endpoint was written quite quickly: the model that was defined did most of the work here.

However, the initial take on this was database intensive and slow. This was linked into Laravel's caching layer to increase speed. Laravel provides a simple caching API, where depending on implementation on the server, cache may be stored on disk or in memory.

3.1.2.3 Back to Modelling: Accessors

```
proposer: 14,
supervisor: 14,
sndreader: 0,
room_id: 4,
```

Figure 9: Missing details

Despite a functioning JSON endpoint, a significant amount of data is absent. For instance, an object containing the IDs of the project's second reader and supervisor is returned, but this does not contain their actual names to display in app. It's clear that the Project class has `proposer_string()`, `sndreader_string()` and other methods; so it must be possible to use those while serialising the projects.

```
class Project extends Model
{
                protected $appends = ['proposer_name', 'supervisor_name',
                                      'sndreader_name', 'student_name', 'room'];

                public function getProposerNameAttribute(){
                        return $this->proposer_string();
                }

                public function getRoomAttribute(){
                        return \App\Room::find($this->room_id);
                }
}
```

Figure 10: Using the Accessor convention and `$appends` array to add simulated attributes

Laravel's Eloquent ORM allows the developer to define accessor methods using the naming convention `getAttrNameAttribute()` to modify the contents of `$object->attr_name` before returning it. If an attribute doesn't exist, it can still be defined as an accessor method. The name of the simulated attribute can be added to the `$appends` array. This way, the information is appended to the serialisation.

```
{
    id: 218,
    created_at: "2016-09-20 09:02:04",
    updated_at: "2017-04-03 17:39:23",
    title: "Mizen - A Location-Aware Guidance App For Events",
    abstract: "Every year the Final Year Project Open Day gives students an opportunity to present their project
    to the academic staff, research staff, and industry visitors. This project is to develop a smartphone app (for
    Android and iOS) to offer a tailored guide for each person.",
    description: "Mizen provides a framework allowing event organisers to mark areas with BLE beacons and allow
    visitors to get location-specific information as they explore the event, as well as information on other
    locations nearby. This app is built using React Native, which allows for creation of a cross-platform mobile
    application with a common codebase (using JavaScript) that nonetheless behaves on par with native apps, and
    still allowing us to use native APIs. It does this by translating React.js view components into each
    platform's own native view components. In doing so, it encourages code reuse with web-platform React code, and
    allows extensive use of NPM packages. This app for both iOS and Android has been deployed at this event, the
    final year project Open Day, and shows the user which projects are active in their current room. To do this,
    the internal tool for managing final year projects was expanded in a way that allowed it to communicate up-to-
    date project details to the app.",
    type_id: 0,
    proposer: 111,
    supervisor: 111,
    sndreader: 18,
    room_id: 4,
    image: "https://colm.cf/temp.png",
    linkedin: "https://colm.cf/in",
    misc: " location-aware, mobile, cross-platform, ios, android React Native, Bluetooth Low Energy, NPM,
    Estimote",
    proposer_name: "Cormac Sreenan",
    supervisor_name: "Cormac Sreenan",
    sndreader_name: "Ken Brown",
    student_name: "Colm Cahalane",
  - room: {
        id: 4,
        created_at: "2017-04-03 16:45:52",
        updated_at: "2017-04-03 16:45:52",
        name: "Room 1.11",
        minor_number: 1000
    }
},
```


Figure 11: An example project JSON object containing the new name and room fields.

3.1.2.4 Creating the rooms admin panel

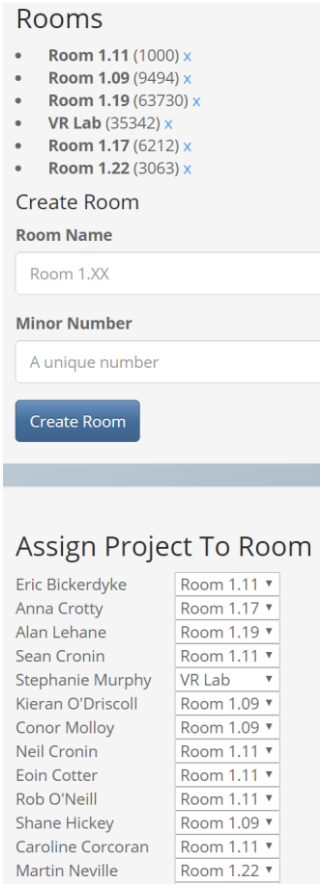


Figure 12: The RoomsIndex page.

It would be far from useful to construct this backend without having a frontend capable of providing the ability to assign projects to rooms.

A new *Rooms Index* view was created that allowed for the creation of new rooms and the assignment of projects to rooms.

A Rooms Controller was built to handle the CRUD (create/read/update/delete) operations on rooms, as well as provide an endpoint for assigning projects to rooms. These were restricted to users with Lecturer or Admin status.

A form allows for the creation of new rooms with their assigned minor numbers.

All projects are listed and have an individual form with a select form.

JavaScript was used to create AJAX calls so that when the room assignment is changed on the page, it updates the backend.

The layout for this page was designed to mirror the rest of the application and was built with Laravel's Blade Template syntax and the Bootstrap CSS framework, mirroring the rest of FPM.

3.2 Developing the App: Native Android

3.2.1 Initial plan: Two separate apps

It was decided early in the project to try and develop two separate native apps. The student had studied Java in modules such as CS2500, so the plan was to develop an Android application first and develop the iOS application in the quieter second semester. The student drew out an original plan for the development time, fortnight by fortnight:

10 Oct	**Start of FYP Plan**	**Start getting up to speed with Android Studio and develop some toy apps and examples working through tutorials. Develop project description.**
24 Oct		Develop Projector web backend (API-first) while working on Android development skills.
7 Nov		Start development of a proof-of-concept Android app that can interact with backend API. This doesn't need to be a final version.
21 Nov		Work more closely on the Android app from a design point of view.
5 Dec	**Study Week & Exams**	Slowed development but try and make the Android app feature-complete.
19 Dec	**Exams & Christmas**	Work on a more polished Android v1 over Christmas.
2 Jan	**Holidays**	Aim to have Android app near-complete by end of holidays - final testing etc.
16 Jan	**Semester 2**	iOS app takes priority from here. Should aim to have access to Mac and iPhone for now. Initial ramp-up time to learn iOS development and start implementing the app.
30 Jan		Make iOS proof-of-concept app that interacts with the API and backend.
13 Feb		Make iOS app feature-complete.
27 Feb		Make iOS app design-ready.
13 Mar		Testing and prep for release on both major platforms and app stores.
27 Mar	**Open Day**	App should be in release preparation stage well before the open day itself - this is because the App Store review process may take some time. Release a V1 and work on potential updates. Start collecting real-life data for the API.
10 Apr		Work on report, and analyse what went well or not during the open day. Survey students, staff and users on effectiveness.
24 Apr		**Report Due at this point, project ends**

However, the student struggled to keep up with the plan. The workload involved in learning the conventions and standards of Android development proved too demanding, and it was unrealistic to balance this with academic and other extracurricular workloads in the first semester. As a result, the student fell significantly behind schedule and had failed to complete the Android V1 by January.

3.2.1.1 Retrofit: Interfacing with the backend API

The Android app used a Java library called Retrofit to connect to the backend API. The JSON, as it would be parsed, would be used to create an instance of a Java class.

This works by creating a class to represent the API service, individual classes to represent object types returned by the API, and a class that represents the form in which these objects are returned. Here is an example of how that would look in Mizen:

```java
public class Project {

    private final long id;
    private final String title;
    private final String student;

    public Project(long id, String title, String student_name) {
        this.id = id;
        this.title = title;
        this.student = student_name;
    }

    public long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public String getStudent(){
        return student;
    }
}
```

Figure 14: A Java class representing a project.

```java
public class ProjectorApiService {

    public static final String DOMAIN = "colmfyp.netsoc.co";

    public static ProjectorApi create() { return create(DOMAIN); }

    public static ProjectorApi create(String domain) {
        OkHttpClient httpClient = new OkHttpClient.Builder()
                .build();

        Retrofit retrofit = new Retrofit.Builder()
                .baseUrl("http://" + domain)
                .addConverterFactory(GsonConverterFactory.create())
                .client(httpClient)
                .build();
        return retrofit.create(ProjectorApi.class);
    }
}
```

Figure 14: A Java class signifying the backend API.

```java
@GET("projects")
Call<ProjectsResponse> getAllProjects();

@GET("projects")
Call<ProjectsResponse> getProjectsByRoomMinor(@Query("minor") String filter);
```

Figure 15: An interface is written declaring methods. Annotations are used to match them to URL routes. Arguments are also specified as URL parameters.

3.2.1.2 Beacons in Android

```java
beaconManager = new BeaconManager(this);
region = new Region("ranged region",
                UUID.fromString("B9407F30-F5F8-466E-AFF9-25556B57FE6D"),null,null);


beaconManager.setRangingListener(new BeaconManager.RangingListener() {
        @Override
        public void onBeaconsDiscovered(Region region, List<Beacon> list) {
                if (!list.isEmpty()) {
                        Beacon nearestBeacon = list.get(0);

                        Toast.makeText( getContext(),
                                        "Nearest Beacon changed " +
                                        nearestBeacon.getMinor(),
                                        Toast.LENGTH_SHORT).show();
                }
        }
});
```

Figure 16: An example of how the EstimoteSDK was employed to range for and react to beacons in the scrapped native Android App.

To interact with beacons in the Native Android app, the Estimote SDK was used. It introduces `BeaconManager`, which can be told to perform ranging on beacons with a specific UUID and given a `RangingListener` that is called when the list of beacons in range changes. The `onBeaconsDiscovered` method of the `RangingListener` receives an ordered list of beacons by their proximity! This allows for an app to be created that reacts properly to changes that occur while the device is in motion throughout a set of beacons.

3.2.2 The Move Away from Native Android

Progress remained very slow on the Android front. Though the toolset to build an application with the required functionality was there, time pressure and a failure to deliver a working interactive proof of concept by Semester Two raised questions on the ability to deliver a working cross platform solution.

