University of Plymouth

School of Engineering, Computing, and Mathematics

COMP3000
Computing Project
2022/2023

Go Phish:

The phishing detection game

Alex Cleverley

Student Number: 10683284

BSc (Hons) Computer Science: Cyber Security

Acknowledgements

Abstract:

This report will describe a mobile application project for a phishing email detection game. There are no plans to develop this app further however an open-source GitHub repository is available should future implementers wish to use the app and implement their own changes.

The report begins with an examination of current cyber-awareness software and highlights the need for an 'on-the-go' style application which this project aims to provide, leading into the project's aims and objectives. Next, the report will discuss the development cycle used throughout the project and highlight the chosen sprint plan approach as well as what was considered for each sprint.

The report will then define the project design elements such as user stories, diagrams, and wireframes. Each design element will additionally show relevance to the project requirements as well as iterative versions as the development cycle continued.

The report will then define the project implementation elements such as the database, API, and mobile application. Each implementation element will highlight key achievements and challenges faced during implementation.

Lastly, a project review is presented which reflects and evaluates which elements of the project went well and what could have been better as well as what potential improvements could be made in the future.

Contents

Statement of Word Count

Word count: 9022

Code Submission

GitHub: https://github.com/AlexCleverley/COMP3000

Introduction:

Phishing cyber-attacks are one of the most common and potentially one of the most dangerous types of attack in the digital space. In 2022, over 80% of identified cyber-attacks in the UK were reported as phishing attacks [1]. The potential impacts of phishing attacks can vary depending on how crafty the attacker is, ranging from financial loss to identity theft. Over 95% of phishing attacks arrive by email, with 1 in every 4,200 emails being a phishing email [2]. Therefore, given this high-profile threat, it is important for all online users to be more cyber aware and be able to spot the signs of a potential phishing scam.

Background

Go Phish provides those who want to be more cyber aware with an intuitive and fun game setting to help them identify the characteristics of a phishing email. The mobile application uses a simple "select what applies" style game, with each level the user plays being randomly selected from the level database. The mobile application has two modes: tutorial mode, for newer users, and play mode. Each level contains an image of a potential phishing email and the user must identify how many signs of a phishing scam are present within the image, returning a score of how many correct signs were identified.

Aims and Objectives:

The main objectives for the Go Phish mobile application are identified below.

Objective	
Allow a User to select either Tutorial or Play modes	Front-End
Randomly select a level image from the database	Back-End
Allow a User to view a randomly selected image	Front-End
Allow a User to select relevant characteristics about the selected image	Front-End
Provide the User with a score for the amount of correctly guessed characteristics	Front-End
In the Tutorial mode, provide examples of each characteristic that the user must identify	Front-End / Back-End

Deliverables:

The final deliverables of the project will consist of a front-end and back-end of a mobile application: A front-end Android studio application, a back-end database to store the levels as well as an interconnecting web API to transfer data between the two elements. All of these elements should successfully implement and fulfil the objectives outlined in the project aims and objectives.

Report Overview:

This report will demonstrate how Go Phish was planned, developed, and implemented.

Firstly, the report will show a literature review, highlighting the main technologies that are used throughout the development of the project and why they were appropriate for the implementation of the project deliverables. Next, the report will discuss the chosen development cycle and identify the sprint plan approach as well as what was considered for each sprint. Next, the report will give regards to the Ethical, Social, and Legal considerations of the project; moving on to also discuss areas of project management.

The report will then define the project design elements such as: requirements, user stories, diagrams, and wireframes. Each design element will additionally show relevance to the project requirements as well as iterative versions as the development cycle continued. The report will then define the implementation of the project deliverables such as the database, API, and mobile application. Each element will highlight key achievements and challenges faced during implementation, relevant to the sprint it was assigned too. Lastly, a project review is presented which reflects and evaluates what elements of the project went well and what could have been better as well as what potential improvements could be made in the future.

Software Review:

Azure Data Studio is "A modern open-source, cross-platform hybrid data analytics tool designed to simplify the data landscape." (Microsoft, [10]). Azure data studio was a proficient tool for the development and implementation of the project's back-end infrastructure. When designing the databases, as shown in the implementation stage later in this report, Azure Data Studio was critical in executing SQL commands and pushing data to the hosted database. Additionally, as shown in the sprint and project management sections, the development of the project was often moved from one computer to another so Azure data studio's ability to be used remotely from the local database helped exponentially in the development of Go Phish.

Visual Studio 2022 is an open-source integrated development environment, IDE, that was a invaluable resource in implementing and testing the Back-end web API. Visual studio allowed for the use of .NET templates, offering Go Phish a web API service to connect the level database to the Android studio mobile application.

Android Studio is an open-source mobile IDE that allows for the creation of android apps and development of android applications. As Go Phish was conceived to be a mobile application, Android studio was chosen to be the core development tool because it is a familiar tool which I had prior experience in using.

Approach Method:

Go Phish was developed with an agile development approach. Agile development encourages an iterative approach to software development which through rapid design develop and testing stages. In the case of the project, agile allowed for dedicated sprints to design and implement the deliverables related to the project objectives.

Sprints were typically done with 2 elements of the project being worked on per week. This was kept on track by the project Gantt chart. Each week, the Gantt chart would be updated to reflect current progress of sprint elements in percentage of completion, allowing for items to be adjusted and refit if one sprint was to take longer than originally expected. This created a dynamic workflow that allowed the project to implement key project objectives early on, leading to a more comprehensive final solution.
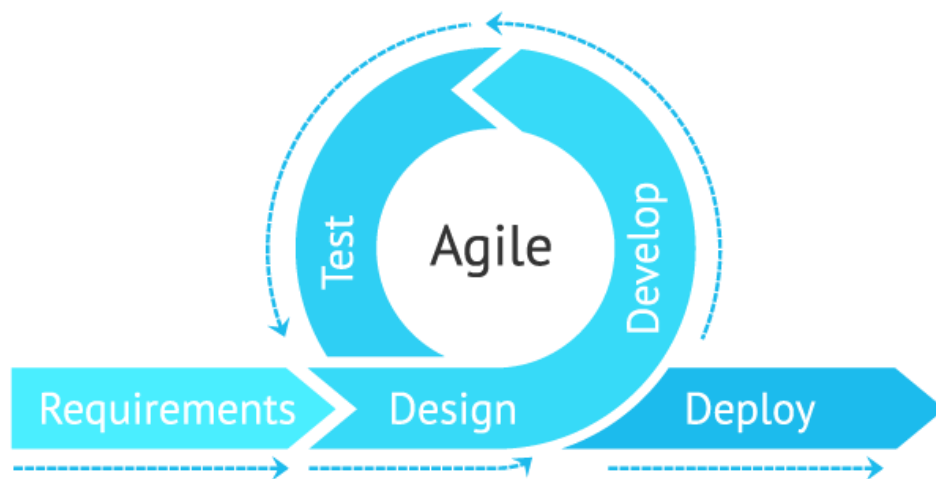
(Agile development model [3])

TeamGantt is an online Gantt chart tool for software development and scheduling. TeamGantt was the primary source of project management during the development of Go Phish. It was used to plan and keep up to date feedback of sprints as well as maintain a consistent working schedule, allowing for smaller sprints to be doubled up and longer sprints to be given more time.

Trello is an online Kanban Board tool which provides the user with a home for processing tasks and to-do lists. During the development of the project, Trello was used to monitor the progress of sprints and the deliverables that corresponded to them. Additionally, Trello was used to maintain a record of the project backlog as well as elements that were Core or Advanced to the development of the project.

Legal, Social, Ethical and Professional Issues:

When Go phish was first conceived, it was crucial that several legal, social, ethical and professional issues that had to be addressed. As the mobile application would allow for user input, a large legal consideration was to protect the app from misuse in regard to the Computer Misuse Act. This act dictates that users should use a computer or computer software for only its intended purpose, and that the computer or software should not be able to maliciously alter the operations of the device [4]. With this in mind it was important to consider how a malicious user might attempt to exploit the mobile application and what steps could be taken to keep Go Phish secure.

Another Legal hurdle was the Copyrights, Designs and Patents Act. This legislation protects unique intellectual property (Ips) and ensures that the author of a property can legally dictate what can and can't be done with their product [5]. As Go Phish is a mobile application, it is important to define how it will stand apart from other mobile apps as well as ensure that the name for the app was not already taken by other parties and was unique.

Throughout Go phish's development, a recurrent Social and Ethical point that had to be addressed was how can Go Phish be intuitive and easy to use. As Go phish is targeted towards people that are less cyber-aware, it was important to test the usability of the app and identify what improvements could be made to the final solution. A notable example of this is development of the tutorial element of the mobile app, allowing a user to understand what types of signs there are and where they would typically be spotted in a phishing email.

Requirements:

These requirements outline the core functionalities as well as the optional functionalities of the project. It was important to establish what features would be necessary rather than optional or additional so that a project prototype could be established, with additional features being added if there was enough development time and resources were available to add them.

| Functional Requirement	Priority
The database should hold a table of levels for the middle and front end to interact with	Required
The database should hold information about each level, such as: Level ID, Level Image, and Level attributes	Required
Level attributes should be in an easy to compute format and all be uniform	Required
The API should be able to connect to the database and host the level table information at a web endpoint for the mobile app to interact with	Required
The mobile app should be able to absorb JSON data from a web API endpoint	Required

The mobile app should be able to select a random level for the user to play	Required
Each random level should have an Image and level attributes; signs for the player to guess	Required
The user must be able to select what attributes apply to the random level via buttons	Required
The mobile app should return a score to the user, based on their correct guesses	Required
The app should appropriate names for training and testing modes. I.e., Tutorial and Play	Required
The app should be able to count down the user so that they have limited time to view the image in the play mode	Optional
The app should be able to recall how well the user did in the previous game, allowing them to view their progression	Optional

| User Requirement	Priority
The user should be able to choose between tutorial or play mode of the game	Required
The user should be able to examine an image by zooming and opening it in full screen view.	Required
The user should be able to select what attributes, signs of a phishing scam, are present through an intuitive system. I.e., buttons	Required
The user must be able to understand what the attribute signs mean.	Required
The user should be able to review the scores of previous games; allowing them to chart their improvements	Optional
In the tutorial mode, the user should be able to look back at the image for a hint	Optional
The user should be able to choose how many levels they would like to do in one session	Optional
The user should be able to choose how many signs they would like to look for per level	Optional

Design:

Design: ERD diagram

Based upon the project requirements and project aims and objectives, an Entity Relationship Diagram was to be created in order to outline how the deliverables of the project would communicate with each other. The Entity relationship Diagram went through progressive iterations as the project continued, acting as foundation which the project elements would later be implemented from.
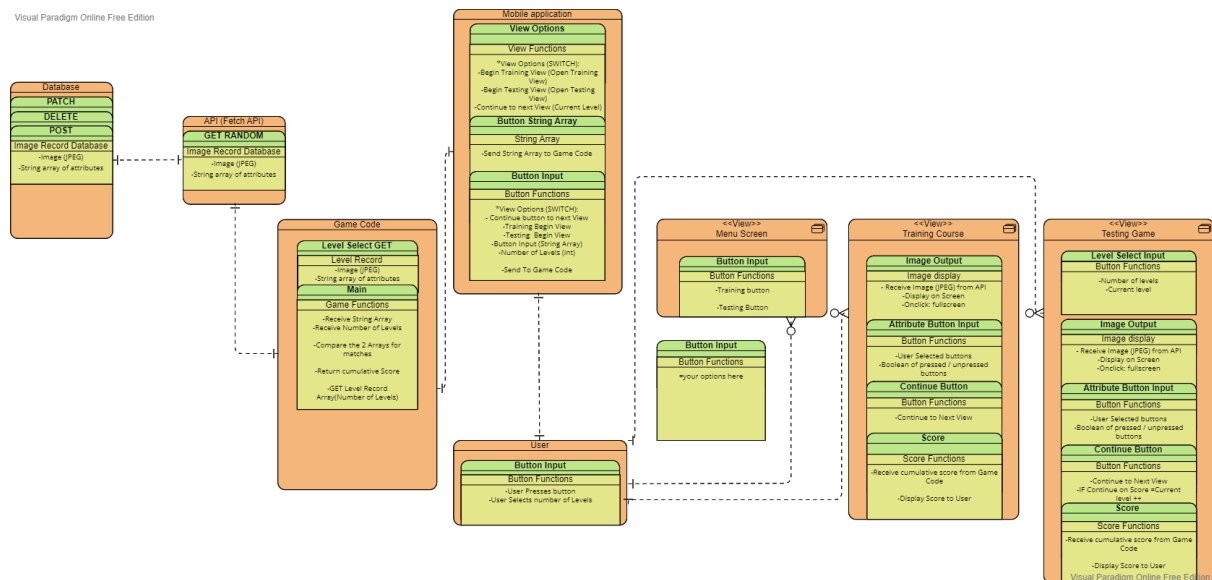
(ERD Diagram V1, Alex Cleverley [6])

The first version of the ERD outlined the core basis of how each project deliverable would communicate to one another. There was one database with one data table that connected to one web API service that would host a random data table value as JSON. This would connect to the game code aspect of the mobile app: able to use the random level to display the level's image and keep a local variable of the level's attributes that would later be compared against the player's own input to return a score. The Game code would connect to a Mobile application front end that controlled how the user was able to navigate through different views and modes of the app. The interactivity was done through buttons which made it important for the ERD to distinguish between view-based buttons and button inputs for the guessing element of the mobile app.

