CA400 Technical Guide

Project Title: NFC Powered DCU
Students: Eamon Crawford & Sam O’Leary
Student Numbers: 16437394, 16358763
Supervisor: Renaat Verbruggen
Completion Date: 07/05/2021

Abstract

NFC powered smart cards can be used for many purposes, such as paying contactless at a store, in a manner which reduces complexity for the end user, such as not having to remember and enter a pin for a debit card. Creating student cards with this ability opens many avenues of possibility, such as booking and signing into study rooms, reducing human error when taking exam attendance, integrating with a car park system, tracking group locations for covid-19 exposure reasons and marking yourself as present at a lecture.
We have created a timetable system that handles attendance by using small android NFC readers to set student attendance.

Table of Contents

1. Introduction

1.A Overview

The system has 3 main components: the backend server, the android NFC reader app, and the website. Eamon created the backend server and the NFC reader app, and Sam created the website.

The backend server contains the postgreSql relational database, a Kotlin Exposed powered DAO (Data Access Object) layer, logic to handle modeling DCU's student-module-lecture system and an API to provide functionality to both the website and the Android device, as well as any future components that could be built, such as a student app.

The android app is a lightweight native application that gets registered to a lecture, and then when it gets tapped by a student card, registers that student as having attended the lecture.

The website serves two types of users, firstly for students to get timetable information, but also for university staff to look at analytics and manage the system. Staff should be able to create, update and delete modules and lectures. They can edit information like students enrolled in each module and lecture's location and time.

1.B Motivation

Our main motivation for this project was to provide a convenient solution to the issue with tracking attendance with accuracy, caused by the onset of the COVID19 pandemic.
DCU has also put a lot of emphasis on preventing student dropouts and we believe that being able to accurately track lecture attendance would provide useful information for dealing with this issue. These motivating factors highlighted a need in the space for a solution, we identified NFC as an ideal method for solving this problem.

1.C Glossary

NFC - Near Field Communication.
DAO - Data Access Object.
Kotlin - JVM compatible language developed by JetBrains.
PostgreSQL - PostgreSQL is a powerful, open source object-relational database system that uses and extends the SQL language.
API - an application programming interface is an interface that defines interactions between multiple software applications.
Ngrok - allows you to expose a web server running on your local machine to the internet.
ORM - Object Relational Mapping
URL- Uniform Resource Locator, also known as a web address.
DSL- Domain-Specific Language, is a computer language that's targeted to a particular problem, rather than a general purpose language that's aimed at any kind of software problem.

2. Research

2.A Researching Tools

Our first task was to research what tools we want to use to build our system. For the front end design we looked into various languages but settled on React. React was chosen due to the prevalence of resources online and available libraries we could use. This would allow us freedom in the design and enable easy connection to the API through the use of libraries such as Axios. We elected to use Axios in this project as it allowed for easier use of Ajax requests, which we were not previously accustomed with.
In order to display the data in a digestible format we elected to visualize the data with graphs and a timetable. For the graphing component we researched several different libraries available to us, we eventually settled on using the react-chartjs-2 library. This library was selected as it has comprehensive documentation and good support for rendering data from Axios requests.
To implement the timetable component we initially looked at implementing our own version from scratch, but after further research we identified this as a large time sync and instead elected for the use of the FullCalendar library. The FullCalendar library allows for easy display of lectures in a presentable format, and also has solid documentation on how to implement axios requests with the calendar.

We needed a relational database and chose to use postgreSQL due to some of its advanced features like storing array types, and its pgAdmin interface. PostgreSQL is also supported by Exposed an ORM(Object Relational Mapping) framework for Kotlin which was important.

We picked Kotlin as the language for our backend service. Kotlin is a language that is growing in popularity. It is JVM compatible so we get access to a huge number of java libraries like junit5. It would also be a good chance to become familiar with a new language. Kotlin can also be used for Android development (which is what it is most popular for), and not switching languages between backend and android is easier for development.

We used Exposed, the ORM framework for Kotlin as it provides a great API for some relational database implementations. We also used Ktor for our http service, allowing us to easily organise many endpoints for the Rest API. We transported data in JSON (Javascript Object Notation) format. We used Netty as our server engine. Both Exposed and Kotlin have support from JetBrains, the company behind Kotlin. JetBrains also develop Intellij IDEA so there is great cohesion in the tools. To expose our locally running backend service we used a free tier of Ngrok to set up a secure URL to our localhost, the free tier unfortunately means our URL changes every 2 hours.

For testing the backend we used Junit5 and MockK, a mocking library for kotlin, as it can mock Kotlin static like objects. For adhoc testing the API, we used a postman collection, as it made it easy to repeatable test the endpoints quickly, and communicate how the endpoints work to my partner. To measure test coverage we used Jacoco to create test metrics.

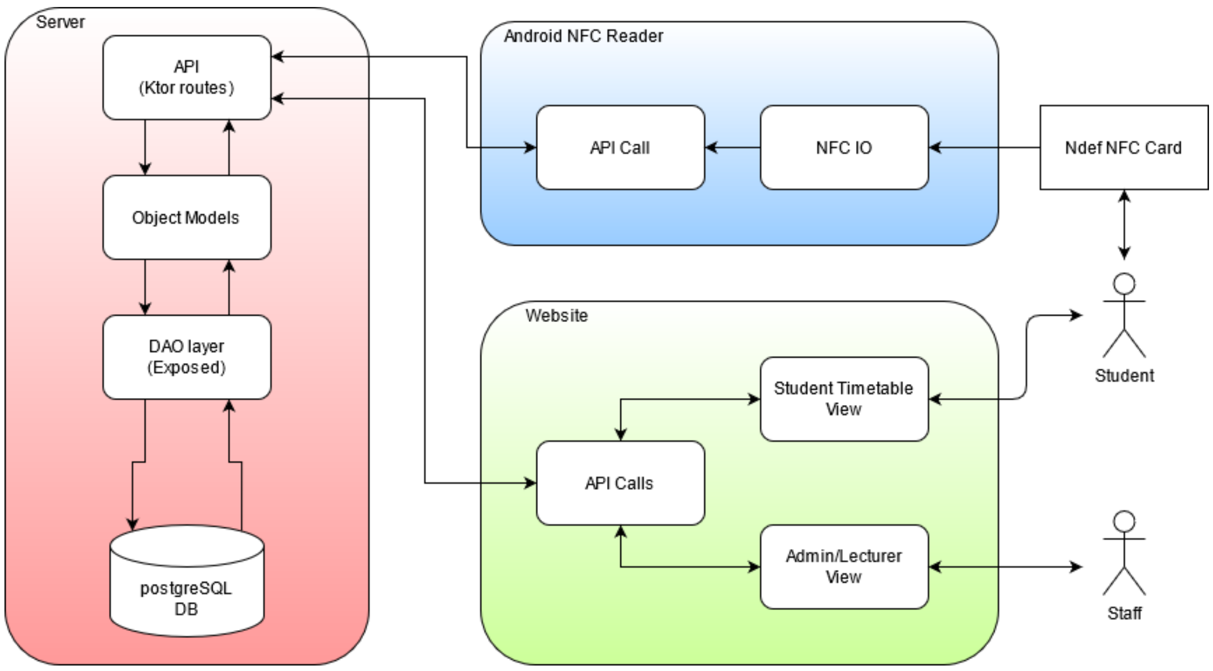
For android development we had to use Android Studio as our IDE, and for ad hoc testing our code we ran the code on an old Samsung android phone with NFC capabilities. We used retrofit for our http calls.
For building both the backend and the android app, we used the gradle build tool.

