REPOTA

A SERVICE REPORT APPLICATION

Applied Project And Minor Dissertation
Bachelor of Honours in Software Development

BY
John Shields

May 7, 2021

Advised by: Andrew Beatty
Department of Computer Science and Applied Physics
Galway-Mayo Institute of Technology (GMIT)

Contents

1 Introduction **3**
- 1.1 Abstract . . . 3
- 1.2 Acknowledgements . . . 3
- 1.3 Project Introduction . . . 3
  - 1.3.1 Objectives . . . 5
- 1.4 Resources URLs . . . 6
- 1.5 Chapter Descriptions . . . 7

2 Methodology **9**
- 2.1 Project Scope & Goal . . . 9
- 2.2 Development Methodology . . . 10
  - 2.2.1 Waterfall . . . 11
  - 2.2.2 Agile . . . 11
  - 2.2.3 Project & Time Management . . . 12
  - 2.2.4 Verification & Testing . . . 13

3 Technology Review **15**
- 3.1 GitHub . . . 15
- 3.2 OpenAPI . . . 16
  - 3.2.1 API Specification . . . 16
  - 3.2.2 How OpenAPI fitted the API's design . . . 16
- 3.3 Golang . . . 17
  - 3.3.1 Go Web Frameworks . . . 19
  - 3.3.2 How Go fitted the Back-end . . . 20
- 3.4 Angular & Ionic . . . 20
  - 3.4.1 How Angular & Ionic fitted the Front-end . . . 22
- 3.5 MySQL . . . 23
  - 3.5.1 Relational . . . 23
  - 3.5.2 ACID . . . 23
  - 3.5.3 How MySQL fitted the Database . . . 24
- 3.6 Amazon Web Services . . . 24
  - 3.6.1 Virtual Machines . . . 24
  - 3.6.2 S3 Bucket . . . 25
  - 3.6.3 Elastic Beanstalk . . . 25

3.6.4 How AWS fitted the project's hosting 26

4 System Design **27**
4.1 The Database 27
4.2 The Back-end 30
4.2.1 Horton API Specification 30
4.2.2 Go Horton 33
4.2.3 Account & Session System 34
4.2.4 Report System 39
4.2.5 3rd Party API 43
4.2.6 Elastic Horton 44
4.3 The Front-end 45
4.3.1 Repota & Horton Integration 46
4.3.2 Account Pages 47
4.3.3 Report Pages 51
4.3.4 AuthGuard 62
4.3.5 Repota UI 64
4.3.6 Repota Bucket 66

5 System Evaluation **67**
5.1 Horton Robustness 67
5.2 Repota Robustness 69
5.3 Mobile Phone Functionality 71
5.3.1 Responsiveness 71
5.4 Issues Encountered 73
5.4.1 Export to PDF 73
5.4.2 Finding the right 3rd Party API 73

6 Conclusion **75**
6.1 Project Conclusion 75
6.2 Objectives Achieved 76

7 Appendices **81**
7.1 GitHub Repository & Web Application URLs 81
7.2 How to Install and Run 81

List of Figures

1.1 Application Logo 4

2.1 Agile VS Waterfall 12
2.2 GitHub Projects 13
2.3 GitHub Issues 13

3.1 Repota App Repository 15
3.2 Go VS Java & C++ [7] 17
3.3 CLI Go Docs on fmt 18
3.4 Go Functions inside the Back-end's openapi Package 18
3.5 Gin Error Handling 19
3.6 Angular & Ionic Home Page Structure 21
3.7 Route of Repota's Home Page 22
3.8 MySQL InnoDB Transaction for inserting data into two tables. . 23

4.1 Example of a Full Report 28
4.2 JOIN QUERY for all Report information 28
4.3 Database Schema 29
4.4 AWS Server 29
4.5 Login Operation 31
4.6 API Paths 32
4.7 Schema Models 32
4.8 InLineObject Struct 33
4.9 Session Struct 33
4.10 WorkerAccount Struct 33
4.11 JobReport Struct 33
4.12 CORS function 34
4.13 Register 35
4.14 RegisterNewUser 35
4.15 isValidAccount 35
4.16 createSessionId 35
4.17 generateSessionId 35
4.18 User in the workers and session tables 36
4.19 Login 36

4.20 verifyDetails . . . 36
4.21 Browser Cookie . . . 37
4.22 Logout . . . 37
4.23 CheckForCookie . . . 38
4.24 Account System . . . 38
4.25 CreateReport Function . . . 39
4.26 Prepare MySQL INSERT QUERIES . . . 40
4.27 Go MySQL Transaction for two INSERTs . . . 40
4.28 GetReports Function . . . 40
4.29 Go MySQL JOIN to SELECT all Reports . . . 41
4.30 ID Parameter from Report . . . 41
4.31 Go MySQL JOIN to SELECT a singe Report . . . 41
4.32 Go MySQL UPDATE a Report . . . 42
4.33 DeleteReport Function . . . 42
4.34 Report System . . . 43
4.35 Back4App Vehicle Data . . . 43
4.36 Elastic Beanstalk Environment . . . 44
4.37 Secure Horton . . . 44
4.38 Repota's Pages . . . 45
4.39 Repota & Horton Connection . . . 46
4.40 Account Service Connection to API . . . 46
4.41 InLineObject Interface . . . 47
4.42 JobReport Interface . . . 47
4.43 Register Method . . . 48
4.44 Regular Expression pattern for Password . . . 48
4.45 Username is already taken - Error . . . 49
4.46 Login Method . . . 49
4.47 Password is incorrect - Error . . . 50
4.48 Hamburger Menu - Logout Button . . . 50
4.49 Logout Method . . . 51
4.50 History Page . . . 51
4.51 ngOnInit Method - Load all user's Reports . . . 52
4.52 User has no Reports . . . 52
4.53 Display Page . . . 53
4.54 ngOnInit Method - Get Report by ID . . . 53
4.55 Report not found . . . 53
4.56 JobReport Object & Push Object to API . . . 54
4.57 Checkboxes - If/Else Statement . . . 55
4.58 ngOnInit method - Back4App Data . . . 55
4.59 Create Page - HTML data list . . . 56
4.60 loadReportById Method . . . 56
4.61 Edit Page - HTML data list . . . 57
4.62 Export Page . . . 58
4.63 onExportPDF Method . . . 58
4.64 Service Report PDF . . . 59
4.65 Delete Report Confirmation . . . 60

4.66 deleteReport Method 61
4.67 Error Message Handlers 61
4.68 JSON extracting & parsing of error message 62
4.69 Please login first 62
4.70 loggedIn Method 63
4.71 canActivate Method 63
4.72 Blocked Routes 63
4.73 Already logged in message 64
4.74 Home Page 65
4.75 About Page 65
4.76 Account Page 65
4.77 Secure Repota 66

5.1 Payload of mock user details 67
5.2 Test Register - POST request 68
5.3 Register test pass 68
5.4 Login Feature 69
5.5 Login Steps 70
5.6 Login Steps Pass 70
5.7 Repota on Chrome DevTools 71
5.8 Repota on Mobile Phone 72

Chapter 1

Introduction

1.1 Abstract

Service reports are essential records for company technicians in the equality of agricultural machinery, automobiles, computers, air-crafts, and military machines. For instance, a history of these reports needs to be stored for easy access as problems may arise with the exact automobiles recorded in the report. Having this information outlines previous problems and allows to identify the problem should it be related to the initial.

1.2 Acknowledgements

I would like to thank my lectures throughout the years and my supervisor, Andrew Beatty, for his advice and for pointing me in the right direction throughout the year. Lastly, I would like to thank my brother for inspiring the idea that turned into a successful final product.

1.3 Project Introduction

In an achievable manner and at a Bachelor of Honours standard, I aimed to develop a service report, SaaS (Software as a Service) application for automobile technicians titled 'Repota' for the *Applied Project and Minor Dissertation.* I decided to complete this application after observing my brother, who is employed in a workplace that specializes in the supply and service of new and used Mobile Plant and Equipment, have difficulties with documenting reports in a timely fashion. The reports are currently completed on paper or with Microsoft Excel. I decided to create an application to write up service reports that is productive, easy to use and effective. After careful consideration, I decided the application's features would consist of a user account system, CRUD (Create, Read, Update and Delete) operations for the reports and an option to export

them to PDFs.

This application is at Level 8 standard as it aimed to be at a full-stack level; it had to contain three components: a front-end, back-end, and a database. The technologies to develop such an application to be Angular and Ionic for the front-end, OpenAPI to design the front-end and back-end APIs (Application Programming Interface), Golang for the back-end and MySQL for the database. The entire application would then be hosted through AWS (Amazon Web Services). Entailing S3 Bucket (Simple Storage Service) for the front-end, Elastic Beanstalk for the back-end and an EC2 (Elastic Compute Cloud) Ubuntu Virtual Machine for the database.

The purpose of this dissertation is to unpack the planning, implementation and reflection of the technologies of the application and the objectives achieved, how it was designed and how it all came to together as a whole. It will firstly look at the methodology of the project's development. Secondly, it will investigate the technologies that were researched and implemented throughout the project. Following on, it will demonstrate the designs and evaluation of the project. The conclusion will then summarise and reflect on the achievements of the project's main objectives.

Figure 1.1: Application Logo

1.3.1 Objectives

The overall aim of the project was to achieve the following objectives:

- Comprehensive MySQL database with all the required information.
  - Users and Service reports.
- OpenAPI specification of Front-end and Back-end APIs.
- Robust and RESTful back-end connected to the database in Golang.
  - CRUD Operations for service reports.
  - Account system for users; register, login and logout.
    - Microservices - Sessions and Cookies for users.
  - 3rd Party API access for vehicle information.
- User friendly front-end with the Angular and Ionic Framework connected to the back-end.
  - Report - CRUD operation pages.
  - Account - Register, Login and Logout pages.
  - Option to export reports to PDFs.
  - Front-end UI responsiveness for multiple devices.
- Testing of back-end and front-end.
  - Suite of integration tests for the back-end's functionality.
  - Suite of high level behaviour tests for the Front-end.
- Database, Back-end and Front-end hosted on AWS.
  - Database hosted on a EC2 Ubuntu Virtual Machine.
  - Back-end hosted with Elastic Beanstalk.
  - Front-end hosted with S3 Bucket.
  - HTTPS for the hosted back-end and front-end.

1.4 Resources URLs

- GitHub Repository: https://github.com/johnshields/Repota-App
- Repota Web Application: https://www.repota-service.com
- Video Demonstration: video_link_here

Repository Contents

- Repota
  - Source code of the Front-end.
- Horton
  - Source code of the Back-end.
- API Specification
  - OpenAPI Specification of the Front-end and Back-end APIs.
- Database
  - Script of the Application's Database.
- Workings
  - Miscellaneous files for rough work.

1.5 Chapter Descriptions

Methodology

Methodology discusses the project's scope, its goal, and any other ideas that were considered. Along with the development methodology used for the project's development approach, its time management, and the initial setup for verification and testing.

Technology Review

The Technology Review discusses the technologies researched and implemented into the project. Along with describing how they fitted the requirements of the project.

System Design

System Design goes through the entire design of the project and how it was all constructed together to become a final product.

System Evaluation

System Evaluation tells how the final product was thoroughly tested through out the front-end and back-end. Including a section for issues encountered that caused long pauses in development.

Conclusion

The Conclusion summaries the project, possible next steps for it, what was learned from the project and addresses the objectives outlined in the Introduction.

Chapter 2

Methodology

2.1 Project Scope & Goal

The project scope had to have a system with several functioning parts that worked with as many modern technologies as possible. Original ideas for this project paralleled the ideas of Desktop Application Development and Game Development. These possible routes for the project were researched with the focus on User Interface and experience. One of these ideas was a desktop application for people to send Curriculum Vitaes (CV) to relatable companies. On this suggested app, the user would upload their CV. That CV would be sent through an Artificial Intelligence system that would read it, pick out the keywords and send it to companies looking for future employees in that field. At first, this was a great concept but constructing an app for this service was an extensive scope as it would have to either have a vast database or a third party API (Application Programming Interface) to correlate all the companies from different sources.

Another initial idea was to develop a fully-fledged open-world Game in the 'Unreal Engine' with C++. This idea was drastically different from the others due to it originating from a creative perspective and personal interest. Old noir films and parallel universes inspired the game. Making a mix of these two genres would be quite exciting and would mean that this open-world would have to be huge. This idea was a favourite and unfortunately had to be forfeited as it was overly ambitious. Perhaps, this project will be pursued in the future with a time allocation to permit it.

In the end, it was decided to develop a service report application for automobile technicians. This would include an app that allowed service reports to be completed for auto dealerships and rental companies. This idea was considered through observations of my brother's line of work which specializes in the supply and service of new and used Mobile Plant and Equipment. The company found challenges when documenting service reports on paper and Microsoft Excel. Ini-

tially, the app was to be a desktop app but after consideration, it transpired that it would would be most suitable to work on a tablet or a mobile phone. With this, the Angular and Ionic framework was chosen for the front-end to build it into a website application. The app needed to ensure confidentially and security as it would include sensitive, work-related data. Therefore, the desktop app focus was not ideal as some app's data may need to be temporarily held on the computer running it. This would make the data less secure than if it were on a back-end database server. [1]

This app would need a quick and robust back-end to provide the front-end's users with time efficient and immediate connection to the database. Google's relatively new language Golang (Go) to implemented the functionality of it and OpenAPI to design the API environment to connect the the back to the front-end were chosen. Go is based off the C programming language and is proven to be as prompt as other languages such as Java and C++. There are several web frameworks for Go available, such as Gorilla Mux, Martini and Gin Gonic. After researching and testing, Gin proved to be the best web framework to use as it consists of the necessary requirements for endpoints, CORS managements. In conclusion, Go was the most suitable choice for the back-end component of the application.

A reliable and strong database was essential as this project's focus was for the application to have the ability to be used in a working environment; it would have to store a large quantity of data that needed to be accessed quickly and be able to add and update the data freely from the front-end through the connection of the back-end. MySQL was chosen for the database as it is proven to have these mentioned qualities with its use of schemas and InnoDB transactions.

The entire application would need to be hosted through a cloud platform to achieve the SaaS requirements. AWS (Amazon Web Services) provides a first year free tier. AWS would be an adequate platform for hosting the application as it has a wide range of services. The ideal setup for the hosting would be S3 Bucket for the front-end, Elastic Beanstalk for the back-end and an EC2 Ubuntu Virtual Machine for the database.

2.2 Development Methodology

The planned approach for the development method was the Agile methodology, Personal Scrum, which consisted of plans and implementations that allowed for flexibility and adaption. Waterfall was considered as it would be ideal to have everything planned and documented before implementation. However, with new frameworks and technologies, Waterfall was not suitable as some problems could arise with these unfamiliar frameworks and technologies. With the use of Personal Scrum, the project was separated into sprints to focus on robustness more than anything else. This section's remainder discusses why and how this

approach benefited the development and shows how a Waterfall approach was not suitable. Also, Figure 2.1 shows how Waterfall and Agile structure projects.

2.2.1 Waterfall

Waterfall is based on Civil Engineering and how software packages for spacecraft mission planning, commanding, and post-flight analysis were developed. [2] It requires almost a complete prediction of the project with a "Big Design Up Front" from planning and documenting the development process. Before coding, the project manager makes a plan and then writes detailed documentation of all plan phases. These phases happen in a strict order. [3]

The Five Phases of Waterfall Life-cycle

- Requirements Analysis and Specification
- Architectural Design
- Implementation and Integration
- Verification/Testing
- Operation and Maintenance

With Waterfall, each stage is required to be fully completed before the next one is attended. This project has a setup of a Front-end and Back-end, meaning if Waterfall were used, it would have most likely required the back-end to be completely built before the front-end was ever touched. This process is not ideal as some cross-origin issues (CORS) can happen between both ends and result in refactoring the back-end in order for it to be able to connect to the front.

2.2.2 Agile

Agile project management and software development is an iterative approach that enables teams to be flexible and adaptable. This method allows for versatility and, depending on the project, allows for delivering a valuable product to consumers faster and without the upfront requirements with Waterfall. Agile employs the Scrum framework, which involves sprints which are fixed-length work iterations. Each sprint has four ceremonies that provide structure. [3]

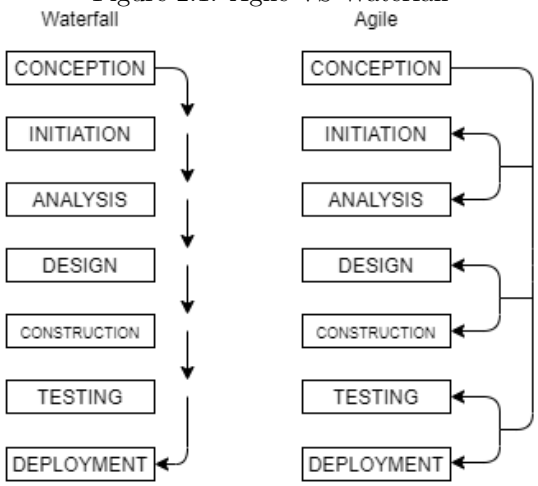
The Four Ceremonies of Scrum

- Sprint Planning
- Sprint Review
- Sprint Retrospective
- Daily Scrum

Personal Scrum

Personal Scrum is an Agile methodology that employs scrum practices for one-person projects; through observation of what work has been fulfilled, adaptation of work that requires refactoring, incremental elaboration for planning, prioritizing and sizing of the essential work with the use of allocated time-frames to improve personal productivity. "While intended for big, collaborative projects, there were a lot of elements of scrum that could be adapted pretty well to individual productivity." [4]
Having this approach allowed to have 1-2 days a week to block out other college work and have them dedicated to the project. These days were usually allocated to the end of the week and during college time off, more days were designated to increase productivity time. The first day mainly consisted of a review the week previous' work, plans and setups for implementations to be done the next day. On the second, if the implementations were either finished or in part, the project's supervisor was informed of the progress, any problems that occurred, and what was next to be done.

Figure 2.1: Agile VS Waterfall



2.2.3 Project & Time Management

In order to have a structured schedule for the project, this required a tracking software with a scrum board. Softwares such as Jira and YouTrack were considered, which seemed a bit over the top for a one-person project. GitHub's repositories have a section titled 'Projects' (Figure 2.2). Here a work board for the repository can be created to organize the work into cards. GitHub issues can be referenced to allow a card to keep track of a particular issue. GitHub's Projects was very convenient, especially with it being in the project's repository. GitHub Issues (Figure 2.3) are a way of tracking enhancements and any problems encountered for projects and are also located in the project's repository.

