



REPOTA
A SERVICE REPORT APPLICATION

**Applied Project And Minor Dissertation
B.Sc. (Honours) in Software Development**

BY
JOHN SHIELDS

APRIL 28, 2021

Advised by: Andrew Beatty
DEPARTMENT OF COMPUTER SCIENCE AND APPLIED PHYSICS
GALWAY-MAYO INSTITUTE OF TECHNOLOGY (GMIT)

Contents

1	Introduction	3
1.1	Project Introduction	3
1.2	Resources URLs	4
1.3	Chapter Descriptions	5
2	Methodology	7
2.1	Project Scope & Goal	7
2.2	Development Methodology	8
2.2.1	Waterfall	9
2.2.2	Agile	9
2.2.3	Project & Time Management	10
2.2.4	Verification & Testing	11
3	Technology Review	13
3.1	GitHub	13
3.2	OpenAPI	14
3.3	Golang	15
3.3.1	Go Web Frameworks	17
3.3.2	How Go fitted the Back-end	17
3.4	Angular & Ionic	18
3.4.1	Routing	19
3.4.2	How Angular & Ionic fitted the Front-end	19
3.5	MySQL	19
3.5.1	Relational	19
3.5.2	ACID	20
3.5.3	How MySQL fitted the Database	20
3.6	Amazon Web Services	21
3.6.1	Virtual Machines	21
3.6.2	S3 Bucket	21
3.6.3	Elastic Beanstalk	22
3.6.4	How AWS fitted the project's hosting	22

4	System Design	23
4.1	The Database	23
4.2	The Back-end	26
4.2.1	Horton API Specification	26
4.2.2	Go Horton	29
4.2.3	Account & Session System	30
4.2.4	Report System	35
4.2.5	3rd Party API	39
4.2.6	Elastic Horton	40
4.3	The Front-end	41
4.3.1	Repota & Horton Integration	42
4.3.2	Account Pages	43
4.3.3	Report Pages	47
4.3.4	AuthGuard	58
4.3.5	Repota UI	60
4.3.6	Repota Bucket	62
5	System Evaluation	63
5.1	Horton Robustness	63
5.2	Repota Robustness	65
5.3	Mobile Phone Functionality	67
5.3.1	Responsiveness	67
5.4	Issues Encountered	69
5.4.1	Export to PDF	69
5.4.2	Finding the right 3rd Party API	69
6	Conclusion	71
7	Appendices	75
7.1	GitHub Repository & Web Application URLs	75
7.2	How to Install and Run	75

List of Figures

1.1	Application Logo	4
2.1	Agile VS Waterfall	10
2.2	GitHub Projects	11
2.3	GitHub Issues	11
3.1	Repota App Repository	13
3.2	OpenAPI Spec - Swagger Editor	14
3.3	Schema Models - Swagger Editor	14
3.4	Go VS Java & C++ [6]	15
3.5	CLI Go Docs on fnt	16
3.6	Go Functions inside the Back-end's openapi Package	16
3.7	Gin Error Handling	17
3.8	Angular & Ionic Home Page Structure	18
3.9	Route of Repota's Home Page	19
3.10	MySQL InnoDB Transaction for inserting data into two tables.	20
4.1	Example of a Full Report	24
4.2	JOIN QUERY for all Report information	24
4.3	Database Schema	25
4.4	AWS Server	25
4.5	Login Operation	27
4.6	API Paths	28
4.7	Schema Models	28
4.8	InLineObject Struct	29
4.9	Session Struct	29
4.10	WorkerAccount Struct	29
4.11	JobReport Struct	29
4.12	Register	31
4.13	RegisterNewUser	31
4.14	isValidAccount	31
4.15	createSessionId	31
4.16	generateSessionId	31
4.17	User in the workers and session tables	32

LIST OF FIGURES

4.18 Login	32
4.19 verifyDetails	32
4.20 Browser Cookie	33
4.21 Logout	33
4.22 CheckForCookie	34
4.23 Account System	34
4.24 CreateReport Function	35
4.25 Prepare MySQL INSERT QUERIES	36
4.26 Go MySQL Transaction for two INSERTs	36
4.27 GetReports Function	36
4.28 Go MySQL JOIN to SELECT all Reports	37
4.29 ID Parameter from Report	37
4.30 Go MySQL JOIN to SELECT a single Report	37
4.31 Go MySQL UPDATE a Report	38
4.32 DeleteReport Function	38
4.33 Report System	39
4.34 Back4App Vehicle Data	39
4.35 Elastic Beanstalk Environment	40
4.36 Secure Horton	40
4.37 Repota's Pages	41
4.38 Repota & Horton Connection	42
4.39 Account Service Connection to API	42
4.40 InLineObject Interface	43
4.41 JobReport Interface	43
4.42 Register Method	44
4.43 Regular Expression pattern for Password	44
4.44 Username is already taken - Error	45
4.45 Login Method	45
4.46 Password is incorrect - Error	46
4.47 Hamburger Menu - Logout Button	46
4.48 Logout Method	47
4.49 History Page	47
4.50 ngOnInit Method - Load all user's Reports	48
4.51 User has no Reports	48
4.52 Display Page	49
4.53 ngOnInit Method - Get Report by ID	49
4.54 Report not found	49
4.55 JobReport Object & Push Object to API	50
4.56 Checkboxes - If/Else Statement	51
4.57 ngOnInit method - Back4App Data	51
4.58 Create Page - HTML data list	52
4.59 loadReportById Method	52
4.60 Edit Page - HTML data list	53
4.61 Export Page	54
4.62 onExportPDF Method	54
4.63 Service Report PDF	55

4.64	Delete Report Confirmation	56
4.65	deleteReport Method	57
4.66	Error Message Handlers	57
4.67	JSON extracting & parsing of error message	58
4.68	Please login first	58
4.69	loggedIn Method	59
4.70	canActivate Method	59
4.71	Blocked Routes	59
4.72	Already logged in message	60
4.73	Home Page	61
4.74	About Page	61
4.75	Account Page	61
4.76	Secure Repota	62
5.1	Payload of mock user details	63
5.2	Test Register - POST request	64
5.3	Register test pass	64
5.4	Login Feature	65
5.5	Login Steps	66
5.6	Login Steps Pass	66
5.7	Repota on Chrome DevTools	67
5.8	Repota on Mobile Phone	68

Chapter 1

Introduction

1.1 Project Introduction

In an achievable manner and at a Bachelor of Honors standard, I aimed to develop a service report application for automobile technicians titled 'Repota' for the *Applied Project and Minor Dissertation*. The inspiration for this application came from my brother, who is employed in a workplace that specializes in the supply and service of new and used Mobile Plant and Equipment. Their reports are generally carried out on paper or with Microsoft Excel. The application's features would have to consist of a user account system, CRUD (Create, Read, Update and Delete) operations for the reports, and an option to export them to PDFs. The application's production aimed to be at a full-stack level. Meaning the application would have 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. Throughout this dissertation I will be going through the technologies of the application, how it was designed and how it all came to together as a whole.

Figure 1.1: Application Logo



1.2 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 Application.
- Database
 - Script of the Application's Database.
- Workings
 - Miscellaneous files for rough work.

1.3 Chapter Descriptions

Methodology

Methodology discusses the project's scope, its goal, and any other ideas that were considered. Along with information on service reports, the development methodology used for the project's development approach, its time management, and an 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 throughout the front-end and back-end. Including a section for issues encountered that caused long pauses in development.

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 new technologies as possible. Original ideas for this project were initially the uniformity 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 Vitae (CV) to companies that their CV related to. On the app, the user would upload their CV. That CV would be then sent through a 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 find all these 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 being from a creative perspective and personal interest. Old noir films and parallel universes inspired the game. Making a mix of these two genres was quite exciting and meant 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.