2.B Researching DCU's current system

Our initial project idea was based on adding some portability and convenience to marking attendance in DCU. Currently there's no real system to track attendance except to pass a sheet of paper around a lecture hall. This was not a serious issue until the COVID19 pandemic occurred and contact tracing became a necessity. While this was a driving factor for the project idea, it also has multiple other uses outside the pandemic, as it can track the attendance for analysis reasons, it could be used to manage the car parking system or to manage study rooms availability. After looking into all these possibilities, with a focus on use within DCU, we elected to limit the project to just lectures and attendance tracking. This would demonstrate the core of our project idea, and could be expanded upon for other use cases once complete.

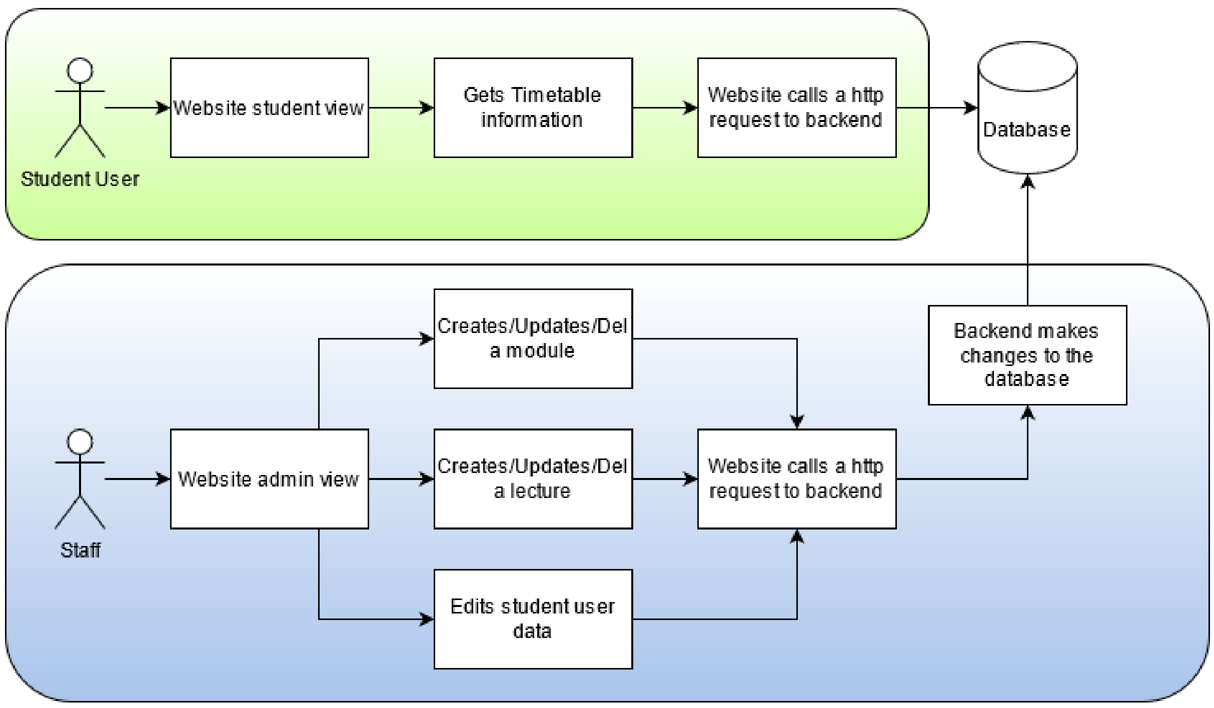
3. Design

3.A System Architecture Diagram

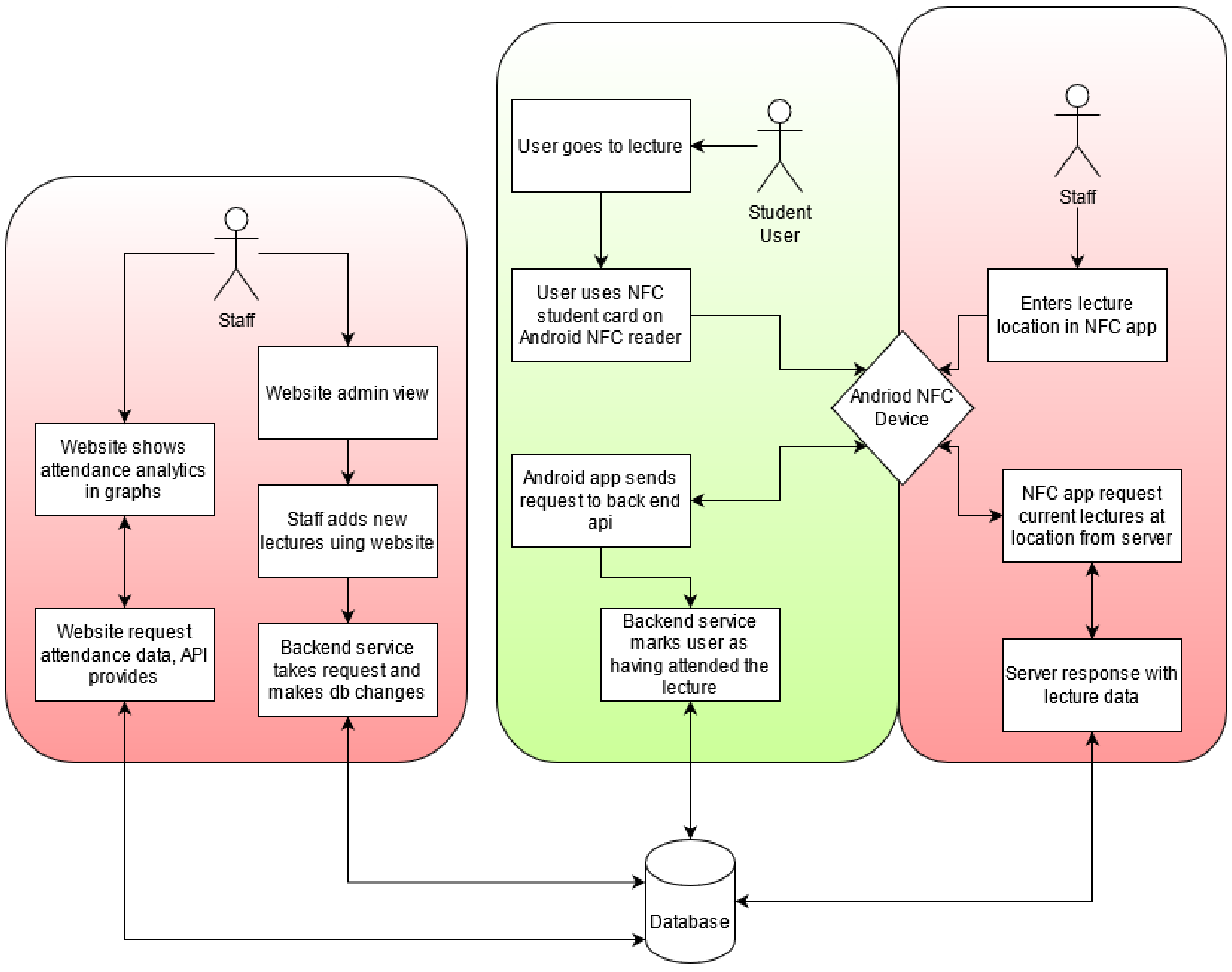


3.B Use Case Diagrams

3.B.1


Student User
Website student view
Gets Timetable information
Website calls a http request to backend
Database
Staff
Website admin view
Creates/Updates/Del a module
Creates/Updates/Del a lecture
Edits student user data
Website calls a http request to backend
Backend makes changes to the database

3.B.2



Staff
Website shows attendance analytics in graphs
Website admin view
Staff adds new lectures uing website
Website request attendance data, API provides
Backend service takes request and makes db changes
User goes to lecture
Student User
User uses NFC student card on Android NFC reader
Android app sends request to back end api
Backend service marks user as having attended the lecture
Andriod NFC Device
Staff
Enters lecture location in NFC app
NFC app request current lectures at location from server
Server response with lecture data
Database

4. Implementation

4.A User Interface

The user interface in this project refers to the React website.
As I was not very knowledgeable working with React at the start, I started off by completing some tutorials on making basic websites and browser games in order to better familiarise myself with React. Once I was more comfortable with it, I began designing mock versions of our site and began researching what kind of libraries were available and might be of use. After completing a mock site that was of a high enough standard and looking into libraries such as axios and chartjs, I decided on React for the User interface design. I then had to begin looking into how to correctly implement libraries for the graphing and the API requests.

While React ships with ajax built in, the Axios library helps to simplify these requests and makes the overall process easier. While researching the chartjs library I found chartjs-2, this library was a continuation of the previous library but is better developed to handle data streams. I had to read the documentation and follow several tutorials on rendering dynamic data as it proved to be more difficult than previously I had thought. Passing in variable paths and the formatting of the data proved to be an issue, the formatting issues were easily solvable with further reading of the documentation and discussion on the backend, but for variable paths we had to look into the use of url templates and the correct method of implementation for them. This period of learning proved to be extremely useful when we went to work on the timetable component.