On February 1st, a friend of the student mentioned React Native in passing as a cross-platform development toolchain without many of the issues faced by Cordova.

Having taken a few days to consider and research the pivot, the decision was made to scrap the Android take of Mizen and develop a cross-platform app using React Native the following day.

The decision was made as:

- There would simply not be enough time to develop separate polished, reliable apps for both platforms in the remaining two-month span, given that the student was approaching a month behind schedule.
- React Native had active community support for beacons with a simple and effective API.
- React Native's community and support was thorough, with many modules available via NPM and GitHub to extend the application.
- React Native's use of JavaScript and its similarity to web application frameworks meant it was reasonably familiar to the student, who had experience with client-side JavaScript, MVC frameworks like Knockout.js, and server-side JavaScript with Node.js.

3.3 React Native – Background

3.3.1 Components

React is a component-based framework. A component can be imagined as a function that takes in arbitrary inputs and returns a display element on screen. For web-based React, the eventual output of a Component could be imagined as HTML: a component specified as `<Navbar />` would output a navigation bar that might reduce to a standard `<ul>` tag with some specific behaviour and styling. Components can contain other components; they nest and define the layout of the application as a tree.

Each React Native application has points of entry defined as `index.ios.js` and `index.android.js` that run boilerplate code and define the root components of the application. The app layout takes shape from this point.

Components are generally declared in their own JavaScript files: for example, a ProjectList component would generally be declared in a file called `ProjectList.js` (and for this project these are generally a Components folder). If necessary to define components differently on iOS and Android, they can be defined as `ProjectList.ios.js` and `ProjectList.android.js`.

```
class Greeting extends Component {
  render() {
    return (
      <Text>Hello {this.props.name}!</Text>
    );
  }
}

class LotsOfGreetings extends Component {
  render() {
    return (
      <View style={{alignItems: 'center'}}>
        <Greeting name='Rexxar' />
        <Greeting name='Jaina' />
        <Greeting name='Valeera' />
      </View>
    );
  }
}
```

Figure 17: Two examples of React Native components; one having the other as a child component, and passing properties to it. [8]

In RN, instead of generating HTML, these reduce to some fundamental basic UI Components such as `<View>`, `<ScrollView>` and `<Text>` which are provided by React Native itself. Other such fundamental components can be written natively and implemented as React Native components. The components that React Native provides are analogous to standard classes in both Android and iOS - `<View>` maps to `UIView` and `android.view`; in web terms, it'd be equivalent to `<div>`.

RN's fundamental components provide options for styling and layout. These are vaguely analogous to CSS (in that most of the properties have similar names, and Flexbox is primarily used for layout).

Components have *state* and *properties*; where *state* is mutable and changes to the state cause the component to re-render, and *properties* are treated as the fixed inputs to the component, passed by their parent.

The clear majority of the development of the application comes in defining these Components, and in planning the shape of the application they comprise both implementation and design; a notable benefit of React development.

3.3.2 Development Environment and Tools

3.3.2.1 React Native: Environment

React Native projects behave similarly to Node.js packages; they can import other packages that are stored in a `node_modules` folder, these packages are installed with **NPM**, and package information is stored in a `package.json` file. Despite this it's not a Node.js application; the code written runs on JavaScriptCore on a phone's OS, or on V8 on the development server.

React Native applications don't run on a desktop operating system (however, Microsoft has started work on adapting React Native to build Universal Windows Platform apps [14]) so **the Android SDK must be installed** (to build and run Android applications) as well as

XCode (to build and run iOS applications) installed. XCode is currently only available on macOS. [15]

React Native comes with a **command line tool** that allows automation of most of the SDK work: `react-native run-android` and `react-native run-ios` will load debug builds of the application in their respective simulator/emulator, and run a development server (which can also be loaded with `react-native start`).

This command line tool is also used to link additional native libraries and assets downloaded from NPM: `react-native link`.

The **development server** is React Native's greatest strength. It allows for instant **live reloads** of the application as soon as code changes on the development machine, rapidly speeding up the development of the application by effectively eliminating app compile time. (There is a JavaScript compile phase to parse JSX etc. but it's significantly faster than rebuilding the app from scratch). This is possible over Wi-Fi allowing for a completely wireless development setup.

Live reloads are also enhanced by optional **hot reloads** where an individual module of the application is reloaded in place; this allows for rapid layout and design changes on a single module.

There is a secondary advantage to the development server in that it provides a **remote debug mode** where the app's JavaScript is instead executed on the development machine inside the Chrome browser using its V8 JavaScript Engine. This allows Chrome Developer Tools to be used within the development process, alongside a console to debug the application state and monitor timings, resources, etc. This is powerful, especially when combined with the live/hot reloading system, in that it provides a hugely efficient workflow in making changes and getting results immediately.

Regardless of if the remote debugging mode is used, React Native offers **interactive error messages** on the device based on a stack trace. Tapping on a line in the stack trace will navigate directly to that line in the text editor on the development server, so identifying errors is generally quite simple.

3.3.2.2 Version Control – Git

Git is a popular version control system (VCS) developed initially by Linus Torvalds to manage the development of the Linux kernel. It is a distributed VCS designed with a focus on speed, support of collaboration and supporting non-linear, branched workflows [16]. Its distributed fashion proved useful – even though the student developed the project alone, he made use of two separate development machines to develop the project. He used GitHub, a

central web platform for hosting Git projects, as a backup and hub for the distributed development of the app.

3.3.2.3 Sublime Text & Babel

Sublime is the student's preferred text editor, and supports RN's interactive error messages. The *Babel* plugin enables it to understand JSX syntax and comes with code completion and snippets.

3.3.2.4 XCode

XCode is an integrated development environment for macOS, necessary to develop iOS applications. Although much of the work is done by the React Native CLI, XCode's own interface is used for building and signing binaries to run on devices and package for release.

3.3.2.5 Gradle & the Android SDK

Gradle is an open source build automation system like Apache Maven that is used in the Android build pipeline and it is at the core of the Android SDK. Most tasks required in the RN development workflow are automated by the RN CLI, but generating a release must be done manually.

3.3.3 Interfacing with the JSON API

React Native provides the Fetch API [17]. This is a simple API for interacting with REST API endpoints. To get an object containing the contents of remote JSON, there are two steps:

```
async function getProjectsFromAPI() {
  let response = await
fetch('https://colmfyp.netsoc.co/projects.json');
  let data = await response.json();
  return data;
}
```

Figure 18: Getting a Response object and parsing it as JSON

Note the use of `await` – Fetch is asynchronous and by default returns a Promise object, which is a JavaScript convention for handling asynchronous functions. In ES2017, if called from a function denoted as `async`, `await` will hold on for the Promise object to return its final value before continuing; effectively removing the complexity of Promise-oriented programming

The fetch response object has a quick method for parsing the response as JSON and returning its value. This should be wrapped in a `try/catch` block to mitigate any exceptions and network/parser errors that may be thrown. Here in this app, that block triggers an alert prompting the user to try again.

```
async function initProjects(){
        try{
                let data = await getProjectsFromAPI();

                store.dispatch({type:'PROJECTS_UPDATE', projects:data.projects});
                store.dispatch({type:'ROOMS_UPDATE', rooms:data.rooms});

                ready = true;
        } catch(error) {
                promptUser(error);
        }
}
```

Figure 19: Calling remotely and handling failure.

The Fetch API provides some more advanced functionality for making POST requests or custom headers etc.; however, it is only required to download the project list, so these extra features are out of scope.

3.3.4 Implementing Beacons in React Native

The package `react-native-beacons-manager` provides a bridge to native libraries for beacons on both iOS and Android by providing this functionality under a (mostly) common API. There are some differences between the iOS and Android implementations, but for the most part the code returned is the same.

By specifying a "region" to search in (specified by the beacons' UUID; see Section 2.2.3), the developer notifies the program to start ranging beacons that match the UUID. Using a React Native module called `DeviceEventEmitter`, a function can be specified to be called every time the `beaconsDidRange` event fires.

The function attached to this event receives an array of beacons; on Android, each beacon has a numeric distance, on iOS it's assigned a Proximity value from this set of values: "Immediate", "Near", "Far" or "Unknown".

As it can be seen above it is very easy to obtain a selection of nearby beacons. However, to use this information in the user experience, there must be a way of managing the application's state.

```
const region = {
        identifier: 'WGB',
        uuid: 'EBD21AB7-C471-770B-E4DF-70EE82026A17'
};
try {
        await Beacons.startRangingBeaconsInRegion(region.identifier, region.uuid);
} catch (error) {
        console.log(`Beacons ranging not started, error: ${error}`);
}
DeviceEventEmitter.addListener('beaconsDidRange', (data) => {
        if(data.beacons.length > 0){
                data.beacons.sort(compareDistance);
                store.dispatch({type:'BEACONS_UPDATE', beacons:data.beacons});
        }
});
```

Figure 20: `Using react-native-beacons-manager` (imported simply as `Beacons`) to start ranging beacons in a region, and attaching a function to be called when a new beacon ranging event fires.

3.3.5 Handling State

How React's component tree works, where each component has *state* and *properties*, is fine for most cases but makes asynchronous changes to the global application state difficult. Considering the layout of Mizen; the same set of projects is to be called upon by different views; and from each of these views it should be possible to change the state of a project by saving it or marking it as done.

3.3.5.1 Application-Level State with Redux

Redux is a state-container for JavaScript that works with any view framework but comes with an excellent React-focused companion package `react-redux`. It has a minimal API and one that is particularly suited to the React style of development. [18]

The state of the app is stored in an object tree inside a *store*. To change the state tree, an *action* is emitted, an object explaining what change is to occur. To interpret actions, a *reducer* is built: a function that takes in the current state and the action and describes how the action will transform the current state into the next state.

Examining previous sample code snippets given in throughout this section so far, some of them already contain these Redux `store.dispatch(action)` calls after new information is received in asynchronous actions. It appeared where the list of projects received from the API is updated, or as the list of nearby Beacons updates (the `beaconsDidRange` event).