The chosen idea was to be a service report application for automobile technicians to fill out service reports for auto dealerships and rental companies. This idea was inspired by my brothers line of work. The company he is employed with specializes in the supply and service of new and used Mobile Plant and Equipment. This app was originally focused on being a desktop app, but it was believed that this app would be very suitable to work on a tablet or a

mobile phone. So, therefore, the Angular and Ionic framework was chosen for the front-end. The app needed to be trusted with work-related data. Therefore, the desktop app focus was not ideal as some app's data may needed to be temporally held on the computer running it. Thus making the data less secure than it would be on a back-end database server. [1]

This app would need a quick and robust back-end and Google's relatively new language Golang (Go) was chosen. This language is relatively new compared to other languages and had not been covered in the course's modules. Go is based of C and is proven to be up to the speed of other languages such as Java and C++. Go being an open-source language made finding suitable packages and web frameworks a challenge. There are several web frameworks for Go out there, such as Gorilla, Martini and Gin Gonic. After researching and testing Gin proved to be the best web framework to use as it proved to have the necessary requirements. After the research and testing was done Go seemed to be the a very good choice for the back-end.

A reliable and strong database was essential as this project purpose was to be used in a working environment meaning 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. MySQL was chosen for the database as it is proven to have these mentioned qualities.

The entire application would need to be hosted through a cloud platform. AWS (Amazon Web Services) provides a first year free tier. AWS seemed to be a great 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.

The Five Phases of Waterfall Life-cycle

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

[3]

With Waterfall, each stage is 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 required to have most likely the back-end 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 with less hassle than Waterfall. Agile employs the scrum framework, which involves sprints which are fixed-length work iterations. Each sprint has four ceremonies that provide structure.

The Four Ceremonies of Scrum

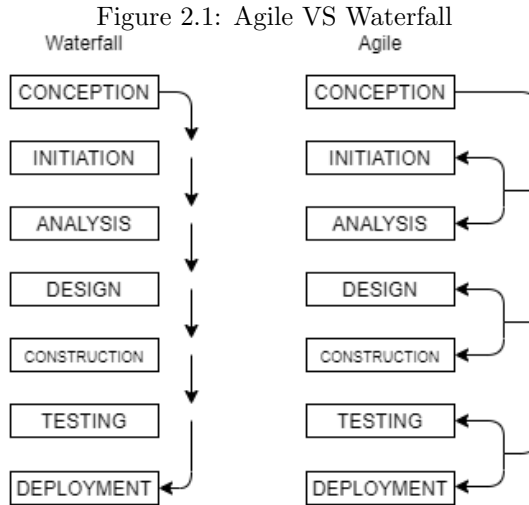
- Sprint Planning
- Sprint Review
- Sprint Retrospective
- Daily Scrum

[3]

Personal Scrum

Personal Scrum is an Agile methodology that employs scrum practices for one-person projects. Observation, adaptation, incremental elaboration, prioritizing and sizing work, and time-boxing are all used in this approach 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. 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. These days were usually allocated to the end of the week, and during college time off, more days were designated to increase productivity time.



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, but they 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. Therefore it was chosen. 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

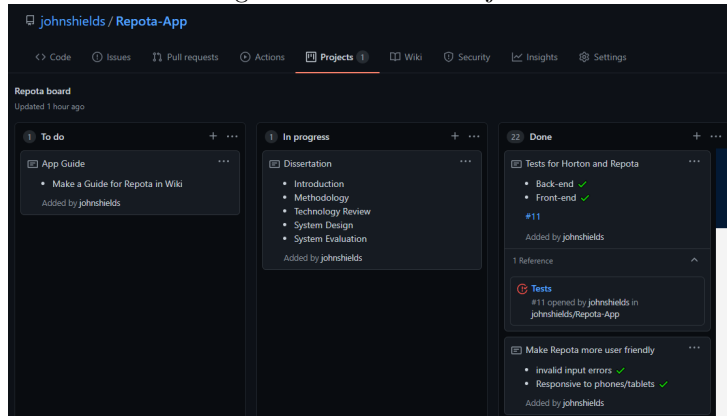
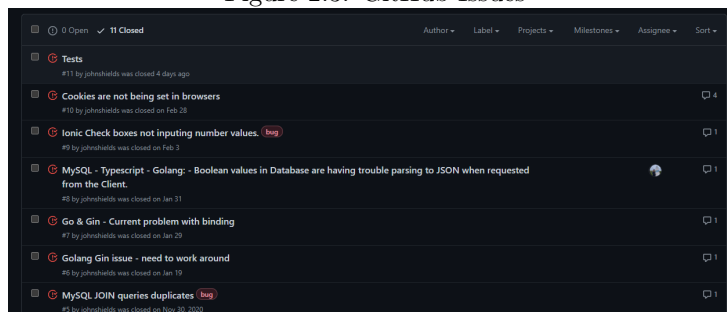


Figure 2.3: GitHub Issues



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 to be dealt with was CORS. Data from the back-end was unable to cross over to the front-end. 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 working, 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.

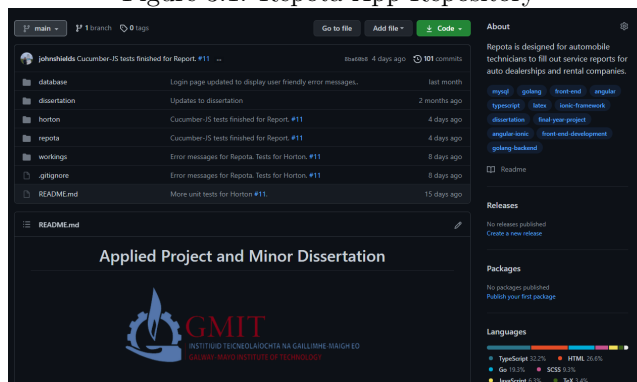
Chapter 3

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 and the issues. It was a wise choice for the project and should be enforced a lot more to use in other modules for projects/assignments.

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 allow for the implementation of specifications for these blueprints. These blueprints can be written in YAML, which is in a human-readable format.

Swagger Tools was highly used for prototyping the project's, server, and client stubs along with their Schema Models. With Swagger, server stubs can be easily designed with the necessary HTTP requests and responses. To follow the server's stubs, the client stubs get generated with the inter-grated design to allow it 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 and easy to use for a first-time user.

Figure 3.2: OpenAPI Spec - Swagger Editor

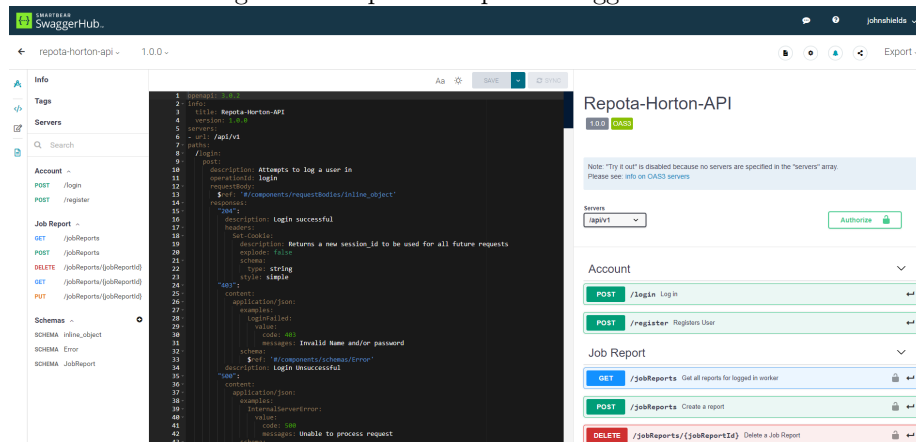


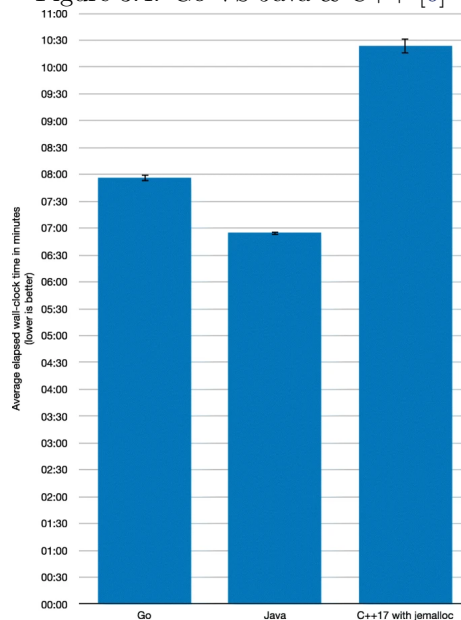
Figure 3.3: Schema Models - Swagger Editor



3.3 Golang

Golang (Go), designed by Google, is an loosely based Object-Oriented Programming (OOP) Language. 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++ (See Figure 3.4). [6] When it comes to OOP, Go is especially great for refactoring and abstraction. Go does not uses classes like Java, the closest thing to a class in Go is a Struct. [7] Go files can pick up functions from others easily as long they are in the same package. A function can be taken from one file and be put into another and still be able to be picked up in the original file 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.