While initially we had planned to design the timetable component from scratch using React's styled components. However upon further researching the method and initial trial and error for the implementation, we realized the actual complexity of designing a component that would be high quality and the large time sink it would be. While this could largely be viewed as lost time, it served as a great learning experience and helped further development to be more realistic. After further research, we chose between the FullCalendar library and the BigCalendar libraries. We tested both libraries and found the FullCalendar library easier to work with and with better available documentation. Implementing the axios requests in this component was a direct continuation of the graphing component which allowed us to spend less time implementing and more time optimizing. In order to test these axios requests we initially had it connect to a copy of our API and database which was being run locally on our computers. This allowed for faster and clearer diagnosis of the connections and any issues we may have had. Once the connections were validated and the components were fully functional, we tested out the connection between our two machines using ngrok to establish a tunnel.

Throughout the implementation of the system we employed ad hoc testing to ensure basic functionality and then once completed, we used unit testing.

4.B Backend Service

For this project, backend service refers to our REST API, the business logic layer, and our DAO. When starting this implementation, I work through some tutorials on the Exposed framework. Exposed allowed me to map my Kotlin objects to my relational database tables. Exposed offers two levels of database access: typesafe SQL wrapping DSL and lightweight data access objects. I used the DAO functionality. Originally I had Exposed running with an in memory H2 database, so I could test the framework before investing my time into the full implementation that we have now. After a short time of familiarising myself with Kotlin and Exposed, I decided both were great choices for our project, partially because of how easy Kotlin was to pick up as someone who is familiar with Java but also because I found the documentation to be clear, concise and complete. With Exposed I create 3 DAO objects that map to the 3 tables in the database, without having to do much SQL code myself. To keep the database access code and helper methods for that separate I created two packages, 'dao' and 'daoUtils'. The objects that map to each table can be found in the 'table' package. The next step was moving off of H2 and onto a proper SQL relational database. Exposed supports many options and I decided to go with PostgreSQL as I had used it in the past. I speak more about PostgreSQL in the Database section.

My research had also discovered Ktor, a Kotlin asynchronous framework for creating microservices, web applications, http services etc. Ktor, like Exposed, gets support from JetBrains, and the standard for documentation and the ease of use for the framework was, for the most part, just as good as Exposed. My Ktor work can be found in the 'routes' package, with 3 different collections of end points, separated by which object they were associated with, each with a different URL. Some supporting data classes for mapping JSON requests can be found in the ‘requestObjects’ package.

For validation I used the Junit5 testing framework for Java and the JVM, which is therefore compatible with Kotlin. Junit5 is a well documented and mature testing framework. For mocking objects in tests, I was originally planning on using mockito, which I had used in the past, until I discovered MockK, which is a mocking library specifically for Kotlin. With MockK I could mock Kotlin’s static-like objects which was a must for my testing. I created both unit tests and integration tests that use a real database set up for testing. I also used postman’s monitoring feature to test my endpoints.

4.C Android Application

The native android application has 3 main functions: getting the current lecture when given a room location, taking data off of NFC student cards and finally using that data to mark students as having attended that lecture.

