

# **Dissertation**

## **Web Application for Tracking Scrum Projects**

Luke Thompson – LDT4

**Supervisor:**

Dr Paula Severi

**Second Marker:**

Dr Hongji Yang

05/2019

Department of Informatics, University of Leicester

**Table of Contents**


Table of Contents .....	b
DECLARATION .....	d
Abstract .....	1
Introduction.....	2
Background Research .....	2
Introduction to Scrum.....	2
Scrum Roles.....	3
Product Owner .....	3
Scrum Master.....	3
Team Member .....	3
Sprint Artefacts .....	3
Tickets .....	3
Product Backlog .....	4
Sprint Backlog .....	4
Scrum Events.....	4
The sprint itself .....	4
Sprint Planning.....	4
Daily Scrum .....	5
Sprint Review .....	5
Sprint Retrospective.....	5
Poker Planning .....	5
Interface Theming.....	6
Requirements .....	6
Users .....	6
Ticket Management .....	6
Sprint Planning.....	7
Sprint.....	8
Retrospective .....	8
Specification and Design.....	9
Frontend Application .....	9
Service Discovery .....	11
API Proxy .....	11
API Services .....	11
Realtime – Team weighting service .....	11
Web Application for Tracking Scrum Projects .....	b

Domain Services .....	11
Database .....	12
Data structures .....	12
Implementation .....	13
Architectural Overview .....	13
Docker .....	14
Service Discovery .....	14
Backend.....	15
Language .....	15
Communication between services .....	16
REST API .....	17
Session Tokens .....	19
Password Security .....	21
Frontend .....	21
Language and Framework.....	21
State Management .....	23
API Communication .....	24
Features .....	25
Backlog Management .....	25
Active Sprint.....	29
Team weighting.....	30
Critical Appraisal .....	33
Critical Analysis .....	33
Users .....	33
Ticket Management .....	34
Sprint planning.....	34
Sprint.....	36
Retrospectives.....	36
Miscellaneous .....	36
Discussion of context .....	37
Personal Development.....	37
Conclusion .....	38
Bibliography.....	39
Glossary .....	40

**DECLARATION**

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

**Name:** Luke Thompson

**Signed:** 

**Date:** 02/05/2019

**Abstract**

Scrum is one of the leading methodologies for software development teams to use, and yet no existing solution is specifically tailored to this. In the project I build a software development tool that aids to integrate with the scrum process and provide a positive experience for the people using it. Functionality such as collaborative team weighting and integrated retrospectives have been developed to solve some of these issues.

## Introduction

The objective of this project is to develop a web-based software development tool that is specifically designed around Scrum and the various roles and events it introduces into any given software development process. Within the next section I will explain the relevant Scrum events and roles team members are to participate in. Following on from that I will cover what the software should do, what I built and how I built it. Finally, I will discuss the success of the final product and where it could go in the future.

Scrum is heavily used across the software engineering industry. Based on a survey across a wide array of software developers, it's found 62% of teams use Scrum [1]. Across the industry, there is a wide array of different issue tracking tools, each suitable to similar purposes. Most tools side on being overly flexible instead of being specialised, to ensure it fits any teams workflow.

The leading issue tracker across teams is Atlassian's Jira [2], which based on a survey taken in 2018 found 69% of developers use the software daily [3]. One of Jira's primary selling points is that it's flexible and can be used for any purpose, this means it's suitable for Scrum teams, Kanban Teams or even support desks. This happens to be one of it's downfalls. Being overly complex means people need a deep understanding of the software to use it, which adds extra overhead.

On the flip side, we have tools such as Trello [4], which features just a basic Kanban style board. Tickets can be created and moved between columns at will. This is great because it's simple, but the downfall is it's overly simple. It's impossible to use efficiently with Scrum due to limited customisability, which is why I've opted to build a tool that is specifically designed around Scrum and the processes and methodologies it provides.

## Background Research

In this section I will discuss Scrum and surrounding methodologies that heavily influence the end product.

### Introduction to Scrum

Scrum is a set of processes and techniques that are applied to a given software development process which are designed to make development easier, quicker and more efficient. These processes are formalised around several events which are spread throughout a given sprint. Some parts of Scrum are open to interpretation and not everyone will enforce Scrum in the same way, but the principles will remain the same. Parts of this

will be influenced by my own experiences on a Scrum team, while others will be taken from official documentation [5]

## Scrum Roles

Scrum mandates that several roles exist within a team, each role has different impacts on the process.