Figure 2.2: GitHub Projects

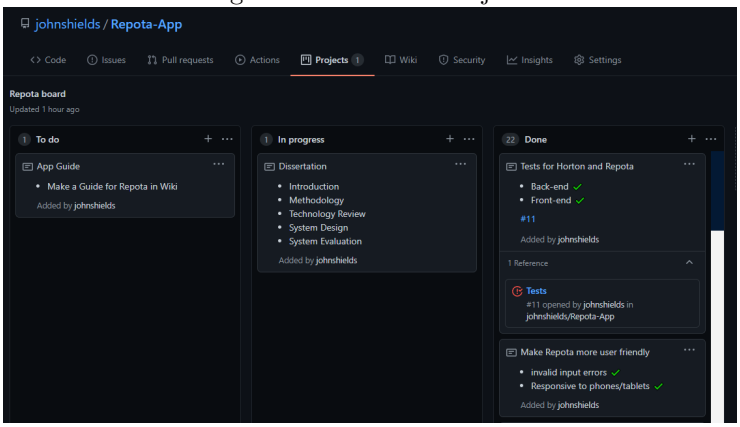
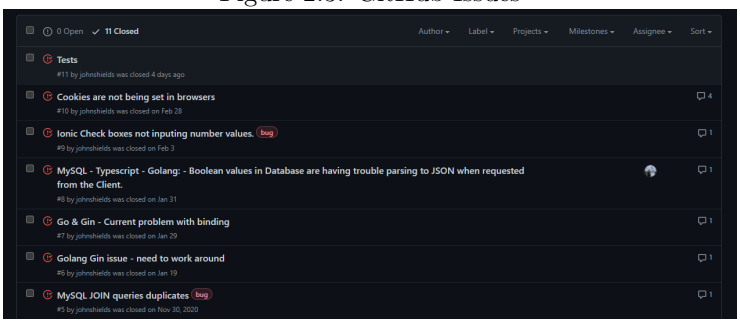


Figure 2.3: GitHub Issues



2.2.4 Verification & Testing

In order to get the project set up, a prototype MySQL database was structured with just an automobiles table. Then a basic Go back-end with the web framework Gorilla Mux was set up to have access to that database. Next was a simple Angular and Ionic front-end that connected to the back-end to present that table on a history page of reports. The first issue was that data from the back-end was unable to cross over to the front-end due to CORS. In order to resolved this issue the web framework was changed to Gin Gonic to allow a CORS function to be implemented into the back-end to allow the cross-over to resolve this issue. Once that was all implemented, an AWS E2C Ubuntu Virtual Machine (VM) was initiated to host the database and the back-end. With the server now in the project, the front-end was altered to connect to the back-end running on the VM. This initial setup fined tuned the environment which, allowed the real development to begin.

Integration and High-level Behavior Tests

To make sure the back-end and front-end were robust, tests were written. Go's standard package for testing was used to perform integration tests for tests multiple operations of the back-end at the same time. While CucumberJS was chosen to test the high-level behavior of the front-end which, tests it from a user's perspective. These tests are explained further in the System Evaluation chapter.

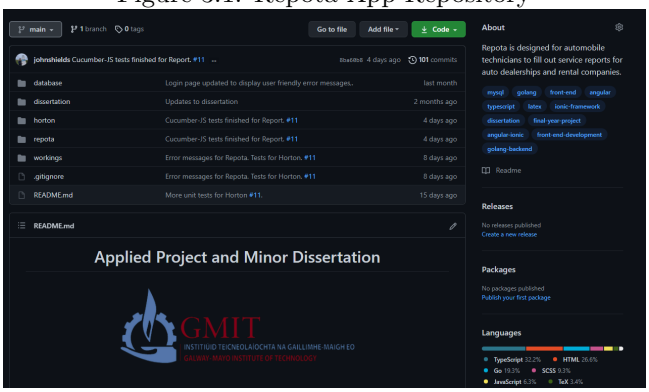
Chapter 3

Technology Review

3.1 GitHub

GitHub is a Version Control Software that stores the entire code history of projects in what they call 'Repositories' (Figure 3.1), with READMEs and Wikis in Markdown for documentation. Repositories can be used for individual usage, and they can add collaborators to commit (contribute) to the repository. For team-based usage, an organization can be created that would contain individual repositories that all team members can commit to the repository. It is an amazing tool. With GitHub, if something went wrong with a project, e.g. a bug or a sudden loss of files, they can all be recovered from the repository making it easier and faster to fix problems. It gives the opportunity to show off work their users have done to future employers without having to send ZIP files or screenshots. GitHub was widely used in this project, from the repository to the projects board, Wikis and the issues.

Figure 3.1: Repota App Repository



3.2 OpenAPI

OpenAPI specification are machine-readable API blueprints for RESTful web services. Client and Server stubs can be generated along with documentation just from these blueprints. OpenAPI enables to perfect with an efficiency of design and testing. Swagger Tools by Smart Bear allow for the implementation of specifications for these blueprints. These blueprints can be written in a human and machine-readable format of YAML. These blueprints can also be written in JSON. YAML almost lives as documentation of an API, whereas JSON may take a longer time to understand as it is more focused towards machines. [6]

3.2.1 API Specification

As said above, an API Specification (spec) works as a blueprint. API specs are broken down into individual parts. The spec always starts with the version of the OpenAPI, followed by a title, description and a version of the API itself as documentation of the spec. The API's base URL (e.g., - /api/v1/) is then defined under the servers section as an URL. The spec is then broken down into paths that are split into a request and a response. The path's request part includes; a HTTP method such as POST, a description of what the path can do, an operationId as a unique identifier for the method (handler function), a request body of a schema model with an example of data objects and any security that is required to make the request to the path such as an API key or a cookie. The response part entails the possible status code and message such as a 200 for success, 400 for a bad request and 500 for internal server errors. Schema models can also be defined within a spec. Schema models contain the following properties; the identifier, name, description and the data fields, types and values. These models represent request payloads to POST/send data to a server in request bodies. [6]

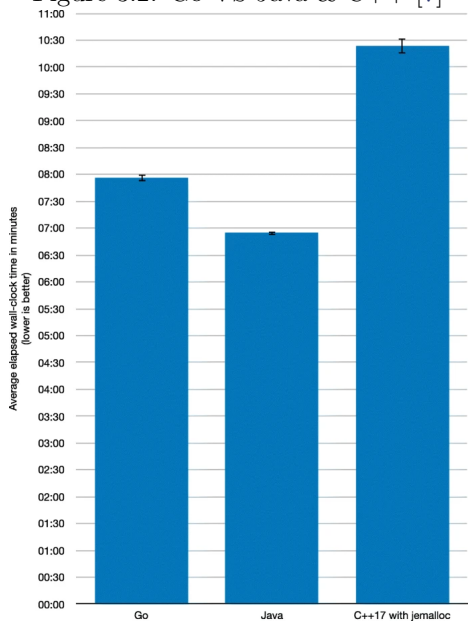
3.2.2 How OpenAPI fitted the API's design

Swagger Tools was highly used for prototyping the project's, server and client API stubs along with their data models that resembled the data in the project's database. With Swagger, from OpenAPI specs, server stubs can be easily designed with the necessary paths, HTTP methods, handler functions and response status codes and messages. To follow the server's stubs, the client stubs can be generated with the integrated design to allow the client to talk back and forth to the server. Swagger made the process of designing the project's OpenAPI specification a structured and straightforward task. It proved to be extremely useful to design an API efficiently especially with the specification's documentation and an easy to use tool for a first-time user.

3.3 Golang

Golang (Go), designed by Google, is inspired by the style of Object-Oriented Programming (OOP) Languages. Go is a relatively new language (first version released on November 10, 2009), open-source and becoming widely used. Go is primarily based on the C programming language and has compile speeds that can be as fast as Java and C++ [7]. Figure 3.4 is from an article that compared the languages Java, C++ and Go. Go's creators desired the security and performance of Java and C++, but with the "lightness and fun" of a dynamically typed interpreted language such as Python. Go's syntax is influenced by C with a mixture of Pascal. [8] When it comes to OOP, Go is especially great for refactoring and abstraction. Go does not use classes like Java does; the closest element to a class in Go is a Struct that can be used for object-like data models. [9] Go files (controllers) can easily pick up functions from others as long as they are in the same package. A function can be abstracted from one controller, put into another, and still be called from in the original controller with no changes or additions. If a function is in another package, it can easily be used outside with an import of that package and a variable declaring that function. Go also employs encapsulation with local variables inside functions rather than outside as they would still be available to other controllers in the same package.

Figure 3.2: Go VS Java & C++ [7]



The main idea of Go is to reduce the development of large servers with big machines, many cores, and developers and bring it down to the use of simple tools that can do all those jobs. [10] Identifying problems in Go is a more straight-

forward process as the tools are simple, allowing concentrating on the problem rather than how to solve it. With the Go SDK installation (Software Development Kit), Go's documentation can be obtained from the CLI (Command Line Interface) with the command 'go doc'. For example, simply typing the command 'go doc fmt' into the CLI lists all the documentation on formatting print statements (Figure 3.4). Go doc can also list all the functions inside a project's package (Figure 3.5). With the command 'godoc -http=:6060', a localhost of Go's complete documentation can be accessed, which would provide great use if there is a weak/no internet connection.

Figure 3.3: CLI Go Docs on fmt

```
C:\Users\johnd\OneDrive\Desktop\Home\_Projects\Repota-App\horton\go>go doc fmt
package fmt // import "fmt"

Package fmt implements formatted I/O with functions analogous to C's printf
and scanf. The format 'verbs' are derived from C's but are simpler.


Printing

The verbs:

General:

    %v  the value in a default format
        when printing structs, the plus flag (%+v) adds field names
    %#v a Go-syntax representation of the value
    %T  a Go-syntax representation of the type of the value
    %%  a literal percent sign; consumes no value
```

Figure 3.4: Go Functions inside the Back-end's openapi Package

```
C:\Users\johnd\OneDrive\Desktop\Home\_Projects\Repota-App\horton\go>go doc
package openapi // import "github.com/GIT_USER_ID/GIT_REPO_ID/go"

func CORS() gin.HandlerFunc
func CheckForCookie(c *gin.Context) bool
func CreateReport(c *gin.Context)
func DeleteReport(c *gin.Context)
func GetCarApiData(c *gin.Context)
func GetReportById(c *gin.Context)
func GetReports(c *gin.Context)
func Index(c *gin.Context)
func Login(c *gin.Context)
func Logout(c *gin.Context)
func NewRouter() *gin.Engine
func Register(c *gin.Context)
func RegisterNewUser(c *gin.Context, username, name, password string) error
func UpdateReport(c *gin.Context)
type Route struct{ ... }
type Routes []Route
```

3.3.1 Go Web Frameworks

Gorilla Mux

Gorilla Mux is a package that implements the combination of a request router and dispatcher for assigning incoming requests to the designated handler. Mux stands for "HTTP request multiplexer" which is used for incoming HTTP requests are compared to a list of the registered routes and the path that meets the URL or other requirements such as a datatype ID for accessing a complete source of data by its unique identifier. [11]

Gin Gonic

Gin Gonic has a Martini-like API but with an efficiency that is said to be 40 times faster than Martini. Martini is another Go web framework, which is no longer maintained. Gin is fast with its "Radix tree-based routing, small memory footprint. No reflection. Predictable API performance." [12] A chain of middlewares can handle incoming HTTP requests. Gin's error management makes it simple to gather all of the errors from responses during a HTTP search. Gin allows the creation of custom HTTP response errors (Figure 3.7). Gin needs to be set as a parameter of a function to respond with these errors to implement these' custom errors'. Suppose a panic occurs in an HTTP request. In that case, Gin can recover it, making it "Crash-free". [12] Also, according to an 'Expert analysis' from a JetBrains, GoLand Blog by Ekaterina Zharova, 41% of Go developers use Gin as their main framework for back-end development. [13] Making it the de facto of Go's web frameworks.

When an API requires endpoints to share a common path, there is a limitation with Gin. For example 'api/movies/:movieId' and 'api/movies/:releaseDate' will result in a issue with Gin. Having colliding paths in APIs is common, and they can usually switch over and back, but with Gin, they can not. However, depending on the paths required for an API, if the paths are for basic CRUD operations (CREATE, READ, UPDATE AND DELETE), this should not be much of an issue. It becomes an issue when an API needs multiple operations other than CRUD which, Gin does not meet the requirements. For example, a search box on an app would be limited to just one data field, limiting users' convenience. On top of that, it limits the 'RESTfulness' of an API.

Figure 3.5: Gin Error Handling

```
if err != nil {
    c.JSON( code: 403, models.Error{Code: 403, Messages: "User is unauthorized"})
    return false
}
```

3.3.2 How Go fitted the Back-end

Back-ends can be developed in many languages, including Java, Python, Ruby, Node.js, and Rust. Go stood out from the others for the back-end for how it handles compile times, syntax, documentation, its use of OOP, and it seemed like a great language to know to expand development skills.

Learning Go for the first time for the back-end was a welcome challenge. It took some time to become comfortable, and the use of 'go doc' made that process smooth. The central part of this challenge was finding a suitable Web Framework that had all the requirements needed. Gorilla Mux was used for the initial setup but failed to effectually meet the requirements to deal with cross-origin issues of HTTP methods for every route with the front-end. Gin Gonic proved to be the best framework to use, but it required Docker to generate the OpenAPI's server's stubs rather than Swagger's standard options. After finalizing the back-end, it can be said that Go is a very nice language to know as it is quite object-oriented and has efficient compile times.

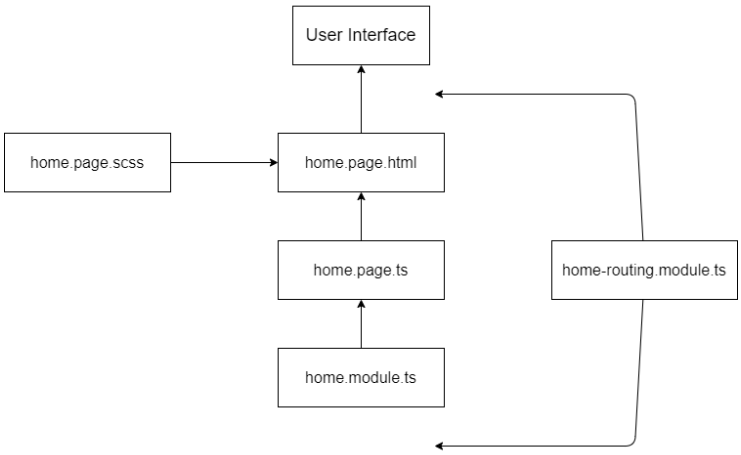
3.4 Angular & Ionic

There are many JavaScript/TypeScript frameworks out in the web development world, including Angular, React, Vue.js, Meteor and Ember.js. These technologies are primarily JavaScript frameworks. A new JavaScript framework appears almost every year. Some of these frameworks can become obsolete as there is always a new one coming right around the corner.

Angular is a TypeScript-based framework for User Interfaces (UI) that has stayed on the market throughout the years [14] and with the combination of Ionic, can make web apps into mobile apps. Also, similar to Go, it was developed by Google after its predecessor AngularJS. Angular has a mighty CLI, which can generate a whole template of an app with routing, testing, and a MVVM (Model-View-ViewModel) for each component. Also, with the CLI, a complete app can be built for production, which translates all the app's code into JavaScript.

Angular is relatively slow to initially compile apps, which is understandable as apps can contain many pages. However, they can be updated once they are up and running without having to rerun them. Angular apps are created with MVVM components, and Ionic turns them into pages. Figure 3.8 shows the structure of a page. There are a lot of benefits of using Angular to build apps such as routing, components and it's built in ngIf, ngFor and ngOnInit, which the following subsections will go into detail.

Figure 3.6: Angular & Ionic Home Page Structure



UI Structure and Styling

- home.page.html & home.page.scss

Controller of UI: Buttons, Input boxes ETC.

- home.page.ts

Imports and Dependencies

- home.module.ts

Path Routing to connect the page to the App's main routing

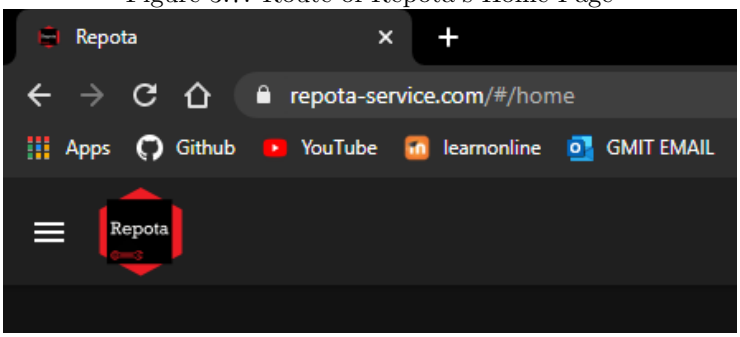
- home-routing.module.ts

Routing

When an Angular and Ionic app is first initialized, template pages are set up. Every page has a routing module that connects to the main app routing module (app-routing.module.ts). This module allows for complete access to navigation through routes. For example, the home page would be on the 'www.app.com/home' route (Figure 3.9). These routing endpoints are similar to what a RESTful API would use. Angular's router setup is quite powerful. With it, specific routes can be blocked to unauthorized users using Angular's 'canActivate' method, while providing a complete structure to an app. Normally with HTML, the 'href' tag is used for navigation over multiple pages causing the app to reload every time a user navigates to a different page, affecting performance and user experience. With Angular's routing, the 'routerLink' tag can be used to go from page to page without reloading the entire app just for one page. Also, with routerLink, an specific data field such as an ID can be loaded

over to the next page to display the data in its entirety. On top of all that, when a user types in a none existing page of the app in the browser's search bar, the router will navigate them to the default visiting page of the app. [15]

Figure 3.7: Route of Repota's Home Page



Components & TypeScript

Components are controllers for the UI (HTML + SCCS). Components can include various tools for the UI such as forms, buttons, loading in data from an API by the built-in ngOnInit method and navigation. Angular uses TypeScript to turn TypeScript's classes into components. TypeScript by itself cannot compile. It has to be generated and then complied with JavaScript to run. This may ask the question, "Why TypeScript? and not JavaScript?". TypeScript's compiler can catch errors/bugs in advance where as JavaScript only notifies them once it is compiled and running. With TypeScript, errors can be avoided for a production build of an app. TypeScript is built off JavaScript, meaning that JavaScript code can be written in TypeScript with no changes. TypeScript is also quite object-oriented, allowing interfaces to define types for data object models. [16]

ngIf & ngFor

Angular uses ngIf as an if statement that can be used across DOMS (Document Body Object). For example, a ngIf can be used if a button on the UI is clicked to load data. ngFor is used as a for loop on the DOM that can loop through arrays of data to avoid DRY code (Don't Repeat Yourself). Meaning that data types can be declared in one HTML div or an Ionic item tag to present multiple data elements on the UI.

3.4.1 How Angular & Ionic fitted the Front-end

Before this project, Angular and Ionic frameworks have proven successful in creating apps for other modules and personal interests. The frameworks with their modules and routing helped to provide smooth navigation throughout the

app for users using buttons and a hamburger menu. While the component/page structure allowed the controllers, HTML, and SCCS to be sufficiently organized throughout the entire front-end. ngFor is used throughout the front-end to present data on the UI from the back-end from the ngOnInit methods. Ionic proved to be very useful as it immensely helped make the front-end responsive for mobile phones using the mobile phone simulator Ionic Lab.