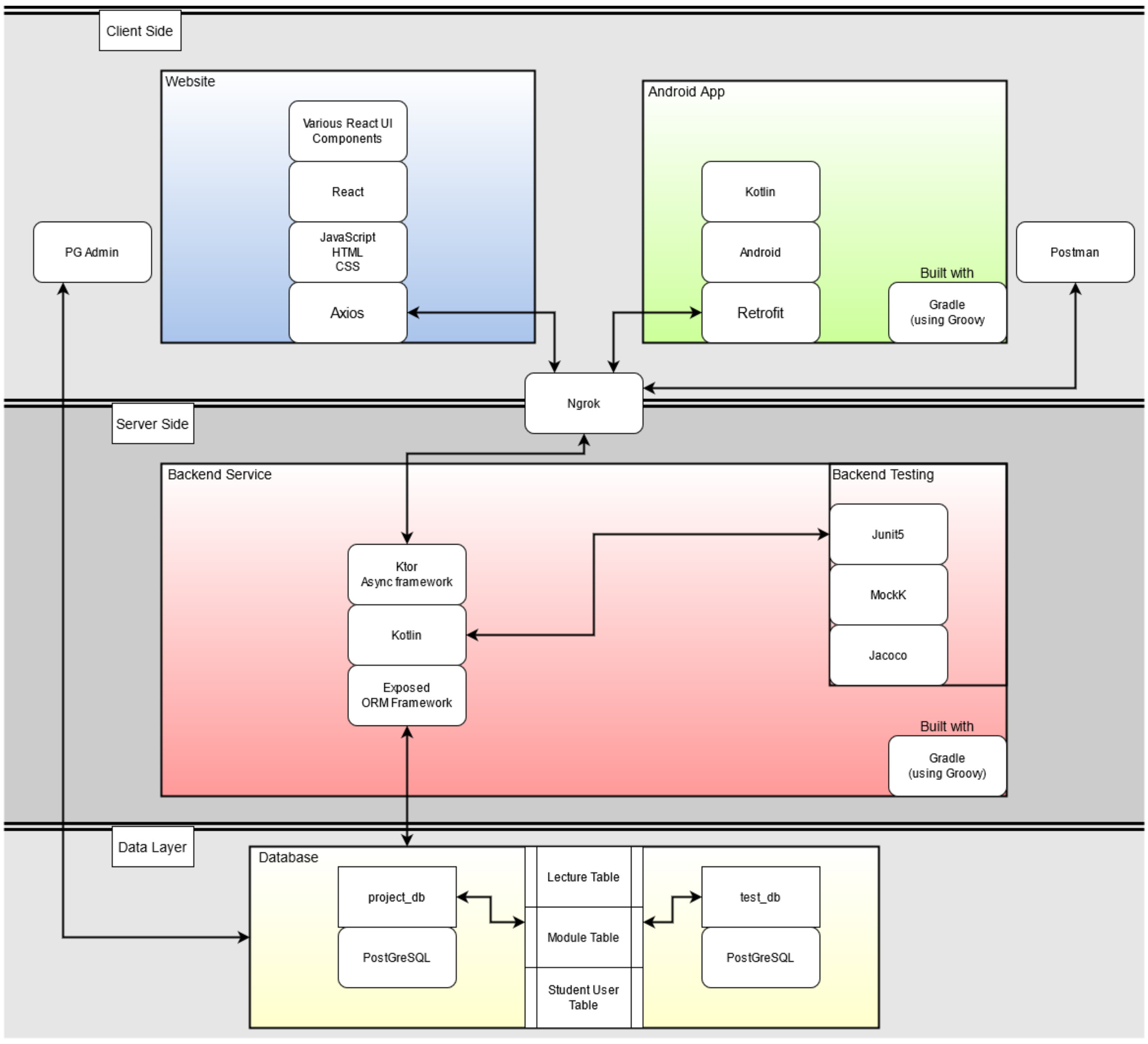
The app uses retrofit2 for its api calls to the server, when a location is entered, the app creates a request that returns the lecture (or multiple lectures in case of a booking clash or no lectures in case of a mistake), that is meant to be happening at that location for now or the near future. The app will display the current set location and the module code of the current set lecture, if one is set. Giving feedback to the users about what stage the app is at in its set up is crucial for the user’s experience.

Getting data off NFC cards can be complex, as there are many different formats. For this project I was only able to test cards with the Ndef (NFC Data Exchange Format) format due to equipment restraints, however the project should in theory also work with MiFare and other formats. I use the native android NFC adaptor as I didn't want to introduce many dependencies to the app, keeping it lightweight so it can run on more devices. The app reads a student’s student Id/Number off of their student card and uses this to mark that user as having attended the lecture. University staff can then look at graphs of attendance data on the website to find trends or issues.

4.D Database

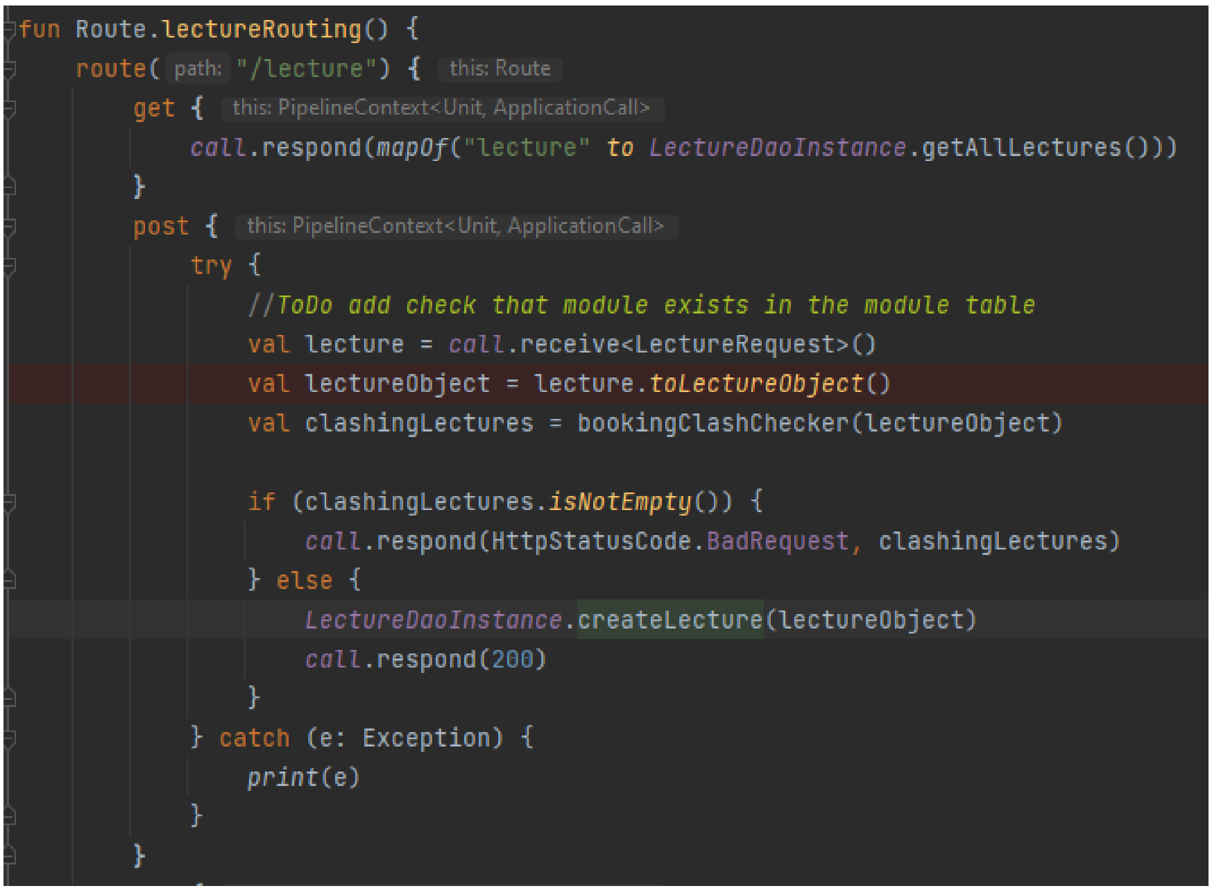
There are actually two databases for this project, one is for live real data and the other is a duplicate for testing. Both databases contain the same 3 tables that map to our 3 data classes through Exposed. These are the lecture, module and studentUser table. Each entry has a UUID so that row can be uniquely identified for editing. Using postgreSQL for the database also gives us, and potential future system admins, the ability to interact with the database using the powerful pgAdmin client.