### Product Owner

The product owner is in charge of choosing what features, bugs and other tickets should be prioritised, whether they are worth doing and when they should be done. They are in charge of ensuring tickets are clear and concise for whomever works on them.

### Scrum Master

The Scrum Master oversees the sprint. They are in charge of enforcing the Scrum process and help aid others in the team understand Scrum. They handle ensuring both the team itself and outside contributors understand the methodology, what is causing impediments and what works well. They will also interface with the Product Owner to ensure tickets and goals are understood and ensure the Product Owner is understood and is effective at prioritising the backlog.

### Team Member

A team member is simply someone who is part of a Scrum team, taking part in a given sprint and working on backlog items. They most commonly communicate with the Scrum master, ensuring impediments are made known and any misunderstandings of the methodology are resolved.

## Sprint Artefacts

### Tickets

A ticket represents a unit of work, commonly known as user stories. It represents a problem or feature that should be implemented and describes accurately what should be done and how it should be completed and resolved.

## Product Backlog

The product backlog is a prioritised list of tickets which represent future development. It should be correctly prioritised at all times, with the most important functionality at the top, and lowering priority as you go down. This is only prioritised by a product owner, the team and scrum master may influence this, but only through discussion with the product owner.

## Sprint Backlog

The sprint backlog is a separate backlog that makes up the contents of a sprint. Typically, tickets are taken from the top of a product backlog and added to a sprint during a Sprint planning meeting, assuming they meet the defined sprint goal.

## Scrum Events

Scrum provides several events which are spread throughout a given sprint. Each marks a different stage in the process.

## The sprint itself

The sprint itself is an event which encapsulates one cycle of the process. It's time-boxed to a specific period which work must be completed within, this helps ensure any tickets are well understood and the scope is manageable. Typically, a sprint is between 2 weeks and a month, although this differs team to team.

## Sprint Planning

Sprint planning occurs at the beginning of a sprint, it's technically what forms the sprint. Typically, the team will decide on a goal and then choose any tickets that they feel meet this goal. A sprint is considered ready when the team is collectively happy that they will be able to complete all the work within a sprint, it's better to underestimate work than to overestimate work. How we estimate how much work can be fitted into a sprint is open to interpretation, but a common method to this is poker planning, and assigning each ticket story points. Poker planning will be discussed later. During the planning meeting it's common to discuss tickets, decide on key implementation details so the team is ready to go and work on them.



### Daily Scrum

Each day of the sprint a short meeting occurs at the exact same time and place. The development team will gather to discuss the what they have done, what they intend to do and any impediments that may be affecting their work. This is formalized into three questions:

- What did I do yesterday that helps meet the sprint goal?
- What will I do today to help meet the goal?
- Do I see any impediments that prevent me or the team from meeting that goal?

The short meeting (typically 15 minutes) provides adequate time for a team to discuss solutions to any problems, with the alternative being the meeting being over quicker. These meetings are timeboxed to ensure discussions aren't elongated, for some impediments it's likely the discussion only impacts a few people, so without timeboxing the meeting would waste others time.

### Sprint Review

The sprint review is the penultimate event in a sprint. The team will gather along with the product owner and any outside contributors (management, other teams which may be impacted). The work completed is discussed, the team discusses what to prioritise next which feeds into the next Sprint planning meeting. This process will also help refine the product backlog, adjusting for new priorities or changes in functionality.

### Sprint Retrospective

The sprint retrospective is the final stage of a sprint in which the team meets to discuss the previous sprint. This is lead by the Scrum master, who will ensure the meeting is productive. The team discusses what went well and worked well in the past sprint, they also discuss what didn't work so well and any impediments that prevented them from working. It's key that this meeting produces actionable feedback, so it's common to come up with ways to resolve these issues so the scrum master can discuss with the product owner and outside influences if necessary. While impediments are discussed during the daily sprint, the retrospective helps formalise this so there is a consistent avenue to discuss problems.