Figure 3.4: Go VS Java & C++ [6]



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. Identifying problems in Go is a more straightforward 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 accessed through the CLI (Command Line Interface). 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 be used to 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 3.5: 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 3.6: 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 a request router and dispatcher for assigning incoming requests to the appropriate handler. Mux stands for "HTTP request multiplexer". `mux.ServeMux` is similar to `http.ServeMux`. Incoming requests are compared to a list of registered routes, and the path that meets the URL or other requirements. [8]

Gin Gonic

Gin Gonic has a Martini-like API but with an efficiency that is up to 40 times faster. Martini is another Go web framework, which is no longer maintained. Gin is fast, "Radix tree-based routing, small memory footprint. No reflection. Predictable API performance." A chain of middle-wares can handle incoming HTTP requests. Gin's error management makes it simple to gather all of the errors during a HTTP search. Gin allows the creation of custom HTTP responses errors (Figure 3.7). If a panic occurs in an HTTP request, Gin can recover it, making it "Crash-free". [9]

There is a limitation with Gin. Endpoints sharing a common path may result in issues. 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.

Figure 3.7: Gin Error Handling

```
if err != nil {
    log.Println(v...: "No cookie found", err)
    c.JSON( 403, models.Error{Code: 403, Messages: "User has no cookie."})
    Logout(c)
}
```

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 seemed like a great language to learn.

Learning Go for the first time for the Back-end was a welcome challenge. It took some time to become comfortable and with 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 prototype but failed to meet the requirements to deal this CORS. Gin Gonic

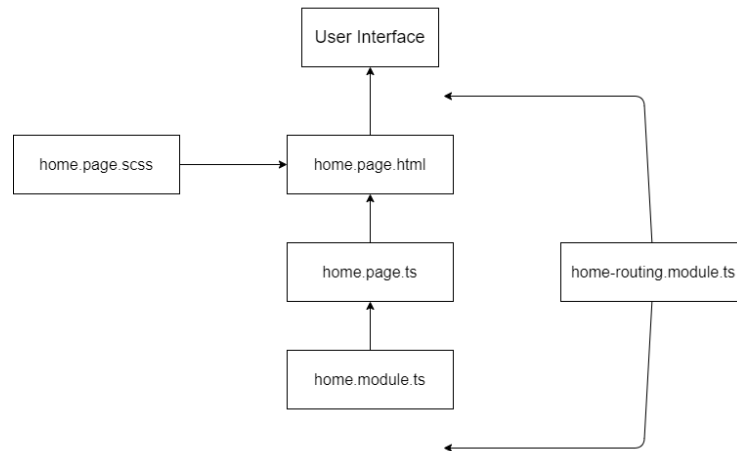
proved to be the best framework to use, but it required Docker to generate the OpenAPI stubs for Gin rather than Swagger's standard options. After finalizing the Back-end, it can be said that Go is a very nice language.

3.4 Angular & Ionic

There are many JavaScript and TypeScript frameworks out there, including Angular, React, Vue.js, Meteor, Ember.js, ETC. These technologies are primarily JavaScript frameworks. A new JavaScript framework almost appears 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 web app framework and with the combination of Ionic can make web apps into mobile apps. Angular is relatively slow to run apps, but they can be updated once they are up and running without having to rerun them. Angular apps are created with components, and Ionic turns them into pages. The figure below shows the structure of a page.

Figure 3.8: 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

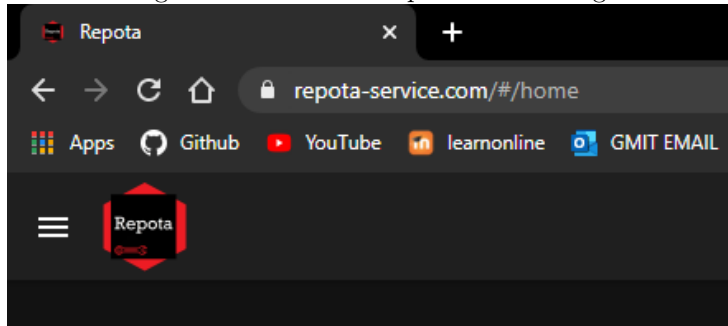
Path Routing to connect the page to the App's main routing

- home-routing.module.ts

3.4.1 Routing

When an Angular and Ionic app is first initialized, template pages are setup. 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.

Figure 3.9: Route of Repota's Home Page



3.4.2 How Angular & Ionic fitted the Front-end

Before this project Angular and Ionic were used in the past to create apps for other modules and personal interest. Meaning that these frameworks were chosen for their comfort-ability and have been successful for other projects. The main reason why an uncomfortable app framework was not chosen as it was thought that learning and becoming component with a new language for the back-end was more important and would be more useful beyond college. Ionic did 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 is a database management system for relational databases (RBMS) that use the ACID model. It is the most common open-source database in the world because it is efficient, simple to set up, and use. [10] MySQL is quite robust, both in it's setup for database tables and its functionality.

3.5.1 Relational

Rather than storing all of the data in one big storeroom, a relational database stores it in 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, views, rows, and columns. [11]

3.5.2 ACID

The ACID model is a collection of database design principles that place a premium on aspects of reliability that are crucial for business data and mission-critical applications. ACID stands for Atomicity, Consistency, Isolation, and Durability. Atomicity is used in InnoDB transactions with statements such as 'COMMIT' and 'ROLLBACK'. Transactions are a way of updating multiple tables at one time (See Figure 3.10). Transactions are usually used when two tables are connected by the use of foreign keys. 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. [12]

Figure 3.10: 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 vehicle 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 is 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

Amazon Web Services (AWS) is a cloud platform with many services. Some of these services are EC2, Route 53, Machine Learning, Elastic Beanstalk, S3 Bucket, 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

A Virtual Machine (VM) is a virtualization or emulation of a computer. 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. EC2 provides the service to have many VMs with different operating systems.

Windows VS Linux

Since Windows OS is a commercial product, only the developers who developed it have the authority to alter it. The Linux operating system 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. [13]

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. 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 deploying and scaling web apps and services. These web apps and services are usually developed in Java, .NET, PHP, Node.js Python, Go, and Docker. Docker can be used through EB. The addition of Docker and the EB CLI makes deploying to EB a satisfying process. Similar to S3 Buckets EBs are HTTP by default. HTTP can be overcome with a somewhat similar process to making a Bucket secure. 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 set of platform-as-a-service products that use OS-level virtualization to deliver applications in 'containers', as Docker refers to them. A container is a file format for packaging software that encapsulates all of an app's code and dependencies in a standard format that allows it to run quickly and efficiently across multiple computing environments.

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 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 worker, session, job report (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 collisions with other workers. Their worker name is set 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 is set this is to be used for 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

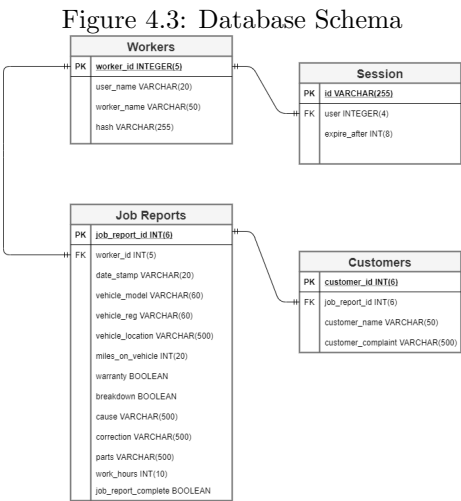
Figure 4.2: JOIN QUERY for all Report information

```
-- 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 QUERIES. 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 in 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

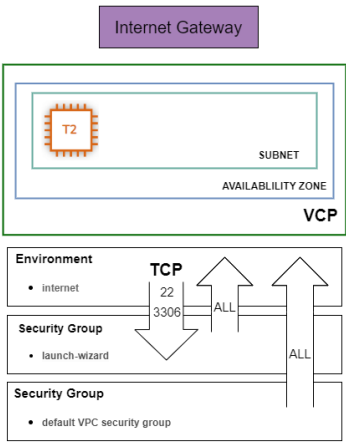
with the addition of 'job_report_complete' to allow workers to keep track of their incomplete reports. The figure below shows the finalized database design.



Database Hosting

The database is hosted on a AWS EC2 Linux Ubuntu 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 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. 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

```

/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. The user and report details were based off the first version of the database.