As the project development continued, the ERD was held back by technical fallbacks in the software's that were used for the project. For example, the mobile application entity had to be injected and customised into each view, allowing view control to branch off from one another in similar nature to the courses that were laid out in the ERD. Nevertheless, it became clear that the ERD was outdated and in order for the design to reflect the technical hurdles faced during development, a 2nd ERD version was made.
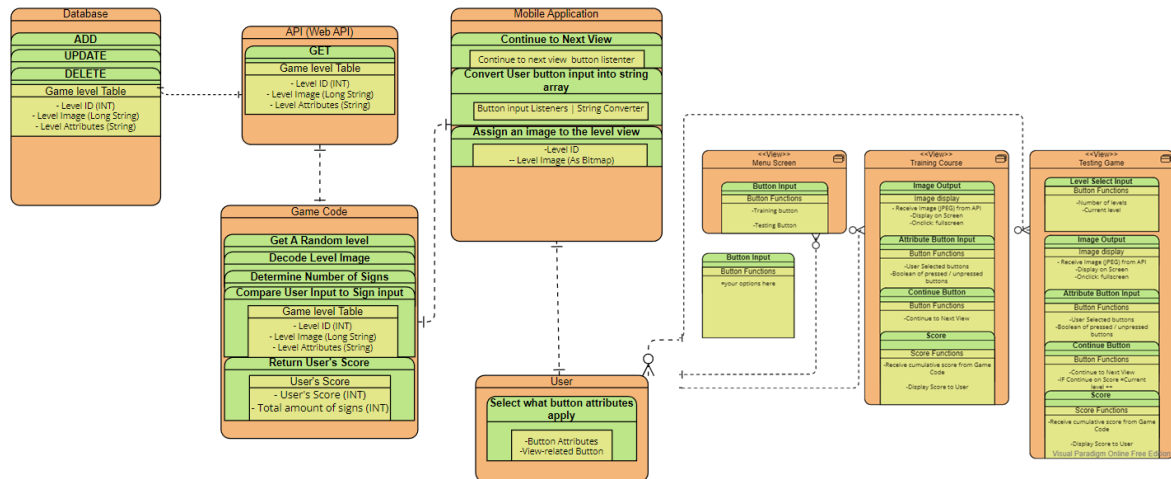
(ERD v2, Alex Cleverley [7])

The 2nd iteration of the ERD adjusted the back-end areas of the project. For example, the selecting a random level function was now a part of the game Code entity. This was because the API lacked the capabilities to select a random level, whereas the java coded game code allowed for capturing the entire JSON list and pulling one record at random. This new ERD better reflected how each of the components fit together, with the views of the training (tutorial) and testing (play) modes being kept, for the most part, the same.

Design: UML diagram

In order to further conceptualise how the end user would interact with the app, a UML diagram was constructed. In the figure below, the UML diagram demonstrates what the use would see in terms of the current view and view buttons. beyond that, within the Mobile app's class code, a button click listener catches when the user presses the button. Depending on the button id, the buttons can continue to the next view, affecting the current view, or convert the button inputs to string attributes, which passes the string attributes to the game code class in order to return a score.
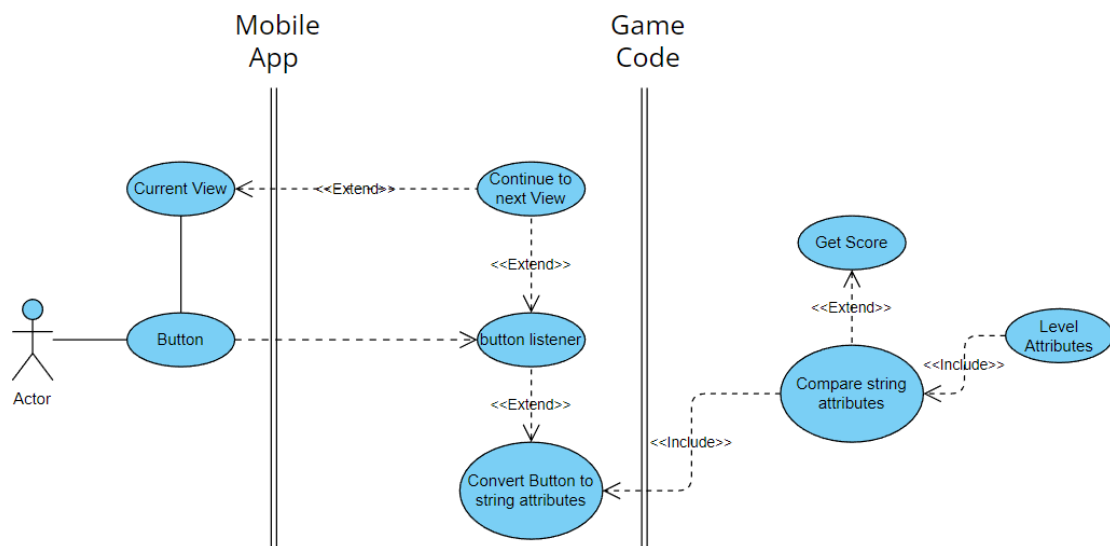
(UML Button diagram, [8])

Design: Wireframes

Building upon both the project requirements as well as the ERD and UML diagrams, Go Phish needed wireframes of the app's views in order to test usability as well as define a uniform layout and identity for the mobile application. Below is an image of the 2nd iteration of wireframes, following improvements and suggestions from usability testing found later in this report. The Wireframes define the view layout of both the Tutorial and Play game modes with application views for: looking at the level image, guessing the signs of the level image and a score page giving you the option to return to menu or, in the play mode, play a new random level.
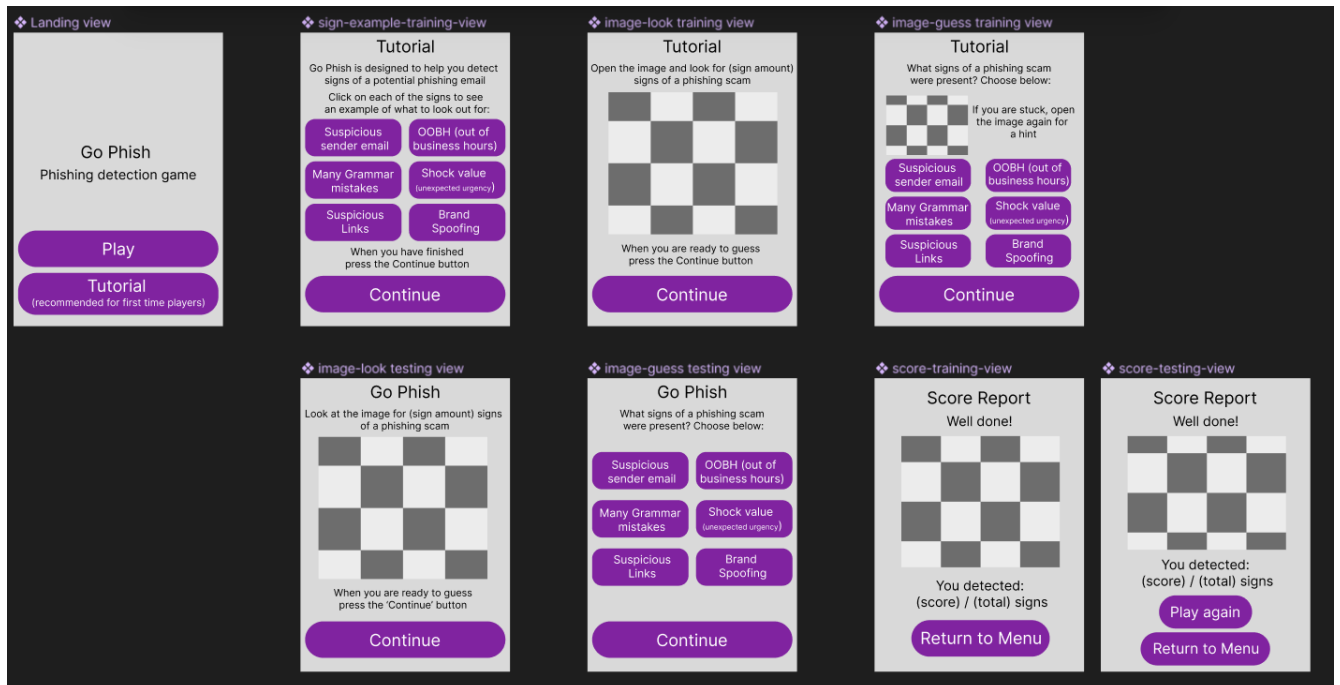
(Project Wireframes, Comp 3000 GitHub)

Implementation:

Implementation: Database

The Implementation of the database was broken up into 2 sprints over the course of December 2022. The first sprint primarily focused creating a database schema and figuring out how the record id, image and attributes should be stored on the database. The second sprint consisted of implementing the schema to the database using Azure Data studio. Over the course of December 2022 I was working remotely at home, so using services such as Azure to remotely connect to the database was a necessary element during this time.

The first sprint towards implementing the database was, in reality, more design documentation. In order to save an image and attributes to the database both of the elements needed to be given a data type. The record ID was an integer and couldn't be an empty value. The record attributes were to be saved as a short string, known in MySQL as a VARCHAR this also was not null as it couldn't be an empty value. The record image was, at first, saved as a BLOB image datatype; however, after learning that the image could not be uploaded remotely, was reconfigured to be a medium sized string value after being encoded into Base 64.

```
Create table [ImageRecordDatabase].[ImageRecord] (

    RecordID INT NOT NULL,

    RecordImage MEDIUM TEXT NOT NULL,

    RecordAttributes VARCHAR (255) NOT NULL);


RecordAttributes: (an array of Boolean variables into a string separated by commas)

    -   Suspicious sender email
    -   OOBH (out of business hours)
    -   Frequent Grammar mistakes
    -   Shock value (unexpected urgency)
    -   Suspicious Links
    -   Brand Spoofing

RecordAttributes Example: "1,0,0,1,0,1"

RecordImage Encoder: https://www.base64encode.org/
```

(Database design documentation v2, GitHub)

Above is the 2nd version of the database design documentation; complete with the code to initialize the data table as well as an example of how the attributes would be configured using ones and zeroes to signify what attribute was true or not. This documentation set the groundwork for how the mobile app would later check for the number of signs per level as well as be able to compare what attributes the user had guessed correctly against the level attributes.