3.5 MySQL

MySQL (Structured Query Language) uses the ACID model in its database management system for relational databases (RDBMS). It is the most common open-source database in the world because it is efficient, simple to set up and use. [17] In addition, MySQL is quite robust, both in its setup for database tables and its functionality.

3.5.1 Relational

Relational can reduce duplication and redundancy within databases. Rather than storing all of the data in one big storeroom, RDBMS stores it in individual and different tables. Physical files are used to organize the database structures, which are designed for pace. The logical model provides a versatile programming environment with artifacts such as databases, tables to organize data, rows, and columns to separate the data inside tables. [18]

3.5.2 ACID

The ACID model stands for the database design principles; Atomicity, Consistency, Isolation and Durability. Atomicity is used in InnoDB transactions with statements including 'COMMIT' and 'ROLLBACK'. Transactions are a way of inserting into/updating multiple tables at one time (Figure 3.10). When two tables are linked by foreign keys, transactions are typically used. Consistency protects data from crashes by having transactions broken down into parts. When several transactions are making adjustments and running queries simultaneously, Isolation fine-tunes the balance between efficiency and reliability, accuracy, and reproducibility of results. Finally, Durability which, stores only the successful transactions in the database. [19]

Figure 3.8: MySQL InnoDB Transaction for inserting data into two tables.

```
START TRANSACTION;
INSERT INTO jobreports
    (job_report_id, worker_id, date_stamp, vehicle_model, vehicle_reg, vehicle_location, miles_on_vehicle, warranty,
     breakdown, cause, correction, parts, work_hours, job_report_complete)
     VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);
INSERT INTO customers (job_report_id, customer_name, customer_complaint) VALUES (LAST_INSERT_ID(), ?, ?);
COMMIT;
```

3.5.3 How MySQL fitted the Database

MySQL was chosen for many reasons. MySQL definitely seemed like the best fit. As discussed in the introduction, Repota is designed for automobile technicians (workers). In a database, workers need their own dedicated table. For any app, website, system for a company of any sort, there needs to be storage for their workers. If the data were all in one place, it would be a mess, and worker information would get lost right in the middle of it. Having separate, connected tables for data was a necessity for the project. Workers are stored in one table. That table then connects to the service reports table which, is then connected to the customers' table. Keeping dedicated tables for workers and customers was seen as a must as their information if very important for the service reports. Given that the reports consist of report and customer information, transactions are perfect to create new reports without hassle. MySQL 'JOIN QUERIES' allow selecting data from multiple tables to present the data as if all the data was originally in one table. Since the complete reports consist of data from two other tables, JOIN QUERIES were the way to go. From these observations, it was quite obvious, MySQL was perfect for the database.

3.6 Amazon Web Services

AWS (Amazon Web Services) is a cloud platform that offers a variety of services. Some of these services are EC2, Route 53, Machine Learning, Elastic Beanstalk, S3 Bucket, Amplify and CloudFront. Deploying to AWS can be as simple as uploading source code files. AWS has its own CLI which, generally works better for deployment. Uploading files can overload the service. Should a significant update needed to be deployed, previous files would first need to be deleted before uploading any new files. With AWS's CLI, command modifiers such as '–delete' can be used during deployment. This modifier deletes any files and deploys the updated ones with ease.

3.6.1 Virtual Machines

Virtual Machines (VM) are virtual emulations of computer environments. [20] For example, VMs allow having a Linux Operating System (OS) accessible from the CLI of a Windows computer with no installation needed. From the Windows side, the connection is generally made in a PowerShell or GIT Bash CLI. AWS's EC2 (Elastic Compute Cloud) service handles VMs as instances. EC2 provides the service to have many VMs with different operating systems.

Windows VS Linux

Since Window's OS is a commercial product, only the developers who developed it have the authority to alter it. The Linux OS is open-source, allowing users to configure their environments to their taste. Linux is considered to be faster than the most recent Windows versions. Linux is extremely stable because it

is simple to find and repair bugs, and in contrast, Windows is vulnerable to viruses and malware because of its vast user base. [21]

3.6.2 S3 Bucket

S3 (Simple Storage Service) provides object storage. Objects are the fundamental entities stored in S3. With S3 Bucket, web apps can be stored and hosted. To deploy/upload to a Bucket source code files can be upload or with the AWS CLI. [22] By default, these Buckets are HTTP, meaning they are not secure for data transfers. Buckets can become secure (HTTPS) using AWS's Route 53, Certificate Manager, and CloudFront. Route 53 allows to setup hosted zones for a custom web domain. Route 53 takes a domain and routes it to an AWS service such as a S3 Bucket or an Elastic Beanstalk. With CloudFront, that custom domain can become secure. Certificate Manager can issue SSL (Secure Sockets Layer) certificates to domains. That SSL certificate can then be set up with CloudFront. From here, Route 53 can then be used to point the Bucket to the secure CloudFront domain.

3.6.3 Elastic Beanstalk

Elastic Beanstalk (EB) is a service for the deployment and scaling of web apps and services. These web apps and services are usually developed in Java, Ruby, .NET, PHP, Node.js Python, Go and Docker. [23] Docker can be used through EB. The addition of Docker and the EB CLI makes deploying to EB a satisfying process. For deployment to EB with Docker, all that is required is a Dockerfile. When deploying to EB with its CLI, it is able to pick up that there is a Dockerfile in a source code's directory. In deployment, the Dockerfile copies over the source code and pushes it to EB to host. Similar to S3 Buckets EBs are HTTP by default. HTTP can be overcome with a somewhat similar approach to creating a secure Bucket. In the EB configuration settings, a load balancer can be set up to be on a HTTPS port with a SSL certificate. Then Route 53 can be used to point a custom domain to that EB. EB uses Environments to host web apps. These environments have logs to access console outputs, warnings, or information from the web app.

Docker

Docker is a series of PaaS (Platform as a Service) products that deliver applications and in 'containers,' as Docker calls them, using OS-level virtual machines. A container is a software packaging file format that encapsulates all of an application's code and dependencies in a single format that allows it to run easily and efficiently across multiple computing environments. [24]

3.6.4 How AWS fitted the project's hosting

AWS seemed like the best fit for hosting as it manages its services very well. The UI itself is well structured and easy to navigate for a cloud platform with several services. The project's database needed to be on a secure server. AWS's EC2 service was opted for the database, hosted on a secure Ubuntu VM. Initially, the project's back-end was also hosted on this VM for prototyping reasons. When the back-end's structure was finalized, it was deployed to EB through Docker. Once the front-end had its core functionality, it was hosted through AWS's S3 Bucket. The back-end and front-end are both HTTPS thanks to AWS's Route 53, CloudFront, and Certificate Manager.

Chapter 4

System Design

4.1 The Database

The MySQL database is the core foundation of Repota. The database stores workers, sessions, job reports (service report) and customer details in their own respected tables. In the database everything is connected to a worker.

Workers & Session

The workers table is in relation of a users table and consists of a worker_id, user_name, worker_name and a hash (password). The ID is set up to be automatically incremented, meaning when a new worker is inserted to the table an ID does not have to be specified. Their username is set as unique to avoid data collisions with other workers in the database. Their worker name is set as as the unique key as this name is to be presented with all their reports on the front-end. Their ID connects them to the session table making it a foreign key. Here the session ID is set up to be a UUID (Universally Unique ID). An expiry time field is setup for user's login sessions. The session ID and expiry time are also used as the set up for browser cookies.

Job Reports & Customers

Workers are connected to their reports by their ID making this a foreign key for the jobreports table. The report's ID is a foreign for the customers table. The jobreports table holds quite a bit of information. This information is split into fourteen fields. The customers table consists of a customer_id, job_report_id (foreign key), customer_name and customer_complaint. The following describes how the tables; workers, jobreports and customers all link together.

Full Report Details

A full report (Figure 4.1) required a lot of information which is in sixteen components. Meaning the information had to spread out between tables to have them all still link to the workers and customers to avoid one large table and still keep the workers and session table intact. All this information is obtained by quite a large JOIN Query (Figure 4.2).

Figure 4.1: Example of a Full Report

job_report_id	121
date_stamp	2021-04-15
vehicle_model	Ford Focus
vehicle_reg	151-DL-2308
miles_on_vehicle	508538
vehicle_location	Gort, Co. Galway
warranty	1
breakdown	0
customer_name	Freddie Quell
customer_complaint	The passenger side door will not open.
cause	The lock on the passenger door was broken.
correction	A new lock has been fitted.
parts	1 DOOR LOCK
work_hours	1
worker_name	John Shields
job_report_complete	1

Figure 4.2: JOIN QUERY for all Report information

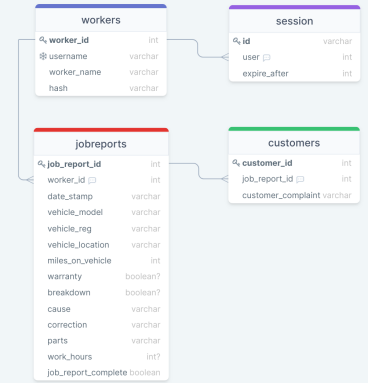
```
-- ALL REPORTS --
SELECT DISTINCT jr.job_report_id, jr.date_stamp, jr.vehicle_model, jr.vehicle_reg, jr.miles_on_vehicle,
       jr.vehicle_location, jr.warranty, jr.breakdown, cust.customer_name, cust.customer_complaint,
       jr.cause, jr.correction, jr.parts, jr.work_hours, wkr.worker_name, jr.job_report_complete
    FROM jobreports jr
    INNER JOIN customers cust
        ON jr.job_report_id = cust.job_report_id
    INNER JOIN workers wkr ON jr.worker_id = wkr.worker_id WHERE wkr.worker_id = ?;
```

Refactoring

The first refactor was due to duplicate results coming from JOIN QUERYs. Even though the data was unique the query had trouble selecting all the report's data. The reports ID were then altered to be drastically different having them is the hundreds instead of 1-2-3-4-5 and the duplicate problem was solved. The jobreports table was originally split into two tables, jobreports, and workdone. When it came to implementing the CRUD operations in the back-end, it made a lot more sense to have this information in one table as it would have made for some monstrous queries for updating and creating. In the old jobreports table, the fields warranty and breakdown were originally varchars for a simple "YES" or "NO". With this refactoring, these fields were changed to booleans

along with the addition of 'job_report_complete' to allow workers to keep track of their complete and incomplete reports. The figure below shows the finalized database design.

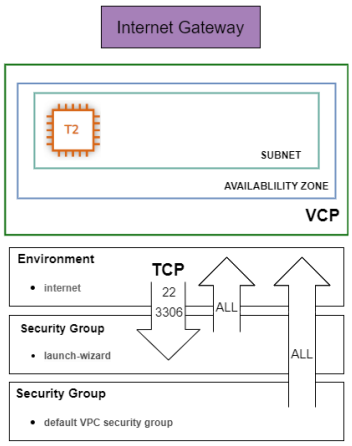
Figure 4.3: Database Schema



Database Hosting

The database is hosted on a AWS EC2 Ubuntu Virtual Machine as a server. This provides a virtual private cloud (VPC) for it with transmission control protocol (TCP) for the ports 22 (SSH login (Secure Shell)) and 3306 (MySQL). This holds the database in a secure environment which only the back-end has access to though a configuration file with the MySQL details. Therefore the data on workers, reports and customers is kept safe. The figure below show the entire set of the server.

Figure 4.4: AWS Server



4.2 The Back-end

Repota depends on the robust and quick back-end nicknamed 'Horton'. Horton handles all user registration, login, logout, sessions, report CRUD operations, and has access to a 3rd Party API named 'Back4App' for vehicle information. Without him Repota would be quite limited in terms of the service it provides.

4.2.1 Horton API Specification

Horton's API specification was designed using the OpenAPI tools from Swagger and is written in YAML. The specification design consists of RESTful routes, operationIds, HTTP methods, status codes and schema models. The specification worked as a prototype to set up the necessary functions and data models for users and their reports. Swagger provided HTML documentation of Horton's specification. This documentation was extremely useful for when it came to implement the main features. The documentation is hosted using GitHub pages. This was mainly to provide easy access. However it can be useful to other developers if they require to develop an API for a service report app or even for a user account system. The documentation displays the API's methods, Login, Register, createReport, deleteReport, getReportbyId, getReports and updateReport. It also provides code samples for said methods with an example of a response HTTP status code and message. Schema Models are also presented to show the data requires in them. The documentation URL is provided below.

Horton API Documentation:
https://johnshields.github.io/horton.api.doc/

Operations

In OpenAPI specification, operations are identified by operationIds which is a unique string identifier. These operations are templates of CRUD operations with HTTP methods, e.g., POST, GET, DELETE, PUT. Operations contain tags, a description, parameters such as an ID, a request body for data, a response status, headers, login requirements and a URL. The user account system and report system were both designed initially as operationIds with their respected paths. Figure 4.5 shows how the login operation was designed. All these operations provided the necessary templates for the handler functions of each path.

Figure 4.5: Login Operation

```yaml
/login:
  post:
    tags:
    - Account
    summary: Log in
    description: Attempts to log a user in
    operationId: login
    requestBody:
      $ref: '#/components/requestBodies/inline_object'
    responses:
      "204":
        description: Login successful
        headers:
          Set-Cookie:
            description: Returns a new session_id to be used for all future requests
            style: simple
            explode: false
            schema:
              type: string
      "403":
        description: Login Unsuccessful
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Error'
            examples:
              LoginFailed:
                value:
                  code: 403
                  messages: Invalid Name and/or password
      "500":
        description: Unable to process request
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Error'
            examples:
              InternalServerError:
                value:
                  code: 500
                  messages: Unable to process request
    security:
    - {}
```

RESTful Routes

Horton's designed routes consist of two paths for user registration and login. For CRUD operations on reports there are five paths (Figure 4.6). These paths provided structure for the API and for HTTP methods.

Schema Models

Figure 4.7 shows the finalized models that were designed to represent user details, status codes with messages and reports. Implementing the API's design resulted in refactoring the initial models designed to accommodate the databases changes and an addition of three other routes for a logout, a 3rd Party API and an Index Handler, which will be discussed in the next section.

Figure 4.6: API Paths

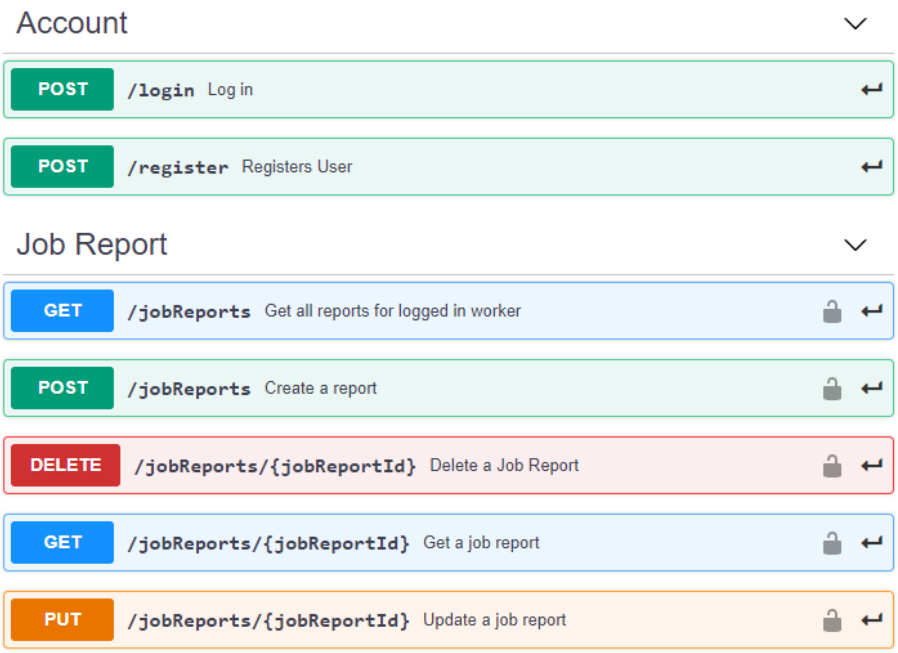


Figure 4.7: Schema Models

4.2.2 Go Horton

Horton himself is written in Go using the web framework Gin Gonic to design and implement him into a robust API. When the API specification had the core fundamentals, it was time to generate the prototype code stubs. Swagger does not provide an option for Gin; this resulted in using another OpenAPI generator with Docker. This prototype set up the necessary functions for registration, login, and all report CRUD operations. The remainder of this section conveys how the final product has come a long way from these generated stubs.

Data Models

The data models from the API specification, 'InLineObject' and 'JobReport', required some light refactoring. InLineObject is for new users in the system. This model now consists of a username, name, and password. The name is now included as for the reports required a name to be displayed with all the other details. For the JobReport model, 'Warranty', 'Breakdown' and the addition of 'JobComplete' were all set as int values (to accommodated the database changes of booleans) to allow them to be check-box values on the front-end. New models were created for login sessions and a 'worker account'. These models are used to send JSON data to the front-end. See the figures below for the Go Structs of the finalized models.

Figure 4.8: InLineObject Struct

```
type InlineObject struct {
    Username string `json:"username,omitempty"`

    Name string `json:"name,omitempty"`

    Password string `json:"password,omitempty"`
}
```

Figure 4.9: Session Struct

```
type Session struct {
    Token  string
    Expiry int
}
```

Figure 4.10: WorkerAccount Struct

```
type WorkerAccount struct {
    Id         int
    Username   string
    WorkerName string
    Password   string
}
```

Figure 4.11: JobReport Struct

```
type JobReport struct {
    JobReportId int32 `json:"jobReportId,omitempty"`

    Date string `json:"date,omitempty"`

    VehicleModel string `json:"vehicleModel,omitempty"`

    VehicleReg string `json:"vehicleReg,omitempty"`

    VehicleLocation string `json:"vehicleLocation,omitempty"`

    MilesOnVehicle int32 `json:"milesOnVehicle,omitempty"`

    Warranty int32 `json:"warranty"`

    Breakdown int32 `json:"breakdown"`

    CustomerName string `json:"customerName,omitempty"`

    Complaint string `json:"complaint,omitempty"`

    Cause string `json:"cause,omitempty"`

    Correction string `json:"correction,omitempty"`

    Parts string `json:"parts,omitempty"`

    WorkHours int32 `json:"workHours,omitempty"`

    WorkerName string `json:"workerName,omitempty"`

    JobComplete int32 `json:"jobComplete"`
}
```

Routers

The API now includes ten paths which each have their respected handler functions. These paths are all managed in routers.go. The API's main function boots up these 'routers' after the CORS requirements have been set (Figure 4.12). The following are the handler functions and their paths.