4.E Tech Stack Diagram



5. Sample Code

5.A Routes using Ktor

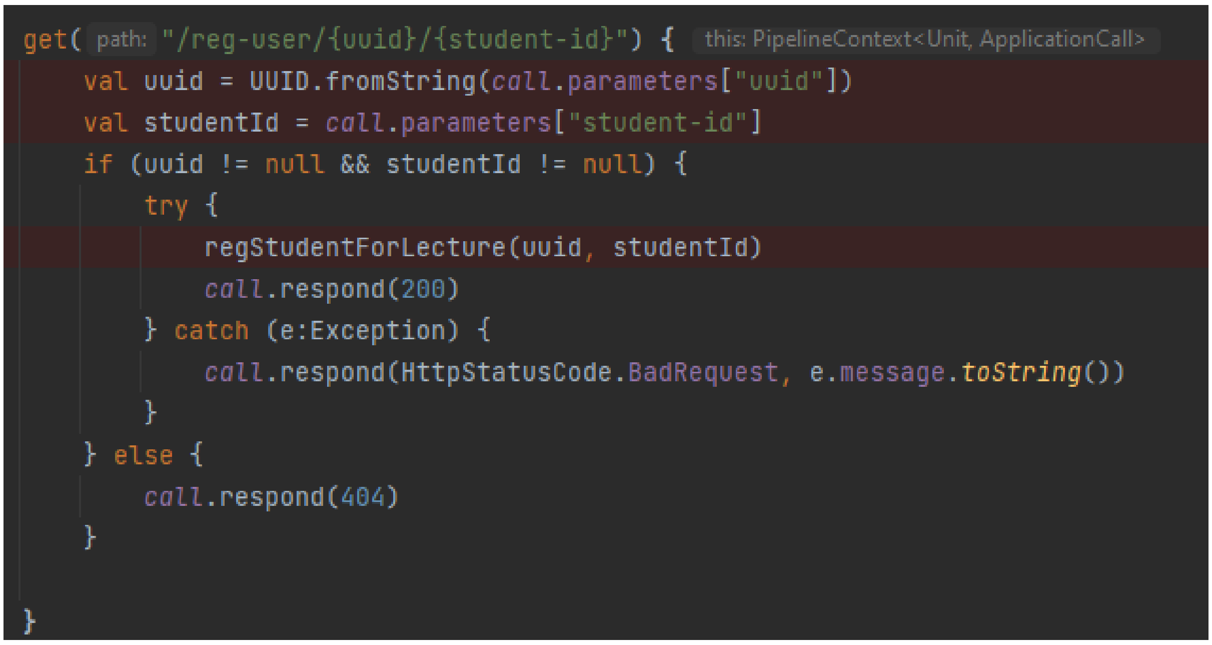


```kotlin
fun Route.lectureRouting() {
    route( path: "/lecture") {  this: Route
        get {  this: PipelineContext<Unit, ApplicationCall>
            call.respond(mapOf("lecture" to LectureDaoInstance.getAllLectures()))
        }
        post {  this: PipelineContext<Unit, ApplicationCall>
            try {
                //ToDo add check that module exists in the module table
                val lecture = call.receive<LectureRequest>()
                val lectureObject = lecture.toLectureObject()
                val clashingLectures = bookingClashChecker(lectureObject)

                if (clashingLectures.isNotEmpty()) {
                    call.respond(HttpStatusCode.BadRequest, clashingLectures)
                } else {
                    LectureDaoInstance.createLecture(lectureObject)
                    call.respond(200)
                }
            } catch (e: Exception) {
                print(e)
            }
        }
```

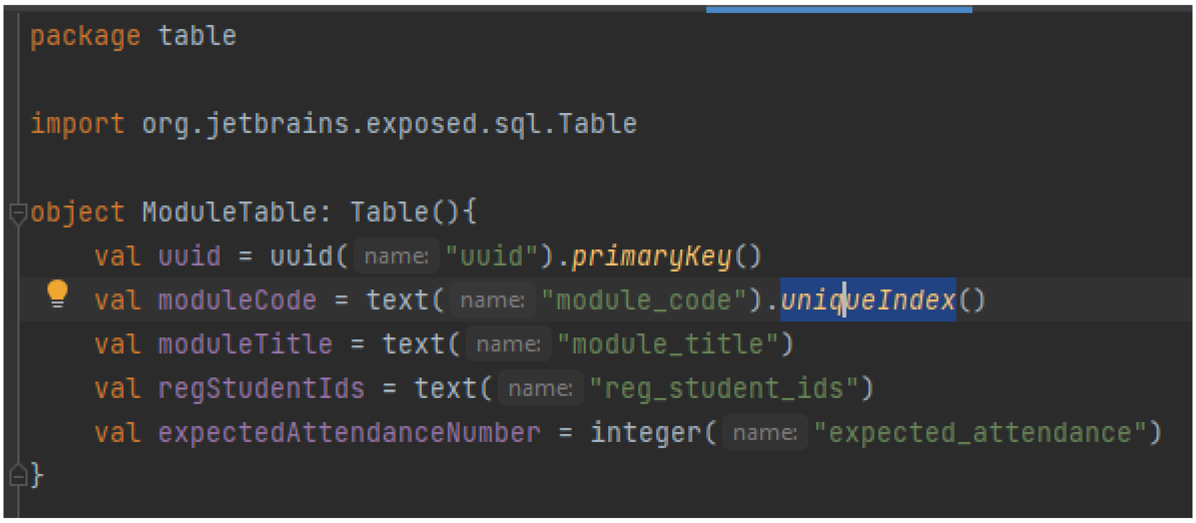
The above code shows the Ktor routes for getting all lectures in the database as well as the post request used to create a lecture. The get has a route of GET baseURL/lecture. It calls the lecture DAO instance method to get all lectures and responses to the call with them. The post request is more interesting, for we take the body the request and map it to the object LetureRequest. The code then calls the LectureRequest method to LectureObject, this method parses data like the dates that as Strings in request object, and creates proper DateTime objects for them. We then need to check if this soon to be created lecture will have a booking clash with any current lectures, we do this by calling the lectureDaoUtils method booking clash checker, which takes the DateTimes and the location for new lecture and returns any other lectures in the db that would clash with it. If there are clashes, we don't create the new lecture and return a list of all the clashing lectures. If there are no clashes we create the lecture and respond with the 200 OK http status code.