With the design documentation completed, the 2nd sprint was focused towards creating and filling the data table in azure data studio. Using the SQL code above, the table was produced and now needed to be filled. The record images, that had been screenshotted as PNG's were encoded into a Base 64 string and added to the table alongside the record attributes using the INSERT INTO SQL command. Each record in the table needed to have two versions; one that had a image with no hints and another that did have hints. This meant that, in the record table, there were always 2 records with the same attributes.

| recordID	recordImage	recordAttributes
1	wolQTkcNChoKAAAADUlIRFIAAAZC…	0,0,0,1,0,0
2	wolQTkcNChoKAAAADUlIRFIAAAbD…	0,0,0,1,0,0
3	w6tQTkcNChoKAAAADUlIRFIAAAY5…	0,0,0,0,1,0
4	w6tQTkcNChoKAAAADUlIRFIAAAbD…	0,0,0,0,1,0
5	wolQTkcNChoKAAAADUlIRFIAAAY/…	1,0,0,0,0,0
6	w6tQTkcNChoKAAAADUlIRFIAAAbi…	1,0,0,0,0,0
7	w6tQTkcNChoKAAAADUlIRFIAAAPD…	0,0,1,0,0,0
8	w6tQTkcNChoKAAAADUlIRFIAAAxl…	0,0,1,0,0,0
9	w6tQTkcNChoKAAAADUlIRFIAAAPD…	0,0,0,0,0,1
10	w6tQTkcNChoKAAAADUlIRFIAAAxi…	0,0,0,0,0,1
11	wolQTkcNChoKAAAADUlIRFIAAAY8…	0,1,0,0,0,0
12	wolQTkcNChoKAAAADUlIRFIAABEw…	0,1,0,0,0,0

(Database screenshot, Azure data studio)

Implementation: API

The implementation of the API was done throughout multiple two week-long sprints over the course of January and into the starting week of February 2023. The first sprint was centred around producing and establishing a connection between the API and the database, essentially implementing the solution. Whist the second sprint was centred towards testing the connection to see if the solution worked and could host the database table in a JSON format as per the project requirements and the ERD design.
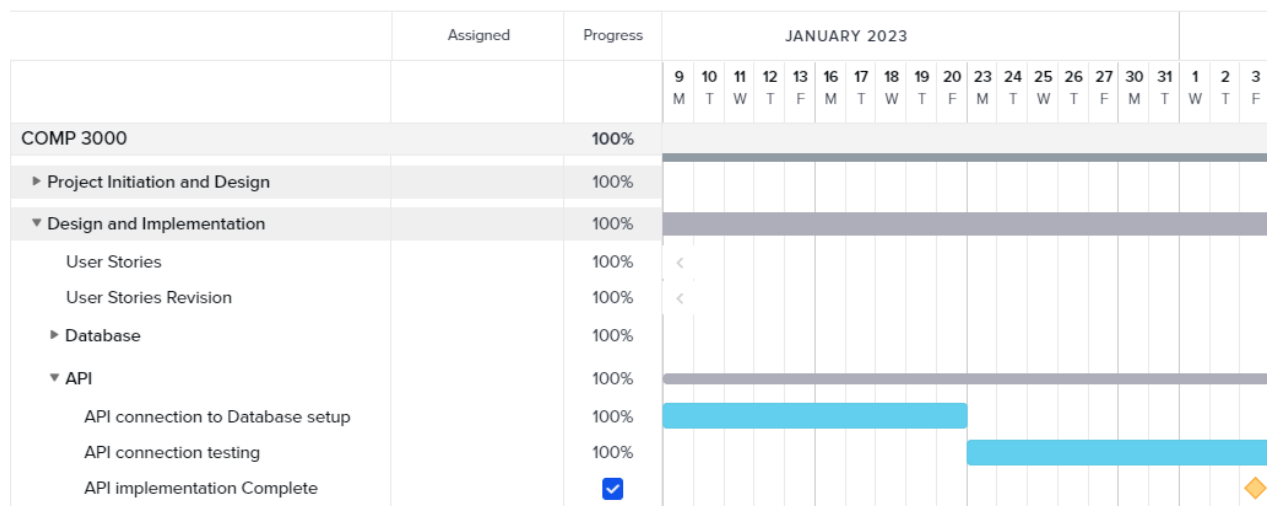

(Screenshot of Gantt Chart, [9])

Within the Software Review section of this report, it is mentioned that Visual studio was used for the benefit of having .NET templates that were used to implement to API solution. When opening Visual Studio, the IDE recommends a variety of templates including the ASP.NET Core Web API template as shown below.
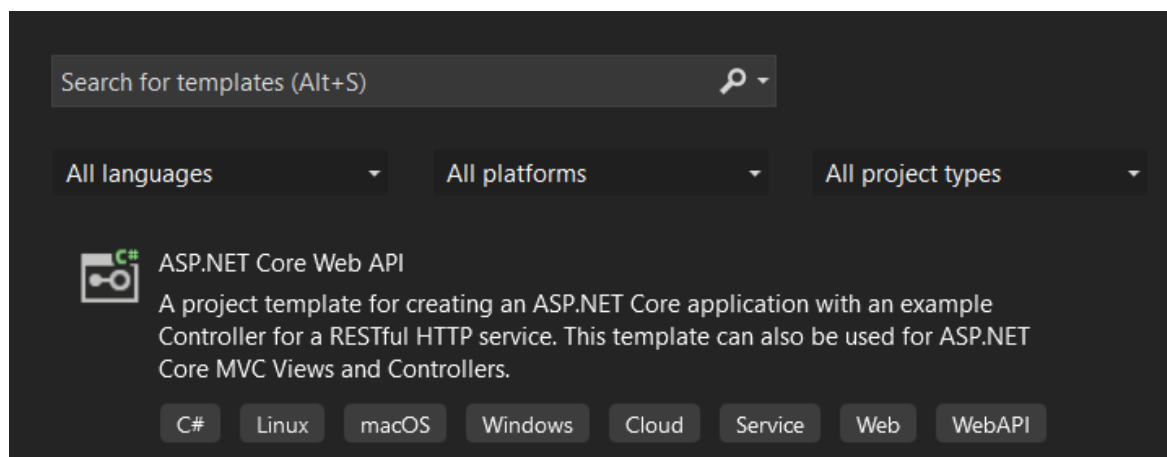
The API implementation was broken up into a variety of smaller components that each worked together to perform the requirement of hosting the database data in JSON on a web endpoint. Below is evidence for all of the folders and files used to create the web API. With each element given more detail later on in this section of the report.
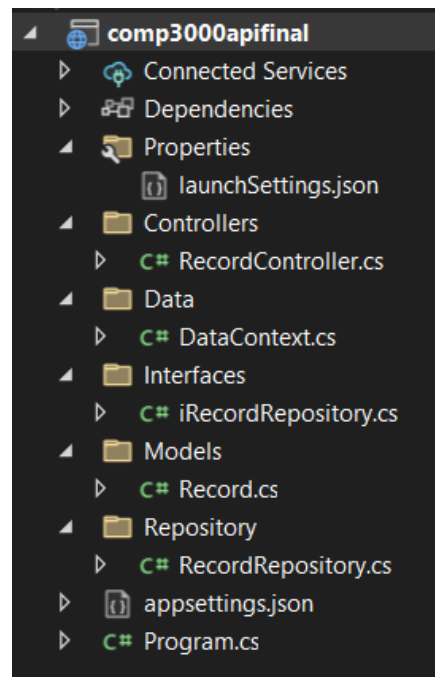
In order to begin implementing the web API solution, additional dependencies needed to be added to the codebase of the web API. Microsoft's Entity Framework dependencies allowed the web API to use entity framework tools and code libraries not included within the base Visual Studio. Additionally, given that the database was in MySQL, "MySqlConnector" and "Polmelo.EntityFrameworkCore.MySql" were used in order to allow connectivity to the MySQL project database from the API. Lastly, the "Swashbuckle" dependency was used to utilize Swagger, an API debugging tool, that helped in the testing sprint of the API implementation.
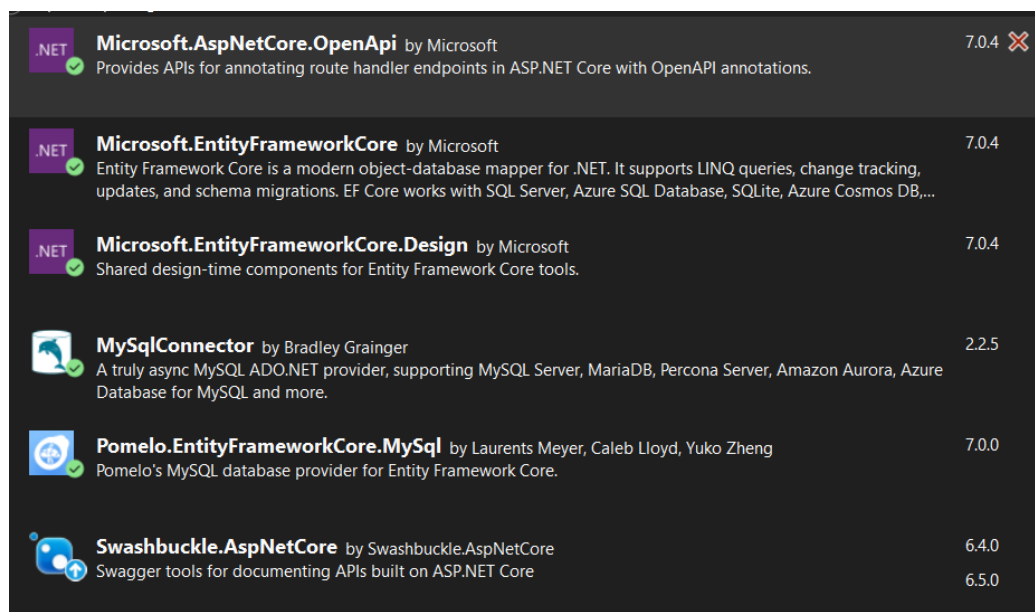
In order to connect the database to the API, a connection string had to be used. A connection string is a collection of key value pair information that provides the API with enough credentials to access the database. For example, the server part of the connection string identifies the what the URL is to connect to the database: which, in this case, is "proj-mysql.uopnet.plymouth.ac.uk". The connection string is included within the appsettings.json file of the API solution.

```
"ConnectionStrings": {
  "RecordDB": "server=proj-mysql.uopnet.plymouth.ac.uk;port=3306;database=COMP3000_ACleverley;user id=COMP3000_ACleverley;password=KlaK351+;"
},
```

In order to reference the data of the database in our API, a model needed to be created that matched the column names of the database table. This provides the API with a model of what data to expect from the database which, in the case of our project, is: a integer for record ID, a string for the encoded record image, and another string for the list of attributes related to the level. This model would be inherited later on into the Data context class, allowing each database entry to be saved as in a JSON list.

```csharp
namespace comp3000apifinal.Models
{
    0 references
    public class Record
    {
        0 references
        public int recordID {  get; set; }
        0 references
        public string recordImage { get; set; }
        0 references
        public string recordAttributes { get; set; }
    }
}
```

The DataContext class is the primary medium for handling the data from the MySQL database and storing it as described in the Record model. DataContext inherits the DbContext class which is provided by Microsoft's Entity Framework Core and allows DataContext to use the base options outlined by DbContext with later options being added in the Program.cs file. In Addition to this, the DataContext class also uses a public DbSet to absorb the information from the database. It uses the Record model created earlier and accesses the database looking for the listed table which, in the case of the project, is imagerecordtable.

```
using comp3000apifinal.Models;
using Microsoft.EntityFrameworkCore;

namespace comp3000apifinal.Data
{
    5 references
    public class DataContext : DbContext
    {
        0 references
        public DataContext(DbContextOptions<DataContext> options) : base(options)
        {

        }

        1 reference
        public DbSet<Record> imagerecordtable { get; set; }
    }
}
```

Now that the API was reading in the data from imagerecordtable, it had to output the data in a JSON format as per the requirements. In order to do this, a interfaced named iRecordRepository was created that ran a command named GetRecords. This interface would allow us to run the GetRecords function but first it needed to be created; a task that the Repository class would undertake.

```
using comp3000apifinal.Models;

namespace comp3000apifinal.Interfaces
{
    4 references
    public interface iRecordRepository
    {
        2 references
        ICollection<Record> GetRecords();
    }
}
```

The Repository class inherits the iRecordRespository interface and also creates a DataContext object named _context. These are both used in the GetRecords function which returns a contextualised list of the imagerecordtable in order of recordID. This function now allowed us to retrieve the data however the function had to be called on by the Controller class.

```csharp
public class RecordRepository : iRecordRepository
{
    private readonly DataContext _context;
    public RecordRepository(DataContext context)
    {
        this._context = context;
    }

    public ICollection<Record> GetRecords()
    {
        return _context.imagerecordtable.OrderBy(p => p.recordID).ToList();
    }
}
```

The RecordController class is defined in 2 main components: the [Route], [ApiController], and [HttpGet] attributes that allow the Controller class to be used as an API controller, as well as the action result which allows the API to run the GetRecords function and check to see if the model state is valid. The controller attributes are apart of Microsoft's ASP Net Core dependency and inform the API that this class is the controller class; also having RecordController inherit from ControllerBase. The action result allows the controller to call and run the GetRecords function which, in turn, calls on the interface that was created earlier. The action result checks to see if the list is the same model state as the Record model, returning a bad request if the model state is not valid else returning the var records as seen below.

```csharp
using comp3000apifinal.Interfaces;
using comp3000apifinal.Models;
using Microsoft.AspNetCore.Mvc;

namespace comp3000apifinal.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class RecordController : ControllerBase
    {
        private readonly iRecordRepository _recordRepository;
        public RecordController(iRecordRepository recordRepository)
        {
            _recordRepository = recordRepository;
        }

        [HttpGet]
        [ProducesResponseType(200, Type = typeof(ICollection<Record>))]
        public IActionResult GetRecords()
        {
            var records = _recordRepository.GetRecords();

            if(!ModelState.IsValid)
                return BadRequest(ModelState);

            return Ok(records);
        }
    }
}
```