Figure 4.12: CORS function

```
// CORS - To handle cross origin issues.
func CORS() gin.HandlerFunc {
    return func(c *gin.Context) {
        c.Writer.Header().Set( key: "Access-Control-Allow-Origin", c.Request.Header.Get( key: "Origin"))
        c.Writer.Header().Set( key: "Access-Control-Allow-Credentials",  value: "true")
        c.Writer.Header().Set( key: "Access-Control-Allow-Headers",  value: "Origin, X-Requested-With, Content-Type, Accept, Authorization")
        c.Writer.Header().Set( key: "Access-Control-Allow-Methods",  value: "GET, HEAD, POST, PUT, DELETE, OPTIONS, PATCH")
```

Handler functions & their paths

- Index - /api/v1/
- Register /api/v1/register
- Login - /api/v1/login
- Logout - /api/v1/logout
- GetReports - /api/v1/jobReports
- GetReportById - /api/v1/jobReports/ID
- CreateReport - /api/v1/jobReports
- UpdateReport - /api/v1/jobReports/ID
- DeleteReport - /api/v1/jobReports/ID
- GetCarApiData - /api/v1/carApiData

The following sections will go into detail on how the handler functions work with their paths and models.

4.2.3 Account & Session System

Registration

When a user registers, they become a 'worker' as the app is revolved around a work environment. Users POST a request to register by their username, name, and password which relies on the InLineObject model. The user's username and name are checked with the function isValidAccount to ensure they do not already exist. The password gets hashed using Go's package 'bcrypt', which implements the Provos and Mazières's bcrypt adaptive hashing algorithm. [25] After the details check and the password hashing, the new user is inserted into

the database, and a session ID (token) is generated with the Session model and is inserted into the database for them (Figures 4.13-4.17). Once a user is in the database, they are uniquely identified by their worker ID. This ID is also attached to their session in the database (Figure 4.18). This session is designed for Cookies to let the user make all future requests to the API.

Figure 4.13: Register

```
func Register(c *gin.Context) {
    var user models.InlineObject

    // Bind user's data to object, else throw error.
    if err := c.BindJSON(&user); err != nil {
        log.Println(err.Error())
        c.JSON( code: 500,  obj: nil)
    }

    username := user.Username
    password := user.Password

    // Register new user and hash the password in RegisterNewUser.
    if err := RegisterNewUser(c, username, user.Name, password); err == nil {

        // Create a session for the new user.
        err, session := createSessionId(username)

        if err != nil {
            log.Print(  v...: "Failed to Register User.", err)
            c.JSON( code: 500,  obj: nil)
        } else {
            c.JSON( code: 200, session) // Session has been created for user.
        }
    } else {
        // Issue with user's details - username or name taken.
        log.Println( v...: "\nUnable to complete request")
    }
}
```

Figure 4.14: RegisterNewUser

```
func RegisterNewUser(c *gin.Context, username, name, password string) error {
    db := config.DbConn()

    // Check if password is null or if username is taken.
    if strings.TrimSpace(password) == "" {
        log.Println( v...: "\nPassword is null")
        return errors.New( text: "password is null")
    } else if isValidAccount(username) {
        c.JSON( code: 409, models.Error{Code: 409, Messages: "Username is already taken"})
        log.Println( v...: "\nUsername taken")
        return errors.New( text: "username is already taken")
    }

    // Hash the password here using golang.org/x/crypto/bcrypt.
    hashedPassword, err := bcrypt.GenerateFromPassword([]byte(password), bcrypt.DefaultCost)
    if err != nil {
        c.JSON( code: 500,  obj: nil)
        log.Fatal( v...: "\nHash Password failed: ", err)
    }

    // Insert new user in the workers table.
    insert, err := db.Prepare( query: "INSERT INTO workers(username, worker_name, hash) VALUES (?, ?, ?)")
    if err != nil {
        c.JSON( code: 500,  obj: nil)
        log.Println( v...: "\nMySQL Error: Error Preparing new user account:\n", err)
    }

    result, err := insert.Exec(username, name, hashedPassword)

    // Return MySQL error if there is a duplicate entry.
    // Most likely the name as password and username have already been checked.
    if err != nil {
        c.JSON( code: 409, models.Error{Code: 409, Messages: "Please make your name more unique"})
        log.Println( v...: "\nMySQL Error: Duplicate entry:\n", err)
        return err
    }
    fmt.Println( a...: "\n[INFO] Printing MySQL Results for user account...\n", result)

    defer db.Close()
    return nil
}
```

Figure 4.15: isValidAccount

```
// Function to do a database look up and check if a username matches one provided.
// Mainly used to check if a user tries to register with a taken username.
// And for selecting users for report functions in API Job Report.
func isValidAccount(username string) bool {
    db := config.DbConn()

    // Check username from workers table.
    selDB, err := db.Query( query: "SELECT * FROM workers WHERE username=?", username)

    if err != nil {
        log.Fatal(err) // error with Query.
        return false
    }

    // Check to see if a true user exists in the table, if not return false.
    if selDB.Next() {
        err = selDB.Scan(&wa.Id, &wa.Username, &wa.WorkerName, &wa.Password)

        if err != nil {
            // No matching username in table (user does not exist).
            log.Println( v...: "\nMySQL Error - no matching username:\n", err)
            return false
        }
        defer db.Close()
        return true
    } else {
        defer db.Close()
        return false
    }
}
```

Figure 4.16: createSessionId

```
func createSessionId(username string) (error, models.Session) {
    db := config.DbConn()

    // Set up the session requirements.
    token := generateSessionId() // Create a new session ID
    expiry := 3600 * 24 * 3      // 3600 * 24 * 3 = 3 days

    // Prepare Insert Query to create session for user.
    insert, err := db.Prepare( query: "INSERT INTO session(id, user, expire_after) VALUES(?, ?, ?)")

    if err != nil {
        fmt.Println(err.Error())
        return errors.New(err.Error()), models.Session{}
    }

    fmt.Println( a...: "\n[INFO] Printing Session Record...", "\nSession Token:", token, "\nWorker ID:", wa.Id,
        "\nExpiry time in seconds:", expiry)

    // Check if user account exists - mainly for attaching session to user.
    if !isValidAccount(username) {
        log.Println( v...: "\nUser has not logged in.", err)
    }

    // Execute query to db (create session for user), handle errors if any.
    if _, err = insert.Exec(token, wa.Id, expiry); err != nil {
        log.Println( v...: "MYSQL Error: Error creating new session record\n", err)
        defer db.Close()
        return errors.New( text: "MYSQL Error: Error creating new session record"), models.Session{}
    } else {
        fmt.Println( a...: "\n[INFO] Session has been generated for User")
        defer db.Close()
        // Returns Session object.
        return nil, models.Session{Token: token, Expiry: expiry}
    }
}
```

Figure 4.17: generateSessionId

```
// Function to create a session ID using UUID (Universally Unique ID) for an authenticated user.
// This ID will be needed for users and their cookies to allow the user to make requests from the client to the server.
func generateSessionId() string {
    return uuid.New().String()
}
```

Figure 4.18: User in the workers and session tables

	worker_id	username	worker_name	hash
1	141	john_shields	John Shields	$2a$10$ttINUB.yZkZUKiKSBqRMf.jzRYIL8.MLMldre63SA5u9DtJjuvMNO

	id	user	expire_after
1	3593ca97-b45d-4511-820e-835f068588f3	141	259200

Login

A user logs in by their username and password which works with the Worker-Account model. This data gets sent through a POST request that checks to see if the username exists in the database, then hashes the entered password and compares it to the one in the database. If the user has an existing session that one is removed and replaced by a new one. If their details are correct and the new session has been created, a cookie of three days is set for them then, the user has been logged in successfully (Figures 4.19-4.21).

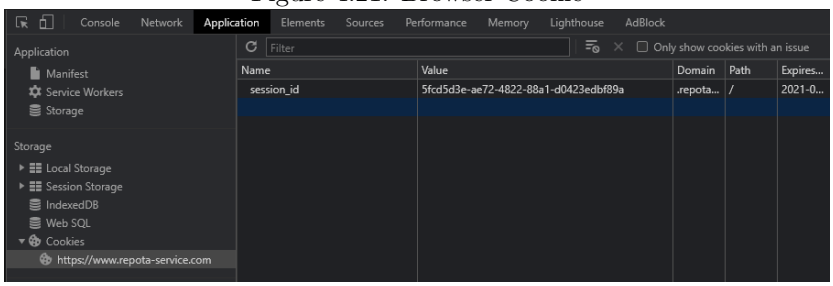
Figure 4.19: Login

```
func Login(c *gin.Context) {
    db := config.DbConn()
    // Object to bind user data too.
    var workerForm models.WorkerAccount

    if err := c.BindJSON(&workerForm); err != nil {
        fmt.Println(err.Error())
    }

    username := workerForm.Username
    password := workerForm.Password

    // Check if user exists in the database and check password is not null.
    if err := verifyDetails(username, password); err != nil {
        c.JSON( code: 403, models.Error{Code: 403, Messages: "Username does not exist"})
        return // Return as there is issues with the username or password.
    }
    // Compare the hash in the db with the user's password provided in the request using golang.org/x,
    if err := bcrypt.CompareHashAndPassword([]byte(wa.Password), []byte(password)); err == nil {
        // Check for existing session, remove if one exits.
        if removeSession(wa.Id) {
            // Create new session ID for user who logged in.
            err, session := createSessionId(username)

            if err != nil {
                log.Print(err)
                c.JSON( code: 500, models.Error{Code: 500, Messages: "Unable to create new session"})
            } else {
                // Set a cookie for logged in user.
                c.SetCookie( name: "session_id", session.Token, session.Expiry,  path: "/",
                     domain: "repota-service.com",  secure: true,  httpOnly: false)
                // User has been logged in and cookie has been set.
                c.JSON( code: 204,  obj: nil)
            }
        } else {
            log.Println( v...: "Unable remove old session", err)
            c.JSON( code: 500, models.Error{Code: 500, Messages: "Unable to remove old session"})
        }
    } else {
        log.Println( v...: "Password is incorrect for User", err)
        c.JSON( code: 401, models.Error{Code: 401, Messages: "Password is incorrect"})
    }
    defer db.Close()
}
```

Figure 4.20: verifyDetails

```
// Function to check password for null and if user exists when users login
// With the help of isValidAccount.
func verifyDetails(username, password string) error {
    if strings.TrimSpace(password) == "" {
        return errors.New( text: "password is null")
    } else if !isValidAccount(username) {
        return errors.New( text: "username does not exist")
    } else : nil ;
}
```

Figure 4.21: Browser Cookie



Logout

A user is logged out by removing their current session and replacing it with a new one. The new session is set to expire after one second. A new cookie is then set for the user with this session, and then the user is logged out after the cookie expires the one second (Figure 4.22). This was mainly to accommodate how browser cookies work. Only users have control of removing their cookies, and setting a new cookie of one second was the best way to log them out as a user requires a cookie to make all report requests to the API. In the session controller, there is a function for cookie checking (Figure 4.23). The purpose of this function is to block requests on reports if a user have no cookie. In other words an unauthorized user.

Figure 4.22: Logout

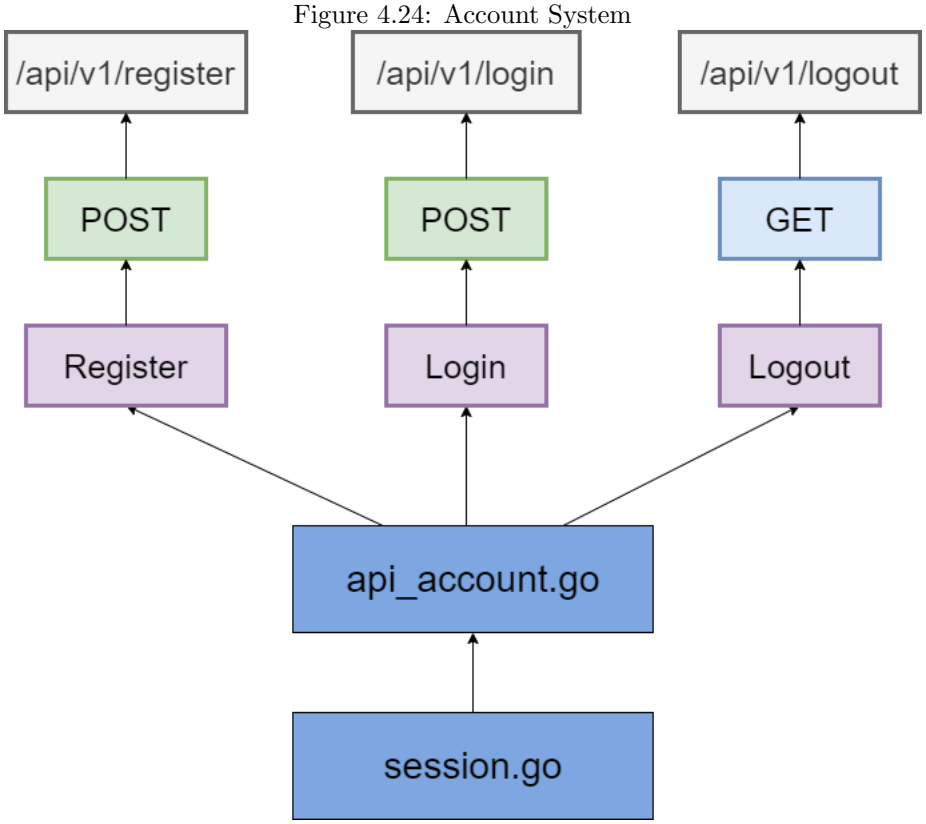
```
// Logout
// Works with removeSession & createSessionId to remove the user's current session.
// Then creates a new one that expires in one second.
// Set a new Cookie with the new session to logout the user after one second.
func Logout(c *gin.Context) {
    db := config.DbConn()
    username := wa.Username
    // Check for existing session, remove if one exits.
    if removeSession(wa.Id) {
        // Create new session ID for user who logged out.
        err, session := createSessionId(username)

        if err != nil {
            log.Println( v...: "Could not logout User", err)
            c.JSON( code: 500, models.Error{Code: 500, Messages: "Unable to logout User"})
        } else {
            // Set a cookie of one second for logged out user.
            c.SetCookie( name: "session_id", session.Token,  maxAge: 1,  path: "/",  domain: "repota-service.com",  secure: true,  httpOnly: false)
            c.JSON( code: 204, models.Error{Code: 204, Messages: "User has been logged out"})
            fmt.Println( a...: "User has been logged out.")
        }
    } else {
        log.Println( v...: "Could not logout User")
        c.JSON( code: 500, models.Error{Code: 500, Messages: "Unable to logout User"})
    }
    defer db.Close()
}
```

Figure 4.23: CheckForCookie

```go
// CheckForCookie
// Check if user has a cookie.
// Used to abort requests made from the client if a user has no cookie (not logged in).
func CheckForCookie(c *gin.Context) bool {
    _, err := c.Cookie( name: "session_id")

    // Return false if no cookie is found.
    if err != nil {
        c.JSON( code: 403, models.Error{Code: 403, Messages: "User is unauthorized"})
        return false
    }
    // Else return true as one exists.
    return true
}
```

Please view the figure below for an overview of how the account and session system works with the controllers, handler functions, HTTP methods, and paths.

Figure 4.24: Account System



4.2.4 Report System

Users are required to be logged in for all the report operations. This requirement is ensured with the CheckForCookie function shown above. Before each function gets involved with the database, the user is checked for a cookie. If they have one, the function request continues; if they do not, it is aborted.

Create A Report

Creating a report is handled by the functions, CreateReport, CheckForCookie, InsertJobReport and isValidAccount. The process begins with a POST request from the client (front-end) of a new report with the information bound as JSON data with the JobReports. Then the CheckForCookie function comes in, and if that checks out, InsertJobReport is called with the new report data as a parameter along with the user's username and one for Gin to handle status codes (Figure 4.25). The data is then prepared as MySQL INSERT QUERIES for the tables: jobreports and customers in the database (Figure 4.26). Once these are in place, the user is again checked with isValidAccount. This second check is mainly to have the user's 'worker name' in the report's details. Next, a MySQL transaction is begun to execute the INSERTs into the two tables (Figure 4.27). Once the transaction is committed, a new report has been successfully created and a status code of 201 is sent to inform the client it was created.

Figure 4.25: CreateReport Function

```
// CreateReport
// Works with CheckForCookie & InsertJobReport.
// If the user has a cookie call InsertJobReport to create a report from user input data.
func CreateReport(c *gin.Context) {
    var report models.JobReport

    // Bind entered JobReport data from user to object, else throw error.
    if err := c.BindJSON(&report); err != nil {
        fmt.Println(err.Error()) // Failed to Bind data.
        c.JSON( code: 500,  obj: nil)
    }

    // Check for user's cookie - if they do not have one abort the request.
    // Status code handled by CheckForCookie.
    if !CheckForCookie(c) {
        log.Println( v...: "User is unauthorized to create a report")
        return
    }

    // Call InsertJobReport to create the report.
    if err := InsertJobReport(c, report, wa.Username); err == nil {
        c.JSON( code: 201, models.Error{Code: 201, Messages: "Report created successfully"})
    } else {
        c.JSON( code: 401, models.Error{Code: 401, Messages: "Not able to create Report"})
    }
}
```

Figure 4.26: Prepare MySQL INSERT QUERIES

```
// Insert into the table jobreports.
insertReport, err := db.Prepare(
     query: "INSERT INTO jobreports(worker_id, date_stamp, vehicle_model, vehicle_reg, vehicle_location, " +
        "miles_on_vehicle, warranty, breakdown, cause, correction, parts, work_hours, job_report_complete) " +
        "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)")

// Insert into the table customers.
insertCustomer, err := db.Prepare( query: "INSERT INTO customers (job_report_id, customer_name, customer_complaint)" +
    " VALUES (LAST_INSERT_ID(), ?, ?)")
```

Figure 4.27: Go MySQL Transaction for two INSERTs

```
// Begin MySQL transition to create a new report with input data from user.
_, err = db.Query( query: "BEGIN")
// Execute insert into the table jobreports.
reportResult, err := insertReport.Exec(wa.Id, report.Date, report.VehicleModel, report.VehicleReg, report.VehicleLocation,
    report.MilesOnVehicle, report.Warranty, report.Breakdown, report.Cause, report.Correction, report.Parts,
    report.WorkHours, report.JobComplete)
// Execute insert into the table customers.
customerResult, err := insertCustomer.Exec(report.CustomerName, report.Complaint)
_, err = db.Query( query: "COMMIT") // Commit MySQL transition.

if err != nil {
    log.Println( v...: "\nMySQL Error: Error Inserting Report Details.\n", err)
    c.JSON( code: 500,  obj: nil)
    return errors.New( text: "error creating Report")
}
fmt.Println( a...: "\n[INFO] Printing MySQL Results for new Report...\n", reportResult, customerResult)
```

Get Reports

Retrieving reports owned by a user are handled by the functions GetReports, CheckForCookie and isValidAccount. This starts off with the client sending a GET request. The user is first checked with the functions CheckForCookie and isValidAccount (Figure 4.28). This is to ensure the user owns these reports. If they do a MySQL JOIN QUERY is initiated to select all the user's reports (Figure 4.29) in the database. The reports are then send to the client with a status code of 200 - meaning the retrieval was a Status OK.