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

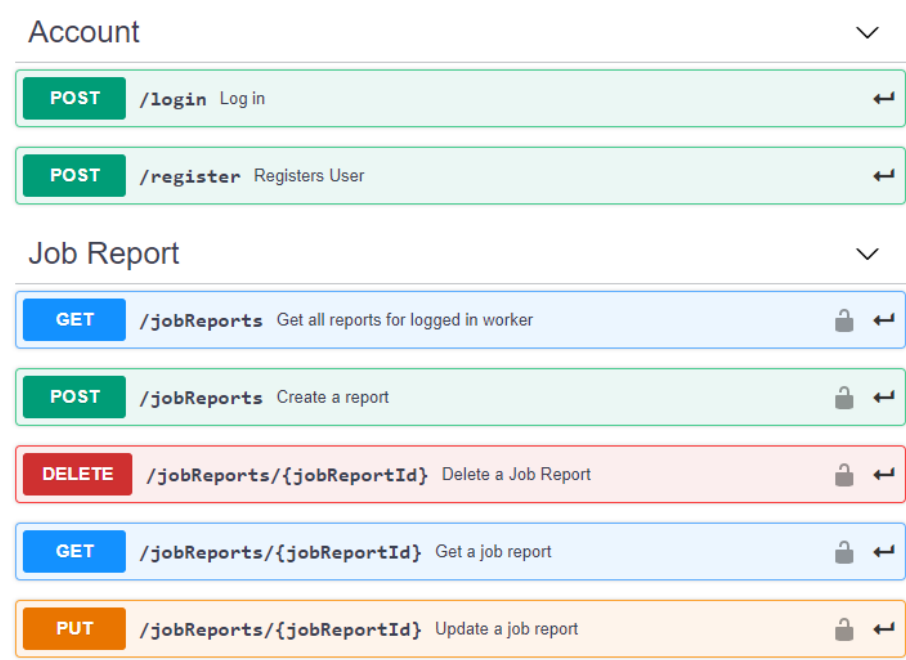


Figure 4.7: Schema Models



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 accommodate 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. The following are the handler functions and their paths.

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 to ensure they do not already exist in the database. The password gets hashed using Go's package 'bcrypt', which implements the Provos and Mazières's bcrypt adaptive hashing algorithm. [14] 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 4.12-4.16). 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 4.17). This session is designed for Cookies to let the user make all future requests to the API.

Figure 4.12: 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: 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()
    //db := mocks.MockDbConn() // for unit tests

    // 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 4.13: RegisterNewUser

```
func RegisterNewUser(c *gin.Context, username, name, password string) error {
    db := config.DbConn()
    //db := mocks.MockDbConn() // for unit tests

    // 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) {
        log.Println( v... "\nUsername taken")
        c.JSON( code 409, models.Error( Code: 409, Messages: "Username is already 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
    }

    fat.Println( a... "\n[INFO] Printing MySQL Results for user account...\n", result)

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

Figure 4.15: createSessionId

```
func createSessionId(username string) (error, models.Session) {
    db := config.DbConn()
    //db := mocks.MockDbConn() // for unit tests

    // 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 {
        fat.Println(err.Error())
        return errors.New( err.Error(), models.Session{} )
    }

    fat.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 {
        fat.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 4.16: 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 4.17: User in the workers and session tables

worker_id	username	worker_name	hash
141	john_chields	John Chields	\$2a\$10\$tttINuB.yZaZUMIKSbQmF.jzRVIL8.MLHdreo3S45u9Pt3jovWNO

id	user	expire_after
3593ca97-b45d-4511-820e-835f0a8588f3	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 there is an existing session for the user that 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 4.18-4.20).

Figure 4.18: 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 exists.
        if removeSession(wa.Id) {
            // Create new session ID for user who logged in.
            err, session := createSessionId(username)

            if err != nil {
                log.Println(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 "reputa-service.com", secure true, httpOnly: false)
                // User has been logged in and cookie has been set.
                c.JSON( code 204, cbj 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 4.19: 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 4.20: Browser Cookie

Name	Value	Domain	Path	Expires...
session_id	5fcd5d3e-ae72-4822-88a1-d0423edbf89a	.repota...	/	2021-0...

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 4.21). 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 4.22). The purpose of this function is to block requests on reports if a user have no cookie. In other words an unauthorized user.

Figure 4.21: 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 exists.
    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()
}
```

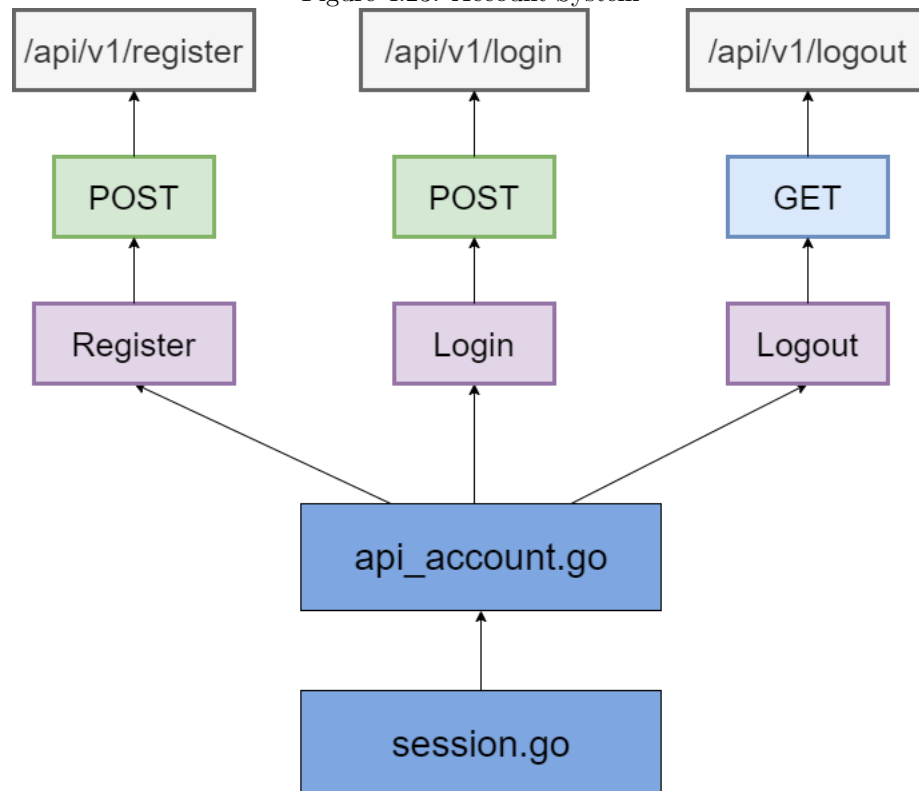
Figure 4.22: CheckForCookie

```
// 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("session_id")

    // Return false if no cookie is found.
    if err != nil {
        c.JSON(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 4.23: 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 4.24). The data is then prepared as MySQL INSERT QUERIES for the tables: `jobreports` and `customers` in the database (Figure 4.25). 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 4.26). 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 4.24: 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 4.25: 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 4.26: Go MySQL Transaction for two INSERTs

```
// Begin MySQL transaction 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 transaction.

if err != nil {
    log.Println(v: "MySQL 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 begins with a GET request from the client. The user is first checked with the functions `CheckForCookie` and `isValidAccount` (Figure 4.27). 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 4.28) 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 4.27: GetReports Function

```
func GetReports(c *gin.Context) {
    db := config.DbConn()
    //db := mocks.MockDbConn() // for unit tests

    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 4.28: 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 4.29). 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 4.30) to select it from the database, and the report is sent to the client with a status code of 200.