With all the prior elements and files in place, the Program file needed to be configured so that the API would recognise the Repository, Interface, and Data context. These were included through the use of builder services that came with the API .NET template. In order to add the Repository and Interface the AddScoped command was used which informed the API that the iRecordRepository interface should be linked to the RecordRepository repository. Next, the builder added the DataContext base options that were defined in the DataContext file. In addition to the base options, the connection string needed to be paired to the DataContext file, so a new MySqlConnection was used to define the connection string as well as defining the MySqlServerVersion which culminated in the options.UseMySql option being used, allowing the MySQL database to be connected to the API through the connection string.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.AddScoped<iRecordRepository, RecordRepository>();
builder.Services.AddDbContext<DataContext>(options =>
{
    var connection = new MySqlConnection(builder.Configuration.GetConnectionString("RecordDB"));
    var serverVersion = new MySqlServerVersion(new Version(10, 4, 27));
    options.UseMySql(connection, serverVersion);
});

var app = builder.Build();
```

With all of the elements of the first sprint now completed, the next sprint was to utilise Swagger and test if the API would return the data in a desired format. In the image below, swagger was used to execute the data retrieval which returned the imagerecordtable data in a JSON format. The testing was a success.
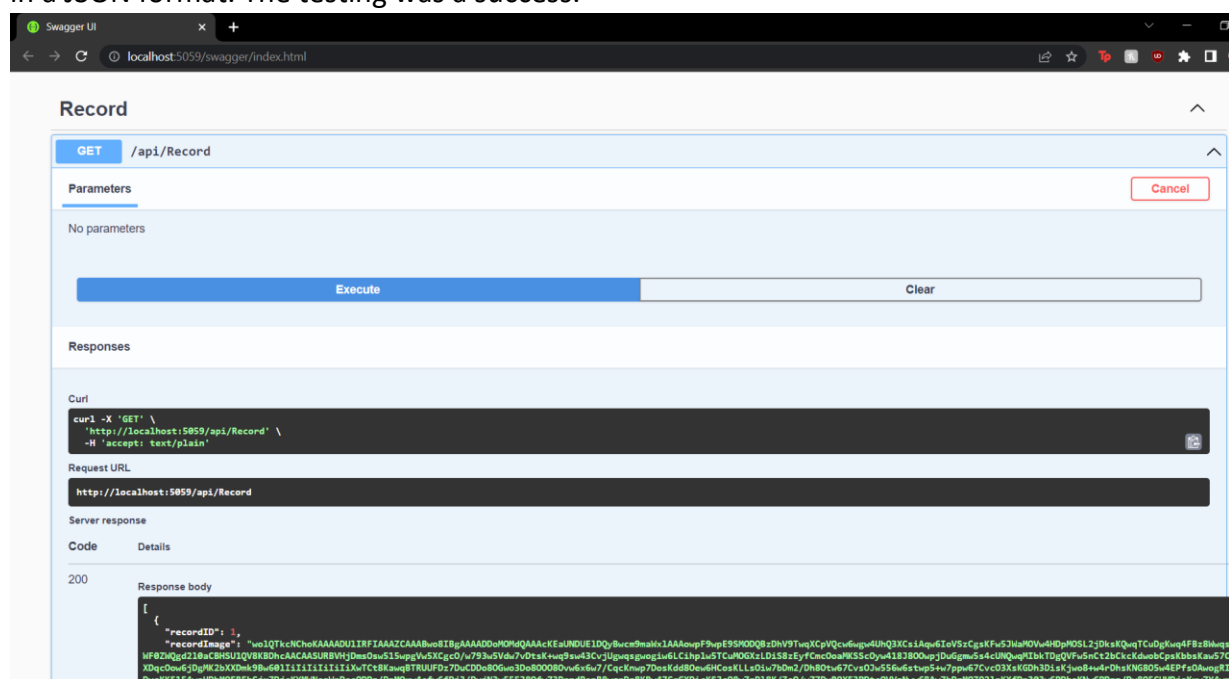
Implementation: Game Code

The java game code was planned and later implemented in two separate sprints; a pseudocode sprint that ran alongside other design elements of the project, and a 5 week-long sprint alongside the implementation of the mobile application. During the prolonged sprint, the views of the mobile application were made, and the java game code was implemented alongside it; in keeping with the agile model of rapid development and testing, each view was coded and tested against the game code for errors, allowing the project to be in the prototype stage in a short number of sprints.
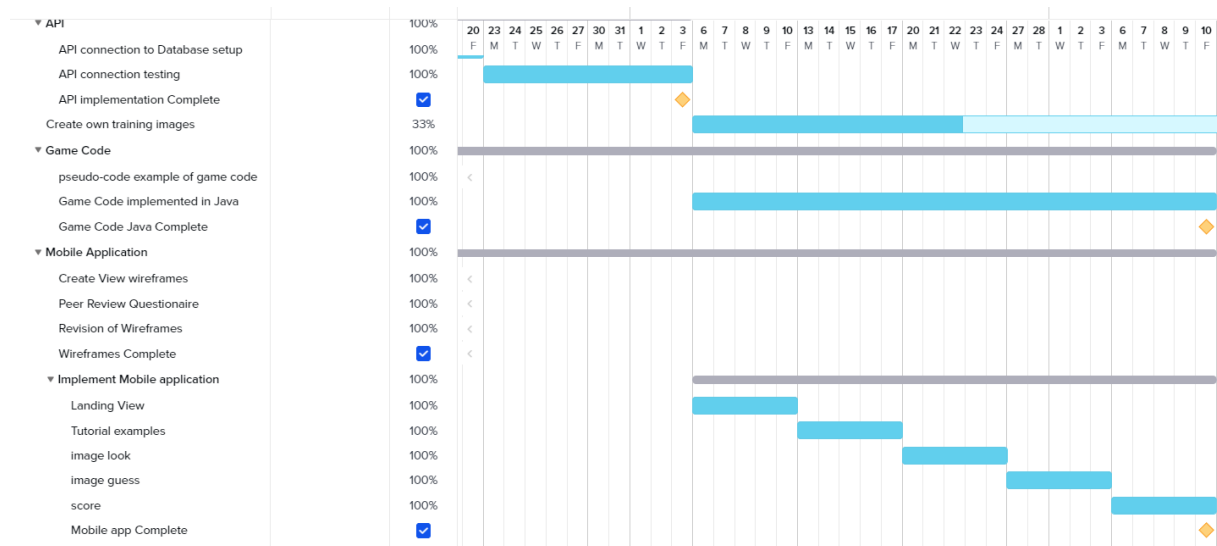
(Screenshot of Gantt Chart, [9])

The game code needed to be able to perform a variety of tasks. Based upon the ERD and project requirements, as well as technical requirements of android studio, the game code needed to:

-   Access the Web API
-   Make a local copy of the JSON data from the API
-   Store the local copy in a format that allowed for searching (JSON Object Array)
-   A function that allowed for searching a specific Image
-   A function that counted up the number of signs in a level
-   A function that kept track of what random level the user was on
-   A function that selected a random level
-   A function that counted up the users score on a level
-   A function that returned the users score back in a readable format

With those requirements in mind, the first step in implementing the game code was to implement a function that could be called at when the app first opens which accessed the API and read in the JSON data values.

The AccessAPI function uses a OkHttp request builder to check an active connection or, in this particular case, the web API connection. When a successful link is established, the OkHttp client then tries a call request to get the JSON data from the API, saving it locally as JsonData. This try catch also checks for Input or Output exceptions alerting the app if it the JSON retrieval was not successful.

```java
//Access the API record using OkHttp
1 usage
public static void AccessAPI() {

    OkHttpClient client = new OkHttpClient();
    String url = "";
    String jsonData = null;

    //create new OkHttp request (Get a URL example on the Okhttp website)
    Request request = new Request.Builder()
            .url(url)
            .build();
    //If successful response, store Json array into String for conversion
    try {
        Response response = client.newCall(request).execute();
        if (response.isSuccessful()) {
            jsonData = response.body().string();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Once the data retrieval from the API is complete, the next step was to save the data in a format for indexing and searching later in the app. As the variable JsonData is only stored as a string, the function needed to convert the data back into a JSON Object Array for querying. This was done in the following code below:

```java
//convert jsonData String into a new list of Records
records = new ArrayList<>();
try {

    //create new Json array object with jsonData
    JSONArray jsonRecords = new JSONArray(jsonData);

    //for each record in the json array, copy object values for list
    for (int i = 0; i < jsonRecords.length(); i++) {
        JSONObject jsonRecord = jsonRecords.getJSONObject(i);
        int recordId = jsonRecord.getInt( name: "recordID");

        //get the string of recordImage in jsonRecord then convert to byte[] array
        String recordImageText = jsonRecord.getString( name: "recordImage");
        byte[] recordImage = Base64.decode(recordImageText, Base64.DEFAULT);

        String recordAttributes = jsonRecord.getString( name: "recordAttributes");

        //add record to arraylist
        Record record = new Record(recordId, recordImage, recordAttributes);
        records.add(record);

    }
} catch (JSONException e) {
    e.printStackTrace();
}
```

The Json data was being copied to a JSON object called Records. Records consisted of an integer named "recordID", a byte array of the level image encoded into Base 64 named "recordImage", and lastly a string of record attributes named "recordAttributes". In order to store these three values in a single record, a record JSON object was created which was defined in the Record class as shown below:

```java
public class Record {
    3 usages
    private int recordID;
    3 usages
    private byte[] recordImage;
    3 usages
    private String recordAttributes;

    1 usage
    public Record(int recordID, byte[] recordImage, String recordAttributes) {
        this.recordID = recordID;
        this.recordImage = recordImage;
        this.recordAttributes = recordAttributes;
    }

    2 usages
    public int getRecordID() { return  recordID; }

    public void setRecordID(int recordID) { this.recordID = recordID; }

    1 usage
    public byte[] getRecordImage() { return recordImage; }

    public void setRecordImage(byte[] recordImage) { this.recordImage = recordImage; }

    1 usage
    public String getRecordAttributes() { return recordAttributes; }

    public void setRecordAttributes(String recordAttributes) {
        this.recordAttributes = recordAttributes;
    }
}
```

The get and set variations of recordID, recordImage and recordAttributes allowed the Record class to be used in conjunction with the conversion of JsonData code within the game code class. This resulted in being able to store a Record object of all the data within the database, now able to be queried and searched through. With that in place, the game code was ready to be implemented alongside the sprints outlined for the mobile application.

Implementation: Mobile application

The implementation of the mobile application views was done in one overarching sprint goal that was broken up into 5 weekly sprints. Below is a screenshot from the project Gantt chart showcasing how the development of the mobile application was planned out.
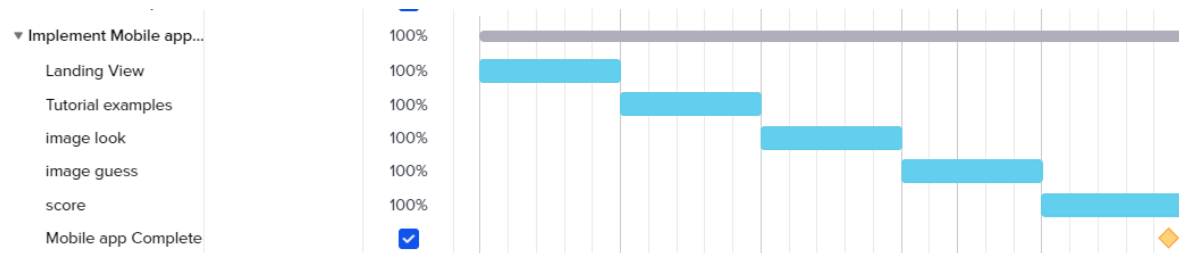

(Screenshot of Gantt Chart, [9])

The 5 weekly sprints each addressed a certain view within the mobile application. Based upon the needs of each sprint, a list of requirements for each view was set up, based upon the overall project requirements as well as the design wireframes and UML and ERD diagrams.