### Poker Planning

Poker planning [6] is a common methodology used to provide some weighting to tickets. The team decides on two tickets, anchors, which provide a low and a high point for reference. This is normally a simple ticket, with a very low weighting and a more complex with a high weighting. As the team is learning to weight tickets, these 2 tickets will provide

the scale. As teams practice poker planning more, the estimates are likely to become more and more accurate. Poker planning provides opportunities for discussion when reviewing the ticket but also during the voting phases. The team will decide on a scale, whether it be doubling numbers (1, 2, 4, 8, 16, etc..) or the Fibonacci sequence and when a ticket is presented, the team will assign this ticket a value from this set. If everyone decides on the same value then it's safe to assume that is the weighting of the ticket. If people choose different, then normally the lowest and highest values discuss their reasoning, the team will then revote and repeat the process until a consensus is met.

## Interface Theming

Ensuring the end product is intuitive and enticing to use is important, as any software development tool is used constantly by development teams, their entire workflow is based around this tool. Based on data from JetBrains, who produce an array of the leading development environments, the colour palette I opted for was primarily dark. This falls inline with their data that 77% of developers prefer dark themes in their development environments [7]. It's safe to assume they would want similar from their software tracking tools too.

## Requirements

The requirements for this project are able to be grouped into 5 feature areas:

### Users

- Users should be assignable to one of two roles, user and administrator. The administrator role should provide deeper control over the system.
- Users should be able to authenticate by username or email address and a password.
- Users should not be able to signup, but creatable by users with the administrative role.
- Users should be able to request password resets and reset their password through an email sent to them without intervention of an administrator.

### Ticket Management

- Tickets should be able to be created with an assigned title, description, priority and category.

- Ticket description should provide some rich functionality, enabling tables, images, etc to be included.
- Tickets should be able to be commented on using similar rich content.
- Tickets should be modifiable, displaying a revision history for any changes made.
- Tickets should be visible in a backlog which is divided into prioritised and pending. New tickets will automatically be in pending and once prioritised will be moved into the prioritised section.
- The backlog will display the ticket title and the priority.
- The backlog should be sortable, indicating the priority of the ticket.

### Sprint Planning

- A sprint should be created with a goal and duration.
- A sprint should contain a list of all team members that will be taking part in the sprint.
- Sprints should only be allowed to be created by administrators.
- A function should exist that enables tickets in the backlog to be weighted by a team.
  - If the team is put into this mode, each team member should see the current ticket and weightings for them to choose from (functional on mobile and desktop viewports).
  - Once everyone has chosen, weightings from each team member will be displayed.

- If there are major outliers, instigator of planning will have the option to exclude them and the software will take the average value within reason.
- If the weightings over a team are extremely dispersed, the team will be forced to revote (Mechanisms to discuss reasoning behind weightings is out of the scope of this tool).
- The software should provide the ability to forecast an estimation of the next sprint.
  - This should factor in sprint duration
  - This should factor in team members present compared to previous sprints
  - A method of obtaining feedback when work is completed should be factored in to improve weighting accuracy.

### Sprint

- The sprint should be a set of columns which tickets can be moved between to indicate status.
- The columns and how they may move between them should be definable by administrators.
- The sprint view should clearly display the goal and duration remaining which factors in any known public holidays.
- A view should be toggleable which enables you to view what has changed between yesterday. Tickets closed, tickets waiting testing, new tickets injected.

### Retrospective

- The retrospective should display a graph of the ticket process throughout the sprint.
- The retrospective view should display the ability to enter what a team thought went well and what could be improved on.
- The retrospective should allow the team to reflect upon the last sprints “could improve” and optionally add them to the current if they feel they didn’t improve.

## Specification and Design

The overall architecture of the system is split into several tiers. It uses the concept of microservices to ensure the project would be scalable and ensures that logic is encapsulated in single services avoiding language lock-ins. The structure starts with a single page application, which connects to a set of REST APIs. The REST APIs have access to a set of domain-oriented services which contain all of the logic. These services are able to call each other at will, ensuring each service is able to only do 1 job, well.