The reducing function itself is very long, so for the purposes of this report, the example will be used of the reducer with only one action in Figure 21.

```
function state(state = defaultState, action) {
  switch (action.type) {
    case 'PROJECTS_UPDATE':
      action.beacons = state.beacons;
      action.rooms = state.rooms;
      action.nearbyRooms = state.nearbyRooms;
      return action;
  }
}

store = createStore(state);
store.dispatch({type:'PROJECTS_UPDATE', projects:data.projects});
```

Figure 21: A look at an example reducer for changing project state

In the first part of the figure, the reducer function is defined. This reducer is made into a store using the createStore method in the second section. When a list of projects is dispatched to the store with the action type `PROJECTS_UPDATE`, as is done in the figure, the reducer takes action. The other fields from the current state, merge them with the action object that contains the new Projects array, and returns this new state. Anything that subscribes to this state is notified.

3.3.5.2 *Connecting Components to Redux*

The remaining challenge is to attach these changes in state to React components. Luckily, Redux provides a way of doing this. It provides a component called `Provider` that takes a store as a property, and subscribes to the store, interpreting its state and managing how that effects its child component.

By writing a function that maps the state to the desired properties of the child component, conventionally referred to as `mapStateToProps`, this function can be connected to a component, and as the state changes, these changes are reflected in the properties of the child component. This concept does take some getting used to: however, as the design and implementation of the project is discussed further, examples of how this was employed in developing Mizen will become apparent.

3.3.5.3 *Persistent Storage – React Native's `AsyncStorage`*

React Native provides an `AsyncStorage` object that implements a key-value storage on both iOS and Android. It is analogous to the web's `LocalStorage`. On Android, it uses either RocksDB or SQLite as its backend depending on what is available; on iOS, it provides its own implementation using file-based storage. It operates globally, but it tends to run slow and it doesn't provide an option for subscription like Redux. In Mizen it is only used to offer persistence in storing what projects are saved or marked done. In this implementation, Redux actions are used to change a project's status, and redundantly store these changes to `AsyncStorage`.

Whether a project is saved or not is represented by storing its ID in a JSON array of Saved Projects:

```
async function updatePersistentStore(id, todo){
  let savedProjects = [];
  try {
    s = await AsyncStorage.getItem('savedProjects');
    if(s != null){
      savedProjects = JSON.parse(s);
    }
  } catch(error) {/* suppress silently and use blank */}
  if( todo === "save" && !savedProjects.includes(id) ){
    savedProjects.push(id);
  } else if(todo === "unsave" && savedProjects.includes(id)){
    savedProjects.splice(savedProjects.indexOf(id), 1);
  }
  try {
    AsyncStorage.setItem('savedProjects', JSON.stringify(savedProjects));
  } catch(error) {/* suppress silently */}
}
```

Figure 22: A function that updates the persistent store when a project is saved or unsaved.

The list of saved projects is retrieved when the app first opens and whether a project is saved is tested using `savedProjects.includes(id)`.

3.3.5.4 *Modifying State on User Interaction*

JavaScript's variable scoping can make it difficult to access the store to perform a `store.dispatch(action)` call. It should be possible to add methods to projects that allow them to modify themselves and update the state of the application so that a project saved once is saved everywhere it appears.

In Mizen, this is implemented by attaching a method to the project within the reducer, as the store is within the scope at that point. Deeper into the application, it should still be possible to call this method on the project and receive the desired effect.

```
case 'PROJECTS_UPDATE':
  for(i in action.projects){
    action.projects[i].changeStatus = function(todo){
      store.dispatch({type:'PROJECT_CHANGE_STATUS',
                      projectID: this.id, todo:todo});
    }
  }
  action.beacons = state.beacons;
  action.rooms = state.rooms;
  action.nearbyRooms = state.nearbyRooms;
  return action;
```

Figure 23: An updated PROJECTS_UPDATE attaches a method that modifies the project

`react-redux` does offer a parallel to `mapStateToProps` called `mapDispatchToProps` for updating the state at a component level, but in practice it felt more natural to just map these changes in state to methods like this: at lower levels of the application, it will be clearer how this works.

3.3.6 User Interaction

3.3.6.1 *React Native's Bundled Components*

Concepts discussed thus far in the report form most of the pieces required to frame the application. However, how React Native handles user interaction has not yet been covered.

Certain elements in React Native are considered "Touchable" such as `Button` and `TouchableOpacity` and therefore have an `onPress` property that is set to a function that should be evaluated on that event. `onLongPress` is also supported. `TouchableOpacity` is one of a class of components: it changes the opacity of the view on press; but there are several others such as `TouchableFeedback` that react differently.

RN also provides a `TextInput` component and a class of other such components like `Picker` and `DatePicker` for input fields that mirror the expected native behaviour.

3.3.6.2 *react-native-searchbar*

The `react-native-searchbar` package generates a search bar that has a native look and feel, but also implements a filter method on a provided collection across all fields, and

generates results that are presented to a `handleResults` method. This is significant in that it essentially provides a complete search functionality out of the box that that can be plugged in without much effort to the Project List view and add much to the user experience.

3.3.7 Navigation

In many ways, this is the final barrier to fulfilling all the requirements. It has been demonstrated above how to build interactive views that have the required traits, all that remains is to provide a way for the user to navigate between different views:

- A list of all projects.
- A nearby-focused view.
- A list of projects grouped by rooms.

Under each of the views, there is also a need for the ability to navigate between individual projects pages and the lists that they're generated under.

In React Native there are a few ways to go about this; it's a difficult topic to come to grips with and there are inherent flaws with many of the approaches.

RN provides stock navigation elements: the stack based `Navigator`, a reworked iOS only version with a more native feel called `NavigatorIOS`, and the deprecated `NavigatorExperimental`. There's also some platform-specific components in the form of `TabBarIOS` and `ViewPagerAndroid`. However, navigation in React Native has never been *easy*; the APIs for many of these components are difficult to understand for newcomers. This, combined with poor performance on the stock `Navigator` caused Facebook to officially start endorsing community solutions in favour of their stock components in March 2017. [8]

React Navigation (reactnavigation.org [19]) is a developer collective that has produced the newly-preferred navigation libraries. Their solutions are superior in that they:

- Built on native components where available.
- Have a simpler API: defining a navigator, its child elements and naming routes can be done in a line of code for each route.
- Nest nicely within other navigators.
- Use the `Animated` class for native 60fps animations.
- Are reasonably and reliably customisable and open for styling.
- Obey platform-specific conventions to a very fine level of detail. The Android navigators will cast heavier shadows and follow Material Design conventions on touch and interaction. The iOS ones are suitably more minimal and run on speedier animations.

- On Android, it recognises the platform's hardware/software Back button and responds as expected to change.

The student had attempted to implement another stack navigator in the project to no avail prior to switching to React Navigation's `StackNavigator`. However, the student had difficulty implementing React Navigation's `TabNavigator` in this project – it appeared fine on Android but not on iOS. He had earlier succeeded in implementing the package `react-native-tab-view`, so he continued to use it.

`react-native-tab-view`'s `TabView` took significant extra effort for initial setup over the React Navigation alternative, in that there was more manual state management involved. However, this paid off dividends in providing a reliable, native-feeling tab navigator that ran cross-platform with ease.

3.3.8 Miscellaneous Layout Challenges

- On iOS, the area used by the status bar is not reserved by the system, so the status bar clashes with the app's layout by default. Also, the status bar may change size depending on the phone's context. Using a package called `react-native-status-bar-size` the application monitors for the size of the status bar and add padding as necessary.
- On both platforms, if the keyboard displays on screen, a RN app doesn't by default give space to the keyboard, meaning that in search results, some results will be hidden behind the keyboard. `react-native-keyboard-spacer` provides a `KeyboardSpacer` component that occupies this space where necessary, which is placed near the root component.

3.4 The React Native Application

Having described a significant amount of the conventions, development strategies and challenges that went into creating the React Native application, this next stage of the report will focus on the exact process of the implementation, and what design choices were made throughout this process.

3.4.1 Initialising the application

Figure 25: The Mizen splash screen (Android)

When Mizen is opened, it needs to perform some basic tasks:

1. Set up the Redux store.
2. Initialise the Bluetooth Beacon library and start ranging for beacons.
3. Download the information from the backend.
4. If there are saved projects or projects marked done in `AsyncStorage` carry them over to the newly fetched projects.

Figure 24: Android and iOS ActivityIndicator equivalents

Steps 2-4 should take place asynchronously. In the main thread, a splash screen is initially displayed that gives some usage details and an animated loading activity icon (`ActivityIndicator`, which shows a platform appropriate icon). If an error occurs in the network setup, an error message is displayed with a button to re-trigger the setup – see Section 3.3.3.

The component then checks for readiness until the setup tasks are completed.

The Splash Screen uses the colour `#2196f3`. This is Material Blue 500, and serves as the root colour for the app. The text displayed in this screen aims to provide some guidance on the usage of the app.

3.4.2 The Tabs layout

It was decided to divide the app into three main sections:

- Projects – List of all projects for searching and filtering.
- Nearby – A view of the projects in the current room and details on what other rooms are in range.
- Rooms – Allowing a user to manually look through projects room-by-room.

This division was necessary to provide all the functional requirements of the application in a way that allowed for a simple, minimal, clean design. Each of the three sections of the app will need to be represented as individual `StackNavigator` components; `react-native-tab-view` was used as the main navigator for the application.

```
const NearbyNav = StackNavigator( {
  Nearby: { screen: NearbyScreen },
  IndividualProject: { screen: IndividualProjectScreen }
} );

const AllProjectsNav = StackNavigator( {
  AllProjects: { screen: AllProjectsScreen },
  IndividualProject: { screen: IndividualProjectScreen }
} );

const AllRoomsNav = StackNavigator( {
  AllRooms: { screen: RoomListScreen },
  IndividualRoom: { screen: IndividualRoomScreen },
  IndividualProject: { screen: IndividualProjectScreen }
} );
```

Figure 26: The three sections of the app defined as StackNavigator components

In each of these components, there are a few possible pages that can be viewed: Either the section's main screen or an individual project. Rooms has an intermediary screen, where each of the rooms has a list of projects as well. In StackNavigator, each of these "screens" are represented by a component and are given a name used for navigation.