Sprint	Requirements
Landing View	Button links to Play and Tutorial views, create layout and java class pages for future sprints.
Tutorial Examples	Button links that displayed specific images from the database, continue button to go to tutorial image look view.
Image Look	Split up into both Play and Tutorial modes. Tutorial: Display random image, allow image click for Fullscreen, Display number of signs to look for, continue button to tutorial image guess. Play: Display random image, allow image click for Fullscreen, Display number of signs to look for, continue button to play image guess.
Image Guess	Split up into both Play and Tutorial modes. Tutorial: Display thumbnail image of hint image, allow image click for Fullscreen, Button options to select what signs / attributes apply to the image, continue button to score. Play: Button options to select what signs / attributes apply to the image, continue button to score.
Score	Split up into both Play and Tutorial modes. Tutorial: Display number of signs guessed / number of total signs, display level image, return to landing view button. Play: Display number of signs guessed / number of total signs, display level image, return to landing view button, play again button.

When beginning implementation for the project's mobile application, the first step was to create the relevant class and layout files that would be needed for the app to function. Each view of the app needed one layout file and one Java Class file: Layout files were used to outline the visual design of the view whilst the Java Class file is used to program the interactivity of the view.
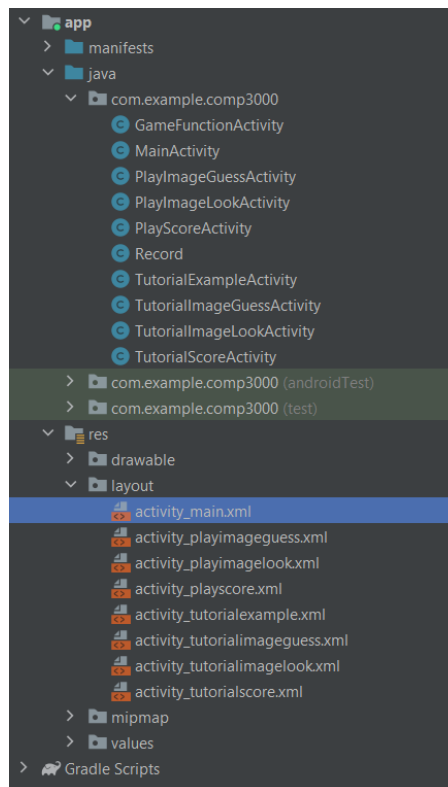
The Layout file for the Landing view included two buttons: one for the Play mode and the other for the Tutorial mode. The Tutorial mode also had additional text, recommending the mode to first time users, based upon feedback from the wireframe designs. Each element of the layout had to be constrained so that its place on screen was uniform throughout any device. The buttons are constrained to the parent edges of the screen offset by different amounts for the play and tutorial buttons respectively.
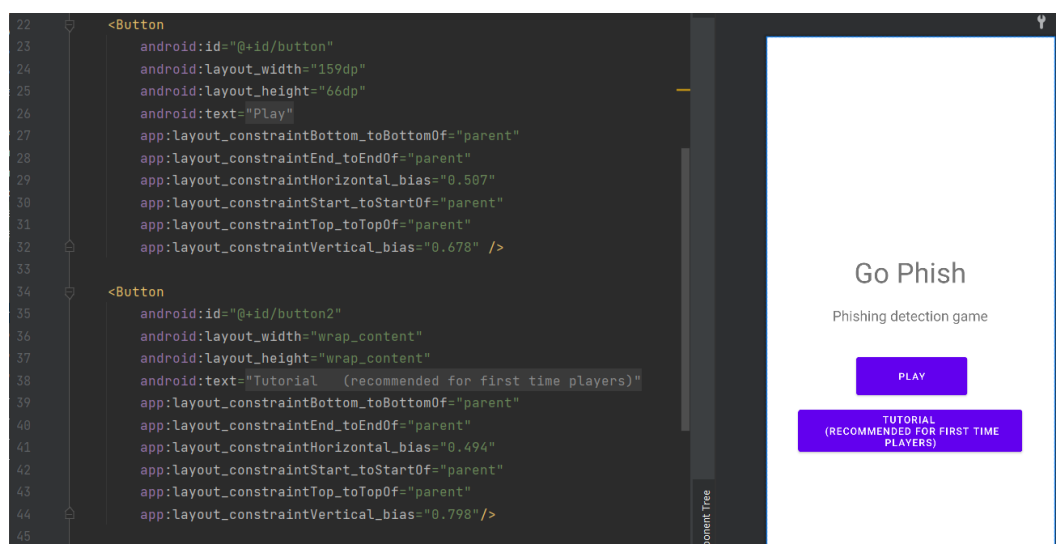
The Java Activity Class file was used to program the interactivity of the buttons through button listeners, linked to the button's ID. The Class file calls the AccessAPI function from the game code and then, based upon which button is clicked, redirects the user to a different function, taking them to the relevant view.

```java
public class MainActivity extends AppCompatActivity {
    2 usages
    private Button playbutton;
    2 usages
    private Button tutorialbutton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //run the accessAPI function in the game function activity
        GameFunctionActivity.AccessAPI();

        playbutton = (Button) findViewById(R.id.button);
        tutorialbutton = (Button) findViewById(R.id.button2);

        playbutton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) { openPlayImageGuessActivity(); }
        });

        tutorialbutton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) { openTutorialExampleActivity(); }
        });
    }
```

The separate functions call an Intent, allowing the android application to navigate to the relevant class. For the Play button it navigates to PlayImageLookActivity whilst for the Tutorial button, it navigates to TutorialExampleActivity. With interactivity of the buttons completed, the next stage was to begin the Tutorial Example view sprint.

```java
    1 usage
    public void openPlayImageGuessActivity() {
        Intent intent = new Intent( packageContext: this, PlayImageLookActivity.class);
        startActivity(intent);
    }

    1 usage
    public void openTutorialExampleActivity() {
        Intent intent = new Intent( packageContext: this, TutorialExampleActivity.class);
        startActivity(intent);
    }
}
```

The Tutorial Example view is a unique view as its purpose is to display examples of the signs a user can choose from. Based upon this sprint's requirements, the tutorial example view needs to access specific images from the database to showcase each example and, by extension, needs to take the image from the database and host it in a imageview for the user to see.
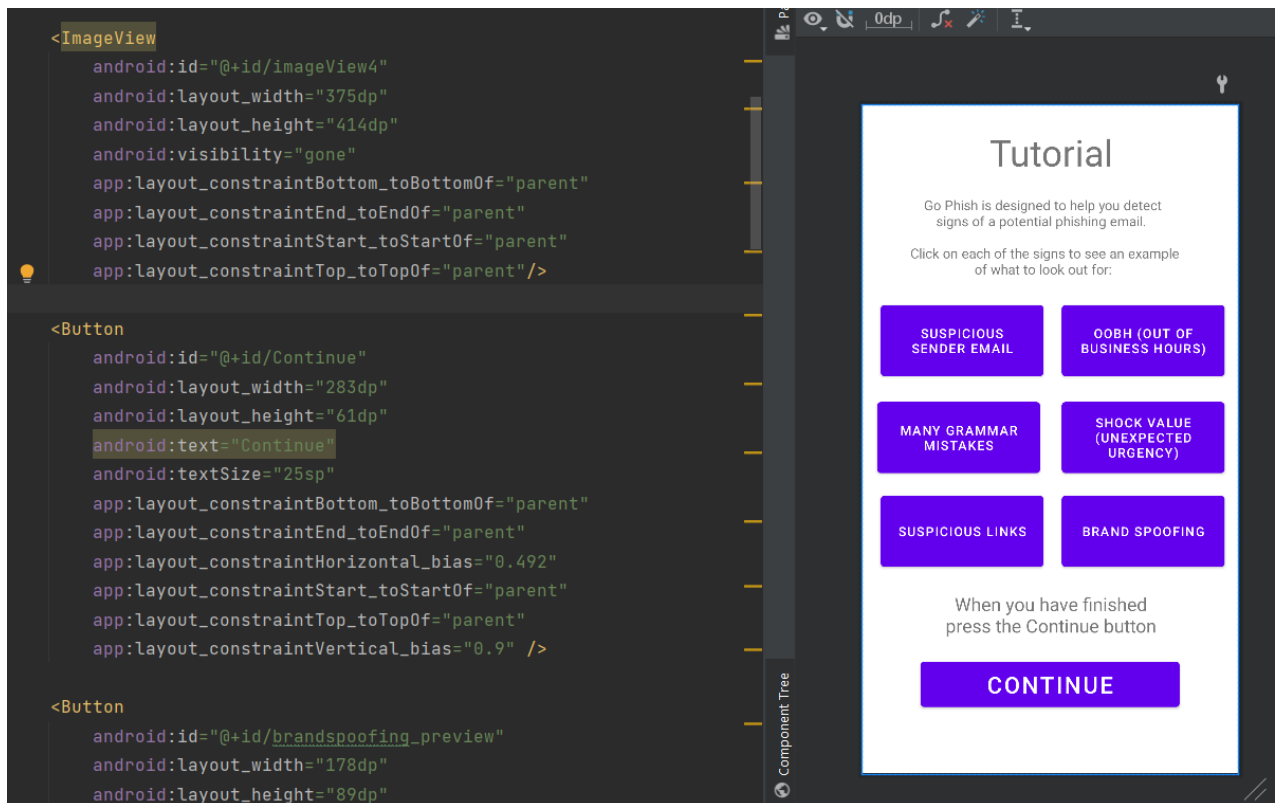
In the Layout file for the tutorial examples, there are a number of buttons that are each tailored to a specific example of phishing sign. In order to display the image for each example, a hidden ImageView using the setting *android:visibility="gone"* which would allow the image to appear after being selected with a button.

In the Tutorial Example Activity Class, each button and imageview needed to be identified and given their own variable name in order to program them in the java class. The button listeners for each of the example signs all were configured the same way: On the button being clicked, run the GetImageBitmap function from the game code class with a corresponding id number unique to each example sign. The function would return a bitmap value of the image stored at the id that was listed, allowing the bitmap to be set to the imageview and then made visible.

```java
Continue = (Button) findViewById(R.id.Continue);
brandspoofing_preview = (Button) findViewById(R.id.brandspoofing_preview);
suspicousemail_preview = (Button) findViewById(R.id.suspicousemail_preview);
oobh_preview = (Button) findViewById(R.id.oobh_preview);
grammarmistakes_preview = (Button) findViewById(R.id.grammarmistakes_preview);
shockvalue_preview = (Button) findViewById(R.id.shockvalue_preview);
suspicouslinks_preview = (Button) findViewById(R.id.suspicouslinks_preview);
imageView = (ImageView) findViewById(R.id.imageView4);


Continue.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) { openTutorialLook(); }
});
brandspoofing_preview.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        int id = 10;
        Bitmap bitmap = GameFunctionActivity.GetImageBitmap(id);

        imageView.setImageBitmap(bitmap);
        imageView.setVisibility(View.VISIBLE);
    }
});
```

```java
public static Bitmap GetImageBitmap(int id) {
    //A byteArray variable for ArrayList recordImage to be copied into
    byte[] imageData = new byte[0];

    //for each record in the arraylist
    for(Record record : records) {
        //if the recordID matches, copy the image byte array
        if(record.getRecordID() == id) {
            imageData = record.getRecordImage();
        }
    }

    //convert the byte array into a bitmap (BitmapFactory method in Android studio)
    Bitmap bitmap = BitmapFactory.decodeByteArray(imageData, offset: 0, imageData.length);
    //return bitmap to imageview
    return bitmap;
}
```