Figure 4.29: ID Parameter from Report

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

Figure 4.30: Go MySQL JOIN to SELECT a single 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 4.31: 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 4.32: 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
    }

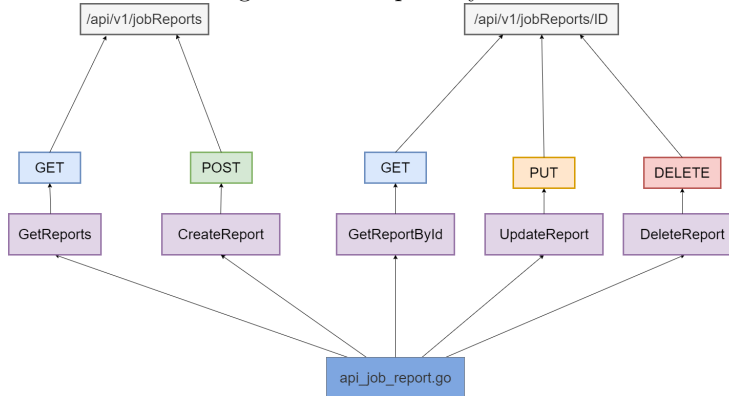
    // Create query to delete the report with its requested ID.
    res, err := db.Exec( query: "DELETE FROM jobreports WHERE job_report_id=?", reportId)
    if err != nil {
        log.Printf( format: "Report failed to delete.")
        c.JSON( code 500, obj nil)
    }

    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 4.33: 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. The figure below shows a sample of the data from Back4App.

Figure 4.34: Back4App Vehicle Data

```

x Headers Preview Response Initiator Timing Cookies
{...}
results: [{"Make": "Audi", Model: "Q3", createdAt: "2020-01-27T20:44:17.665Z", objectId: "Z8gPP9dBM",...}]
[0 - 99]
> 0: {"Make": "Audi", Model: "Q3", createdAt: "2020-01-27T20:44:17.665Z", objectId: "Z8gPP9dBM",...}
> 1: {"Make": "Chevrolet", Model: "Malibu", createdAt: "2020-01-27T20:44:17.665Z", objectId: "cpt81CINSL",...}
> 2: {"Make": "Cadillac", Model: "Escalade ESV", createdAt: "2020-01-27T20:44:17.665Z", objectId: "LuzYwMYJpM",...}
> 3: {"Make": "Chevrolet", Model: "Corvette", createdAt: "2020-01-27T20:44:17.665Z", objectId: "r0kHakOBKP",...}
> 4: {"Make": "Acura", Model: "RLX", createdAt: "2020-01-27T20:44:17.665Z", objectId: "7G1VT2pSNO",...}
> 5: {"Make": "Chevrolet", Model: "Silverado 2500 HD Crew Cab", createdAt: "2020-01-27T20:44:17.665Z", objectId: "4qL9FAU2S",...}
> 6: {"Make": "BMW", Model: "3 Series", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}
> 7: {"Make": "Chrysler", Model: "Pacifica", createdAt: "2020-01-27T20:44:17.665Z", objectId: "ALbinxfzMI",...}
> 8: {"Make": "Chevrolet", Model: "Colorado Crew Cab", createdAt: "2020-01-27T20:44:17.665Z", objectId: "9eRgmVvii",...}
> 9: {"Make": "BMW", Model: "X3", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}
> 10: {"Make": "Acura", Model: "TLX", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}
> 11: {"Make": "Chevrolet", Model: "Silverado 3500 HD Crew Cab", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}
> 12: {"Make": "BMW", Model: "7 Series", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}
> 13: {"Make": "Ford", Model: "Fusion", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}
> 14: {"Make": "Buick", Model: "Envision", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}
> 15: {"Make": "Audi", Model: "SQ5", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}
> 16: {"Make": "Audi", Model: "R8", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}
> 17: {"Make": "Chevrolet", Model: "Traverse", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}
> 18: {"Make": "Acura", Model: "MDX", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}
> 19: {"Make": "INFINITI", Model: "QX80", createdAt: "2020-01-27T20:44:17.665Z", objectId: "q6d5H3Yd",...}

```

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 4.35) 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 4.36).

Figure 4.35: Elastic Beanstalk Environment

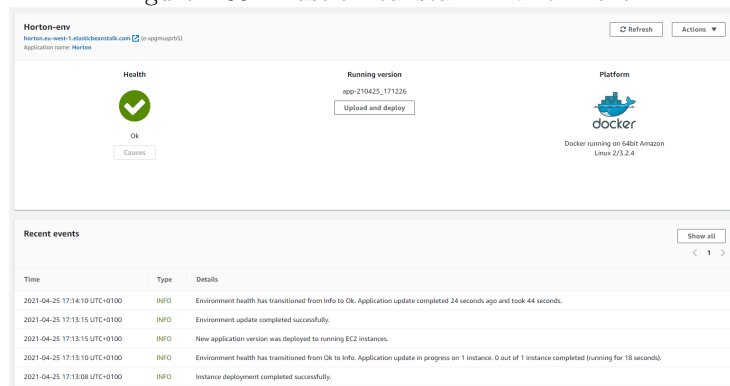
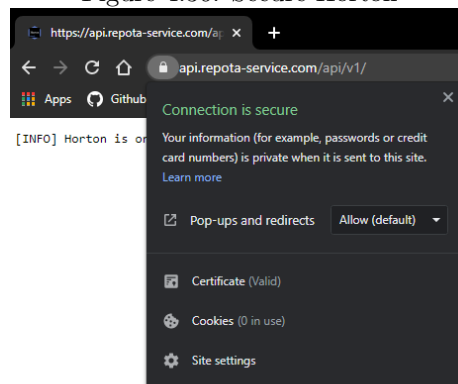
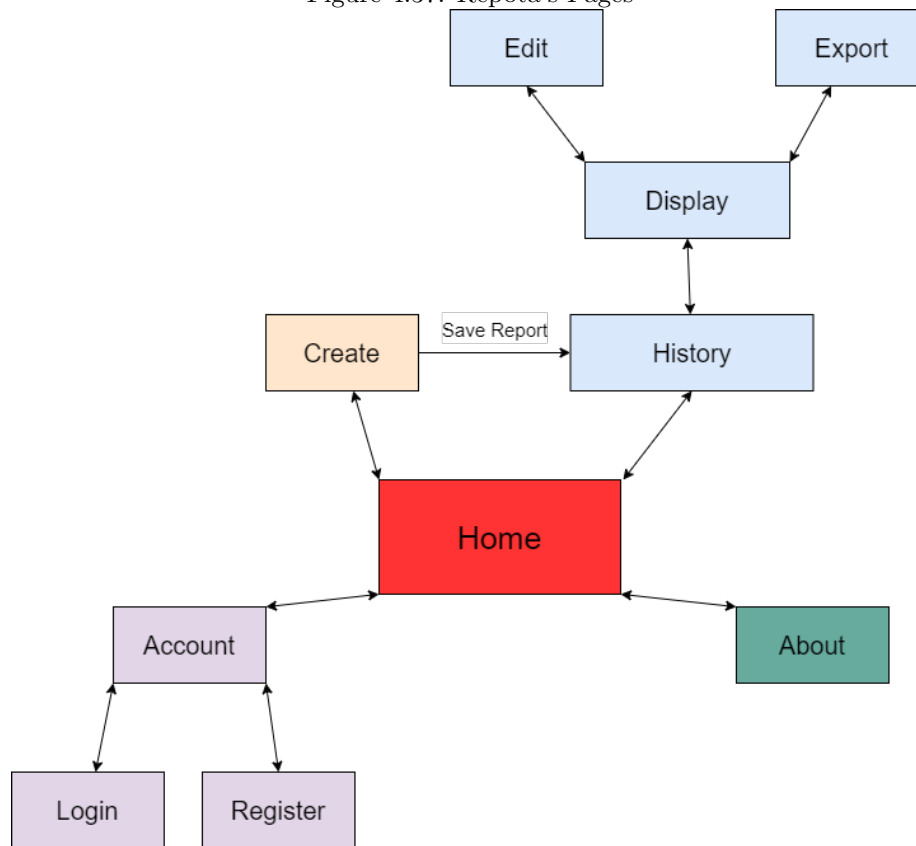


Figure 4.36: Secure Horton



4.3 The Front-end

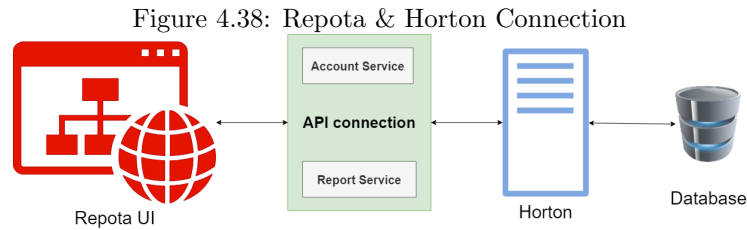
Figure 4.37: Repota's Pages



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 above).