There must be a Tab View that encapsulates these three panes.

`react-native-tab-view`'s `TabView` doesn't take in the same style of inputs as the `StackNavigator` components. Instead, it has a `renderScene` method that returns JSX representing the required view for the currently selected route.

```
_renderScene = ({ route }) => {
  switch (route.key) {
    case '1':
      return (
        <AllProjectsNav style={styles.stackNav}
                screenProps={this.props} />
      );
    case '2':
      return (
        <NearbyNav style={styles.stackNav}
               screenProps={this.props} />
      );
    case '3':
      return (
        <AllRoomsNav style={styles.stackNav}
               screenProps={this.props} />
      );
    default:
      return null;
  }
};
```

Figure 27: The StackNavigators are assigned to different panes of the TabView.

Whichever route is currently selected is managed in the State of the component that contains the `TabView`, which must also contain a method for setting that state and handling changes in the currently selected tab. This makes it possible to use Redux to manage state if desired, but local state works in this case.

Navigating between different tabs of the app are performed by interacting with a `TabBar` that the tab view provides. Within the tabs though, the StackNavigator will need to be provided instructions by a function.

Each of the StackNavigators receives a `screenProps` of `{this.props}`; meaning that the properties of the Tabs layout are passed as properties to each of the individual screens. This is how the Redux store is passed down to the individual screens of the app.

The Tabs layout also features the spacers discussed in Section 3.3.8.

The TabBar provided in the tab view package ended up being the core reference point for the application's design. Although it is customisable, the default look it uses is based around Material Blue 500 (`#2196f3`) and Material Yellow 500 `#ffeb3b` on Android; UIColor Blue `#007aff` and the same yellow on iOS. The Android blue was used for the app's branding elements on both platforms.

On iOS and Android, it also sets the tone by having platform-specific shading, colours and fonts:

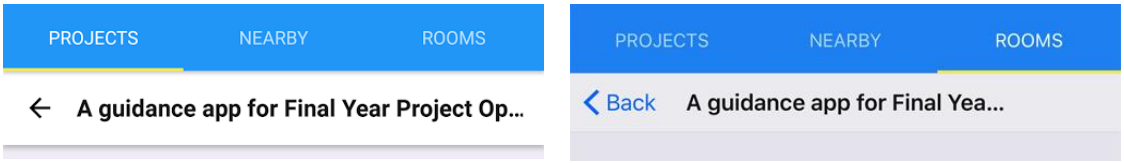


Figure 28: Android (left) and iOS (right) forms of the nested navigation layout established for the app

3.4.3 React Native `ListView` in use, and connection to state

Each of these navigators features a first screen that contains some form of a list element, and the elements in this list are used to navigate. The Project List is the first one that the user sees in the app, so is a good starting point for analysing this setup.

3.4.3.1 Connecting to state, and creating the lists's datasource

To recap on the structure of the app: within the main `Tabs` navigator, and within the `AllProjectsNav` is the `AllProjectsScreen`, which contains the `AllProjectsList` component, wrapped in a `Provider`, which receives the app's Redux *store*.

```
<Mizen>
  <Tabs>
    <AllProjectsNav>
       <AllProjectsScreen>
          <Provider store={store}>
             <AllProjectsList navigation={ap_nav} />
          </Provider>
       </AllProjectsScreen>
       …
    </AllProjectsNav>
    …
  </Tabs>
</Mizen>
```

Figure 29: An approximation of the app layout so far. Note that this isn't valid JSX.

An extended version of this diagram is included in Appendix D: The Application Layout

`AllProjectsList` *connects* the Redux Store to the generic, reusable `ProjectsList` component. It takes the current state and maps the current list of projects in the state to properties of the `ProjectList`. A List Data Source that contains the projects is also created: this is a quirk of React Native List Views, where rows can be grouped by section headers or data from different elements in an array can be grouped into a single row. In this case, each element of the array maps to an element of the data source one-for-one.

```
const mapStateToProps = (state, ownProps) => {
  let datasource = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
  props = {
    ds : datasource.cloneWithRows(state.projects),
    projects: state.projects
  }

  if(typeof ownProps.navigation !== "undefined"){
    props.navigation = ownProps.navigation;
  }

  return props;
}

export const AllProjectsList = connect(mapStateToProps)(ProjectList);
```

Figure 30: Example of a React-Redux connection and mapStateToProps

When the state changes, the mapStateToProps function is re-called, and the component reloads.

3.4.3.2 *React Native's `ListView`*

```
<ListView
  style={this.ListViewStyle}
  dataSource={this._getDS()}
  renderRow={(rowData) =>
    <ProjectRowItem
      project={rowData}
      callOnClick={
        (project) =>
        navigate('IndividualProject', { project: project })
      }
    />
} />
```

Figure 31: The `ListView` in the `ProjectsList` component

This is the `ListView` made using the list of projects. The `ListView` takes a `renderRow` property that takes a function that shows how the data of each row of the list is rendered in the layout. For this, the student created a new component to avoid cluttering this section of the code.

An arrow function describing how each project can navigate to their own page within the `StackNavigator` is passed down to the individual row items. Some quick observations and explanations:

- The `DataSource` provided to the `ListView` is fetched by a function, not specifically the one created in `mapStateToProps`. This is done so that as the user filters and search through the list of projects, a data source containing search results can be provided instead.
- The `callOnClick` property is just like any other property – it must be tied to the `onPress` property of a `Touchable` component for it to have any effect.
- Navigating within a `StackNavigator` is simple; provide the name of the screen that is being navigated to, and providing *parameters* that the screen uses while rendering.

The row item components are relatively simple components; taking in the project, they layout some of the project's details in a `TouchableOpacity` component mapped to navigate to the project's own screen, and Save and Done buttons to the project's `changeStatus` function.

```
<TouchableOpacity style={styles.main} onPress={() => this.pressHandler()}>
  <View style={styles.main}>
    <Text style={[styles.title]}>
      {this.props.project.title}
    </Text>
    <Text style={[styles.owner]}>
      {this.props.project.student_name} (with {this.props.project.supervisor_name})
    </Text>
  </View>
</TouchableOpacity>
```

Figure 33: An excerpt from `ProjectRowItem` showing the layout of the touchable project info.

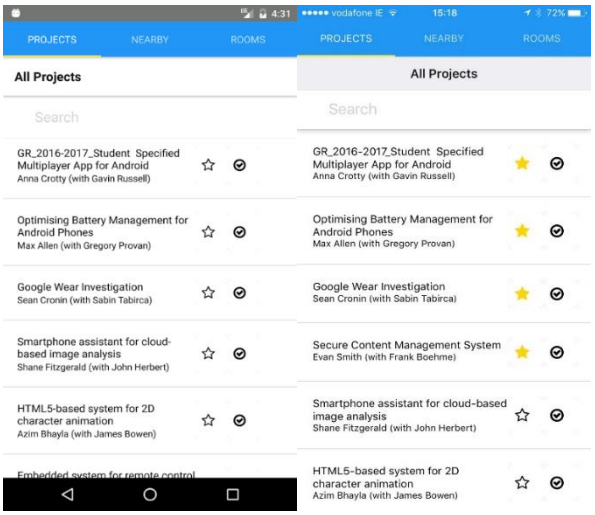


Figure 32: The final All Projects view on both iOS and Android

The `IndividualProjectScreen` component is simple and mostly layout based, so it won't be covered in this section of the report.

3.4.3.3 *Beacons in this workflow*

```
        return {
                room: state.nearbyRooms.shift(),
                nearbyRooms: state.nearbyRooms,
                projrm: projrm, // Projects In Room
                nearbyDS: ds.cloneWithRows(projrm),
                projects: state.projects, // All Projects
        }
```

Figure 34: `NearbyRoomInfo` **contains a much more complex** `mapStateToProps`.

The relatively simple example of the Projects list outlines how app state, layout and navigation can be intertwined in React Native (and the dual nature of *implementation and design* in this style of application development). The Nearby screen shows a more complex case.

`ProjectList` is reused – rather than use `AllProjectsList` and wrap it in a `Provider`, the projects specific to a room are provided as properties – these are determined in the `mapStateToProps` function. A ListView Data Source is also generated and used.

A list of other nearby rooms is also determined from the state.

These details change frequently; a change in the order of nearby rooms, or change in room altogether, can occur every few seconds. Still, this view must be completely up to date. Any detected change must be instantly visible.

The `mapStateToProps` function handles this flawlessly, reloading the page and yet still maintaining details like a user's position in any ListViews etc. The changes propagate to lower levels of the layout tree with ease.

3.4.4 Defining a Look and Feel for the Application

React Native provides a CSS-like styling system for the View components it provides. Given that many of the interactive components come with their own styling by default, this was made somewhat easier; the only items that required much in-depth styling were the individual rows in the List views and the projects page itself.

3.4.4.1 *Flexbox*

Flexbox styling was introduced in CSS3, allowing responsive elements in a container to align themselves in each direction and share space by given ratios. [20]

In React Native these rules are reimplemented. As the sizes of mobile device screens are unknown and React Native doesn't allow for styling by percentages, flexbox is the preferred way to allow items to fill space on the screen.

3.4.4.2 *Row Items*

Row items in the Project List follow the typical iOS menu item style, which looks well also on Android. Items are given a thin border, ample padding, and a white background.

Placed across from the item are two buttons. They were sourced from the `react-native-vector-icons` package, which implements the FontAwesome icon set. A star was used as the Favourite icon, and the checkmark implemented as the done icon; they change shape and colour when activated to reflect the project's status in a way that should be semantically obvious to users.

3.4.4.3 *Project pages*

Project pages are contained in a ScrollView, contain header text, subheadings, action buttons (from the same `react-native-vector-items` set) and an optional project image, roughly modelled on the appearance of apps in App Stores. App Store was an original source of inspiration for many of the design focuses throughout Mizen.