With the previous two sprints completed, and with the code being created in a format suitable for agile development, the Tutorial and Play image look views were next to be implemented. This sprint would use much of the same coding principles as the previous two sprints, utilizing an imageview which would be assigned a bitmap value by the relevant activity class. However, this sprint introduced a new challenge; how to calculate the number of signs per level.
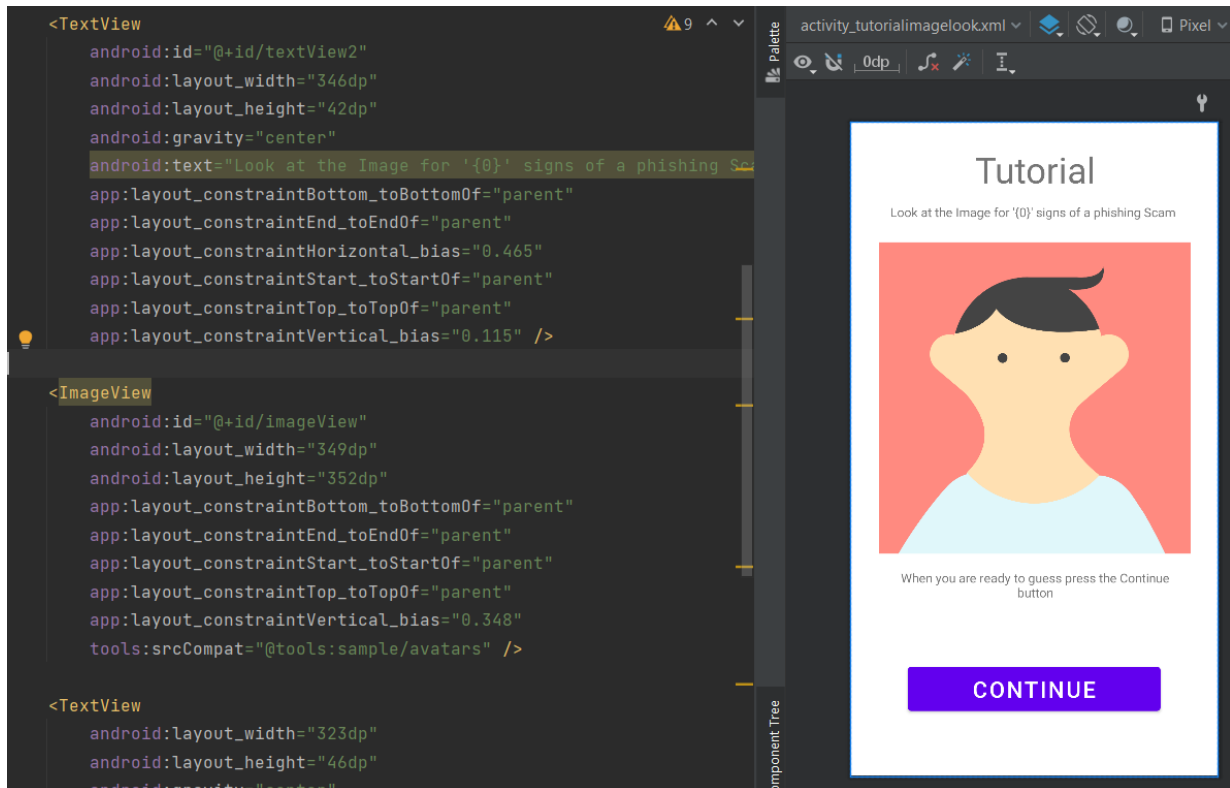
For the Layout file for both the Play and Tutorial modes, the layout was exactly the same. Both shared a textview which displayed the number of signs in a level with a corresponding id that would allow us to reference it later on, as well as having an imageview for the randomly selected level to be placed into.

```java
        Continue = (Button) findViewById(R.id.Continue);
        imageView = (ImageView) findViewById(R.id.imageView);
        Signamount = (TextView) findViewById(R.id.textView2);

        //get random level ID
        int int_random = GameFunctionActivity.GetRandomNumber();

        //get bitmap of random image for level
        Bitmap bitmap = GameFunctionActivity.GetImageBitmap(int_random);
        imageView.setImageBitmap(bitmap);

        //get signamount of random level
        String signAmountText = GameFunctionActivity.GetSignAmount(int_random);
        //insert new text into Signamount textview
        Signamount.setText(signAmountText.toCharArray(), start: 0, signAmountText.length());

        Continue.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) { openTutorialGuess(); }
        });
}

1 usage
private void openTutorialGuess() {
    Intent intent = new Intent( packageContext: this, TutorialImageGuessActivity.class);
    startActivity(intent);
}
```

In the imagelook Class files, the code was identical between tutorial and play as well. The code called a random integer from the game code function named GetRandomNumber(). This produced a random number using the Random datatype, but additionally allowed for setting the limit of where the number could generate. For this case, the upper limit was set to the maximum size of records (our JSON object Array). In addition to generating the number, the GetRandomNumber function also stores the random number in a variable local to the game code class, allowing us to reference it later on.

```java
    //Get a random record id number
    public static int GetRandomNumber() {
        Random rand = new Random();
        int upperbound = 0;

        //produce Random number limit between records 12 - Max of the records list
        upperbound = records.size();
        int int_random = rand.nextInt(upperbound);
        RandomLevelIDStorage = int_random;
        return int_random;
    }
```

Now with our random number, the imagelook class could reference the random number as an index id, allowing us to reuse the same code as the tutorial example view to return an image however the number of total signs needed to be implemented to work alongside this. As seen from the code snapshots above, the sign amount is returned as a string named SignAmountText, using the integer of random_int to identify the level.

The GetSignAmount function was implemented in the game code class and followed similar principles as the GetImageBitmap function. The code would search through the record JSON array and compare the recordID against the integer random_int. When the program had found a match, the level attributes of that the array entry would be stored locally as a variable named LevelAttributes. Then, the level attributes were checked each character at a time, adding up each time there was a 1. At the end, the total count of the signs was added to a integer variable called numberofSigns.
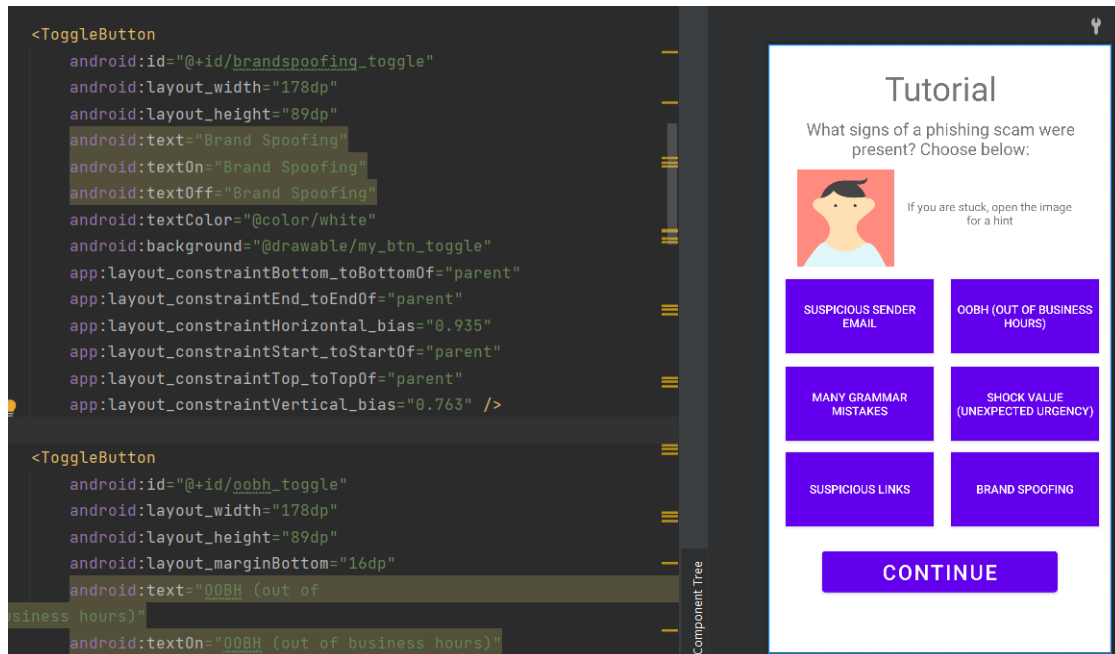
```java
public static String GetSignAmount(int int_random) {

    //String for Record Attributes to be copied into
    String levelAttributes = "";
    int numberofSigns = 0;
    String signamountText = "";

    //for each record in the arraylist
    for(Record record : records) {

        //if the recordID matches, copy the level attributes
        if(record.getRecordID() == int_random) {
            //store level attributes to variable
            levelAttributes = record.getRecordAttributes();
            LevelAttributesStorage = levelAttributes;
        }
    }
    //Find out Sign amount
    for(int i = 0; i < levelAttributes.length(); i++) {
        //if the char in levelAttributes is a 1
        if(levelAttributes.charAt(i) == '1') {
            //add to the sign amount counter
            numberofSigns ++;
        }
    }
    //set number of signs for score use later
    TotalSigns = numberofSigns;
```

The variable numberofSigns was then used in to create a message format, a custom string that allows us to input the number of signs using '{0}' as an annotation for where the number would be. In addition to this, two message formats were created if the number of signs was 1 or more, allowing for correct grammar on the message. Once formatted, the function returned signamountText, which then replaced the textview text in the imagelook activity class.

```java
//set the sign amount into a message for the textview
if(numberofSigns == 1) {
    signamountText = MessageFormat.format( pattern: "Look at the Image for {0} sign of a phishing Scam", numberofSigns);
}
else
    signamountText = MessageFormat.format( pattern: "Look at the Image for {0} signs of a phishing Scam", numberofSigns);
return signamountText;
```

The next sprint of the mobile application was focused on both the Tutorial image guess view as well as the Play image guess view. As shown in the requirements for the sprint the Tutorial guessing view needed additional work, with the inclusion of an image view that the user could use for a hint if they were stuck. Both views needed to have a series of buttons so the user could select which signs apply to the level image, this was done with the use of toggle buttons. The toggle buttons changed colour to a lighter purple, allowing the user to keep track of what signs were selected.

In the activity java class for the guessing views, a series of variables were made to reference the toggle buttons as well as the imageview and continue button. In order to collect the same image as before to give the user a hint, the program needed to call the previous random number instead of providing a new one. The CallRandomNumber() function allows us to do this.

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_tutorialimageguess);

    Continue = (Button) findViewById(R.id.Continue);
    brandspoofing_toggle = (ToggleButton) findViewById(R.id.brandspoofing_toggle);
    suspicousemail_toggle = (ToggleButton) findViewById(R.id.suspicousemail_toggle);
    oobh_toggle = (ToggleButton) findViewById(R.id.oobh_toggle);
    grammarmistakes_toggle = (ToggleButton) findViewById(R.id.grammarmistakes_toggle);
    shockvalue_toggle = (ToggleButton) findViewById(R.id.shockvalue_toggle);
    suspicouslinks_toggle = (ToggleButton) findViewById(R.id.suspicouslinks_toggle);
    imageView = (ImageView) findViewById(R.id.imageView2);


    //get previous record id
    int int_random = GameFunctionActivity.CallRandomNumber();

    //get bitmap of random image for level
    Bitmap bitmap = GameFunctionActivity.GetImageBitmap(int_random);
    imageView.setImageBitmap(bitmap);
```

```java
public static int CallRandomNumber() { return RandomLevelIDStorage; }
```

The code snippet above shows the CallRandomNumber function, as we previously stored the random integer in RandomLevelIDStorage, we can reference it again here with only a return value. This allows us to use the GetImageBitmap function, using the integer from the random level id storage, giving us the same image as last time whilst minimising the amount of repeated code.

In order to compare the users button response against the level attributes, they both needed to be the same data type. When the user has selected their buttons and hits the continue button, the button listener for the continue button runs a little extra bit of code before conventionally opening the next view. It reads the state of each toggle button, adding either a 1 or a 0 to a string, it is done in the order shown below in order to match the same order in level attributes; outlined by the database design documentation. Once a list is complete, the program then runs the SaveButtonInput function, passing over the newly made String list of ones and zeroes.

```java
Continue.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        //when user clicks continue, check the state of the toggle buttons and add to list array
        if (suspicousemail_toggle.isChecked()) {ToggleButtonValues += "1";} else {ToggleButtonValues += "0";}
        if (oobh_toggle.isChecked()) {ToggleButtonValues += ",1";} else {ToggleButtonValues += ",0";}
        if (grammarmistakes_toggle.isChecked()) {ToggleButtonValues += ",1";} else {ToggleButtonValues += ",0";}
        if (shockvalue_toggle.isChecked()) {ToggleButtonValues += ",1";} else {ToggleButtonValues += ",0";}
        if (suspicouslinks_toggle.isChecked()) {ToggleButtonValues += ",1";} else {ToggleButtonValues += ",0";}
        if (brandspoofing_toggle.isChecked()) {ToggleButtonValues += ",1";} else {ToggleButtonValues += ",0";}

        //send to game function activity to compare against level list and return score
        GameFunctionActivity.SaveButtonInput(ToggleButtonValues);

        openTutorialScore();
    }
});
```

```java
private void openTutorialScore() {
    Intent intent = new Intent( packageContext: this, TutorialScoreActivity.class);
    startActivity(intent);
    finish();
}
```

In the SaveButtonInput function, the java game code saves a public local variable of the button input, allowing the code to reference it again when on the score view.