4.3.1 Repota & Horton Integration

Account & Report Service



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 4.38). This class allows the HTTP methods POST for register and login and GET for logout. `JobReportService` 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 4.39: Account Service Connection to API

```

@Injectables()
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 4.40: InLineObject Interface

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

Figure 4.41: 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 4.41) 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 must have at least eight characters, one uppercase and lowercase letter, one number, and a special character. These requirements are ensured on the HTML side with a regular expression to provide extra security (Figure 4.42). Once the user has the form filled out, they click the register button on the UI to send the data, and once the data is in the database, the user is navigated to the login page. If there are any errors with the registration process (e.g., the username is already taken), an error message is displayed on the UI to the user (Figure 4.43).

Figure 4.42: 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);
  });
}

```

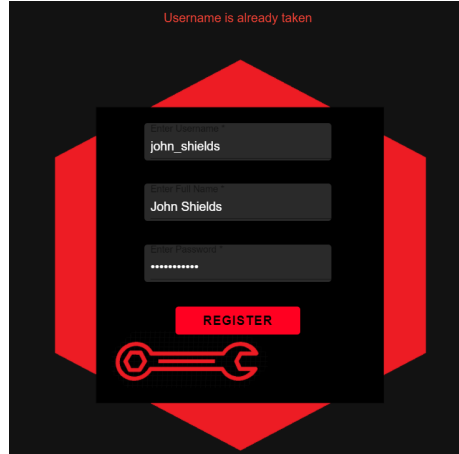
Figure 4.43: Regular Expression pattern for Password

```

<!-- 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])(?=.*?[0-9])(?=.*?[!@#&^_~`~\W_])\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 4.44: Username is already taken - Error



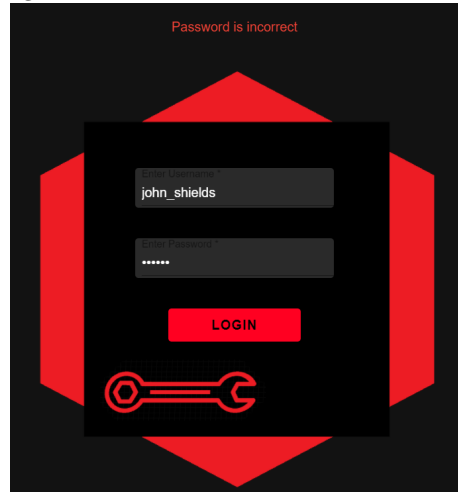
Login

User login is managed on the Login Page. This process is handled by the login method (Figure 4.43). 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 data is then checked with what is in the database for the user. 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 4.44).

Figure 4.45: 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 4.46: Password is incorrect - Error



Logout

Logout is available to a logged-in user on any page of the front-end. The logout button is located in the hamburger menu (Figure 4.46). When a user clicks on the logout button, this calls the logout method (Figure 4.47). The 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 4.47: Hamburger Menu - Logout Button

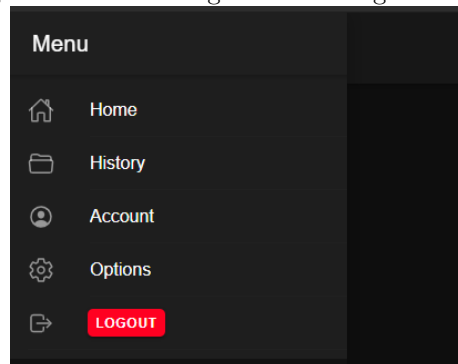


Figure 4.48: Logout Method

```

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 4.48). The reports are loaded in from the API by the JobReportService's getReports method. The method ngOnInit (Figure 4.49) calls getReports from JobReportService, and the reports are automatically loaded for the user when they navigate to this page. If the user does not have any reports, a message is displayed to notify them (Figure 4.50).

Figure 4.49: History Page

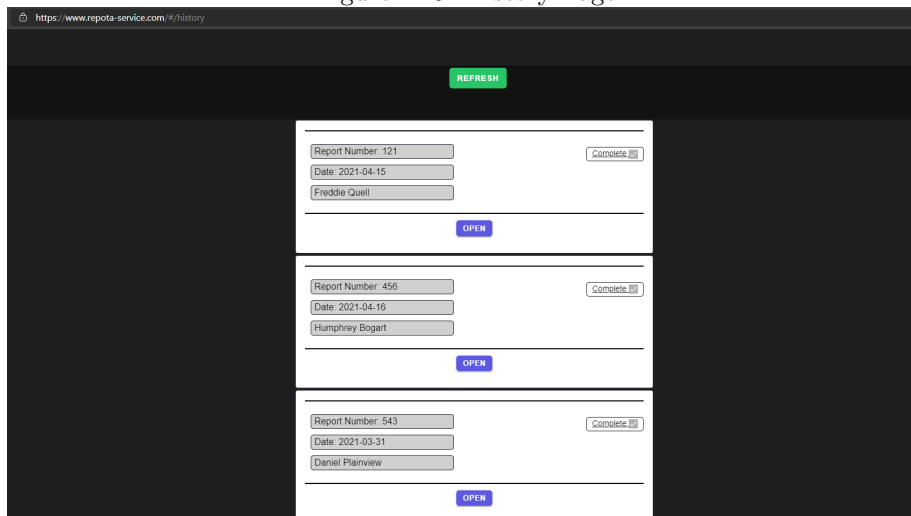
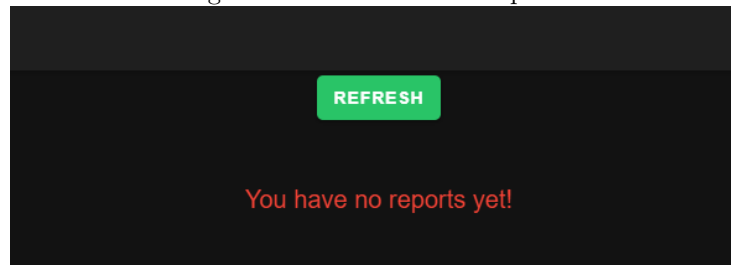


Figure 4.50: ngOnInit Method - Load all user's Reports

```
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 4.51: 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 with the options to edit, export it to a PDF and delete (Figure 4.51). The report is loaded by the ngOnInit method (Figure 4.52) 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 4.53).

Figure 4.52: Display Page

https://www.repota-service.com/#/display/456

Report

Report Number: 456
Date: 2021-04-25

Vehicle Details:

Model: Ford Mustang
Reg: 54-SF-135
Location: Furbogh, Co. Galway
Mileage: 1007538
Warranty: ☒
Breakdown: ☐

Customer Details:

Name: Humphrey Bogart
Complaint: The car needs new tyres

Work Details:

Cause: Worn out tyres
Correction: New tyres have been fitted
Parts: 4 TYRES
Work Hours: 1
Technician: John Shields
Complete: ☒

EDITPDFDELETE

Figure 4.53: 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 4.54: Report not found

https://www.repota-service.com/#/display/4565

Report Number 4565 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 4.54). 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 4.55). 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 through the `ngOnInit` method (Figure 4.56) from the API and is displayed to the user as a HTML data list (Figure 4.57).

Figure 4.55: 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 4.56: 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 4.57: 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 4.58: 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 4.58) in HTML data lists (Figure 4.59). 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 4.59: 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 4.60: 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 to the create page's structure for them.

The user submits the edit with the edit button. This submission calls the `editReport` method and corresponds 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 loaded 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 4.60). When a user clicks on the Export PDF button, this calls the method `onExportPDF` (Figure 4.61), which brings up a file explorer 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 4.62).

Figure 4.61: Export Page

Report Number: 456
Date: 2021-04-25

Vehicle Details:

Model: Ford Mustang
Registration: 54-SF-135
Location: Furbogh, Co. Galway
Mileage: 1007538
Warranty: ☒
Breakdown: ☐

Customer Details:

Name: Humphrey Bogart
Complaint: The car needs new tyres.

Work Details:

Cause: Worn out tyres.

Correction: New tyres have been fitted.

Parts: 4 TYRES


Work Hours: 1
Technician: John Shields
Complete: ☒

EXPORT PDF

Figure 4.62: onExportPDF Method

```
onExportPDF() {
  const content = document.getElementById( '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( 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 4.63: Service Report PDF


Report Number: 456
Date: 2021-04-25

<u>Vehicle Details:</u> <div>Model: Ford Mustang</div> <div>Registration: 54-SF-135</div> <div>Location: Furbogh, Co. Galway</div> <div>Mileage: 1007538</div> <div>Warranty <input checked="" type="checkbox"/></div> <div>Breakdown <input type="checkbox"/></div>	<u>Customer Details:</u> <div>Name: Humphrey Bogart</div> <div><u>Complaint:</u> The car needs new tyres.</div>
--	---

Work Details:

Cause:
Worn out tyres.