Figure 4.28: GetReports Function

```
func GetReports(c *gin.Context) {
    db := config.DbConn()

    worker := wa.Username
    var res []models.JobReport
    var report models.JobReport

    if !CheckForCookie(c) {
        log.Println( v...: "User is unauthorized to get these Reports")
        return
    }

    if !isValidAccount(worker) {
        log.Println( v...: "\nUser does not own these reports.")
        c.JSON( code: 401, models.Error{Code: 401, Messages: "User does not own these reports"})
        return
    }
```

Figure 4.29: Go MySQL JOIN to SELECT all Reports

```
// JOIN Query to get user's job reports.
selDB, err := db.Query( query: "SELECT DISTINCT jr.job_report_id, jr.date_stamp, jr.vehicle_model, "+
    "jr.vehicle_reg, jr.miles_on_vehicle, jr.vehicle_location, jr.warranty, jr.breakdown, "+
    "cust.customer_name, cust.customer_complaint, jr.cause, jr.correction, jr.parts, jr.work_hours, "+
    "wkr.worker_name, jr.job_report_complete FROM jobreports jr INNER JOIN customers cust "+
    "ON jr.job_report_id = cust.job_report_id "+
    "INNER JOIN workers wkr ON jr.worker_id = wkr.worker_id WHERE wkr.username = ?", worker)

if err != nil {
    log.Println( v...: "\nFailed to process Reports.")
    c.JSON( code: 500,  obj: nil)
    return
}
```

Get A Report by ID

GetReports and GetReportById are similar to each other. The difference being GetReports retrieves all reports owned by a user, and GetReportById gets a single report. GetReportById gets the report's ID from its parameter of the path e.g. /jobReports/ID (Figure 4.30). Retrieving a single report begins with a GET request from the client. The user is then checked the same as if they were getting all their reports. Then a MySQL JOIN QUERY is done with the requested report's ID (Figure 4.31) to select it from the database, and the report is sent to the client with a status code of 200.

Figure 4.30: ID Parameter from Report

```
// Get ID from request.
reportId := c.Params.ByName( name: "jobReportId")
fmt.Printf("Get Report with ID: " + reportId)
```

Figure 4.31: Go MySQL JOIN to SELECT a singe Report

```
// JOIN Query to get report by requested ID and username.
selDB, err := db.Query( query: "SELECT DISTINCT jr.job_report_id, jr.date_stamp, jr.vehicle_model, "+
    "jr.vehicle_reg, jr.miles_on_vehicle, jr.vehicle_location, jr.warranty, jr.breakdown, "+
    "cust.customer_name, cust.customer_complaint, jr.cause, jr.correction, jr.parts, jr.work_hours, "+
    "wkr.worker_name, jr.job_report_complete FROM jobreports jr INNER JOIN customers cust "+
    "ON jr.job_report_id = cust.job_report_id "+
    "INNER JOIN workers wkr ON jr.worker_id = wkr.worker_id "+
    "WHERE jr.job_report_id = ? AND wkr.username = ?", reportId, worker)

if err != nil {
    log.Println( v...: "\nFailed to process Report.", err)
    c.JSON( code: 500,  obj: nil)
    return // Problem with QUERY.
}
```

Update A Report

Updating a report starts with a PUT request from the client with the report's ID as a parameter and its changed data. The user is then checked and the report data is then bound to JSON with the JobReports model. Next, a MySQL UPDATE QUERY is done with the reports new data in the database and a status code of 202 is sent to the client to inform them it has been updated - Figure below.

Figure 4.32: Go MySQL UPDATE a Report

```
// Read in values from client request and build object - update the report with the user's inputted data.
update, err := db.Exec( query: "UPDATE jobreports jr SET jr.date_stamp = ?, jr.vehicle_model = ?, "+
    "jr.vehicle_reg = ?, jr.vehicle_location = ?, jr.miles_on_vehicle = ?, jr.warranty = ?, "+
    "jr.breakdown = ?, jr.cause = ?, jr.correction = ?, jr.parts = ?, jr.work_hours = ?, "+
    "jr.job_report_complete = ? WHERE jr.job_report_id = ?", report.Date, report.VehicleModel, report.VehicleReg,
    report.VehicleLocation, report.MilesOnVehicle, report.Warranty, report.Breakdown, report.Cause, report.Correction,
    report.Parts, report.WorkHours, report.JobComplete, reportId)

if err != nil {
    log.Println( v...: "\nMySQL Error: Error Updating Report:\n", err)
    c.JSON( code: 503, models.Error{Code: 503, Messages: "Error Updating Report"})
} else {
    fmt.Println( a...: "\n[INFO] Processing Job Report Details...", "\nReport ID:", reportId)
    // Report has been successfully updated.
    c.JSON( code: 202, gin.H{})
    fmt.Println( a...: "\n[INFO] Print MySQL Results for Report:\n", update)
    defer db.Close()
}
```

Delete A Report

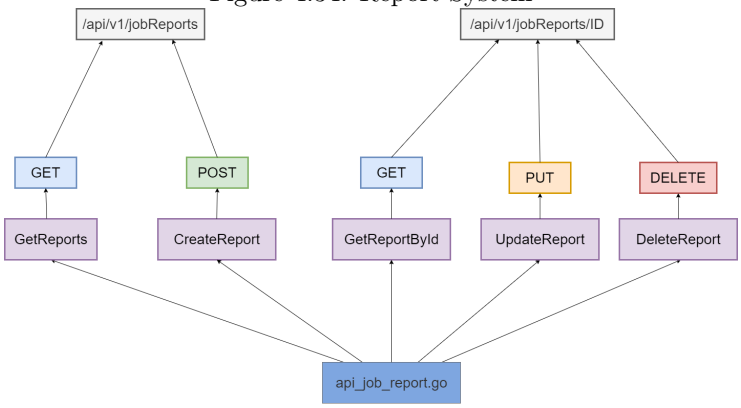
Deleting a Report begins with a DELETE request from the client request report by its ID. The user is checked and a MySQL DELETE QUERY is executed to remove the report from the database. Then a status code of 204 (No Content) is send to the client to notify the deletion of the report was successful - Figure below.

Figure 4.33: DeleteReport Function

```
func DeleteReport(c *gin.Context) {
    db := config.DbConn()
    // Get ID from request.
    reportId := c.Params.ByName( name: "jobReportId")
    fmt.Printf("Get Report with ID: " + reportId)

    if !CheckForCookie(c) {
        log.Println( v...: "User is unauthorized to delete this Report")
        return
    }

    // Create query to delete the report with its requested ID.
    res, err := db.Exec( query: "DELETE FROM jobreports WHERE job_report_id=?", reportId)
    if err != nil {
        log.Printf( format: "Report failed to delete.")
        c.JSON( code: 500,  obj: nil)
    }

    affectedRows, err := res.RowsAffected()
    if err != nil {
        log.Printf( format: "Report failed to delete.")
        c.JSON( code: 500,  obj: nil)
    }

    fmt.Printf( format: "\nThe statement affected %d rows\n", affectedRows)
    c.JSON( code: 204,  obj: nil) // Report has been deleted successfully.
}
```

Please view the figure below for an overview of how the report system works with the controllers, handler functions, HTTP methods, and paths.

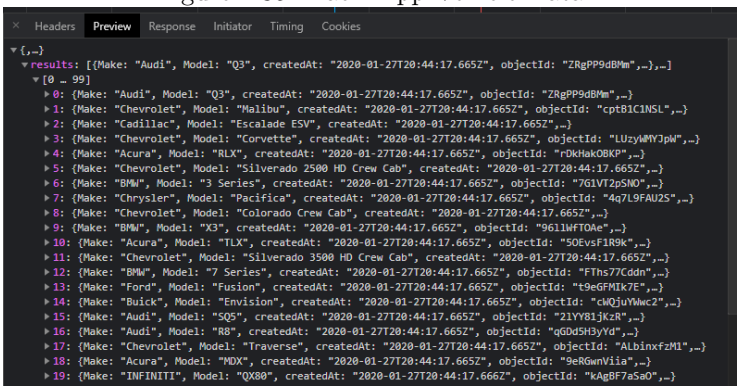
Figure 4.34: Report System



4.2.5 3rd Party API

The back-end has access to a 3rd Party API named Back4App which lists vehicle make and model data to make the creating and updating of reports convenient for the users. The data is gathered from the API with a GET request. Before this request is done, authorization headers are set from configuration files to be allowed to use the API's service. After the headers are set the CheckForCookie function comes in to stop the request if the user is not logged in. This done as only the first 10k requests are free with Back4App and unauthorized users could take up all the requests. Next the GET request is done and the JSON data is decoded from the response body and sent the client with a status code 200 and the response body is closed. Figure 4.35 below shows a sample of the data from Back4App.

Figure 4.35: Back4App Vehicle Data



4.2.6 Elastic Horton

The back-end is hosted on AWS's Elastic Beanstalk (EB). This means it is constantly running to have it available to the front-end at all times. This process was done through the EB CLI and a Dockerfile. First, an EB environment (Figure 4.36) was set up for the back-end, then the Dockerfile was used to deploy and update the running back-end to EB. The domain name 'repota-service.com' was bought from GoDaddy. This domain was bought to have the back-end and front-end to be secure (HTTPS). A SSL certificate attached to the domain by the name of 'api.report-service.com' was acquired from AWS's Certificate Manager with the domain bought. Then the EB environment was altered to listen on port 443 for HTTPS with the SSL certificate. Route 53 was then used to point the bought domain to the EB environment. In summary, the back-end is hosted with a very secure connection (Figure 4.37).

Figure 4.36: Elastic Beanstalk Environment

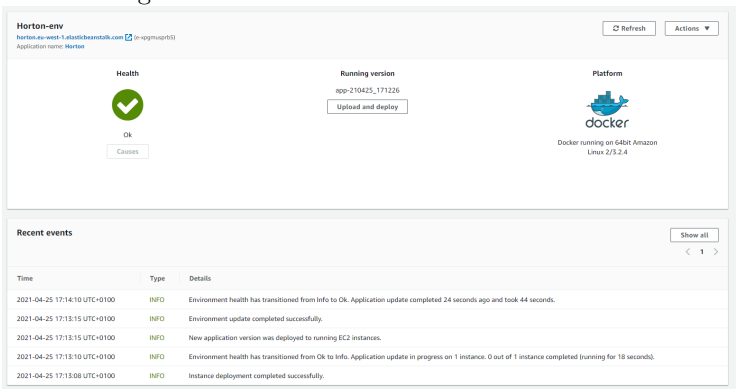
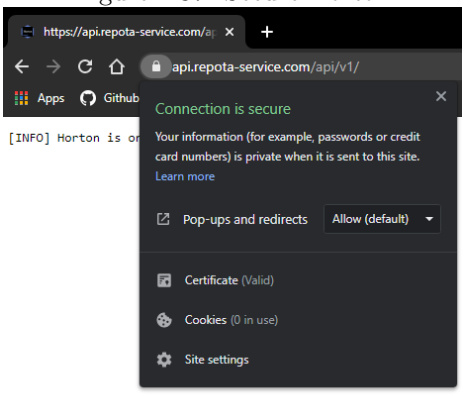


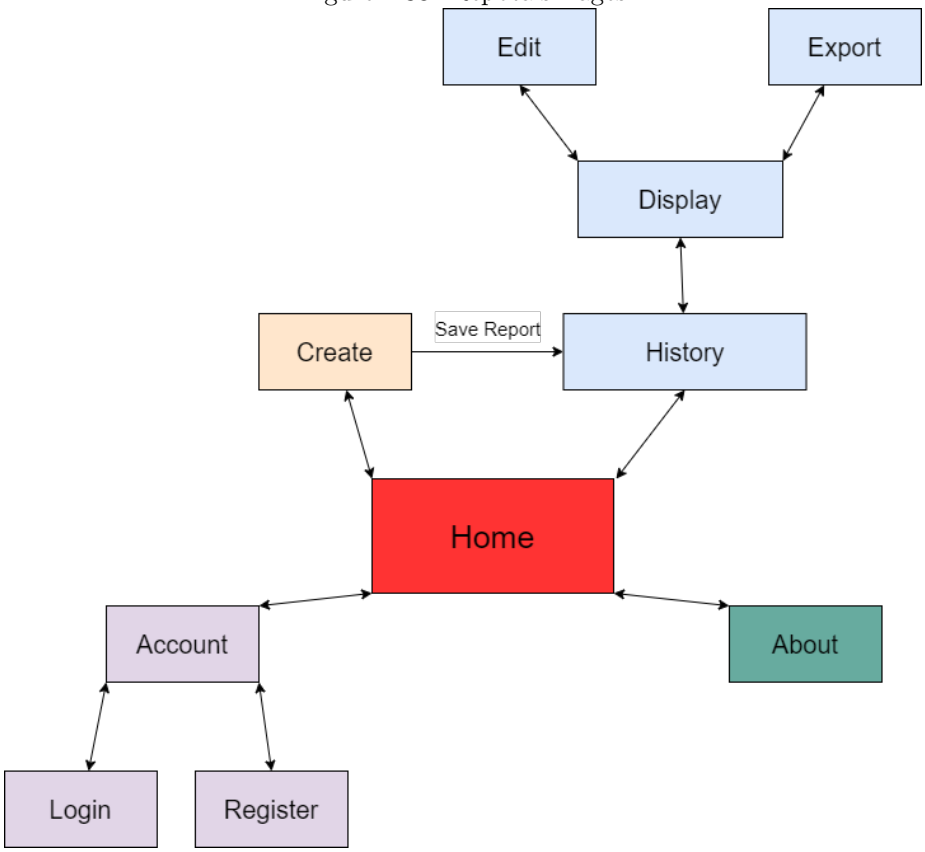
Figure 4.37: Secure Horton



4.3 The Front-end

The front-end (Repota) is the face of it all. Repota itself was built with Angular and Ionic, while the OpenAPI specification provided the TypeScript code stubs to link Repota to Horton. The remainder of this section will explain how Repota and Horton are integrated for the users and their reports through the app's pages - Figure below.

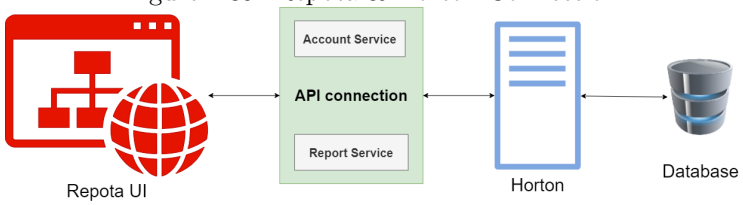
Figure 4.38: Repota's Pages



4.3.1 Repota & Horton Integration

Account & Report Service

Figure 4.39: Repota & Horton Connection



The figure above displays how Repota (the front-end) is connected to Horton (API) and the database. The API connection is made through the classes AccountService and JobReportService, which came from the OpenAPI specification and required little editing. The only editing required was to change the default URL where the connection is made to the API and the addition of two methods named logout and getCarApiData - for retrieving Back4App's data from the API. AccountService handles user registration, login, and logout. AccountService is connected to API by the URL (basePath - Figure 4.40). This class allows the HTTP methods POST for register and login and GET for logout. JobReport service is similarly connected to the API to AccountService. JobReportService handles all the CRUD operations and HTTP methods: GET (Get reports & get a report by ID), POST (Create a Report), DELETE (Delete Report), PUT (Update a Report) for reports, and retrieving Back4App's data from the API. These methods, their functionality and how they work for users will be explained below in the respected titled subsections.

Figure 4.40: Account Service Connection to API

```
@Injectable()
export class AccountService {

    protected basePath = 'https://api.repota-service.com/api/v1'; // Connection to API
    public defaultHeaders = new HttpHeaders();
    public configuration = new Configuration();
```

Model Interfaces

The model interfaces named InLineObject and JobReport (Figures below) also came from the OpenAPI specification. These models are used for the register, login, create report and edit report forms to interact with the models with the API's models and to put the data into the database.

Figure 4.41: InLineObject Interface

```
export interface InlineObject {
    username?: string;
    name?: string;
    password?: string;
}
```

Figure 4.42: JobReport Interface

```
export interface JobReport {
    jobReportId?: number;
    date?: string;
    vehicleModel?: string;
    vehicleReg?: string;
    milesOnVehicle?: number;
    vehicleLocation?: string;
    warranty?: number;
    breakdown?: number;
    customerName?: string;
    complaint?: string;
    cause?: string;
    correction?: string;
    parts?: string;
    workHours?: number;
    workerName?: string;
    jobComplete?: number;
}
```

4.3.2 Account Pages

Registration

User registration is done through the Register Page. The method register (Figure 4.43) is set up as a form that uses the InLineObject model to have the user's enter details (username, name, and password) as an object to push the data with a POST request to the API with the AccountService's register method and insert the details into the database. The user must have their username at least eight characters long, their name to be their full name (as this is displayed on their reports). The password is ensured by a regular expression on the HTML side to have at least eight characters, one uppercase and lowercase letter, one number, and a special character to provide extra security (Figure 4.44). Once the user has the form filled out, they click the register button on the UI to send the data, and once the data in the database, the user is navigated to the login page. If there are errors with the registration process (e.g., the username is already taken), an error message is displayed on the UI to the user (Figure 4.45).

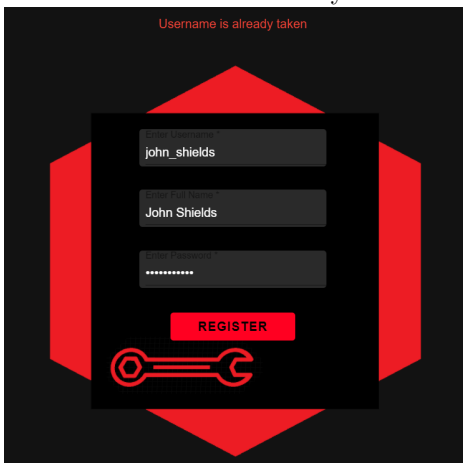
Figure 4.43: Register Method

```
register(form: NgForm) {
    const object: InlineObject = {
        username: form.value.username,
        name: form.value.name,
        password: form.value.password,
    };

    // Push data to API to register user.
    this.api.register(object).subscribe( next: () => {
        this.setErrorMessage(''); // clear error message.
        form.reset();
        this.router.navigate( commands: ['/login']);
    }, error: error => {
        // Get error from response.
        let errorMessage = JSON.stringify(error.error.messages);
        this.setErrorMessage(JSON.parse(errorMessage));
        console.log(error);
    });
}
```

Figure 4.44: Regular Expression pattern for Password

```
<!-- Password input -->
<mat-form-field class="password">
    <!-- Minimum 8 characters, at least 1 uppercase & lowercase letter, 1 number and 1 special character -->
    <input matInput type="password" name="password" ngModel required
           pattern="^(?=.*?[A-Z])(?=(.*[a-z]){1,})(?=(.*[\d]){1,})(?=(.*[\W_]){1,})(?!.*\s).{8,}$"
           placeholder="Enter Password" #password="ngModel" id="password"/>
    <mat-error *ngIf="password.errors?.pattern || password.invalid">
        Enter a <u>STRONG</u> password.
    </mat-error>
</mat-form-field>
```

Figure 4.45: Username is already taken - Error



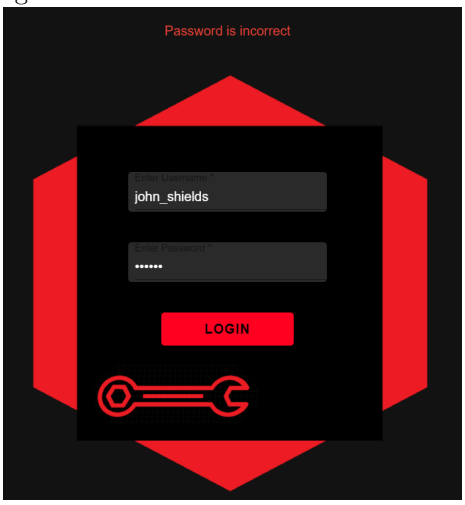
Login

User login is managed on the Login Page. This process is handled by the login method (Figure 4.46). Similar to register, login is set up as a form and uses the InLineObject model. The user's entered username and password are attached to the model. Once the user clicks the login button, the data is sent to the API by a POST request through the AccountService's login method. The entered data by the user is then checked with the user's details in the database. If the information is correct, a cookie is set for three days for the user and then they are navigated to the home page. If there is a problem or the information is not correct, an error message is displayed for the user (Figure 4.47).