Below is the Ktor route for registering a student as having attended a lecture. This route gets called by the android app after a success NFC event. You can see how the route is more complex, as there are variable for the lecture’s UUID and the students Id. These variables are taken as parameters. For this method we also return a custom error message if an exception is thrown by the regStudentForLecture function.


```kotlin
get( path: "/reg-user/{uuid}/{student-id}") { this: PipelineContext<Unit, ApplicationCall>
    val uuid = UUID.fromString(call.parameters["uuid"])
    val studentId = call.parameters["student-id"]
    if (uuid != null && studentId != null) {
        try {
            regStudentForLecture(uuid, studentId)
            call.respond(200)
        } catch (e:Exception) {
            call.respond(HttpStatusCode.BadRequest, e.message.toString())
        }
    } else {
        call.respond(404)
    }

}
```

5.B Object Table Maps

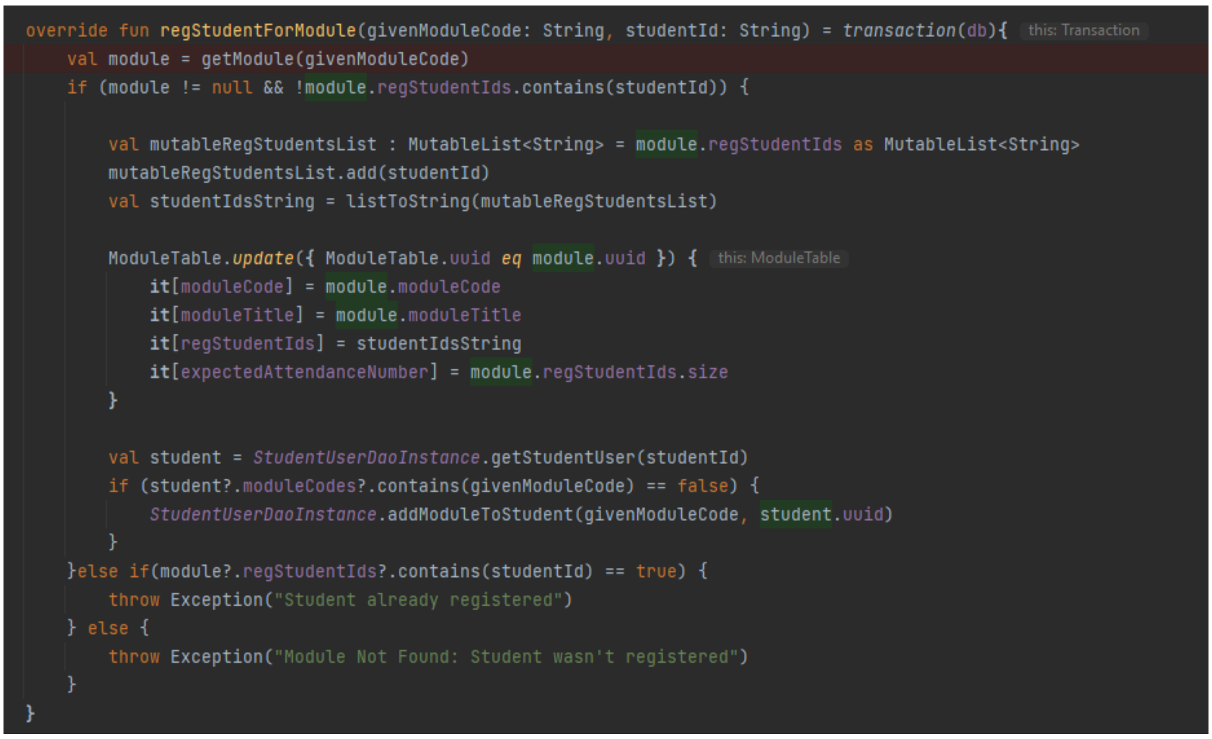

```kotlin
package table

import org.jetbrains.exposed.sql.Table

object ModuleTable: Table(){
    val uuid = uuid( name: "uuid").primaryKey()
    val moduleCode = text( name: "module_code").uniqueIndex()
    val moduleTitle = text( name: "module_title")
    val regStudentIds = text( name: "reg_student_ids")
    val expectedAttendanceNumber = integer( name: "expected_attendance")
}
```

Above is the code that maps the module object as a SQL table. We can see UUID gets set as the primary key, but the moduleCode is also set as a uniqueIndex. Both of these columns must be unique to that row and not appear in any other row. regStudentIds is interesting as its name implies an array but it's only a string. PostgreSql supports having arrays as a column data type but Exposed does not support this. As I was planning to use this postgreSQL feature I didn't want to spin up a lot of join tables each time I wanted to store an array. So instead I translate my lists into strings when storing them, and translate the string into lists when calling them from the DB. This may create scaling problems and was only done as a quick fix due to time constraints.