Correction:
New tyres have been fitted.

Parts: 4 TYRES

Work Hours: 1

Technician: John Shields

Complete ☒

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 4.63). When the user clicks 'OK' on the popup box, the method `deleteReport` (Figure 4.64) 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 4.64: Delete Report Confirmation

The image shows a web application interface for 'Repota'. A dark grey modal popup is centered on the screen, asking for confirmation to delete a report. The popup text reads: 'www.repota-service.com says Are you sure to delete Report Number 456?'. It has two buttons: 'OK' (blue) and 'Cancel' (grey). Below the popup, the main content area shows the 'Repota' logo, the report number '456', and the date '2021-04-25'. The report details are organized into three sections: 'Vehicle Details', 'Customer Details', and 'Work Details'. Each section contains several input fields with text or checkboxes. At the bottom of the page, there are three buttons: 'EDIT' (blue), 'PDF' (green), and 'DELETE' (red).

www.repota-service.com says
Are you sure to delete Report Number 456?
OK Cancel

Repota
Report Number: 456
Date: 2021-04-25

<u>Vehicle Details:</u>	<u>Customer Details:</u>
Model: Ford Mustang	Name: Humphrey Bogart
Reg: 54-SF-135	Complaint:
Location: Furbogh, Co. Galway	The car needs new tyres.
Mileage: 1007538	
Warranty <input checked="" type="checkbox"/>	
Breakdown <input type="checkbox"/>	

Work Details:

Cause:

Worn out tyres.

Correction:

New tyres have been fitted.

Parts: 4 TYRES

Work Hours: 1

Technician: John Shields

Complete ☒

EDIT PDF DELETE

Figure 4.65: 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 4.46). 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 4.66). 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 4.66: Error Message Handlers

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

Figure 4.67: 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 4.66). 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 4.68) 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 4.69) in the AuthGaurd class. canActivate is then used in the class AppRoutingModuleModule to block the pages listed above (Figure 4.70).

Figure 4.68: Please login first

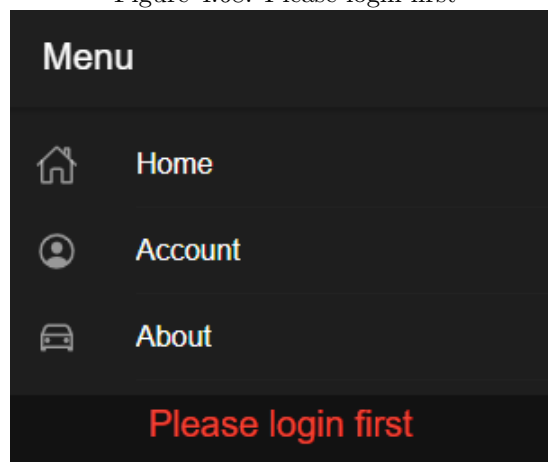


Figure 4.69: loggedIn Method

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

Figure 4.70: 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;
  }
}
```

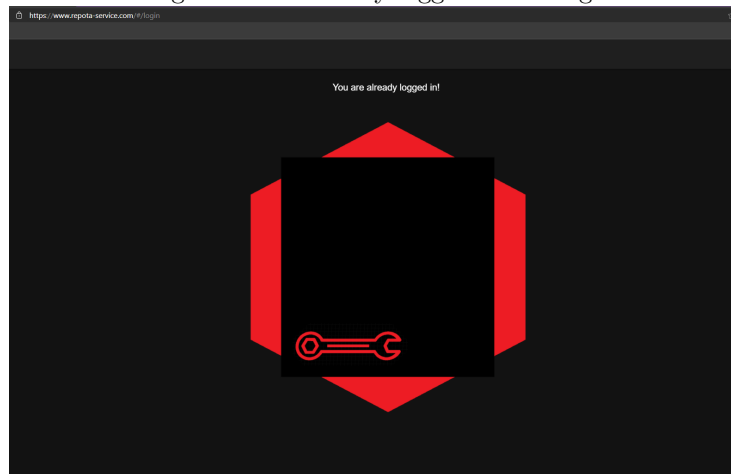
Figure 4.71: Blocked Routes

```
{
  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 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 4.71).

Figure 4.72: 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 4.72). The about page (Figure 4.73) can be accessed 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 4.74). 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 4.73: Home Page

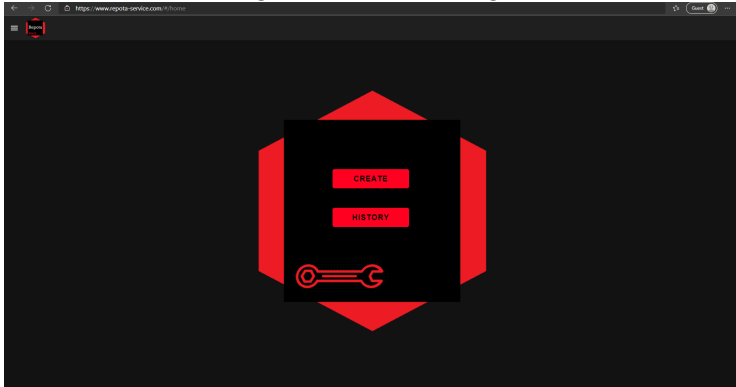


Figure 4.74: About Page

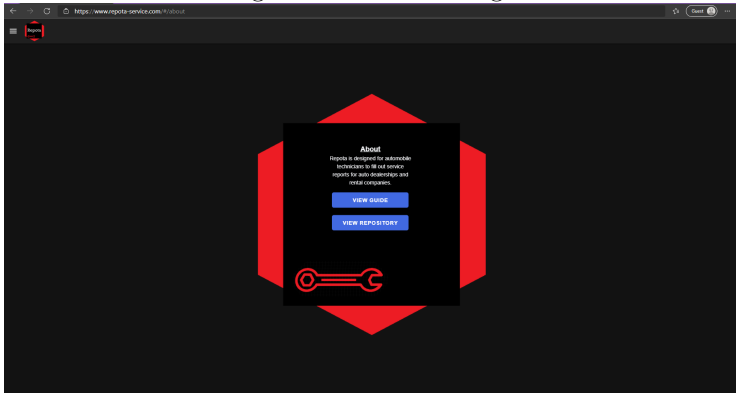
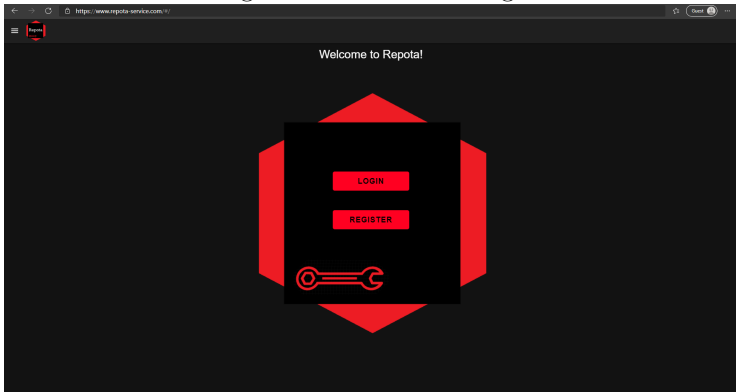