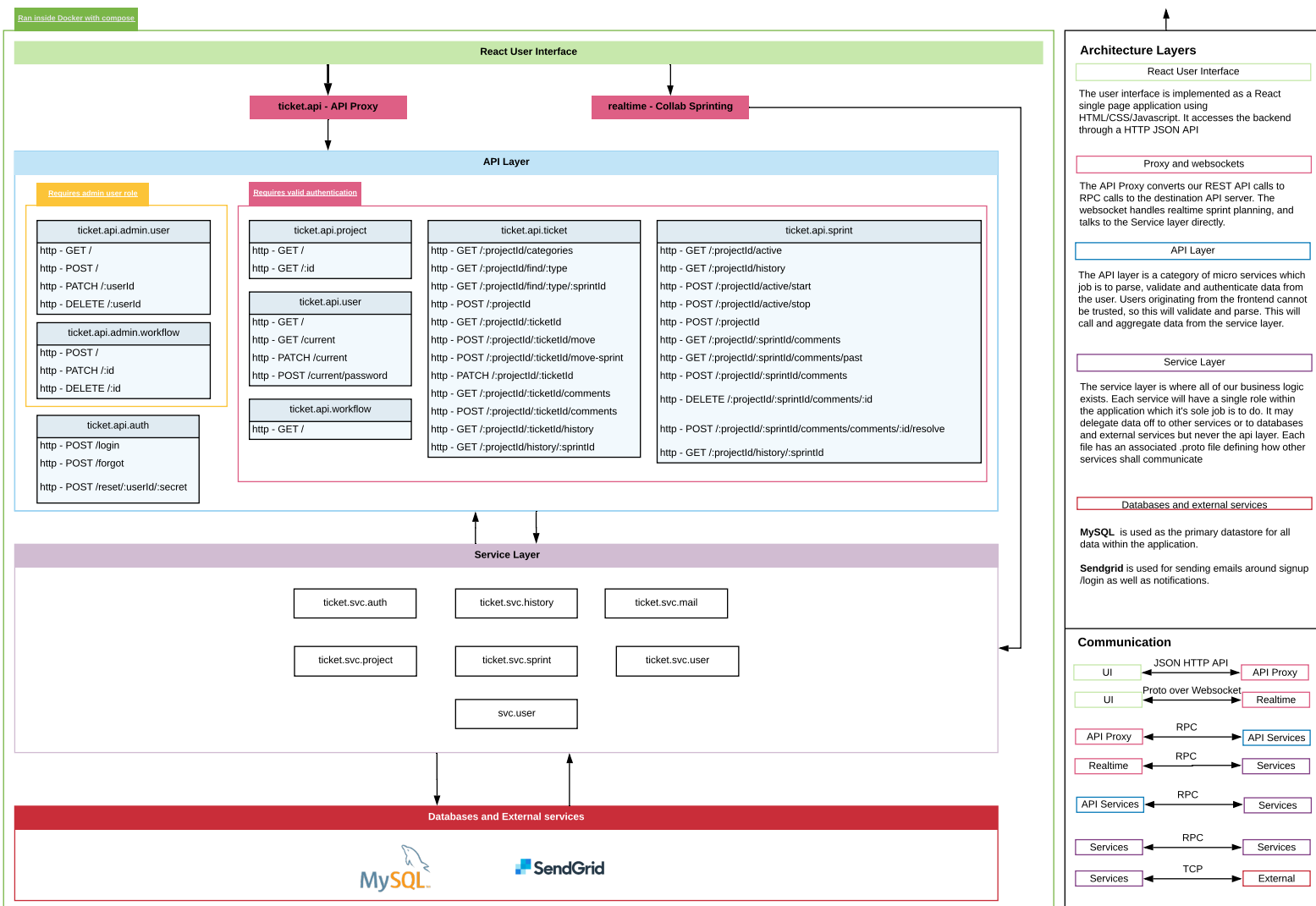


Figure 1 Service Architecture

## Frontend Application

The frontend is implemented as a single page application in React [8]. React is a framework based around Components, which encapsulate all of the logic for a single feature (or part of a feature). Components are reused and connected to make up a full application. React

provides the presentation layer but state is managed in a state container called Redux. Redux is framework agnostic, we are able to use it to store and manage our state. We are then able to connect the store to the React components to provide the state. When state is updated, the components are rendered.

The principles of redux are that state is stored in an immutable tree. This makes it reliable and prevents the chance of unexpected things mutating it. We have actions, which are fired by user interactions. Actions indicate we wish to change something in state, we may have an action to fetch the user data. When doing this, we need to set a “fetching” flag to true, so we can display that we are fetching data to the user. Redux has the concept of Epics, which listen for actions and trigger side effects. All side effects dispatch another action when completed (e.g calling the API to get user data), which in turn changes state. State itself is changed by a set of reducers, which take the current state, modify it and return a new copy. This is managed outside our presentation logic meaning business logic and UI logic are entirely separate.