Figure 4.46: Login Method

```
login(form: NgForm) {
    // User (worker) account model.
    const object: InlineObject = {
        username: form.value.username,
        password: form.value.password
    };

    // Push data to API to login user.
    this.api.login(object).subscribe( next: () => {
        this.setErrorMessage(''); // clear error message.
        this.ngOnInit();
        form.reset();
        this.router.navigate( commands: ['/home']); // allow user in.
    }, error: error => {
        // Get error from response.
        let errorMessage = JSON.stringify(error.error.messages);
        this.setErrorMessage(JSON.parse(errorMessage));
        console.log(error);
    });
}
```

Figure 4.47: Password is incorrect - Error



Logout

Logout is available to a logged-in user on any page of the front-end. The logout button is available to users in the hamburger menu (Figure 4.48). When a user clicks on the logout button, this calls the logout method (Figure 4.49). That method then sends a GET request with the AccountService's logout method to the API, and after one second, the user is officially logged out and navigated to the account page. The wait for one second is to allow the logged-out cookie set by the API to expire; therefore, the user does not send the logout request multiple times within that one second.

Figure 4.48: Hamburger Menu - Logout Button

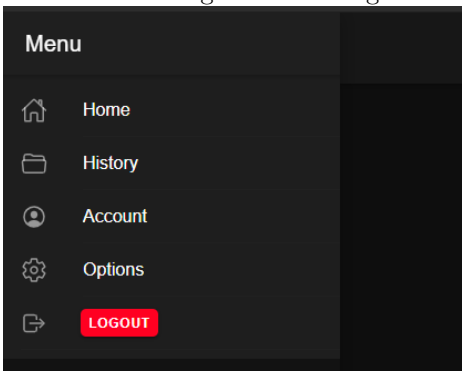


Figure 4.49: Logout Method

```
logout() {
    const delay = ms => new Promise( executor: res => setTimeout(res, ms));
    if (this.authService.loggedIn()) {
        this.api.logout().subscribe( next: async () => {
            this.setErrorMessage('');
            await delay( ms: 1000); // Wait for cookie to expire.
            await this.router.navigate( commands: ['']);
        }, error: error => {
            // Get error message from response.
            let errorMessage = JSON.stringify(error.error.messages);
            this.setErrorMessage(JSON.parse(errorMessage));
            console.log(error);
        });
    }
}
```

4.3.3 Report Pages

History

The History Page displays all the reports owned by a user. Only the report's number (ID), date, customer name, and the complete box are displayed here (Figure 4.50). The reports are loaded in from the API by the JobReportService's getReports method. The method ngOnInit (Figure 4.51) calls getReports from JobReportService, and the reports are automatically loaded for the user when they navigate to this page from a ngFor on the HTML side. If the user does not have any reports, a message is displayed to notify them (Figure 4.52).

Figure 4.50: History Page

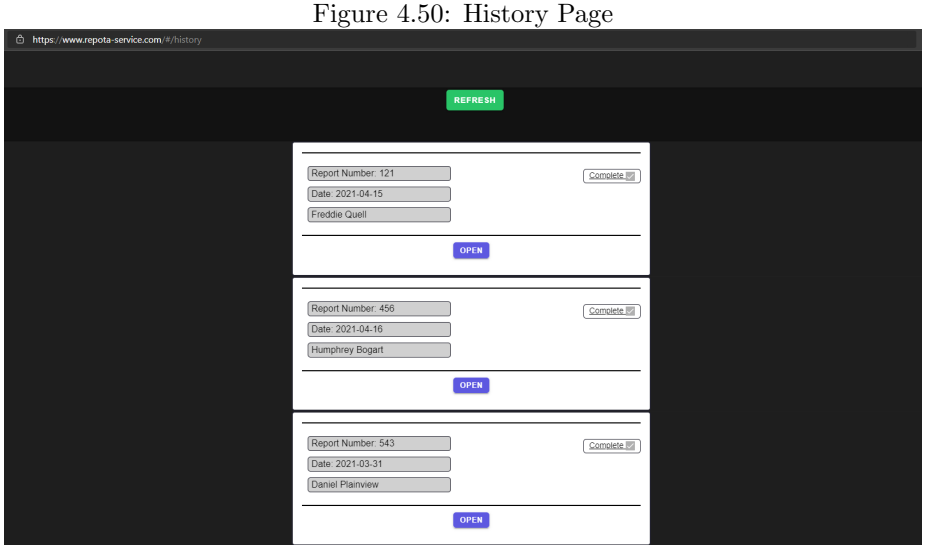
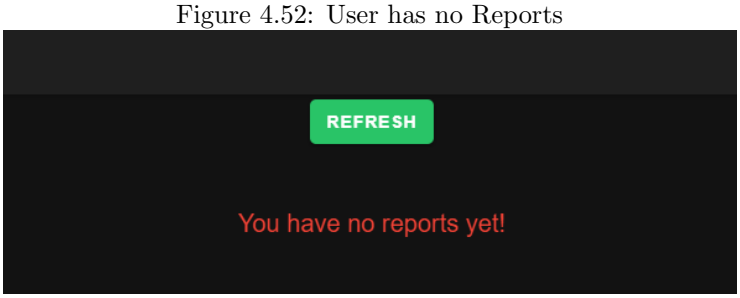


Figure 4.51: ngOnInit Method - Load all user's Reports

```
ngOnInit() {
    this.api.getReports().subscribe( next: data => {
        this.reports = data;
        this.setErrorMessage(''); // clear error message.
        // If User has no reports.
        if (data == null) {
            this.setErrorMessage('You have no reports yet!');
        }
    }, error: error => {
        // Get error from response.
        let errorMessage = JSON.stringify(error.error.messages);
        this.setErrorMessage(JSON.parse(errorMessage));
        console.log(error);
    });
}
```

Figure 4.52: User has no Reports



Display

The Display Page shows a single report. A user can see the report by clicking the open button on a report from the history page. This button navigates the report with the selected report's ID as a parameter to the display page. The full data of the report is displayed here by a ngFor with the options to edit, export it to a PDF and delete (Figure 4.53). The report is loaded by the ngOnInit method (Figure 4.54) and from the API through the JobReportService's getReportById method. If the ID parameter is not the ID for a report in the database or the user does not own the report, a 'Report not found' message is displayed (Figure 4.55).

Figure 4.53: Display Page

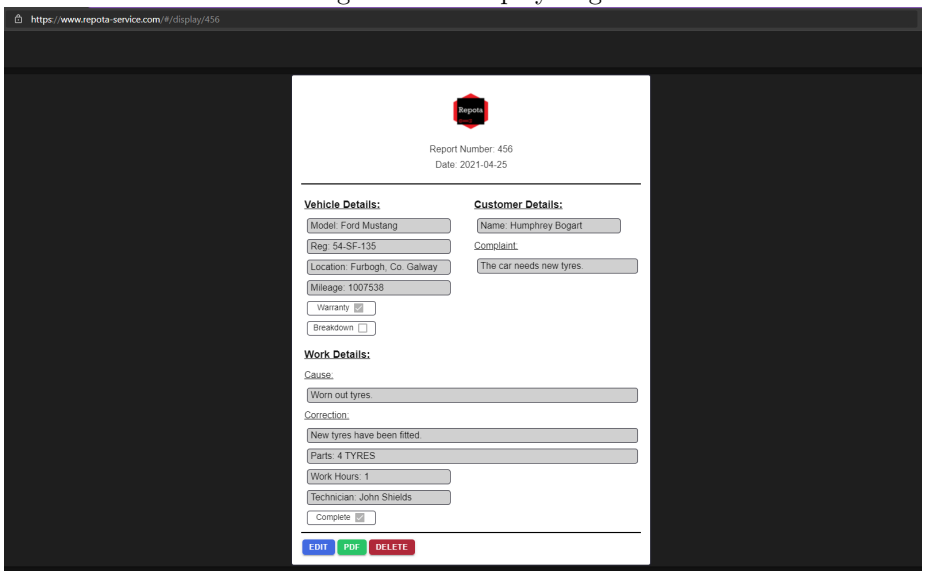
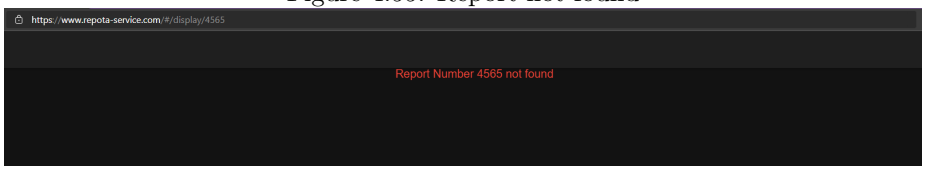


Figure 4.54: ngOnInit Method - Get Report by ID

```
ngOnInit() {
    this.api.getReportById(this.route.snapshot.params['jobReportId']).subscribe( next: data => {
        this.report = data;
        this.setErrorMessage(''); // clear error message.
        if (data == null){
            this.setErrorMessage('Report Number ' + this.route.snapshot.params['jobReportId'] + ' not found');
        }
    }, error: error => {
        // Get error from response.
        let errorMessage = JSON.stringify(error.error.messages);
        this.setErrorMessage(JSON.parse(errorMessage));
        console.log(error);
    });
}
```

Figure 4.55: Report not found



Create

Users construct their reports through a form on the Create page. The method createReport uses the JobReport model as an object for all the data the user enters (Figure 4.56). The values, warranty, breakdown, and complete are checkboxes. These values required lists to input the data along with an if/else statement to turn the check box values from true and false to 1 and 0 (Figure 4.57). This alteration is to accommodate the values on the Go side of the model and the database. Once the user has entered the report's information and clicks the create button, this data gets sent through a POST request to the API through the JobReportService. After the report is in the database, the user is navigated to the history page. The data from Back4App is loaded in though the ngOnInit method (Figure 4.58) from the API and is displayed to the user as a HTML data list (Figure 4.59).

Figure 4.56: JobReport Object & Push Object to API

```
// Use JobReport Model
const object: JobReport = {
    date: form.value.date,
    vehicleModel: form.value.vehicleModel,
    vehicleReg: form.value.vehicleReg,
    milesOnVehicle: form.value.milesOnVehicle,
    vehicleLocation: form.value.vehicleLocation,
    warranty: this.checkBoxValue1,
    breakdown: this.checkBoxValue2,
    customerName: form.value.customerName,
    complaint: form.value.complaint,
    cause: form.value.cause,
    correction: form.value.correction,
    parts: form.value.parts,
    workHours: form.value.workHours,
    jobComplete: this.checkBoxValue3
};

// Push data to API to create report using the model.
this.api.createReport(object).subscribe( next: data => {
    this.api.createReport(data);
    this.setErrorMessage(''); // clear error message.
    this.router.navigate( commands: ['/history']);
}, error: error => {
    // Get error from response.
    let errorMessage = JSON.stringify(error.error.messages);
    this.setErrorMessage(JSON.parse(errorMessage));
    console.log(error);
});
```

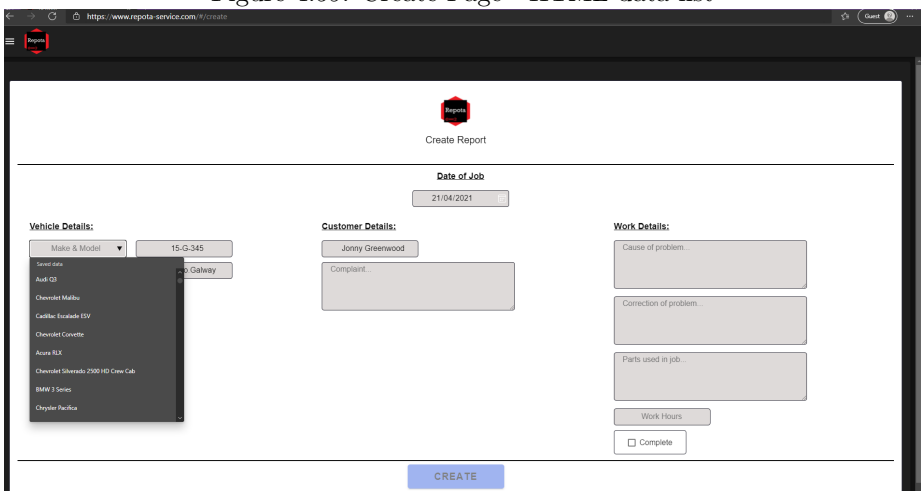
Figure 4.57: Checkboxes - If/Else Statement

```
createReport(form: NgForm) {
    // Make the true/false values of check boxes to 1s and 0s.
    // warranty
    if (form.value.warranty === true) {
        this.checkBoxValue1 = 1;
    } else {
        this.checkBoxValue1 = 0;
    }
    // breakdown
    if (form.value.breakdown === true) {
        this.checkBoxValue2 = 1;
    } else {
        this.checkBoxValue2 = 0;
    }
    // job complete
    if (form.value.jobComplete === true) {
        this.checkBoxValue3 = 1;
    } else {
        this.checkBoxValue3 = 0;
    }
```

Figure 4.58: ngOnInit method - Back4App Data

```
ngOnInit() {
    // Get vehicle data from API.
    this.api.getCarApiData().subscribe( next: data => {
        // Strip out the keys from data.
        for (let key in data) {
            this.vehicles = data[key];
        }
        this.setErrorMessage('');
    }, error: error => {
        let errorMessage = JSON.stringify(error.error.messages);
        this.setErrorMessage(errorMessage);
        console.log(error);
    });
```

Figure 4.59: Create Page - HTML data list



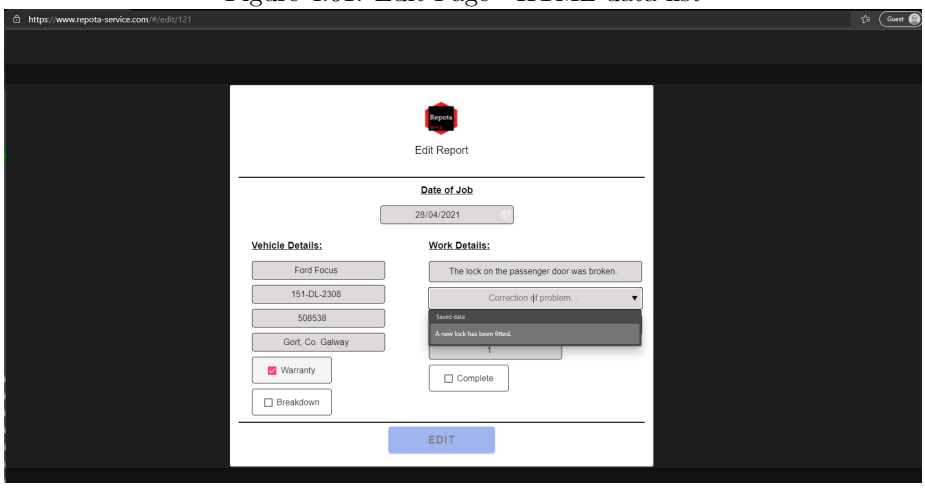
Edit

Users can make changes or complete their reports on the Edit Page. They can access the edit page from the display page by clicking the edit button on the report. This carries over the report's ID to the edit page. The report's data is available through the loadReportById method (Figure 4.60) in HTML data lists (Figure 4.61). This setup was an improvise as when the data was loaded straight into the input boxes, it was unchangeable, and users would still have to retype it all. If the data were loaded into an input box, there would be two inputs updated in the database. For example, the 'Make & Model' box would have the value 'Ford Focus', and the user types in 'Ford Focus' again and submitted the edit. In the database, the value would be 'Ford Focus Ford Focus'. This method was not suitable. The HTML data list solved this problem as the user only has to click the data to have it in the input box again.

Figure 4.60: loadReportById Method

```
loadReportById() {
    this.api.getReportById(this.route.snapshot.params['jobReportId']).subscribe( next: data => {
        this.report = data;
        this.setErrorMessage('');
        if (data == null){
            this.setErrorMessage('Report Number ' + this.route.snapshot.params['jobReportId'] + ' not found');
        }
    }, error: error => {
        let errorMessage = JSON.stringify(error.error.messages);
        this.setErrorMessage(JSON.parse(errorMessage));
        console.log(error);
    });
}
```

Figure 4.61: Edit Page - HTML data list



Similar to the create page, the Back4App's data is retrieved from the API to the Make & Model input box. The only difference is that the request is made through the method loadCarData and, along with loadReportById, are called in ngOnInit. This abstraction was done to ensure code quality and performance. Initially, all the code for retrieving Back4App's and the report's data, along with the lists for the checkboxes, was in the ngOnInit method. The checkboxes' structure is in relation the the create page's structure for them.

The user submits the edit with the edit button. This submission calls the editReport method and correspondent with the createReport method. editReport uses the JobReport model as an object with the report's altered data and calls the updateReport method from JobReportService with a PUT request to the API. This process updates the report in the database and navigates the user back to the history page.

Export

Users can export their reports to PDFs from the Export Page. Exporting reports is accessed on the display page of a report by the PDF button. When a user clicks on this button, the report is brought and load over by its ID through the ngOnInit method from the API on the export page. The export page presents the report in a printable format (Figure 4.62). When a user clicks on the Export PDF button, this calls the method onExportPDF (Figure 4.63), which brings up a file explore window for the user to save the report locally. onExportPDF uses the libraries dom-to-image and jsPDF. dom-to-image converts the HTML DOM (Document Object Model) to a PNG image, then this image is added to a PDF made by jsPDF. Next, the report is saved to a file named service_report.pdf (Figure 4.64).