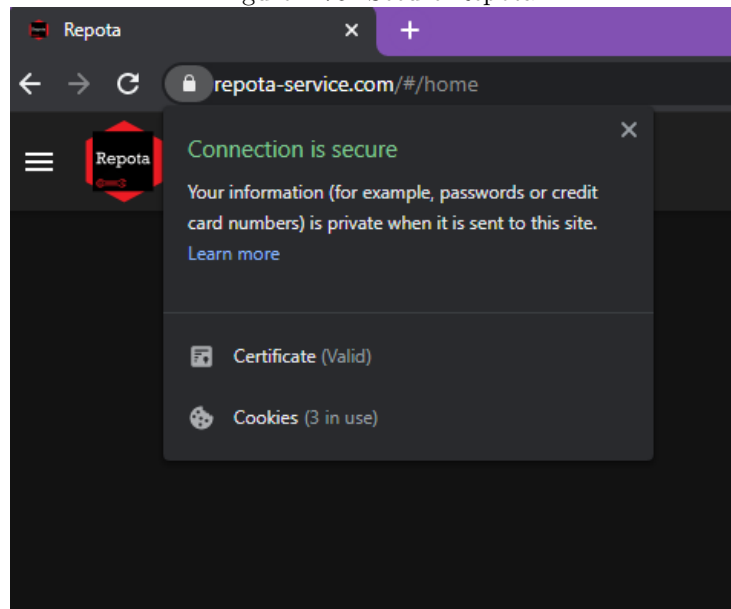
Figure 4.75: Account Page



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 4.76: 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 behavior-driven 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

```
// 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 "([^\s]*)"/, 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 "([^\s]*)"/, 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

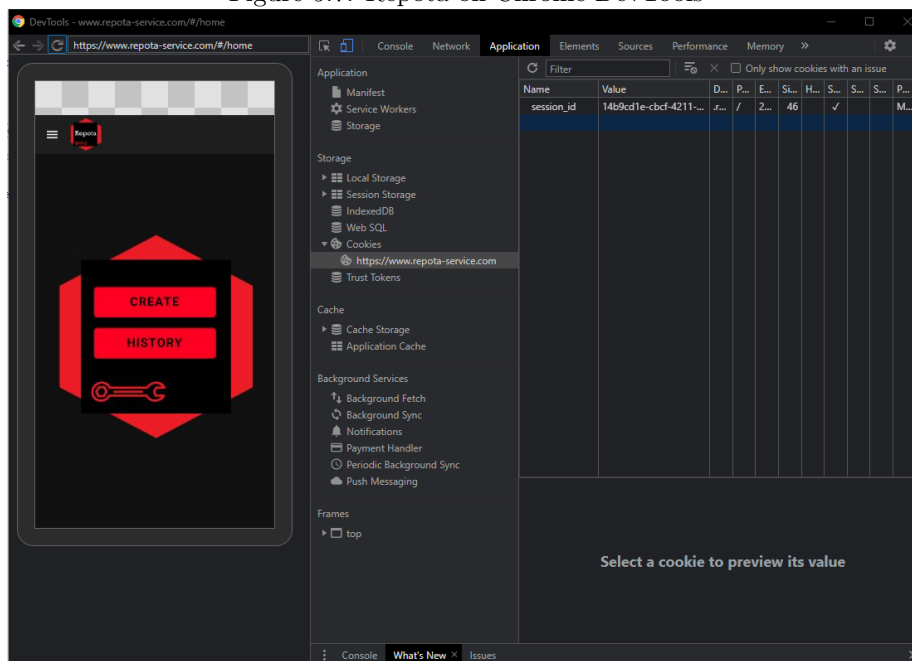
```
....User successfully Logged in
*

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

The screenshot displays the Repota mobile application interface on a smartphone. At the top, the status bar shows the time as 20:17, along with icons for alarm, Wi-Fi, cellular signal, and a 74% battery level. The browser's address bar shows the URL `repota-service.com/#/create`. Below this is a dark navigation bar with a hamburger menu icon on the left and the Repota logo in the center. The main content area is white and features the Repota logo at the top, followed by the heading "Create Report". A horizontal line separates the heading from the form fields. The first field is labeled "Date of Job" and contains the date "27/04/2021" with a dropdown arrow. Below this is the section "Vehicle Details:", which includes four stacked input fields containing "BMW 3 Series", "151-G-2308", "664355", and "Gort, Co. Galway". At the bottom of the form are two checkboxes: "Warranty" (checked) and "Breakdown" (unchecked). The bottom of the screen shows the standard Android navigation bar with back, home, and recent apps buttons.

20:17 74%

repota-service.com/#/create

Repota

Repota

Create Report

Date of Job

27/04/2021

Vehicle Details:

BMW 3 Series

151-G-2308

664355

Gort, Co. Galway

☒ Warranty

☐ Breakdown

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. I made many attempts 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, but it was 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 and a post on Stack Overflow. 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 came 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 or the post on Stack Overflow. 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 got no response. As a result this API was ruled out too. Some

more research and exploring was done and Back4App was finally discovered and suited the needs perfectly.

Chapter 6

Conclusion

Repota turned out to be a great success. It has all its intended functionality. I aimed to develop a service report application for automobile technicians using MySQL, OpenAPI, Go, Angular and Ionic. The finalized Repota provides the service of creating, storing, editing, and deleting of service reports. I believe the idea of Repota could one day be very successful. If I am to continue with this idea, I would definitely continue using MySQL, OpenAPI, and Go. Perhaps I would change the front-end technologies to the likes of Rust or Flutter as these JavaScript/TypeScript based frameworks can become obsolete relatively quickly since there is always a new "big" one every 1-2 years.

Overall, I am very happy with the final product and what I accomplished. I think the scope was a perfect size for a one-person project, resulting in a satisfactory output from hard work, problem-solving, and improvisation. While having slow initial compile times, Angular and Ionic being familiar frameworks made the development of Repota a somewhat satisfying process. Swagger turned out to be a surprisingly easy tool for a first-time user and is great for creating an OpenAPI specification. Learning Go for the first time was a welcome challenge that resulted in a pleasing outcome for Horton and proved to be a great language to learn as it is very fast and quite object-oriented. The combination of MySQL, Go and Angular, and Ionic worked out quite well.

Independently learning a new language has improved my skills as a developer immensely. Before this year, I was getting through by the skin of my teeth. In the third year of this course, I had to repeat a module for the first semester. This setback helped me mature a lot about college and prepare my head for this project. With learning Go came a lot of debugging since I was uncomfortable with it. However, through this learning process, I have become better at other languages used throughout this course. From learning Go, I have a better understanding of programming itself as I can now formulate solutions that could possibly lead to something successful rather than having no idea where to begin with, which I suffered a lot from throughout the past years in the course.

Having said all that, I am very proud of myself for getting to the other side, and I would like to thank my lectures 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 giving me the inspiration that turned into a successful final product.

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] 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>
- [7] Structs Instead of Classes - OOP in Go. (2020, June 3). Golang Bot.
URL: <https://golangbot.com/structs-instead-of-classes/>
- [8] Gorilla Mux - GitHub
URL: <https://github.com/gorilla/mux>
- [9] Gin Gonic Documentation
URL: <https://gin-gonic.com/docs/introduction/>
- [10] MySQL Documentation - Getting Started with MySQL
URL: <https://dev.mysql.com/doc/mysql-getting-started/en/c>
- [11] MySQL Documentation - What is MySQL?
URL: <https://bit.ly/3t1woVi>

- [12] MySQL Documentation - InnoDB and the ACID Model
URL: <https://dev.mysql.com/doc/refman/5.6/en/mysql-acid.html>
- [13] Pedamkar, P. (2021, March 24). Linux vs Windows. EDUCBA.
URL: <https://www.educba.com/linux-vs-windows/>
- [14] Niels Provos and David Mazières - A Future-Adaptable Password Scheme
- Proceedings of the FREENIX Track: 1999 USENIX Annual Technical
Conference
URL: [https://www.usenix.org/legacy/event/usenix99/provos/
provos.pdf](https://www.usenix.org/legacy/event/usenix99/provos/provos.pdf)

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 updated:
 - 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>