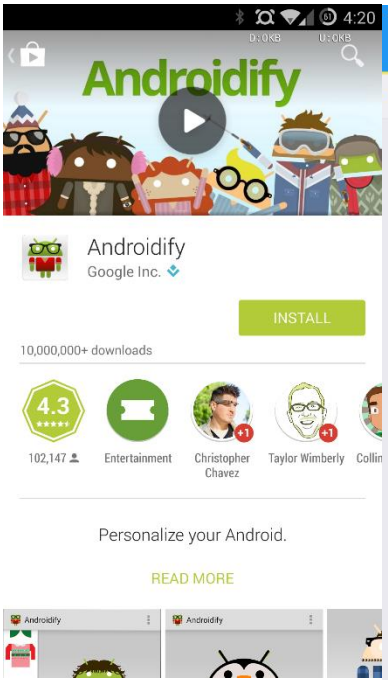


Figure 37: An App on the Google Play Store

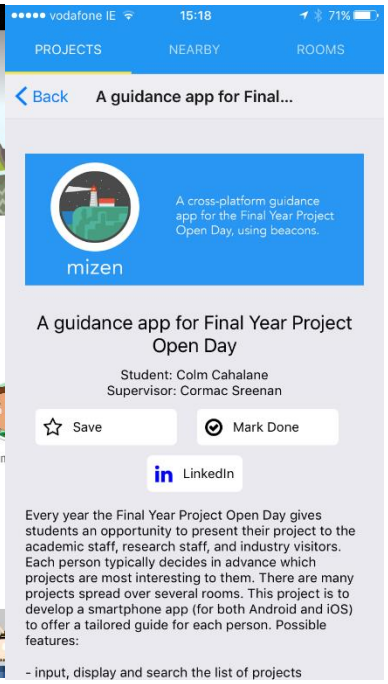


Figure 37: A Mizen listing

3.5 Design & Branding

3.5.1 Mizen

The concept of the *lighthouse* as a metaphor for beacon technology is a common one; it's found in the Kontakt.io whitepapers [10] and in the naming of Google's *Eddystone,* after the world's first open-ocean lighthouse (built in 1699) [21].

Paying homage to this theme with a local twist, the student selected an Irish lighthouse, *Mizen*, found on Ireland's southernmost point. References to Mizen Head are common in weather forecasting and Irish geography and as such would be familiar to guests at the event.

Mizen is not the name of any existing app and is a short five-letter, two-syllable word. It is perfect for an application and would perform well in search engines.

3.5.2 The Mizen wordmark

mizen

The Mizen wordmark uses Avenir Light. It was picked as it paired well with both Android's Roboto and iOS' San Francisco, and embodied the clean look of the app and logo.

3.5.3 The Lighthouse Logo

Figure 38: The lighthouse logo

This icon was made by *Flat Icon* for Flaticon.com and is licenced under the Flaticon Basic Licence. [22]

This icon was selected for its aesthetic qualities. It also exhibits traits of both the iOS flat design and Google Material Design styles for its use of lighting and colour. It combines well with the wordmark. It was available for free download in vector formats on FlatIcon.com under licence.

3.5.4 Mizen in blue

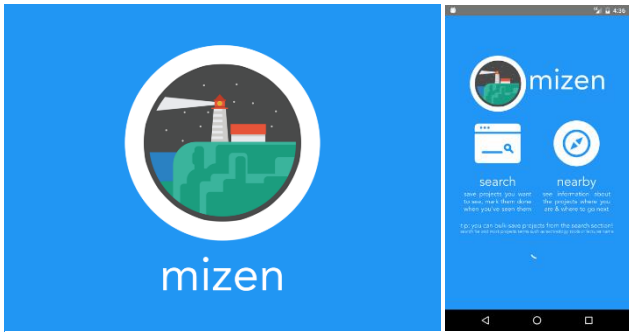


Figure 39: Examples of Mizen's use of blue in its identity

To introduce the application in the initial splash screen and usage instructions Mizen uses a colour palette derived from the application's colours. The logo is modified with a white ring replacing the darker outer ring from the original mark, and the wordmark is written in white.

Figure 40: Mizen "daylight" concept

An alternate all-blue "day" look was tried out, but was rejected after poor results on a 24-hour Twitter poll.

Figure 41: Results of a Twitter poll rejecting this concept

Figure 42: Mizen's icon on iOS

3.5.5 iOS identity

Apple enforces square icons with rounded corners on iOS. The icon is redesigned and tweaked for display on the App Store and on the iOS home screen ("springboard").

Various sizes had to be produced; vector icons were not permitted.

3.5.6 Mizen in Print

Adobe InDesign was used to design an A4 advertisement that was distributed to visitors on the Final Year Project Open Day. It uses the blue brand identity defined by the app's own colours, though slightly washed out; the bolder colours are kept for the app as it appears on screen.

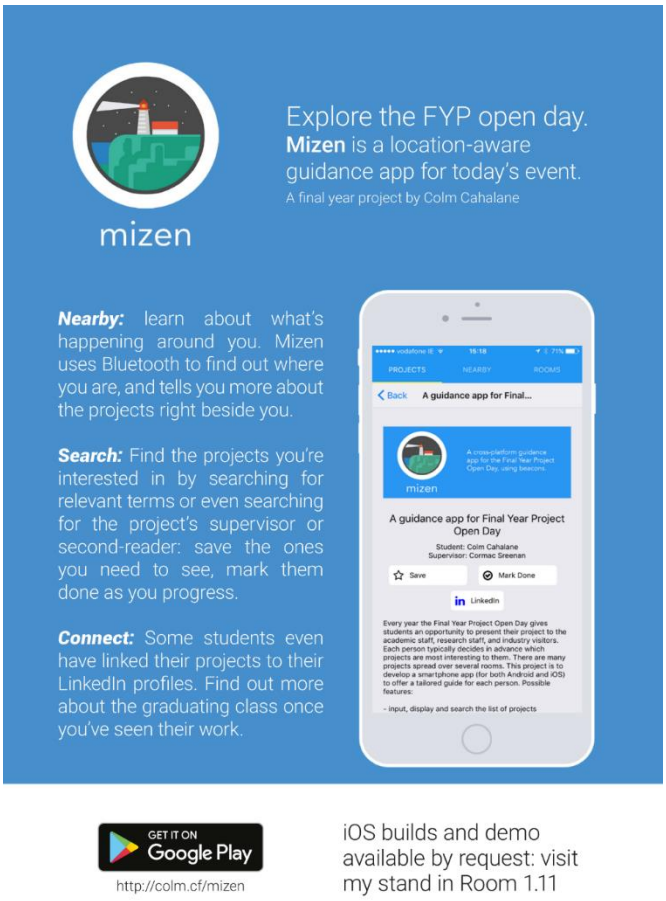


Figure 43: The Mizen poster from the FYP Open Day

4 Release

4.1 Android & The Google Play Store

4.1.1 Building the application for release

Android applications are bundled as application package (APK) files that contain the application and some metadata. An APK can be installed manually on any device using the package installer, or over USB by Android Debug Bridge, but the most viable option for mass distribution is the *Google Play Store*, loaded on the clear majority of Android phones.

While React Native's CLI allows for debug builds, for release, the code must be signed. A signing key is generated with the JDK `keytool` and the details of this should be added to the Gradle config and app build config.

React Native provides a gradlew script, and by running `gradlew assembleRelease` the project is built for release. Scripts and assets are bundled, and the code is signed. An `app-release.apk` file is generated for upload to the Play Store.

Different APKs can be produced for different devices.

4.1.2 Uploading to Google Play

- The APK must be uploaded to Google Play, with a version number and version code.
- A title, description, application type and category must be provided.
- Screenshots, a high-resolution icon and feature graphic must be included.
- A rating questionnaire must be completed to determine the audience for the app and its safety.
- Depending on the app's usage of permissions, a privacy policy may be necessary.

The APKs may be released in production, beta and alpha channels for testing.

Three Android releases were rolled out:

1. 1.0 (initial).
2. 1.1: Fix for a bug in the Search section.
3. 1.2: Fix for a bug in the Rooms section.

34 people installed the Android app, mostly on the day before the Open Day, suggesting the primary audience was other students (staff and visitors weren't informed about the app until the Open Day itself).

An Android App release has very little turnaround time between releases being initiated and going live. Notably, the initial release had only a three-hour turnaround. All students that had downloaded version 1.1 prior to release were updated to version 1.2 prior to the event.

4.2 The iOS Release Process & The App Store

The relative ease of the Android deployment contrasts harshly with the myriad of difficulties faced in the iOS deployment process.

4.2.1 Developer Accounts & Code Signing

Apple Developer Accounts permit users to sign code. Code signing is particularly important for iOS development, as Apple has always placed strict rules on what can and can't run on the iOS platform. The UDID (Unique Device Identifier) numbers of each of the devices the app is built for must be included in the Apple Developer Account, and in the Provisioning Profile specific to the app and the signing certificate used to sign it.

XCode provisions to automatically manage signing for debug executables, the ("iOS Developer") signature. The phone must also explicitly trust the developer's signature.

There do exist other forms of code signing for iOS, however:

- **Ad-Hoc Distribution**
  The signed code is archived as an IPA file that the user can distribute to devices. However, the UDIDs of the devices must be included in the provisioning profile. This differs from the Debug\iOS Developer build in that it can be installed "over the air" in browser and does not require trust management.
- **In-House Distribution**
  Subscribers to the Apple Developer Enterprise Program can develop self-signed applications that they can distribute Ad-Hoc but without the requirement to know UUIDs of target devices.
- **App Store Distribution**
  The app is signed for release on the App Store.

For these, the developer must generate their own signing certificate and create a corresponding provisioning profile in their Apple Developer Account.

This complicated the process: the student developed the app initially under their own iCloud account using the build identifier `cf.colm.mizen`, but had to change the build identifier to a unique one (in this case `net.cahalane.mizen`) when building the app for distribution under UCC's Apple Developer Account. The automatic code signing for debug executables refused to work under the new build identifier, so it had to be switched back for debug builds.