5.C ModuleDao code to register a student for a module in the db



```
override fun regStudentForModule(givenModuleCode: String, studentId: String) = transaction(db){ this: Transaction
    val module = getModule(givenModuleCode)
    if (module != null && !module.regStudentIds.contains(studentId)) {

        val mutableRegStudentsList : MutableList<String> = module.regStudentIds as MutableList<String>
        mutableRegStudentsList.add(studentId)
        val studentIdsString = listToString(mutableRegStudentsList)

        ModuleTable.update({ ModuleTable.uuid eq module.uuid }) { this: ModuleTable
            it[moduleCode] = module.moduleCode
            it[moduleTitle] = module.moduleTitle
            it[regStudentIds] = studentIdsString
            it[expectedAttendanceNumber] = module.regStudentIds.size
        }

        val student = StudentUserDaoInstance.getStudentUser(studentId)
        if (student?.moduleCodes?.contains(givenModuleCode) == false) {
            StudentUserDaoInstance.addModuleToStudent(givenModuleCode, student.uuid)
        }
    }else if(module?.regStudentIds?.contains(studentId) == true) {
        throw Exception("Student already registered")
    } else {
        throw Exception("Module Not Found: Student wasn't registered")
    }
}
```

The above code directly interacts with the database, when it calls ModuleTable.update and again when it calls StudentUserDaoInstance.addModuleToStudent. One useful feature about kotlin is its null safety. In the above code the line

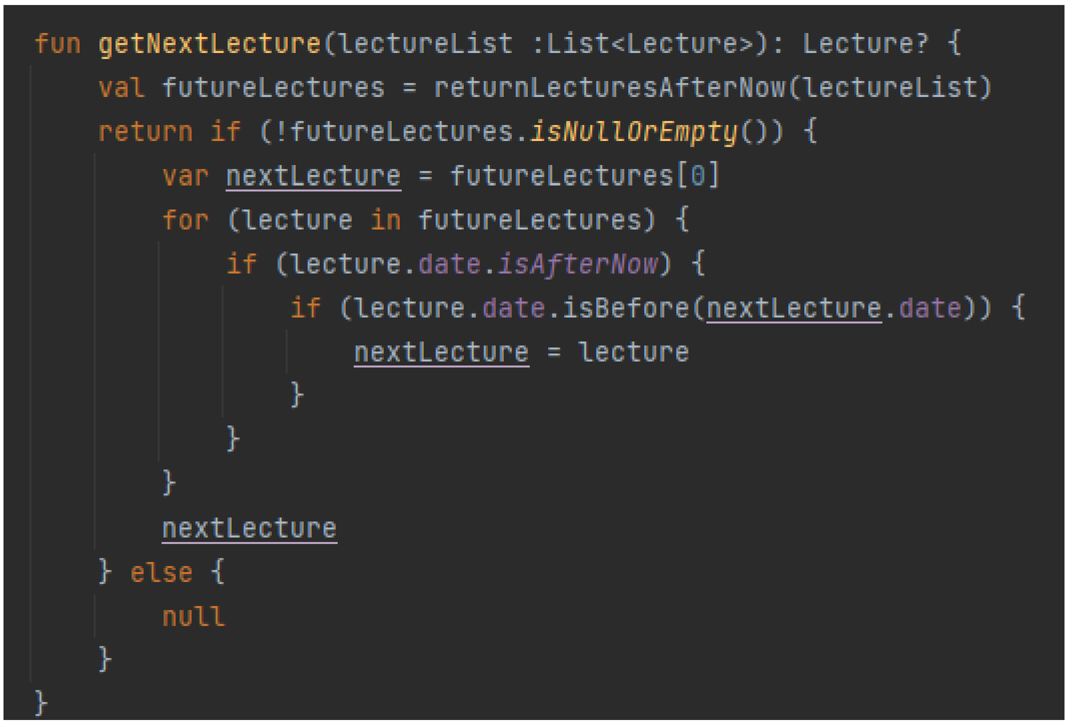
```
if(student?.moduleCodes?.contains(givenModuleCode) == false)
```

Is the same as writing the following in Java

```
if (student != null) {
    if (student.moduleCodes.contains(givenModuleCode)) {
```

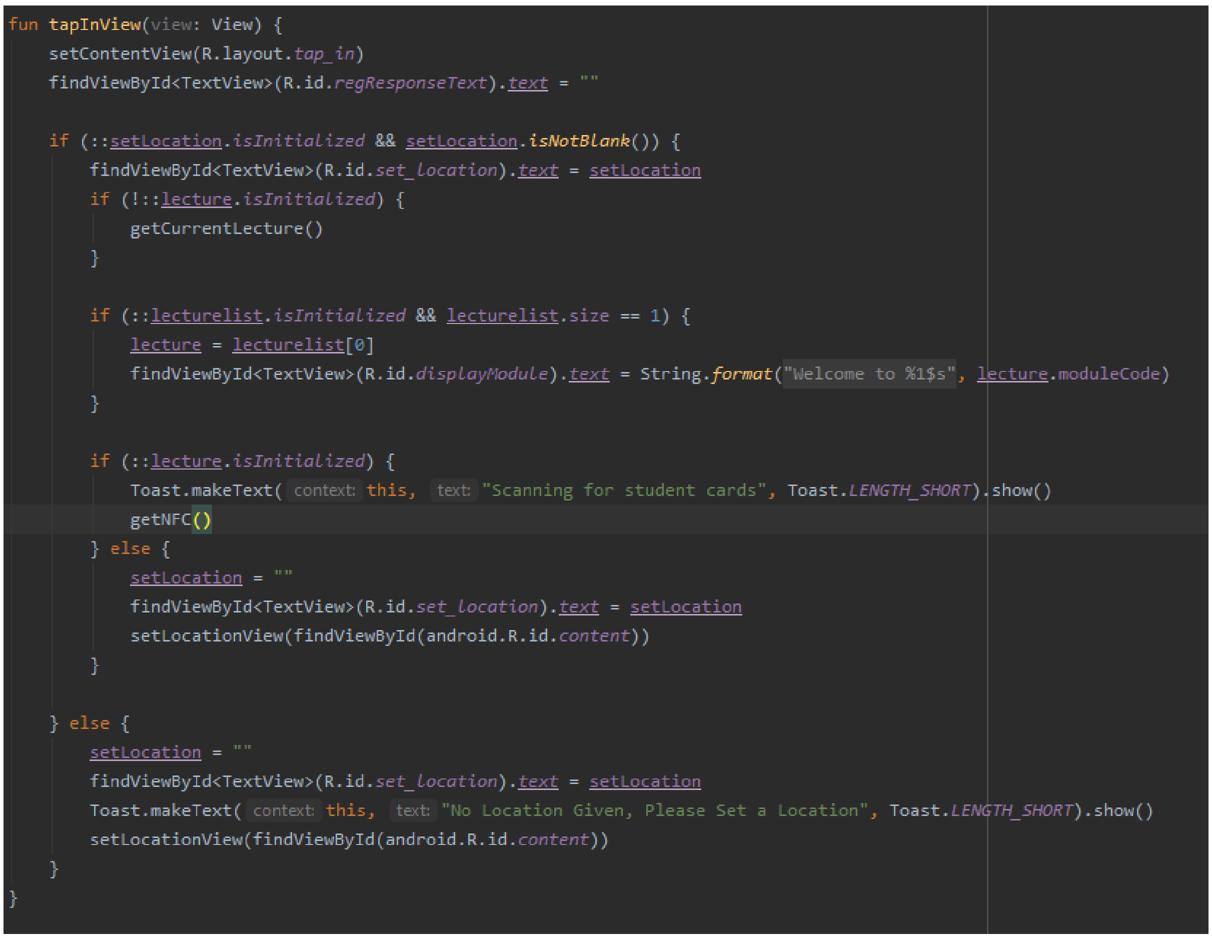
This pattern comes up a lot.