```java
//Save users toggle button input
2 usages
public static void SaveButtonInput(String ToggleButtonValues) { UserButtonInput = ToggleButtonValues;}
```

The last sprint of the mobile application involved implementing a score view so that the player could see how many signs the guess correctly. To do this, the layout file used the same design principles as the other sprints, using an image view for the level image whilst the textview displayed their score. In addition, navigational buttons would display at the bottom of the screen to allow the player to return to the main menu or to play a new random level.
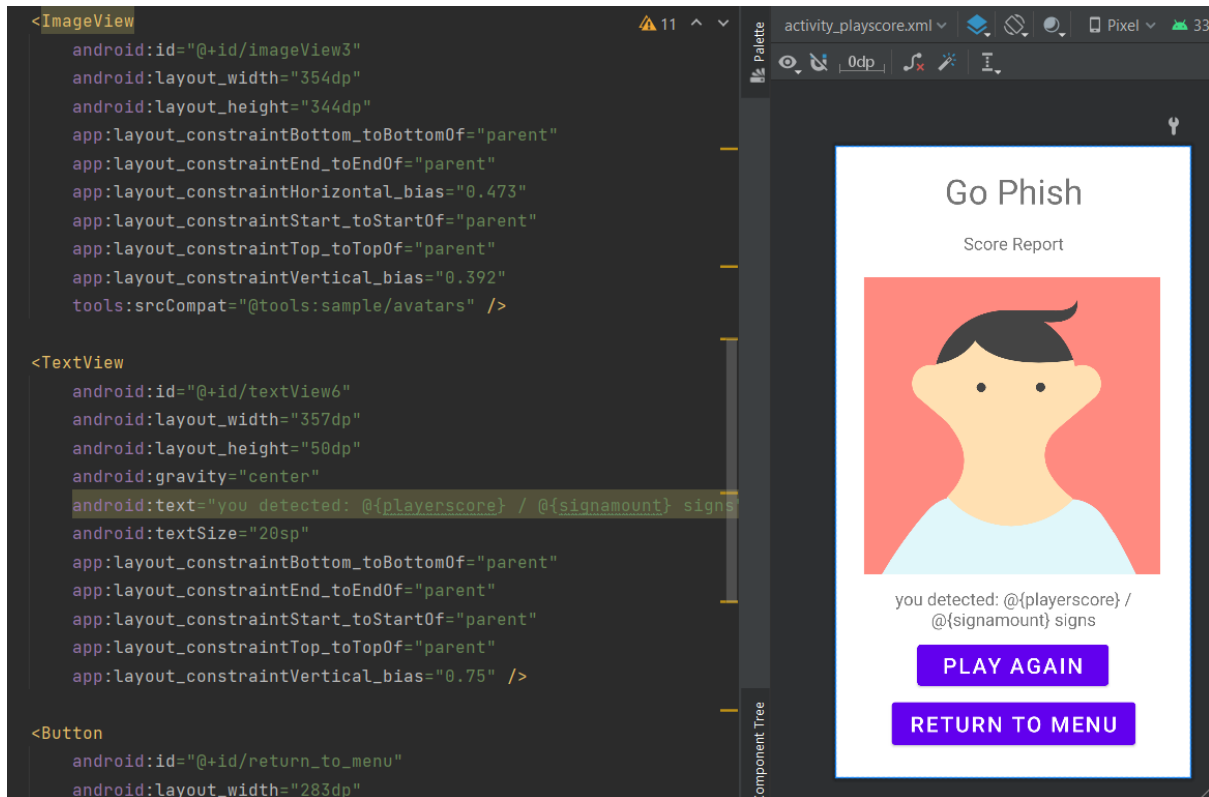
Within the Score activity class, the program needed to be able to change the bitmap of the image view as well as edit the text to display the users score. The image view and navigational buttons were all things I had programmed in previous sprints so that was able to be implemented quickly. However, in order to change the textview a new function would need to be called; ReturnScore from the game code class.

```java
public class PlayScoreActivity extends AppCompatActivity {
    2 usages
    private Button ReturnToMenu;
    2 usages
    private Button PlayAgain;
    2 usages
    private TextView ScoreReport;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_playscore);

        ReturnToMenu = findViewById(R.id.return_to_menu);
        PlayAgain = findViewById(R.id.play_again);
        ScoreReport = findViewById(R.id.textView6);

        String ScoreText = GameFunctionActivity.ReturnScore();
        ScoreReport.setText(ScoreText.toCharArray(), start: 0, ScoreText.length());

        ReturnToMenu.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) { openstartView(); }
        });
        PlayAgain.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) { openPlayImageLook(); }
        });

    }
}
```

```java
private void openPlayImageLook() {
    Intent intent = new Intent( packageContext: this, PlayImageLookActivity.class);
    startActivity(intent);
    finish();
}

private void openstartView() {
    Intent intent = new Intent( packageContext: this, MainActivity.class);
    startActivity(intent);
    finish();
}
```

The ReturnScore function compares the UserButtonInput string against the LevelAttributeStorage string: looping through each string to check if both characters are a 1. For every correct 1 there is a counter that increments and keeps track of the users score. Then, the function creates a message format string named scoreamountText, adding the players score and the total number of signs into the message and returning scoreamountText.

```java
//compare user input to level record and return score result
public static String ReturnScore() {
    String scoreamountText = "";
    int ScoreNumber = 0;

    //compare list of UserButtonInput and LevelAttributes
    if (LevelAttributesStorage != null) {
        for (int i = 0; i < UserButtonInput.length(); i++) {

            //check if Userbuttoninput is a 1
            if (UserButtonInput.charAt(i) == '1') {

                //check if User guess correctly according to the level attributes
                if (UserButtonInput.charAt(i) == LevelAttributesStorage.charAt(i)) {
                    ScoreNumber++;
                }
            }
        }
    }

    //Return score amount in a message for the textview
    if(ScoreNumber == 1) {
        scoreamountText = MessageFormat.format( pattern: "you detected: {0} out of {1} sign", ScoreNumber, TotalSigns);
    }
    else
        scoreamountText = MessageFormat.format( pattern: "you detected: {0} out of {1} signs", ScoreNumber, TotalSigns);
    return scoreamountText;
}
```

Testing:

After the implementation of the project deliverables, the next and final stage of the project was to test and debug features of the deliverables as well as consider what additional testing could be performed with regards to usability, interactivity, and design layout. Earlier in the project's development cycle these points were considered with regards to the project wireframes, leading to a better overall end solution.

Wireframe test plan and improvements:

The project wireframes underwent a testing and improvement process during December of 2022. During this time, it was important to receive feedback on the design aesthetic, interactivity, and useability of the app. Whilst it would have been much preferable to have an MVP (minimum viable product) to perform this feedback, the test plan was still able to provide enough information so that targeted improvements could be made and deliver an overall more accessible final solution.
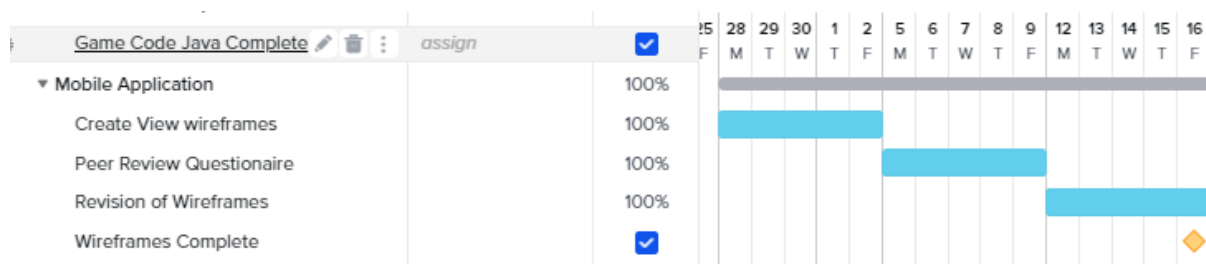
The test plan consisted of the project wireframes presented to the user in the form of a Wizard of Oz style paper-based prototype and additionally recorded both a pre and post-test questionnaire that the user would complete for each test [11]. As Alex was working from home during this stage of the project development, only two test users were able to be chosen for the evaluation test plan: Test user 1, Rachel Cleverley and test user 2, James Cleverley. This is due to the fact that as per the Ethical Approval guidelines, each test user must be over 18 years of age as well as each test user must not be a vulnerable adult.

The questionaries provided the test users with the ability to express their feedback within the Formative evaluation test plan [11], which was subsequently analysed and concluded upon at the end of the test plan document. The reoccurring message from the feedback was that less cyber-aware users would not be able to understand the how the app works or what key words, such as the level attribute names, mean.
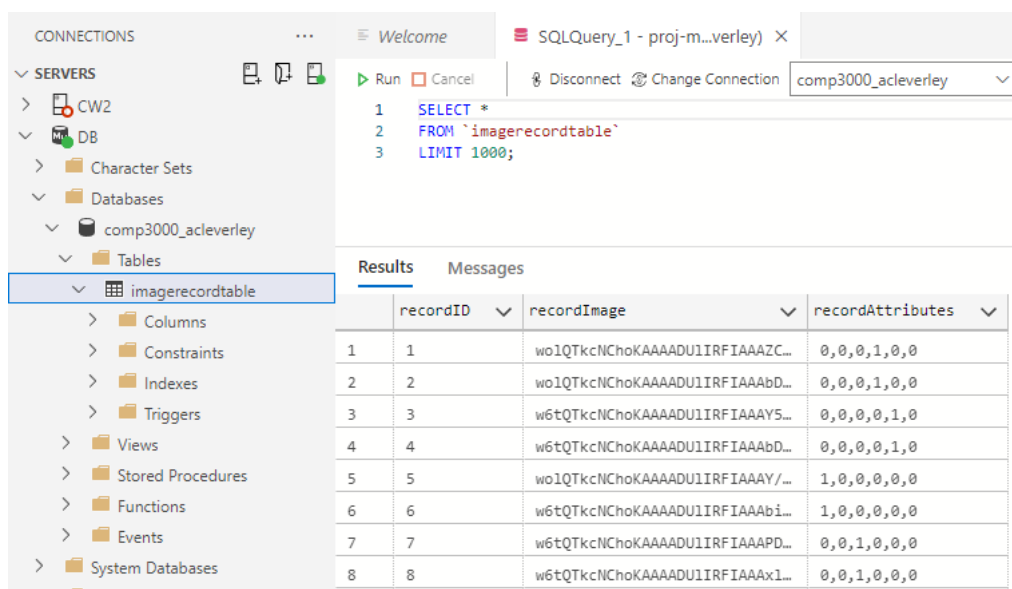
In order to resolve this, test user 1 suggested that the "Training" element of the mobile app should be renamed to "Tutorial" and focus on introducing first time users to how the app works and what the keywords for potential signs, such as OOBH, means [11]. This would manifest in the form of the tutorial example view in the 2nd iteration of the project wireframes, providing buttons to each sign that show images that highlight what part of the phishing email the potential sign refers too as well as being implemented into the mobile application solution in the Tutorial example view.

Test user 2 suggested that a feature that let the user choose how many signs a level has making the level easier or more difficult depending on how many signs the user selects [11].

Testing and debugging of project deliverables:

The testing stage of Go phish was broken up into three major sprints that were ongoing until the conclusion of project's development. These were: testing the connection of the MySQL database and ensuring the table records were hosted properly, testing the connection of the web API to ensure that the database information was being hosted correctly in a JSON format, and lastly the testing and debugging of the mobile application. Due to the nature of the project deliverables, the testing had to be performed in this order, making the debugging stage of the mobile application the last step before the project could be completed.

In order to test the connection of the MySQL database, Azure Data studio was used to interact with the database tables and perform a SELECT * FROM command. This allowed the imagerecordtable database table to be shown through Azure data studio in its current state and allow us to test to see if the data that was passed to the database had been stored without any errors. Below is screenshot of the SELECT command being used, showing the imagerecordtable with appropriate in its fields. With this test passing we can confirm that the database is working and move on to test the next deliverable for the project solution.

In order to test the .NET web API, Swagger was used to execute a get command that would retrieve the database information and display it for the user to see. This would allow the web API to be tested for connectivity to the database and additionally test to see if the API was hosting the data in a JSON format. Below is a screenshot of the swagger command being executed at "api/Record". This is the URL extension of Localhost that hosts the data for our mobile application to access. We can confirm that the database table is successfully being hosted and proceed to the testing stage of the mobile application.
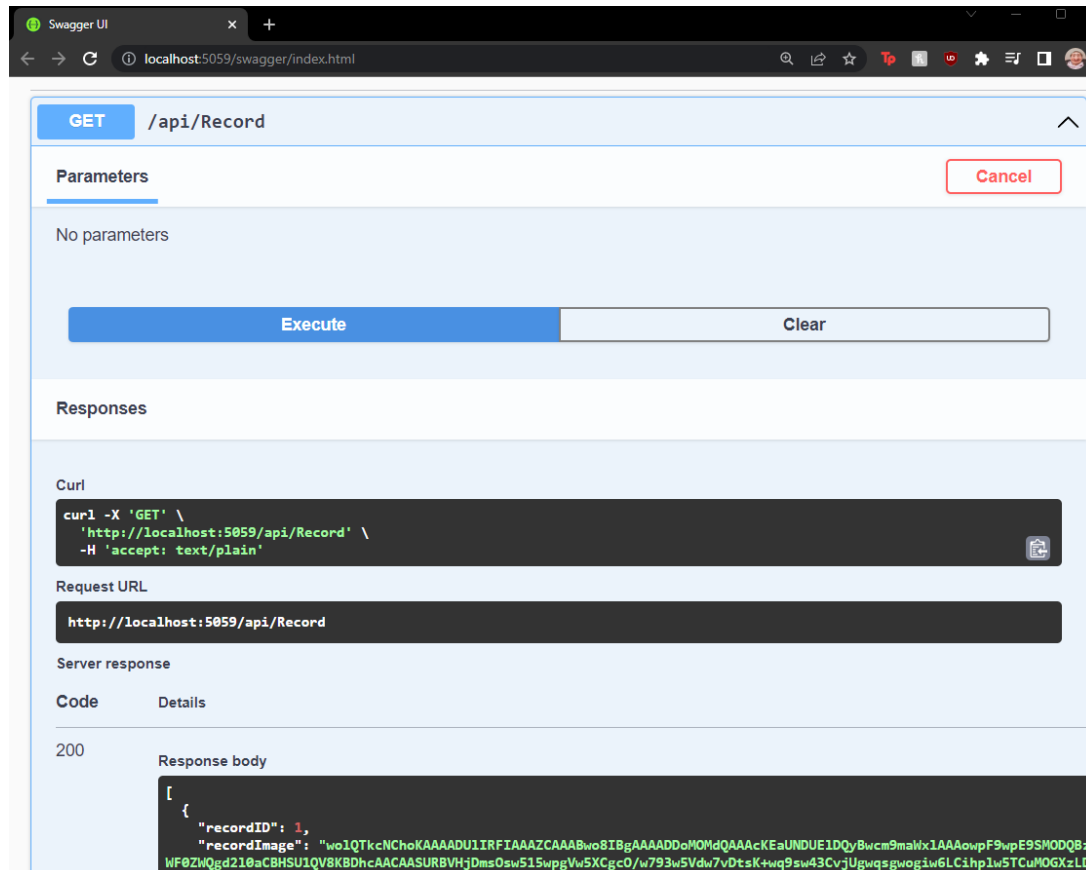
The first step in testing the mobile application was to ensure that the OkHttp request to the API was connecting successfully and retrieving the database information. Upon building and running the app for the first time, a Network on main thread error was encountered as evident in the screenshot below.

In order to solve this error, the AccessAPI function within the GameFunction activity needed to be separated into two components, the network focused OkHttp request to retrieve the String JSON data that should run on the background thread and the conversion of the json data into a Record JSON object array that should be performed on the main thread. To use the background thread, an asynchronous task needed to be created that would return the string json data back to the original AccessAPI function. This was implemented as shown below.