XCode provides an "archive" option for projects, and upon exporting an archived app or uploading it to iTunes Connect, the developer is provided these options to sign it.

4.2.2 iTunes Connect

Further to the barriers described above, iTunes Connect accounts used for App Store deployment are completely separate entities to Apple Developer Accounts, despite the fact that they both rely on iCloud login IDs. Another barrier exists in the form of the $99-per-year iTunes Connect account for iOS Developers, in stark contrast to Android's one-time $25 membership fee.

For the most part, the information required to release an app on the App Store is like the Google Play process in the provision of graphics and text content. Some sections are generally more complex and difficult to fill out. There are also far more requirements for screenshots with different device types.

Upon uploading an application from XCode, giving the application some time to run through Apple's processing, the app enters a review phase.

4.2.3 Application Review on iOS

The App Store review process has collected a degree of infamy for its unfriendliness, and this has been recognised throughout the media and the developer community; efforts to improve upon this process have been the subject of much report and anticipation [23].

While unofficial averages placed the time of an App Store review at roughly two days at the time of submission [24], as a submission of a first version, a slightly longer review time was allowed for.

The student issued a request for expedited review, as Apple provides this service in the case of applications relating to time-sensitive events. This was denied with no reason provided. (Refer to Appendix A: Denial of Expedite Request)

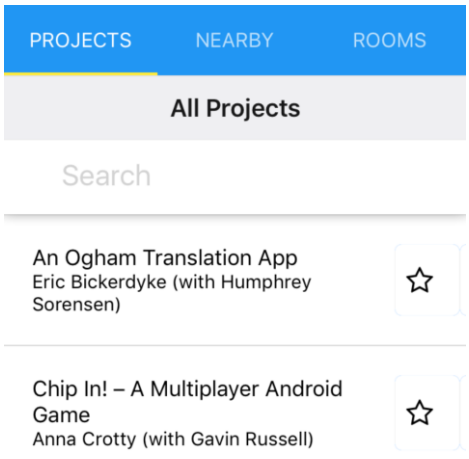


Figure 44: Excerpt from screenshot provided by Apple

What was not expected was that beacon-specific issues would hold up the review process. A call for further information was sent to the student by Apple two days after release. (Refer to Appendix B: Request for Information – Beacons)

Finally, the app was rejected from the App Store (refer to Appendix C: Rejection Notice). In a message, Apple declared "that [the] app or its metadata includes irrelevant third-party platform information" and "Specifically, we have found Android referenced upon launch of the application.", providing a screenshot.

The screenshot showed the Mizen All Projects screen. Android was mentioned in the name of one project: "Chip In – A Multiplayer Android Game". This *mention* of Android was deemed enough to block the app from release.

As it should be obvious that this should be permitted in context, it could be assumed the review process was carried out somewhat automatically, and not by a human. However, it's not clear why Apple did not request the information required at the first delay earlier, or why Android mention didn't flag sooner.

The app was distributed Ad-Hoc to interested students, staff and industry professionals using a tool called Diawi *("Development & In-house Apps Wireless Installation")* [25] that would allow the user to install the app within the browser if their UDID was included in the provisioning profile for the app.

5 Evaluation

5.1 Automated Testing in React Native

Due to working with a new framework and the time pressure involved in implementing and releasing Mizen after the pivot to start from scratch in February, the student didn't have enough time to implement a proper unit testing workflow.

That said, test-driven development is possible in React Native. Facebook has created the Jest testing framework for JavaScript that was designed specifically with React and React Native in mind. [26]

However, it is not possible *at this point of the project* to expect meaningful results from Jest. Jest is based on snapshot testing; creating a snapshot of expected results and periodically running tests against these snapshots. If a test fails, the developer has the option to confirm/deny that the changes are as expected, thus updating the snapshot.

However, given that development has ceased, running Jest will not give any further information.

5.2 Peer Testing

The app was distributed to an audience of peers before general release to find and identify any possible bugs. Two were resolved:

- Under certain conditions, the search bar was not functional at all on some devices. This was resolved in the 1.1 Android release, and an update to the iOS ad-hoc distribution.
- The initial fix for the search bar bug caused the individual rooms page to break on Android; the changes were initially only tested on iOS. The incorrectly-referenced variable was corrected and this was resolved in the 1.2 Android release.

5.3 Additional Bugs

- After the beacons were deployed in UCC, a field test was conducted. It was noted that although the Android version reliably detected the current/nearest room without fail, the iOS version was incredibly inaccurate. After ruling out hardware failure, it was found that the sorting function used to sort the beacons on Android (by a numeric *distance*) was incompatible with the iOS implementation of the beacons library, which implemented a *proximity* field with an enum value. An update to the sorting algorithm to account for the implementation on iOS proved

successful. It is still not known why no exception was thrown when the array of beacons was being sorted by a non-existent attribute.

- On devices with very large screens such as the iPhone 7 Plus, Project Row Items don't reach the edge of the screen. Conversely, on devices with far below standard display size (such as an iPad emulating an older iPhone) the "mark done" button may exist over the edge. This was a result of failing to reach a proper balance when mixing flexbox and absolute layouts, and not being able to emulate more demanding devices like the iPhone 7 Plus on the slower development machine.
- On Android, some testers remarked their saved projects had become missing. A logic error caused saved projects to be unrecovered when the app is reloaded; the relevant code was mistakenly put in an iOS section. This was fixed in the code repository on GitHub but as this issue was discovered after the day there was no corresponding Play Store release.

5.4 Issue on open day

During some early demos on the deployment day for staff members and students, the Android version of the application began crashing. On a debug unit, the error message was unclear, referring to malformed JSON. One student present remarked that the app successfully loaded on Android if started while Bluetooth was disabled.

The room where the student was located (Room 1.11) did not have an Estimote Beacon, as five beacons were provided to the student to handle six rooms. The student had downloaded an application called *mactsAsBeacon* to simulate the final beacon with his MacBook Pro. For unknown reasons, this was outputting malformed data that caused the Android app to crash if detected in the initial setup phase.

The 6th beacon was instead implemented on an Android phone running *Beacon Toy*.

No further issues were reported on the day itself.

5.5 Overall accuracy of beacons

It was found that the application performed very well at identifying the current room, although there was a small chance of flicker between two nearby beacon-equipped rooms (such as 1.11 and 1.19) depending on the beacon's position in the room.

At greater distances, the accuracy of the detection of nearby rooms proved unreliable. Some rooms such as the VR Lab appeared to have a much greater chance of appearing regardless of actual proximity.

Having spoken to Prof. Ken Brown about this at the open day, the student learned that this fall-off is common in beacon projects. True accuracy for beacons at greater distances would

involve a certain level of fingerprinting on the environment to determine and account for these anomalies.

5.6 App Performance and Usability

Though the student did not collect metrics on this matter, review from peers such as other students and staff was generally positive; the app felt like and behaved like a native application; no complaints relating to speed were heard.

However, omitting the room number from individual project pages was identified as a serious error where usability is concerned; it should've been possible for a user to identify what room a project is in from a search. This was noticeably absent from the application and should've been identified at the requirements analysis phase.

5.7 Revisiting the requirements analysis

5.7.1 Functional Requirements

1. The user MUST be able to see at a glance what projects are in the room they are in.

Success. The Nearby view works, and the beacons appear to be accurate and reliable on both platforms as discussed above. It displayed this information at a glance and further information was available on each project.

2. The user MUST be able to search and filter through a list of all projects – all fields of the project SHOULD be searchable.
    a. Title, Description, Student Name, Supervisor Name and Second Reader Name MUST be searchable.

Success. `react-native-searchbar` was used to filter through the list of projects on all fields. This meant that even elements of the project that weren't displayed in the final version of the app (such as a *misc* field containing keywords) were searchable.

3. The user MUST be able to bookmark projects to view later, and to mark them as done/seen.

Success. "Saved" and "done" states were added to projects. However, a plan was originally implemented so that saved projects appeared at the top of Project Lists to improve upon this feature. It ended up being scrapped as peer testers felt like projects moving around in the list as they worked with it was unexpected behaviour. It was not found to be possible to both prevent this behaviour and keep state synchronised between tabs.

4. The user MAY be able to separately "like" a project.

Unimplemented. This, and a separate option to take notes on a project, were scrapped from Mizen based on time constraints and this feature's status as an optional requirement. Implementing it would be trivial (using the same logic as Saved and Done) but it would only complicate the UI for little gain.

5. The user MAY be provided suggested projects based on the projects they have previously shown an interest in.

Unimplemented. This was considered highest priority among the secondary objectives of the project at first, as the student was curious about the implementation of recommender systems in this and other applications. However, the sheer amount of additional complexity that this would add to the project, including requiring a far more detailed web backend, comprehensive user tracking, and possibly some form of account system, was deemed too much given the project's time constraints. It remains a suitable area for future work

6. The user MUST be able to see at a glance what other rooms are nearby.

Reasonably successful. This was implemented, and even showed the saved/seen status of projects in these rooms, but a fall-off in quality of suggestions was noted as distance between rooms increased.

7. The user MUST be able to see additional information about the project such as its full abstract, student and supervising lecturer.

Success. Each project has its own individual project page containing this information.

8. The user SHOULD be able to follow links to additional content such as the student's online presence or related documents to the project.

Success. This was implemented in the application, and the service to add this information was advertised to students, but nobody except the developer of the project added these fields. If the student could update FPM directly rather than their clone at colmfyp.netsoc.co, this functionality would be complete.

5.7.2 Non-Functional Requirements

The non-functional requirements of the project were fulfilled in the creation of the updated FPM backend and allowing using the Fetch API to communicate with it.

6 Conclusions

As is clear from the evaluation section, the project was a technical success and achieved most of its goals, except for an iOS App Store deployment.

In this section, the non-technical accomplishments and results of the project will be considered; the place of beacons and React Native in the industry, and possible future developments.