5.D LectureDaoUtils code to gets the next soonest lecture form a list



```
fun getNextLecture(lectureList :List<Lecture>): Lecture? {
    val futureLectures = returnLecturesAfterNow(lectureList)
    return if (!futureLectures.isNullOrEmpty()) {
        var nextLecture = futureLectures[0]
        for (lecture in futureLectures) {
            if (lecture.date.isAfterNow) {
                if (lecture.date.isBefore(nextLecture.date)) {
                    nextLecture = lecture
                }
            }
        }
        nextLecture
    } else {
        null
    }
}
```

5.E Android function to set screen and depending on variables state



```kotlin
fun tapInView(view: View) {
    setContentView(R.layout.tap_in)
    findViewById<TextView>(R.id.regResponseText).text = ""

    if (::setLocation.isInitialized && setLocation.isNotBlank()) {
        findViewById<TextView>(R.id.set_location).text = setLocation
        if (!::lecture.isInitialized) {
            getCurrentLecture()
        }

        if (::lecturelist.isInitialized && lecturelist.size == 1) {
            lecture = lecturelist[0]
            findViewById<TextView>(R.id.displayModule).text = String.format("Welcome to %1$s", lecture.moduleCode)
        }

        if (::lecture.isInitialized) {
            Toast.makeText( context: this,  text: "Scanning for student cards", Toast.LENGTH_SHORT).show()
            getNFC()
        } else {
            setLocation = ""
            findViewById<TextView>(R.id.set_location).text = setLocation
            setLocationView(findViewById(android.R.id.content))
        }

    } else {
        setLocation = ""
        findViewById<TextView>(R.id.set_location).text = setLocation
        Toast.makeText( context: this,  text: "No Location Given, Please Set a Location", Toast.LENGTH_SHORT).show()
        setLocationView(findViewById(android.R.id.content))
    }
}
```

6. Issues Resolved

6.A Axios variable path request

One of the issues we ran into was needing our requests to have variable paths. As we wanted the user to be able to either input their student ID to display lectures or select a course to graph the attendance of. Axios does not have variable paths built in which cause us some problems. The solution to this problem was to use template strings. This allowed us to pass whatever variable we wanted into the url for our requests.

6.B TimeZones and storing DateTime

TimeZones have long been a sore point for developers. Because our project is currently only required to work in Dublin I thought I could avoid working with Time zones and keep all dates in the UTC format, which is correct for Ireland for half the year due to daylight savings time anyway. However I discovered that somewhere between posting a http request that contains a datetime string as part of its JSON object, translating it into a proper joda DateTime object, storing that object in the database and recalling it later, all datetimes were having an extra hour added to them on top of being turned into Irish Standard Time (GMT/UTC +1 hour).

This was only discovered when ad hoc testing datetime comparison functions. The error was that postgreSQL was storing the date as IST(Irish Standard Time) but it does not store any timezone information in the field, so when retrieved it was also acting as if it was translating UTC to IST again. This issue was solved by removing the Z at the end of DateTime Json strings that indicate UTC time, and making the default time zone of the server Europe/Dublin.

6.C Android Multithreading

Before this project I had little to no experience writing android code natively. I found I underestimated the lack of my knowledge, which caused some issues, the worst of which was multithreading problems. While my code was waiting for an NFC card on one thread, it was looping requests to the server in another thread. After refactoring my code with better understanding I got past these issues.

6.D Modeling DCUs student-lecture-module Relationship

The core of the programming logic was based around model DCU’s student-lecture-module relationship. Keeping everything in sync so that each module knows its students and each student knows its modules is challenging. Modelling time based elements to, for example, get a student's next lecture for a certain module, took a lot of functional logic and patience design. I am happy with what I have accomplished here, although there is room for improvement.

6.E Maintaining a Stable Build with Gradle and Intellij IDEA

For various reasons over the course of my work on the project, I ran into build failure issues. There were a few times I had to invalidate intellij’s cache or re import the gradle project. Some of these were due to my intellij project unsyncing with the gradle build, causing very odd looking errors. After refactoring my code structure into the default project structure, and removing my custom pointers to src/main and scr/test, I had less issues. Twice I tried replacing my Gradle build files that use Groovy, with the more modern gradle KTS files that use Kotlin build scripts with Gradle, neither time led to a successful build, although I did learn more about Gradle.

6.F Testing DB

I wanted to use a H2 in memory database for testing my DAO code with a real database, without having to set up a new postgres database. However I was unable to do this as H2 wouldn’t support a uniqueIndex call for one of my table objects. I was unable to resolve this issue and so instead created a full postgreSQl database for testing.

7. Results

While compared to our functional specification our project had to be scaled back, we still managed to explore the technologies that were interesting to use and produce a working product. We both feel like we learnt a lot coming out of this, after using new tools for the first time and after facing issues and overcoming them.

For Eamon, who focused more on frontend development during his internship, getting the chance to build up a backend service from the ground up was a brand new challenge. It was also a great opportunity for learning Kotlin, which is now my favourite language. This project was also the first I spun up an SQL relational database, as in my third year project we used mongoDB, a noSQL database. This was also my first time writing an Android app nativity, as previously I had just used the React Native library for Android.

For Sam, who missed his internship due to covid, this was his first time using the very popular React library for frontend development. Sam also became much more comfortable with git over the course of this project.

8. Future Work

8.A Development

The goal for continuing this project would be to flesh it out with more functionality and give the user more access and control through the UI. A login system that authenticates users with the DCU authentication system, enabling them to edit personal information or allows administrators greater liberties in managing module registration etc.

Once this more natural progression of development was achieved, we would aim to diversify the uses for the system from just university use to also being able to implement it in any desired organisation. An example case of this would be with gyms, as they also provide fitness lessons but these lessons have limited places available so the system could be used to ensure places in classes do not go to waste.

We would also like to develop an accompanying mobile phone app for the system. This would add portability to the system and the mobile device could also be enabled as a substitute ‘tag’ for the student ID.

Another challenge launching this project as a product is the current lack of security. We didn't invest work into encrypting the data stored on the NFC student cards or any of the http traffic, due to time constraints. This leaves our system vulnerable.

8.B Research

Future research in this area would heavily centre on mobile phone applications as there is a noted move towards adding convenience into every aspect of modern society. Security is also an area at the forefront of people’s minds. As such we would like to focus on researching secure ways to implement NFC enabled mobile phones into other areas.