Figure 4.62: Export Page

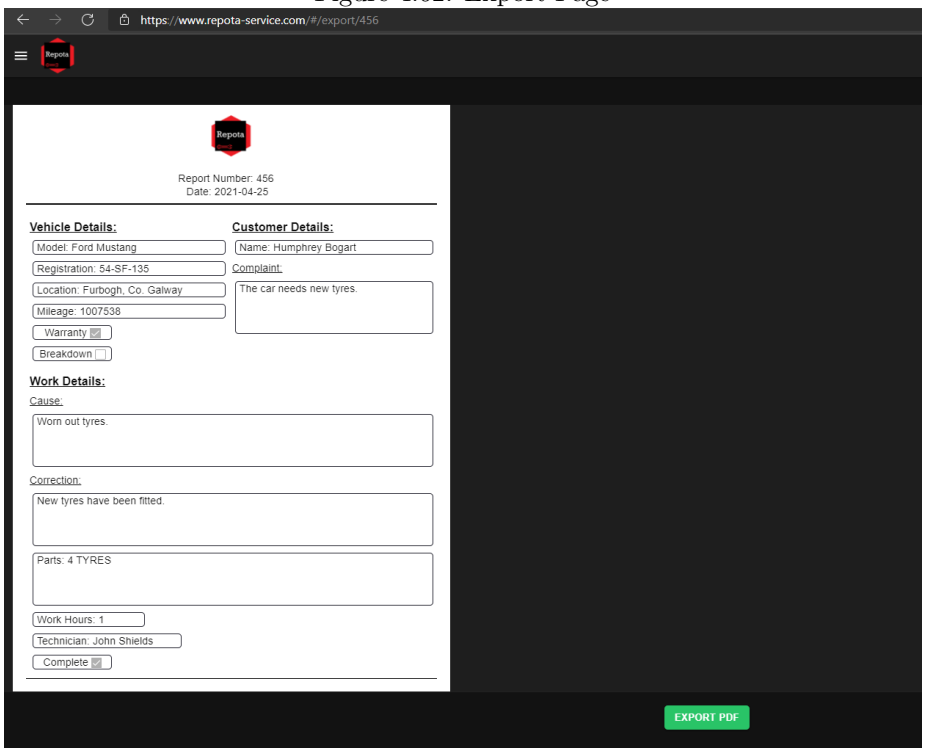
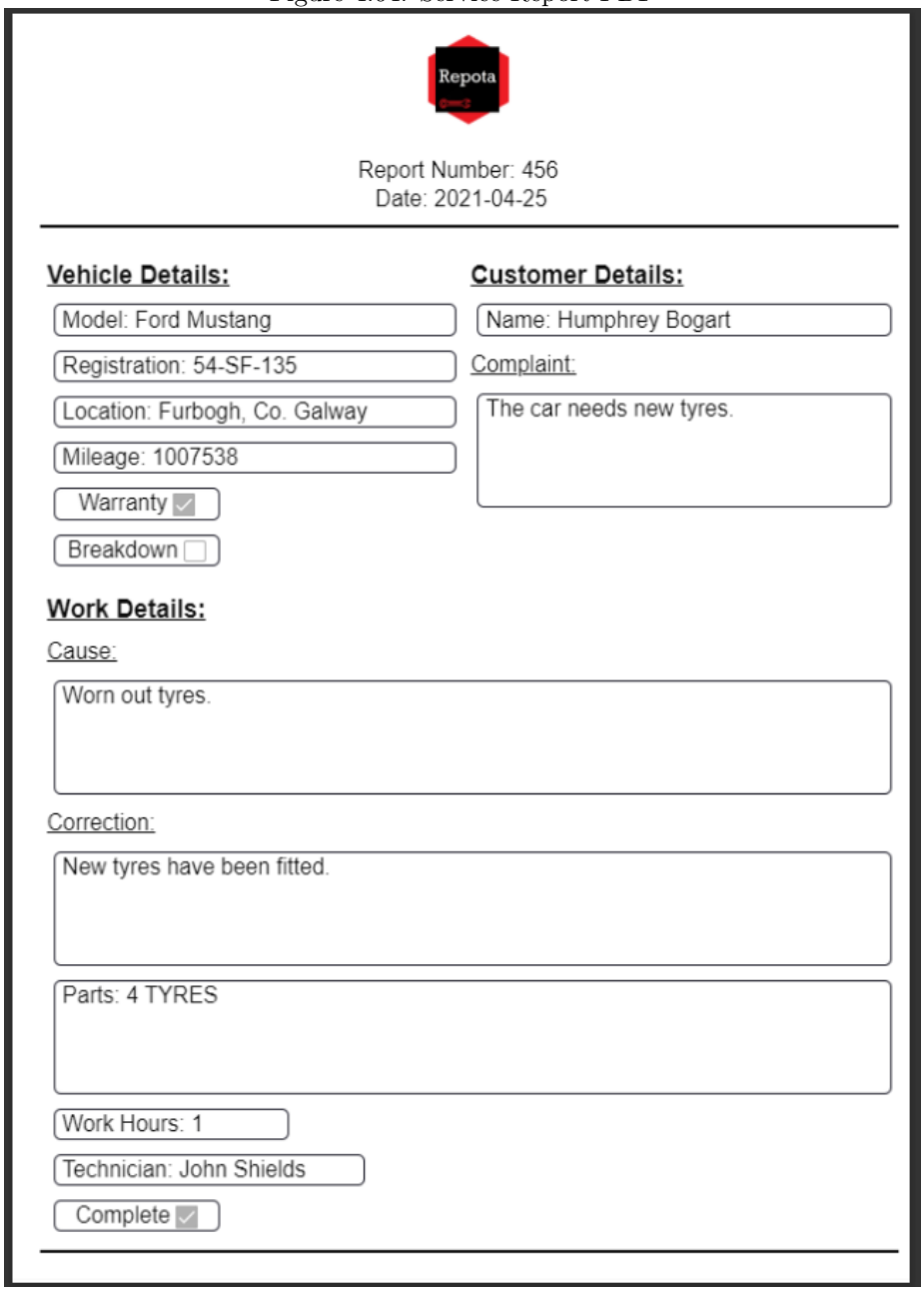


Figure 4.63: onExportPDF Method

```
onExportPDF() {
    const content = document.getElementById( elementId: 'job-report');
    // Set the PDF requirements.
    const options = {background: 'white', width: this.pdfWidth, height: this.pdfHeight, quality: 1};
    // Convert the dom to an image.
    domtoimage.toPng(content, options).then(
        (dataUrl) => {
            // Add the image to a PDF.
            const doc = new jspdf.jsPDF( orientation: 'portrait', unit: 'mm', format: 'a4', compressPdf: true);
            doc.addImage(dataUrl, format: 'jpeg', x: 0, y: 0, this.imgWidth, this.imgHeight);
            doc.save( filename: 'service_report.pdf');
        });
}
```

Figure 4.64: Service Report PDF



Delete

The deletion of a report is handled on the display page. When a user clicks the delete option, a popup box displays asking them are they sure to delete it (Figure 4.65). When the user clicks 'OK' on the popup box, the method deleteReport (Figure 4.66) is called to send a DELETE request to the API through the JobReportService with the corresponding method. Once the request is complete, the report is removed from the database, and the user is navigated back to the history page.

Figure 4.65: Delete Report Confirmation

Figure 4.66: deleteReport Method

```
deleteReport(id: number) {
    // Pop up box to make sure the User wants to delete the Report.
    if (confirm("Are you sure to delete Report Number " + id + "?")) {
        this.api.deleteReport(id).subscribe( next: () => {
            this.setErrorMessage('');
            this.router.navigate( commands: ['/history']);
        }, error: error => {
            let errorMessage = JSON.stringify(error.error.messages);
            this.setErrorMessage(JSON.parse(errorMessage));
            console.log(error);
        });
    }
}
```

Error Messages

All error messages for user registration, login, and logout, along with the CRUD operations for reports, are handled with getters and setters (Figure 4.67). As shown and discussed above, error messages are displayed for invalid user inputs and 404s (Not found errors). Any errors from the API are extracted into a string format and then set as the parsed message to the setter (Figure 4.68). The getter is then used on the HTML side to display the error to the user, as shown above. These errors can also let users know of any serious errors from the API. For instance, an error with the status code 500 (Internal Server Error) if a request went horribly wrong on the API's side. For example, an internal server error could be a loss of a database connection, and the users must be notified immediately to avoid confusion.

Figure 4.67: Error Message Handlers

```
setErrorMessage(error: String) {
    this.errorMessage = error;
}

getErrorMessage() {
    return this.errorMessage;
}
```

Figure 4.68: JSON extracting & parsing of error message

```
}, error: error => {
    // Get error from response.
    let errorMessage = JSON.stringify(error.error.messages);
    this.setErrorMessage(JSON.parse(errorMessage));
    console.log(error);
});
```

4.3.4 AuthGuard

When an unauthorized user visits Repota the pages home, history, display, create, edit, export are blocked. If the user tries to navigate to the home page through the hamburger menu the error message 'Please login first' displays (Figure 4.69). Also the history page and logout button are also not available. This security is a result of the AuthGuard implementation into the app. The AuthGuard blocks pages to users that do not have cookies (unauthorized user). The classes AuthGaurd and AuthService work together. AuthService uses the method loggedIn (Figure 4.70) to check if a user has a cookie or not. If the user is not logged in a boolean is set as false with the method canActivate (Figure 4.71) in the AuthGaurd class. canActivate is then used in the class AppRoutingModule to block the pages listed above (Figure 4.72).

Figure 4.69: Please login first

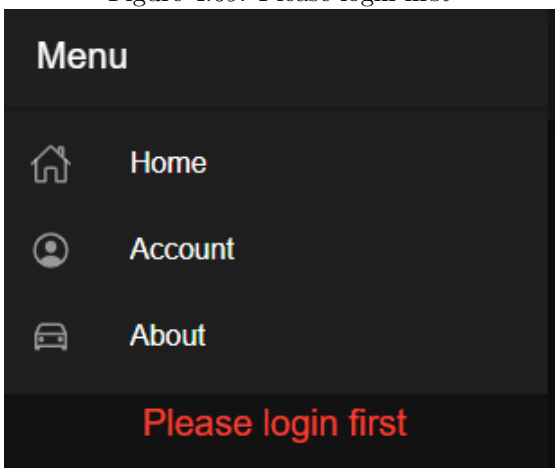


Figure 4.70: loggedIn Method

```
/**
 * @title Logged In
 * @desc Get the cookie 'session_id' set by API.
 */
loggedIn() {
    return !!this.cookieService.get('session_id');
}
```

Figure 4.71: canActivate Method

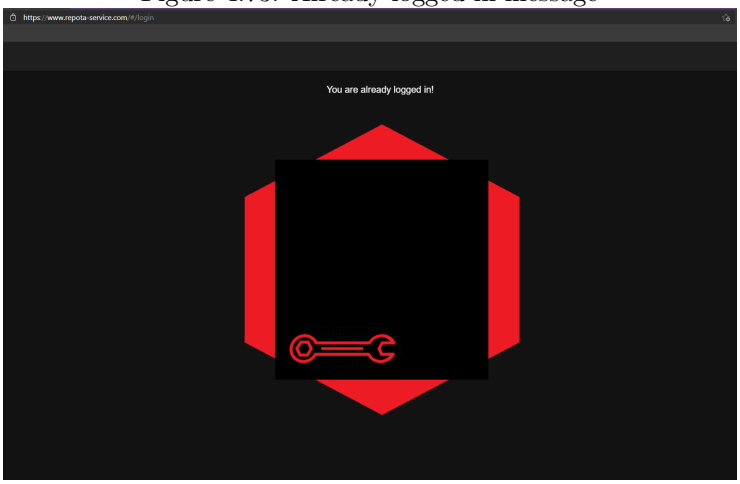
```
canActivate(): boolean {
    // If user is logged in return true.
    if (this.authService.loggedIn()) {
        return true;
    }
    // If user is not logged in return false and navigate to account page.
    else {
        this.router.navigate( commands: ['']);
        return false;
    }
}
```

Figure 4.72: Blocked Routes

```
{
    path: 'home',
    loadChildren: () => import('./home/home.module').then(m => m.HomePageModule),
    canActivate: [AuthGuard] // Block User if they are not logged in (no cookie).
},
{
    path: 'create',
    loadChildren: () => import('./create/create.module').then(m => m.CreatePageModule),
    canActivate: [AuthGuard]
},
{
    path: 'history',
    loadChildren: () => import('./history/history.module').then(m => m.HistoryPageModule),
    canActivate: [AuthGuard]
},
```

As mentioned above, when a user logs in, they receive a cookie. Once they are logged in, all the pages are unblocked, and they are free to explore the app through the hamburger menu. The method loggedIn is used throughout the app with Angular's ngIf to disable the login and register if a user is already logged in/registered. Suppose a user is logged in and navigates to the login or register page a message displays to notify them that they are already authorized (Figure 4.73).

Figure 4.73: Already logged in message



4.3.5 Repota UI

Throughout this chapter, a majority of Repota's UI has been shown. This section will go through the UI pages Home, Account, and About. As seen on the login and register page, the app's logo is used throughout these pages to provide consistency and an iconic style. When a user logs in, they are navigated to the home page. There are two buttons to navigate to the pages create and history (Figure 4.74). The about page (Figure 4.75) can be accessed and navigated to through the hamburger menu. This page displays a short description of the app and two buttons as hyperlinks for the app's guide located at the project's Wiki of the GitHub repository and the repository itself. The initial visiting page of the app is the account page (Figure 4.76). When a user is not logged in, a welcome message is displayed, controlled by the loggedIn method from the AuthService class. The buttons on the page provide navigation to the pages login and register.

Figure 4.74: Home Page

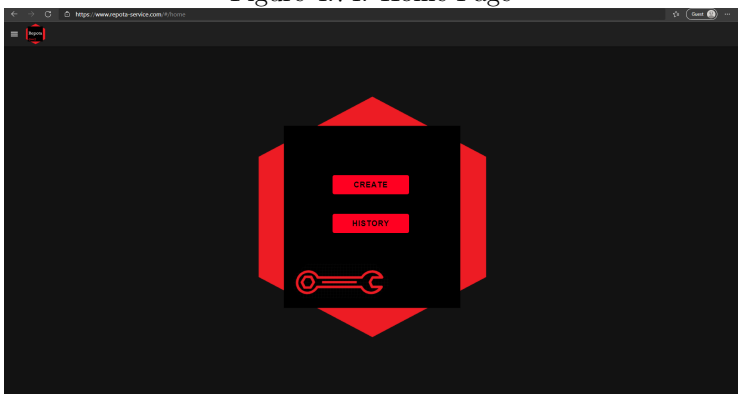


Figure 4.75: About Page

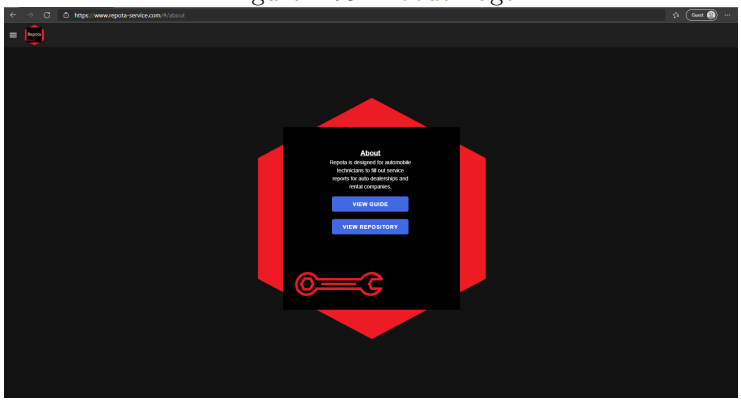
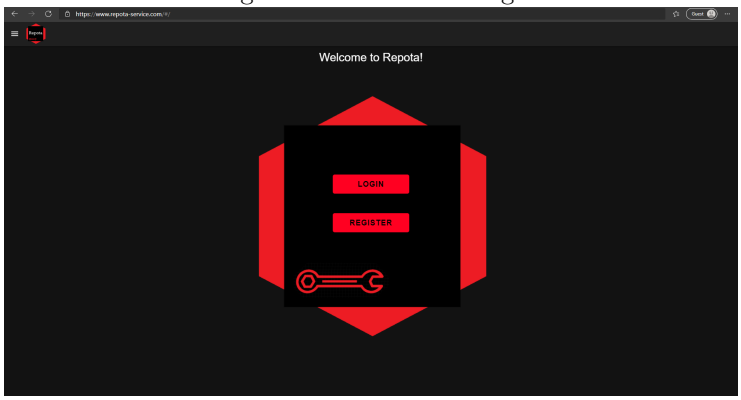


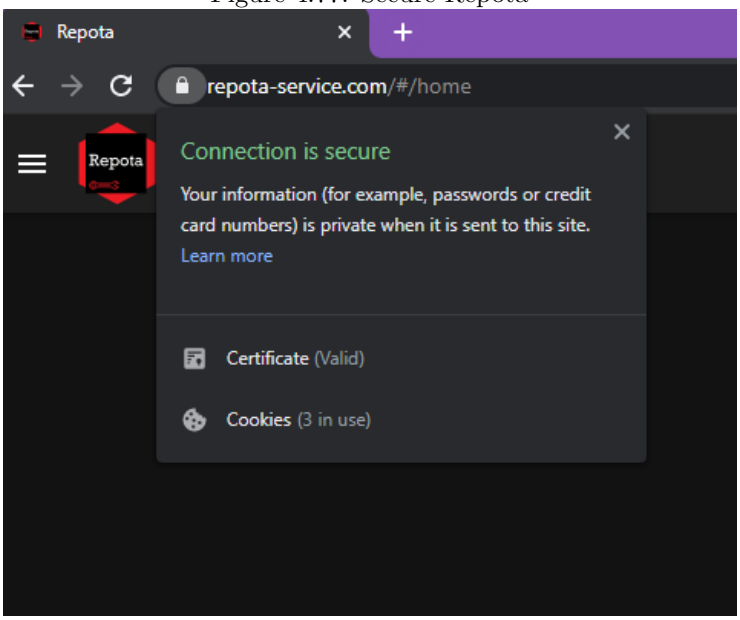
Figure 4.76: Account Page



4.3.6 Repota Bucket

Repota is hosted on AWS's S3 bucket. This procedure was reached through the AWS S3 CLI. The app was built for production using the Angular command 'ng build –prod'. The production directory was then deployed to S3 Bucket. Resembling Horton, Repota is also HTTPS. This process was achieved through AWS's Route 53, CloudFront, and Certificate Manager. The bought domain from GoDaddy was set to 'www.repota-service.com' through AWS's Route 53 to point to the bucket. Then the SSL certificate was retrieved from the Certificate Manager. CloudFront was then used to have the bucket attach to the SSL certificate. Resulting in Repota being secure (Figure below).

Figure 4.77: Secure Repota



Chapter 5

System Evaluation

The final product has all its intended functionality. Horton is robust and quick, while Repota is user-friendly and has met all the required features. Horton was tested thoroughly through integration testing with the tests package from Go. Repota's functionality was verified through high-level behaviour tests with CucumberJS (JavaScript). In order to perform these tests, a mock database was constructed especially for these processes. The reason for a mock database was to avoid conflict with the production database and avoid affecting the real data of users and reports.