6.1 On React Native for cross-platform development

> *"React Native is the future."*
>
> - Stanislav Vishnevskiy, CTO, Discord [27]

The switch to React Native was the best decision made during the development of the application – without it, the application would either remain unfinished or it would be released as a substandard Cordova application. RN successfully delivered on its promise of a rapid development framework that could be used on both iOS and Android with ease.

In this sense, it's easy to see React Native find its niche. It allows businesses or individuals to develop applications that reach the widest possible audience with much less specialist knowledge required, reducing development time and cost. When *Electron* – a framework for developing desktop apps with HTML and JavaScript, like a Cordova for the desktop – arrived, it became massively popular among start-ups; suddenly, web-focused full-stack developers could bring their skillset to the desktop for the first time. Cordova failed to fill this gap on mobile, yet React Native shows far more promise.

A flaw remains: as an application becomes increasingly complex, so too increases the chance of needing to use native code to provide a final app. While the community continues to provide solutions in the form of open-source applications, it's harder to tell at an early stage if this problem will apply to any given project. The question of *"if I'll need to write native code anyway, why not start that way?"* hovers over small teams.

However, the fact that native code can be wrapped around a React Native app suggests a possibility of it standing out as a language for rapid agile development and prototyping. It also suggests a home for it on the opposite end: larger teams that now only need a smaller number of mobile specialists.

Looking at some of the teams that use React Native for their applications is a beacon of hope for the future of the technology. Facebook, who initially pioneered the technology, are the obvious highlight; the Facebook [28] and Instagram [29] apps which were built with React

Native [30] are the only third-party Android apps with over a billion downloads. Other notable teams are on this list, however; the communications start-up Discord [27] published a particularly valuable document, including claims of 98% code reuse between mobile and desktop in the state-handling core of the app.

There's a definite space for React Native in the *future*, but it's far from perfect within the present. The fact there isn't a fast, simple, bundled implementation of Navigator is one such cause for concern; the tendency for updates to have breaking effects is another. React Native has no final API and is still on a zero-point version; meaning that no support is guaranteed between updates. In the face of breaking changes, community contributions are often left abandoned; stalling progress. The package `react-native-ibeacon` is one such example; support was abandoned after a breaking change in `react-native 0.40.0` even though a community solution was available. Eventually, a community fork was made available (the `react-native-beacons-manager` referred to throughout the report), but this tendency to change makes React Native inaccessible to some at this point in its lifetime.

6.2 On beacon-driven location

Field testing for Mizen showed that displaying the current room worked as expected, but more distant beacons were much harder to differentiate. In the early stages of development, the student was unable to get reliable differences between two beacons in a small space. A common question from staff and students during the day was whether it would be possible to determine using beacons what individual *project* the user was viewing, and for the purposes of the open day, that would've been reasonably impossible, given how projects were located only a few metres apart – unless work was done over time to measure and fingerprint the room and allow for any variance.

Room-level context is far from useless: Mizen is proof of that. Also, there's nothing particularly wrong with an app that can determine a few nearby beacons to the device even if it can't determine which of the them is the *nearest* – it seems that most beacon-driven applications will show nearby items but not try to guess which of those is the current focus of the user's attention. However, the technology is limited: strong signals can still move through walls; for some sizes and shapes of rooms, managing this may prove difficult. Beacon technology is useful; but its use should be constrained.

6.3 Modern JavaScript

A minor, but important, detail of the application is its use of modern JavaScript. React Native's implementation of ES7 (and even some features from ES2017!) revealed a reliable, versatile and modern language, with excellent features such as strong object orientation and reliable handling of asynchronous tasks. Several issues with older versions of JavaScript such

as finicky binding of the "`this`" keyword, unclear variable scoping, or monolithic singular JS files, were tackled by new changes to the language. Even minor changes, such as the `Array.prototype.includes(v)` method make the language friendlier to work with. With more improvements to syntax such as array comprehensions planned for future releases, modern JavaScript is looking promising.

6.4 Reimplementing Mizen next year

It shouldn't be too difficult to implement Mizen at future final year project open days. If FPM is still in use, these are the necessary steps:

- Implement the new version of FPM that has been submitted along with this project.
    - Copy over the source code.
    - Use `php artisan migrate` to update the database.
- Obtain a SSL certificate for project.ucc.ie, and ensure that it can serve over https. Otherwise, the iOS build will not work.
- Replace references to colmfyp.netsoc.co in the Mizen source code with project.ucc.ie
- Attach beacons to a UUID and use this UUID in the place of the one given in-app.
- Build and deploy new releases of Mizen!

Ideally, this rollout should take place in early February, to give over time for unexpected outcomes in obtaining/securing an iOS release.

If FPM is replaced by a new system, make sure this new system can serve the projects as JSON over HTTPS, can manage rooms and assign them to beacons. Mizen will have to be rewritten to handle the new exported format, but most of its layouts will remain the same.

Contact me@colm.cf for support at any point for help and advice in reimplementing Mizen.

6.5 Future options for the technology behind Mizen

A primary concern when developing Mizen was to imagine a future for the application could be implemented in new contexts beyond the final year project open day.

While the app in its current form is tethered to the open day by being hardcoded to it – it downloads information specifically from the FPM clone, and while the fields displayed throughout the app are specific to this data, the core logic is the same; at a trade show, careers fair etc., switching "project" for "exhibit" and changing the app to fit details from a new data source would be trivial – this could be provided as a service to clients wishing to implement beacon-driven navigation at their event.

However, it's more exciting to imagine the logic behind Mizen as a cross-platform location-aware application implemented in ways that allow users to interact with their surroundings

rather than just be made aware of it. Consider the possible use case of the smart home, where a device knows what room its user is in and displays relevant controls – lighting, heat, media controls etc. – by using this source of context. There hasn't been an elegant solution for this problem yet within the smart home, and this room-level context by beacons could lead the way.

Ideally, Mizen should reach a state where any event manager can manage this manually – buying their own stack of beacons, tagging them to rooms or objects of relevance, and using an app to bind them to information.

An even brighter vision for Mizen might lie in the React Native community.

While developing the application, the student made occasional use of a tool called Expo [31] to test out new libraries. Based on the concepts of React Native, Expo went further on its core concepts:

- by providing a JavaScript SDK linking providing platform-agnostic access to parts of native functionality, such as
- by then dropping the requirement to build an app at all, allowing users to scan QR codes that download and run the JavaScript behind the application

Expo could be considered as an alternative implementation of the idea of the *progressive web application*, where ephemeral web applications with access to native mobile functionality could be a way forward.

Expo wasn't useful for building Mizen itself – not being able to import libraries that relied on native code ruled out implementing beacon functionality. However, Expo could act as a blueprint for a next-generation version of Mizen, where instead of displaying simple information or set controls, experiences with native functionality can be written for individual beacons, and the application can seek out and download these local experiences based on nearby beacons.

Finally, there's room to create more of a fleshed-out backend for Mizen that notes interest in projects and suggests other projects based on overall similarity.

6.6 A final personal statement

I took on developing this project to learn more about mobile development, a field that I'd previously no experience in. I initially underestimated the sheer scale of this undertaking – the deceptively simple scale of the app led me to come at the project from the wrong side.

Opportunity arose from difficulty in the pivot to React Native – I credit it as being the key to not only completing the application on both platforms but improving the quality and

standards of the final product by giving me more time to focus. A three-line function to *fetch* from the API in React Native was a sharp contrast to learning the Retrofit model for Android – and Retrofit was a major community improvement over the pure-Java status quo.

While I ended up failing to achieve some of my personal goals in the project – to learn the Swift programming language, or gain specialist knowledge in beacons and native application state handling on Android and iOS, I've been lucky to gain the knowledge to fill this report, finding new, exciting ways to use the skills I'd already learned to develop for the web, while expanding on those skills. I'm confident in my knowledge of React Native today.

I'm proud of the final application that has been developed, and of the work that has gone into it. With the permission of Prof. Sreenan, I've made the project open-source and will continue to use it as a portfolio piece for years to come.

Appendices

Appendix A: Denial of Expedite Request

Hello Colm,

Thank you for contacting the App Store Review team. We are unable to accommodate your request for an expedited review at this time. While we do our best to accommodate requests for expedited reviews and take individual circumstances into consideration, we are unable to grant every request due to our volume. Helping you get your app, IAPs, or bundle to the App Store is very important to us, and we are working hard to process all submissions as quickly as possible.

Best regards,
App Store Review

Appendix B: Request for Information – Beacons

Guideline 2.1 - Information Needed

We have started the review of your app, but we are not able to continue because we need additional information about your app.

Next Steps

To help us proceed with the review of your app, please review the following questions and provide as much detailed information as you can.

- Does this app detect startMonitoringForRegion:, startRangingBeaconsInRegion:, or both?

- What is the user experience when the app detects the presence of a beacon?

- What features in this app use background location?

- If this app uses 3rd party SDKs for iBeacons, please provide links to their documentation showing that background location is required for it to function.

Please reply to this message in Resolution Center with the requested information.