```java
public static class Access_API extends AsyncTask<Void, Void, String> {

    @Override
    protected String doInBackground(Void... voids) {
        OkHttpClient client = new OkHttpClient();
        String url = "http://10.0.2.2:5059/api/Record";
        String jsonData = null;

        //create new OkHttp request (Get a URL example on the Okhttp website)
        Request request = new Request.Builder()
                .url(url)
                .build();
        //If successful response, store Json array into String for conversion
        try {
            Response response = client.newCall(request).execute();
            if (response.isSuccessful()) {
                jsonData = response.body().string();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return jsonData;
    }
}

public static void AccessAPI() {
    //Access the API record using OkHttp in a background thread
    Access_API access_api = new Access_API();
    String jsonData = null;
    try {
        jsonData = access_api.execute().get();
    } catch (ExecutionException | InterruptedException  e) {
        e.printStackTrace();
    }
}
```

Another error of the mobile application that had to be resolved was that previous activities, such as PlayImageLook, would give errors and crash the testing session even though the mobile application was not on that view. The intents that were used to navigate between activity views throughout the mobile app needed to be closed after the continue button was pressed, otherwise the activities would still be in memory and cause unnecessary errors despite the program not needing them to be in memory anymore. In order to solve this issue, an additioanaly line of code was added to each intent function that declared the current activity view as finished with and removed unnecessary data from memory, subsequently solving the errors from previously used views.

```java
private void openPlayScore() {
    Intent intent = new Intent( packageContext: this, PlayScoreActivity.class);
    startActivity(intent);
    finish();
}
```

End Project Report:

Go Phish is a mobile game application that focuses on phishing email scams and the signs to look out for when viewing them. The app is targeted towards self-employed people and people who want to be more cyber-aware, providing a fun 'brain-training' exercise to verse themselves in signs of a potential phishing email. Unlike traditional 'Crash courses' where users would spend a day learning about cyber-awareness, Go phish is a 'On-the-Go' mobile app that allows the user to freshen their memory of potential phishing signs and highlights what to look out for when viewing one.

Majority of the project objectives have been fulfilled with all required objectives having been successfully implemented. Go phish allows a user to choose between play or tutorial modes and selects a random level from the Record level array for the user to test their detection skills and select the appropriate signs. Additionally, the user is able to view a random phishing email image via an image view however the image is unable to be scaled making it difficult to properly examine the phishing email image and play go phish well. With this in mind, Go phish provides a complete solution however, with the current implementation of how images are displayed to the user and unimplemented additional features, future development of Go phish can continue and, with more time to understand the technicalities of android mobile development, can become a widely accessible and complete solution.

Project Post-Mortem:

What went well?

Despite the complications of working from home and switching development devices, the design stage to the project went very well. Taking place over November and December of 2022, the design, testing and improvement of the project wireframes as well as the previously designed project ERD allowed Go Phish to begin taking shape and conceptualising how the final project solution would look and feel. The Formative evaluation test plan of the wireframes that also took place during this time provided feedback that not only suggested ideas that would help shape the final solution but alleviated any doubts that the project solution was not a good idea and that it would not be possible.

The sequential view sprints that took place throughout February of 2023 were also an area of the project that went very well. Prior to these sprints, the mobile application had not been implemented or prototyped on android studio which left an intense anxiety towards starting this stage of the project. Fortunately, upon beginning the Landing and Tutorial example sprints, Android studio's XML layout files and Java activity files became familiar and once core pieces of code were implemented, a lot the code could be repeated and rapidly implemented alongside the agile workflow that the project utilized.

Another area of the project deliverables that went well was the ability for images to successfully be decoded and displayed as a bitmap image of a random level for the user to examine. As the project is focused on examining phishing email images this step in the project's development was a significant achievement and one which felt key to visualizing the project's final submission.

What did not go well?

Unfortunately, due to only being able to properly test and debug the android studio mobile application in the last stage of testing, a lot of time was spent solving errors and debugging the application that should have been spent adding optional requirement features and optimizing required features of the project to create a sound final solution. This is, in large part, due to the fact that I had basic level experience in android studio and underestimated how difficult it would be to implement and debug a mobile application solution.

Over the course of late January and February I attempted multiple times to create a web API using Visual studio 2019 and .NET 5.0. Whilst a successful API solution would later go on to be implemented using Visual studio 2022 and .NET 7.0 over the Easter break in the beginning of April, a significant portion of time and energy was spent trying to implement an API solution that ultimately was fruitless and caused unnecessary frustration and distress during the project development. This resulted in the implementation of the API taking longer than its designated sprint time and subsequently pushing back the implementation of the rest of the project, notably the mobile application.

Lessons learned & future enhancements:

During the course of the Go phish project, I have gained knowledge and understanding of mobile development techniques, having learned about more advanced computing techniques such as utilizing multithreading as well as when to best use static and non-static functions and classes. In addition to this I have also learned how images can be stores as string variables using Base64 encoding and encoded images can be decoded into byte[] arrays and subsequently bitmap images.

Should the Go phish project be continued in the future, the next step in enhancing it would be to optimise and successfully implement the high priority requirements, allowing the project to have a successful foundation to then continue to add more advanced features such as reviewing the scores of previous sessions in the form of a score tracker that would allow the user to keep track of their score and see their improvements, and an option that allows the user to select how many signs are shown per level, being able to choose the difficulty of the by making the number of signs higher or lower. In future enhancements, the project should aim to include more varied phishing email levels, making the random levels more unique and challenging experiences for the user to be increasingly more cyber-alert and aware.

Conclusions:

This report has examined the growing trends of cyber-attacks and highlighted a need to raise awareness of potential phishing attacks and empower those with less cyber-awareness with tools that can help them to identify the potential signs of a phishing email through the use of the Go Phish mobile application solution. Go phish was developed using modern development tools such as Azure Data Studio, Visual Studio and Android Studio.

Go Phish is unique and ambitious compared to alternative phishing awareness solutions as it offers the user the ability for an 'On-the-go' handheld method of training and awareness,

accessible to those that may want to be more cyber-aware but do not have the time or finances to participate in a Training and awareness day, such as the current medium for majority of cyber awareness.

Whilst the final solution of Go phish is complete, the required features to be added have a foundation and implementation steps that, with more development time, can be continued and brought together for a more optimised package and useable solution.

References:

[1] Ell, M. and Gallucci, R. (2022) Cyber security breaches survey 2022, GOV.UK. Available at: https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2022/cyber-security-breaches-survey-2022#key-findings (Accessed: March 27, 2023).

[2] Rosenthal, M. (2022) Phishing statistics (updated 2022) - 50+ important phishing stats, Tessian. Available at: https://www.tessian.com/blog/phishing-statistics-2020/#:~:text=96%25%20of%20phishing%20attacks%20arrive,message%2C%20we%20call%20it%20smishing. (Accessed: March 27, 2023).

[3] Com, D. (2021) Agile methodology for software development, Amplifyre. Available at: https://www.amplifyre.com/articles/agile-methodology-for-software-development (Accessed: March 29, 2023).

[4] Rosenthal, M. (2022) Phishing statistics (updated 2022) - 50+ important phishing stats, Tessian. Available at: https://www.tessian.com/blog/phishing-statistics-2020/#:~:text=96%25%20of%20phishing%20attacks%20arrive,message%2C%20we%20call%20it%20smishing. (Accessed: March 27, 2023).

[5] Office, I.P. (2021) Copyright act, GOV.UK. GOV.UK. Available at: https://www.gov.uk/government/publications/copyright-acts-and-related-laws (Accessed: March 31, 2023).

[6] Cleverley, A (2022) Project ERD Version 1, GitHub. Available at: https://github.com/AlexCleverley/COMP3000/blob/main/Design%20Documentation/Project ERD.vpd.png

[7] Cleverley, A (2023) Project ERD version 2, GitHub. Available at: https://github.com/AlexCleverley/COMP3000/blob/main/Design%20Documentation/ERD version 2.PNG

[8] Cleverley, A (2023) UML diagram, GitHub. Available at: https://github.com/AlexCleverley/COMP3000/blob/main/Design%20Documentation/UML Button use cases.png

[9] Cleverley, A (2023) Export of COMP 3000 project Gantt chart, GitHub. Available at: https://github.com/AlexCleverley/COMP3000/blob/main/Design%20Documentation/COMP _3000 ganttchart.pdf

[10] Corporation, M. (2023) Azure Data Studio, Microsoft Azure. Available at: https://azure.microsoft.com/en-us/products/data studio

[11] Cleverley, A (2023) Formative evaluation test plan, GitHub. Available at: https://github.com/AlexCleverley/COMP3000/blob/main/Design%20Documentation/Formative%20Evaluation%20Test%20plan%20documentation.pdf