5.1 Horton Robustness

The tests package from Go was used to test all of Horton's functions for the Routers, Account, Session, and Reports Systems and the data request from Back4App. The tests consist of HTTP methods to either check if everything is up and running, get data, or send payloads of data with the models (Figure 5.1) to the respected endpoints (Figure 5.2). The tests pass on the conditions of the response status codes and messages. For example, a '200 OK' would be a pass, and '500 Internal Server Error' would be a fail (Figure 5.3).

Figure 5.1: Payload of mock user details

```
t.Run( name: "register", func(t *testing.T) {
    // Set up InLineObject Payload.
    body := &models.InlineObject{
        Username: "test_user",
        Name:     "Test",
        Password: "@Testing14",
    }
```

Figure 5.2: Test Register - POST request

```go
// Set up /register request.
url := "http://localhost:8080/api/v1/register"
req, err := http.NewRequest( method: "POST", url, payloadBuf)
if err != nil {
    log.Println(err)
}
// Do POST request (Register User).
client := &http.Client{}
res, err := client.Do(req)
if err != nil {
    log.Println(err)
}
defer res.Body.Close()

fmt.Println( a...: "response Status:", res.Status)
if res.Status == "200 OK" {
    // TEST PASSED
    fmt.Println( a...: "\n[PASS] User was Registered successfully")
} else {
    // TEST FAILED
    t.Error( args...: "\n[FAIL] failed to Register User", err)
    t.Fail()
}
```

Figure 5.3: Register test pass

```
Tests passed: 2 of 2 tests – 710 ms
<4 go setup calls>
=== RUN   TestRegister
[TEST] Testing Register...
--- PASS: TestRegister (0.12s)
=== RUN   TestRegister/register
response Status: 200 OK

[PASS] User was Registered successfully
    --- PASS: TestRegister/register (0.12s)
PASS
```

5.2 Repota Robustness

CucumberJS was used throughout Repota to run through the app and test its high-level behavior of the features register, login, logout, create, edit and delete a report from a user's perspective. There are two sides to Cucumber, the Gerkin features, and the step definitions. Features consist of 'User Stories' and a 'Scenarios' (Figure 5.4). A User Story includes the name of a feature to test, the possible stakeholder, and the feature's need. Scenarios are the steps required to test the app's feature. Step definitions then perform these steps (Figure 5.5). The steps for the tests include navigation throughout the app to test multiple features at one time. All the tests were done through a controlled Firefox with Selenium WebDriver. The tests pass on a count of all the steps begin completed successfully (Figure 5.6).

Figure 5.4: Login Feature

```
Feature: Login
  As a User
  So that I can access my reports
  I need to login with my valid Username and Password

  Scenario Outline: Login User
    Given user navigates to the Login Page
    When user enters username "<username>"
    When user enters password "<password>"
    Then user clicks the login button
    Then user should be successfully logged in

    Examples:
      | username      | password   |
      | bob_mock_test | @Testing14 |
```

Figure 5.5: Login Steps

```
// Spin up a controlled firefox while the app is running to see if the Login Page is available.
driver = new Builder().forBrowser( name: 'firefox').build();
Given('user navigates to the Login Page', {timeout: 2 * 5000}, async () => {
    driver.wait(until.elementLocated(By.tagName( name: 'ion-header')));
    await driver.get('http://localhost:8100/#/login');
});

// Mock User enters username into the username input box.
When(/^user enters username "([^"]*)"$/, function (username) {
    let elm = driver.findElement(By.id( id: 'username'));
    return elm.sendKeys(username);
});

// Mock User enters password into the password input box.
When(/^user enters password "([^"]*)"$/, function (password) {
    let elm = driver.findElement(By.id( id: 'password'));
    return elm.sendKeys(password);
});

// Finally Mock User clicks the login button.
Then(/^user clicks the login button$/, function () {
    let elm = driver.findElement(By.id( id: 'login-btn'));
    return elm.click();
});

// Mock User has been logged in.
Then('user should be successfully logged in', function () {
    console.log('User successfully Logged in');
    return driver.close();
});
```

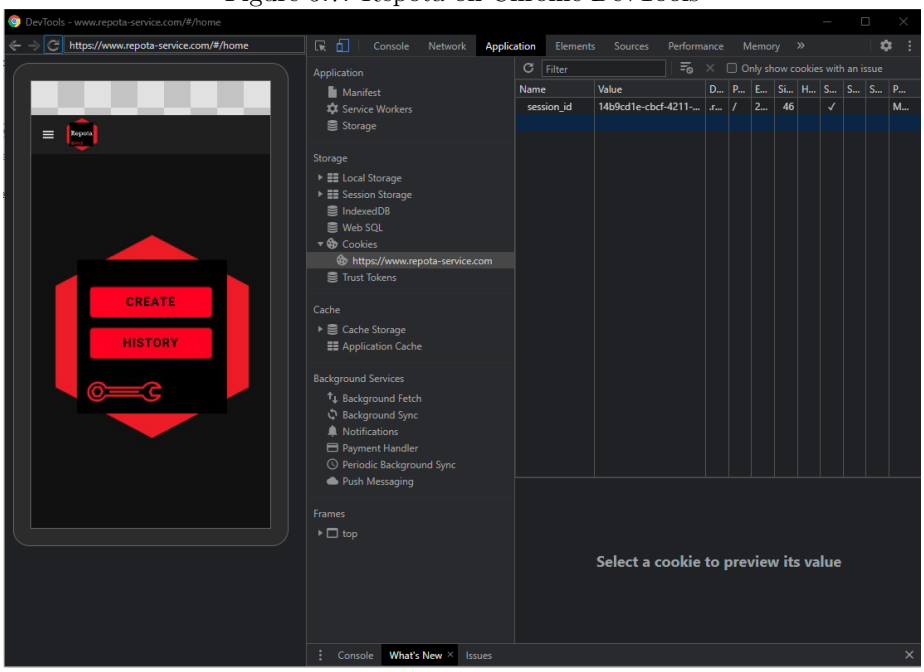
Figure 5.6: Login Steps Pass

```
....User successfully Logged in

.


1 scenario (1 passed)
5 steps (5 passed)
0m10.835s (executing steps: 0m10.816s)
```

5.3 Mobile Phone Functionality

Repota was tested for its functionality on Android through Chrome's DevTools. The DevTools was primarily used to check if all requests were successful in verifying all the features were working. Another vital check was especially to ensure cookies were being set on mobile phones as the app relies heavily on them.

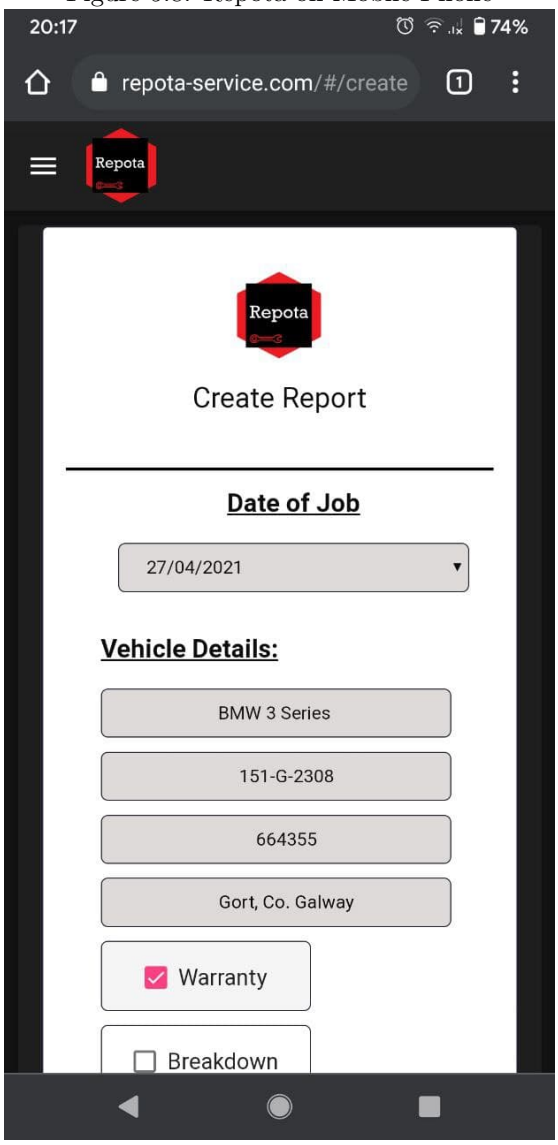
Figure 5.7: Repota on Chrome DevTools



5.3.1 Responsiveness

Repota is responsive on multiple devices. Ionic Lab helped to achieve this immensely. The app's responsiveness has been tested on the browsers Google Chrome, Firefox and Microsoft Edge. While for mobile phones, it was tested on Android and iPhone. The figure below shows a screenshot of the app's create page on a Google Pixel 3a with the Chrome app.

Figure 5.8: Repota on Mobile Phone



5.4 Issues Encountered

This section goes over the issues encountered that caused long pauses in the development of Repota.

5.4.1 Export to PDF

The initial library used for exporting reports to PDFs was html2pdf. The PDF was of excellent quality for computers and mobile phones. Unfortunately, due to Angular's stylesheets connected to multiple pages, html2pdf caused a DOM (Document Object Body) Exception for "Sharing constructed stylesheets in multiple documents is not allowed." This exception caused the hamburger menu button to disappear. Many attempts were made to catch the exception and prevent it. I even went to the extreme of disconnecting the Export Page from all the style sheets and had the CSS right in the HTML, and this did not change anything. I did find a solution for this exception for html2canvas on one of the issues on the html2canvas GitHub Repository. I attempted to translate the solution for html2pdf with no success. Then I created an issue on the html2pdf GitHub Repository. Shortly after this, I came across the libraries dom-to-image and jsPDF. This setup for the PDF caused no exceptions. On computers, the PDF comes out just as expected. However, on mobile phones, the reports comes out half the size of the PDF. Again I made attempts to have the PDF requirements resize when on mobile phones, but this resulted in the report being squashed on the PDF. All this resulted in having only the PDFs exported on computers to have an acceptable format. I got no response for the issue on the html2pdf GitHub Repository. It was then seen that it was better to have the export to PDF working right on one device rather than two devices with a DOM exception.

5.4.2 Finding the right 3rd Party API

Finding the right 3rd party API was not a straight forward process. Two other APIs were tried and tested before discovering Back4App. The first one found is called VINCARIO. I had to request an API key from them. They accepted the request and sent code examples to use their service in Python, PHP and Node.js. I was able to translate the Python example to Go and they asked me for the code so they could have it for other users wanting to use their service with Go and I happily sent it on to them. Once the code was in place it was realised this API has access to only one vehicle at a time. This limitation ruled it out as a very large function or a loop would have been needed to get multiple vehicles at one time. The second is named Car Makes and Models Database API. This API has access to more than one vehicle at time but they require one credit for each request. I was given twenty credits to start off with but this soon ran out and I attempted to pay for more credits but the API's PayPal service was down and still is months later. I contacted them to let them know it was down and I received no response. As a result this API was ruled out

too. Some more research and exploring was done and Back4App was finally discovered which, suited the needs perfectly.

Chapter 6

Conclusion

6.1 Project Conclusion

The purpose of this project was to develop a service report, SaaS application for automobile technicians using MySQL, OpenAPI, Go, Angular, and Ionic at a full-stack level. The finalised application provides users with the service to create, store, edit, and delete service reports on a hosted environment. This paper has used research and analyzed appropriate methods for this project to further explain the application's overall functionality. The main objectives are mentioned below and each shows whether they were achieved.

Overall, the final product was a success and it achieved the objectives; the scope proved to be perfect size for a one-person project, resulting in a successful output from careful consideration, research, implementation and analysis through trial and error. The combination of MySQL, OpenAPI, Go, Angular, and Ionic with AWS worked out quite well. With them, the Repota application turned out to be a great success as it has all its intended functionality. Through this report, it is evident that Repota could be very successful in suitable workplaces. Possible next steps to further enhance the application would be developing the exportation of PDFs for service reports in a format that is suitable for mobile phones to support efficiency. Also, to have it available on the Google Play Store and Apple App Store. Furthermore, the front-end technologies could be altered to be developed in other modern frameworks in the similitude of Flutter; this would support the use of the application on mobile phones and tablets. Eventually, it could expand to be used in numerous workplaces and not limited to service reports on automobiles.

Building a full-stack application gave me the experience of creating what a real-world project expects; this includes a scrum board to plan and implementing development procedures in a structured time frame, problem-solving by either formulating a solution or improvising. Testing through integration and high-

level behavior to ensure the application is working successfully. Using GitHub as a version control software appropriately, regularly monitoring essential commits when changes and implementations have been made will provide a backup for the application if absolutely necessary. Reflecting on the first steps in the initial setup of the project which included just the basic database, back-end and front-end and observing the differences made through targeting and achieving the intended objectives of the project has thoroughly expanded my skills as a developer.

6.2 Objectives Achieved

- Comprehensive MySQL database with all the required information. ✓
  - Users and Service reports.
    - Fully Constructed
- OpenAPI specification of Front-end and Back-end APIs. ✓
- Robust and RESTful back-end connected to the database in Golang. ✓
  - CRUD Operations for service reports.
    - Fully Implemented
  - Account system for users; register, login and logout.
    * Microservices - Sessions and Cookies for users.

    - Fully Implemented
  - 3rd Party API access for vehicle information.
    - Fully Implemented
- User friendly front-end with the Angular and Ionic Framework connected to the back-end. ✓
  - Report - CRUD operation pages.
    - Fully Implemented
  - Account - Register, Login and Logout pages.
    - Fully Implemented
  - Option to export reports to PDFs.
    - Implemented with a limitation for mobile phones.
  - Front-end UI responsiveness for multiple devices.
    - Fully Implemented
- Testing of back-end and front-end. ✓
  - Suite of integration tests for the back-end's functionality. ✓
  - Suite of high level behaviour tests for the Front-end. ✓
- Database, Back-end and Front-end hosted on AWS. ✓

- Database hosted on a EC2 Ubuntu Virtual Machine. ✓
- Back-end hosted with Elastic Beanstalk. ✓
- Front-end hosted with S3 Bucket. ✓
- HTTPS for hosted back-end and front-end. ✓

Bibliography

[1] Faithfull, M. - SecureTeam (2020). Desktop Application Security Assessment.
URL: https://bit.ly/3slk7KY

[2] Winston W Royce. Managing the development of large software systems. In proceedings of IEEE WESCON, number 8, pages 328–338. Los Angeles, 1970.
URL: https://bit.ly/3fBuVRz

[3] Ionos - Digital Guide - Waterfall methodology
URL: https://bit.ly/3mdYudg

[4] Atlassian. (2020). Agile project management.
URL: https://www.atlassian.com/agile/project-management

[5] Pahuja, S. (2015, February 15). Scrum for Individuals. InfoQ.
URL: https://www.infoq.com/news/2015/02/personal-scrum/

[6] OpenAPI Specification - Version 3.0.3 — Swagger. (2021).
URL: https://swagger.io/specification/

[7] Costanza, P., Herzeel, C., & Verachtert, W. (2019). A comparison of three programming languages for a full-fledged next-generation sequencing tool. BMC Bioinformatics, 20(1), pages 3–5.
URL: https://doi.org/10.1186/s12859-019-2903-5

[8] Morris, R. (2016, July 28). Rob Pike: Geek of the Week. Simple Talk.
https://bit.ly/3vOGIBb

[9] Structs Instead of Classes - OOP in Go. (2020, June 3). Golang Bot.
URL: https://golangbot.com/structs-instead-of-classes/

[10] Stamat, D. (2013, May 12). How We Went from 30 Servers to 2: Go. The Iron.Io Blog.
URL: https://blog.iron.io/how-we-went-from-30-servers-to-2/

[11] Gorilla Mux - GitHub
URL: https://github.com/gorilla/mux

[12] Gin Gonic Documentation
URL: https://gin-gonic.com/docs/introduction/

[13] Zharova, E. Z. (2021, February 3). The state of Go — The GoLand Blog.
URL: https://bit.ly/3gWKnZr

[14] Angular Tag Trends. (2021). Stack Overflow.
URL: https://insights.stackoverflow.com/trends?tags=angular

[15] Angular Documentation - Router. (2021). Angular.
URL: https://angular.io/guide/router

[16] Documentation - TypeScript for the New Programmer. (2021). TypeScript Language.
URL: https://bit.ly/3xR4hel

[17] MySQL :: Getting Started with MySQL. (2021).
URL: https://dev.mysql.com/doc/mysql-getting-started/en/

[18] MySQL :: MySQL 8.0 Reference Manual :: 1.2.1 What is MySQL? (2021).
URL: https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html

[19] MySQL :: MySQL 5.6 Reference Manual :: 14.2 InnoDB and the ACID Model
URL: https://dev.mysql.com/doc/refman/5.6/en/mysql-acid.html

[20] What is a virtual machine (VM)? (2021). Redhat.
URL: https://red.ht/2RtyW0x

[21] Introduction to Amazon S3 (4:31). (2021). Amazon Web Services, Inc.
https://aws.amazon.com/s3/

[22] Introduction to AWS Elastic Beanstalk. (2021). Amazon Web Services, Inc.
URL: https://aws.amazon.com/elasticbeanstalk/

[23] Pedamkar, P. (2021, March 24). Linux vs Windows. EDUCBA.
URL: https://www.educba.com/linux-vs-windows/

[24] What is Docker—Container Registry. (2021). Oracle Ireland.
URL: https://bit.ly/3aXYnOv

[25] Niels Provos and David Mazières - A Future-Adaptable Password Scheme - Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference
URL: https://bit.ly/3xKUfeM

Chapter 7

Appendices

7.1 GitHub Repository & Web Application URLs

- GitHub Repository: https://github.com/johnshields/Repota-App
- Repota Web Application: https://www.repota-service.com

7.2 How to Install and Run

- All changes (besides MySQL details, the database, and 3rd Party API config files) have been made for anyone to run locally.
- Requirements
  - Git = https://git-scm.com/downloads
  - NPM = https://www.npmjs.com/get-npm
  - Golang - 1.15.3 = https://golang.org/dl/
  - MySQL = https://dev.mysql.com/downloads/shell/

Clone the GitHub Repository

- Open a directory of your choice in Command-Line and enter:
  - git clone https://github.com/johnshields/Repota-App.git

Repota Front-end

- Open the repository directory in Command-Line and enter:
  - cd repota/repotaApp
  - npm install @angular/cli
  - npm install
  - npm run build
  - ionic serve
    - Running on Localhost: `http://localhost:8100`

Horton Back-end

- Ensure the following have been fulfilled:
  - Create the Database in a MySQL Console.
    - Location - `/database/REPOTA_DB.sql`
  - Edit the config.ini file with your MySQL details.
    - Location - `/horton/go/config/config.ini`
  - Acquire an App ID and API key from Back4App to use their service. (First 10k requests are free).
    - URL - `https://www.back4app.com/`
  - Add in the App ID and API key into the .txt config files.
    - Location - `/horton/go/config/`
- Open the repository directory in Command-Line and enter:
  - cd horton
  - go mod download
  - go build && go run main.go
    - Running on Localhost: `http://localhost:8080/api/v1`