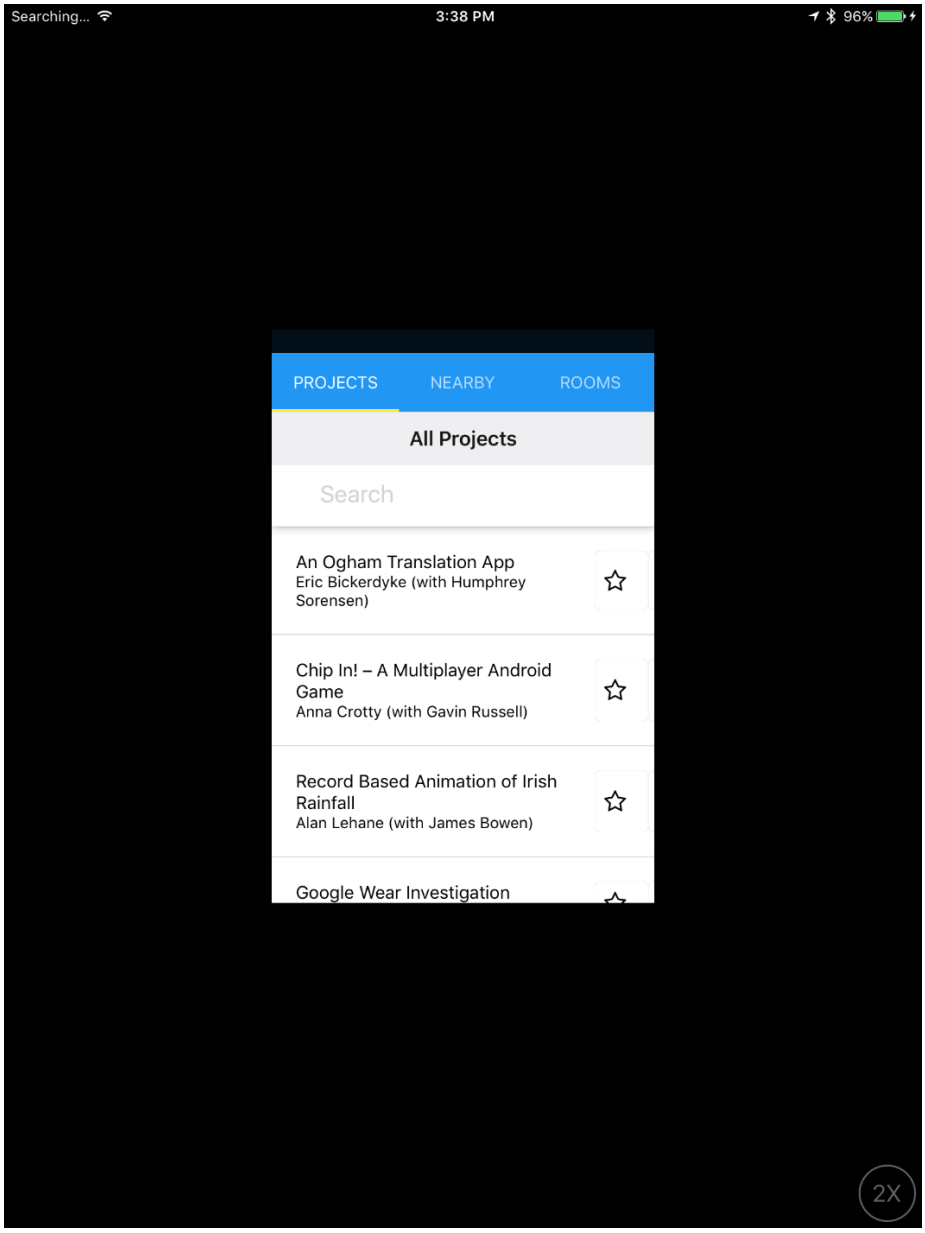
Appendix C: Rejection Notice

Guideline 2.3.10 – Performance

We noticed that your app or its metadata includes irrelevant third-party platform information. Specifically, we have found Android referenced upon launch of the application.

Referencing third-party platforms in your app or its metadata is not permitted on the App Store unless there is specific interactive functionality.

Please see attached screenshots for details.



Next Steps

To resolve this issue, please remove all instances of this information from your app and its metadata, including the app description, What's New info, previews, and screenshots.

Since your iTunes Connect status is **Rejected**, a new binary will be required. Make the desired metadata changes when you upload the new binary.

NOTE: Please be sure to make any metadata changes to all app localizations by selecting each specific localization and making appropriate changes.

Appendix D: The Application Layout

This is an extended version of a figure shown earlier in the application that *approximates* the application layout. It is not valid JSX. It does aim to show how components are linked in a React tree, but it's not particularly useful – simply included for completeness.

```
<Mizen>
  <SplashScreen/>
  <Tabs>
    <AllProjectsNav>
      <AllProjectsScreen>
        <Provider store={store}>
          <AllProjectsList navigation={ap_nav}>
            <ListView>
              <ProjectRowItem/>...
            </ListView>
            <SearchBar/>
          </AllProjectsList>
        </Provider>
      </AllProjectsScreen>
      <IndividualProjectScreen/>
    </AllProjectsNav>
    <NearbyNav>
      <NearbyScreen>
        <Provider store={store}>
          <NearbyRoomInfo navigation={nb_nav}>
            <ProjectList projects={nearbyProjects}>
              <ListView>
                <ProjectRowItem/>...
              </ListView>
            </ProjectList>
            <NearbyInfo rooms={nearbyRooms}/>
          </NearbyRoomInfo>
        </Provider>
      </NearbyScreen>
      <IndividualProjectScreen/>
    </NearbyNav>
    <AllRoomsNav>
      <RoomListScreen>
        <Provider store={store}>
          <RoomList navigation={rm_nav}>
            <ListView>
              <RoomRowItem/>...
            </ListView>
          </RoomList>
        </Provider>
      </RoomListScreen>
      <IndividualRoomScreen>
        <ProjectList projects={room_projects}>
          <ListView>
            <ProjectRowItem/>...
          </ListView>
        </ProjectList>
      </IndividualRoomScreen>
      <IndividualProjectScreen/>
    </AllRoomsNav>
  </Tabs>
</Mizen>
```

References

[1] C. Sreenan, "A guidance app for Final Year Project Open Day," 20 September 2016. [Online]. Available: http://project.ucc.ie/projects/218. [Accessed 17 April 2017].

[2] Pointr Labs, "Beacons - Everything You Need To Know," July 2016. [Online]. Available: http://www.pointrlabs.com/blog/beacons-everything-you-need-to-know/.

[3] Apple, Inc., "Getting Started With iBeacon (version 1.0)," 2 June 2014. [Online]. Available: https://developer.apple.com/ibeacon/Getting-Started-with-iBeacon.pdf. [Accessed 4 April 2017].

[4] Google, "The Physical Web," [Online]. [Accessed 8 April 2017].

[5] Estimote, "Beacon Tech Overview," 2014. [Online]. [Accessed 8 April 2017].

[6] Ionic, "The Last Word on Cordova and PhoneGap," 6 March 2014. [Online]. Available: http://blog.ionic.io/what-is-cordova-phonegap/. [Accessed 9 April 2017].

[7] Facebook, "React - JSX In Depth," 31 January 2017. [Online]. Available: https://facebook.github.io/react/docs/jsx-in-depth.html. [Accessed 9 April 2017].

[8] Facebook, "React Native Documentation (v 0.42)," 1 March 2017. [Online]. Available: http://devdocs.io/react_native/. [Accessed 9 March 2017].

[9] "ModuleCounts," 4 April 2017. [Online]. Available: http://www.modulecounts.com/. [Accessed 4 April 2017].

[10] Kontakt.io, "Boosting Mobile Experiences at Events With Proximity," 2016. [Online]. Available: https://kontakt.io/resources/. [Accessed 9 April 2017].

[11] Kontakt.io, "Making Audio Guides Relevant in the 21st Century with the Power of Proximity," 2017. [Online]. Available: https://kontakt.io/resources/. [Accessed 9 April 2017].

[12] Pulsate, "Pulsate - Customers," 10 April 2017. [Online]. Available: https://www.pulsatehq.com/why-pulsate.

[13] S. Bradner, "RFC2119: Key words for use in RFCs to Indicate Requirement Levels," IETF, Cambridge, MA, 1997.

[14] Microsoft, "react-native-windows on GitHub," [Online]. Available: https://github.com/Microsoft/react-native-windows.

[15] Apple, Inc., "Get Xcode," [Online]. Available: https://developer.apple.com/download/. [Accessed 16 April 2017].

[16] S. C. &. B. Straub, Pro Git (2nd Edition), Apress, 2014.

[17] Mozilla, "Fetch API," 9 April 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API. [Accessed 12 April 2017].

[18] D. Abramov, "Redux.js Documentation," 31 March 2017. [Online]. Available: http://redux.js.org/. [Accessed 12 April 2017].

[19] React Navigation, “React Navigation,” 22 February 2017. [Online]. Available: http://reactnavigation.org. [Accessed 16 April 2017].

[20] W3C, “CSS Flexible Box Layout Module Level 1,” 26 May 2016. [Online]. Available: https://www.w3.org/TR/css-flexbox-1/. [Accessed 15 April 2017].

[21] “Lighthouse,” in *Encyclopaedia Britannica*.

[22] FlatIcon, “Flaticon Basic Licence,” [Online]. Available: http://file000.flaticon.com/downloads/license/license.pdf. [Accessed 14 April 2017].

[23] A. Ghoshal, “Apple is fixing App Store reviews at last,” *The Next Web,* 2017 January.

[24] Shiny Development, “App Review Times: Annual Trend Graph,” 16 April 2017. [Online]. Available: http://appreviewtimes.com/ios/annual-trend-graph. [Accessed 16 April 2017].

[25] Diawi, “Diawi,” [Online]. Available: http://diawi.com. [Accessed 17 April 2017].

[26] Facebook, “Jest - Javascript Testing Framework,” [Online]. Available: https://facebook.github.io/jest/docs/tutorial-react-native.html. [Accessed 7 April 2017].

[27] F. Chen, “Using React Native: One Year Later - Discord Engineering,” 7 June 2017. [Online]. Available: https://blog.discordapp.com/using-react-native-one-year-later-91fd5e949933. [Accessed 17 April 2017].

[28] AppBrain, “Facebook - Android App Statistics,” [Online]. Available: http://www.appbrain.com/app/facebook/com.facebook.katana. [Accessed 17 April 2017].

[29] AppBrain, “Instagram - Android App Staistics,” [Online]. Available: http://www.appbrain.com/app/instagram/com.instagram.android. [Accessed 17 April 2017].

[30] P. De Baets, A. Lang, F. Sagnes and T. Zagallo, “Dive into React Native performance,” 28 March 2016. [Online]. Available: https://code.facebook.com/posts/895897210527114/dive-into-react-native-performance/. [Accessed 17 April 2017].

[31] Expo, “Expo Documentation, version 15,” April 2017. [Online]. Available: https://docs.expo.io/versions/v15.0.0/index.html. [Accessed 17 April 2017].

[32] K. Soltani, “Building Mobile Apps with Cordova, AngularJS and Ionic,” LinkedIn, 2014.