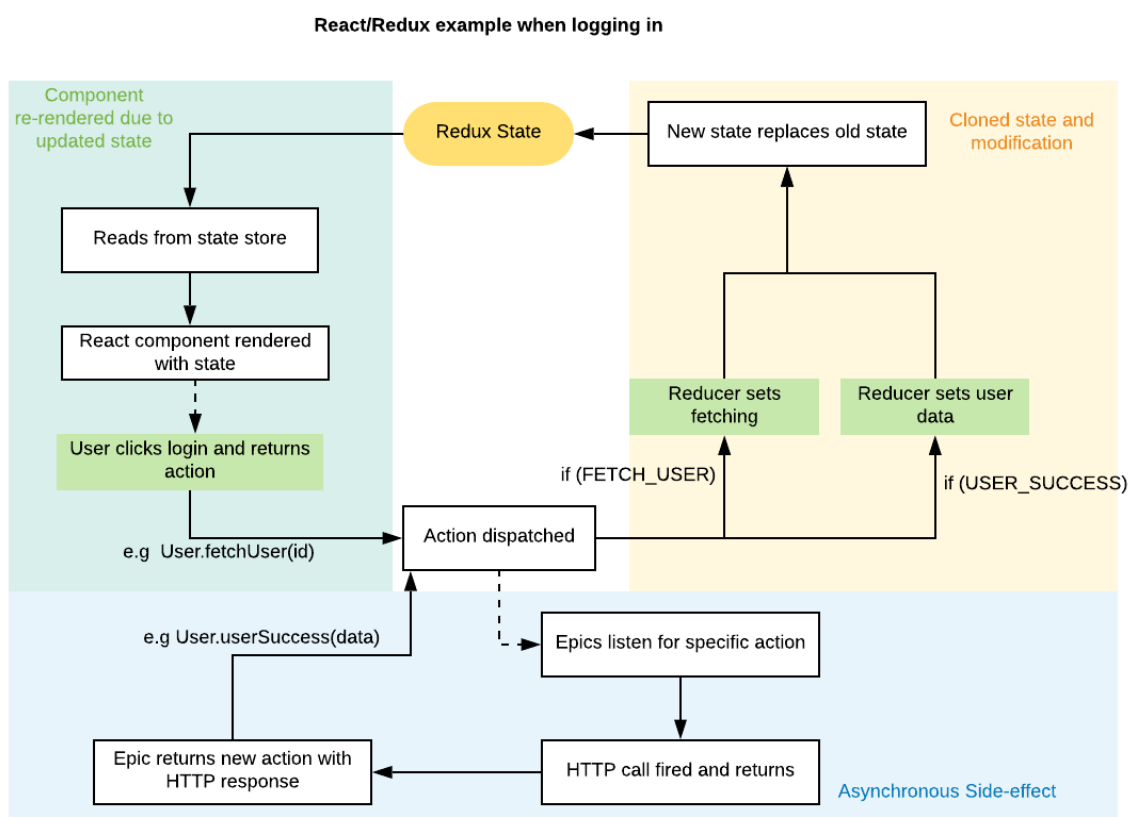


Figure 2 Example of React with Redux state

The frontend application communicates with a REST API, which serves all content through JSON. A websocket is also used, enabling real-time communication between a client and a server.

### Service Discovery

Service discovery is central to the system, all services dynamically communicate by looking each other service up in a registry. This means IP addresses for each service don't have to be hard coded.

### API Proxy

To simplify accessing the REST API, an API proxy exists which takes the HTTP request, transforms it into a protocol buffer and looks up the destination using service discovery. Usage of service discovery means it's entirely dynamic and is just a dumb-proxy forwarding information. This follows the API Gateway pattern [9]

### API Services

A tier of API services exist which define the public API. Authorisation, authentication and validation is done here. API Services will call down to domain services where the logic for validation authentication tokens, fetching users, etc lives. These are lightweight services, transforming data for public use. Each service exposes the same protocol buffer service and is uniquely identified by service discovery. For example:

<http://api/auth/login> -> ticket.api.auth

<http://api/user/get/1> -> ticket.api.user

### Realtime – Team weighting service

The team weighting service is a small JavaScript server which handles the real-time aspects of team weighting. It uses protocol buffers to communicate with the browser. For fetching and updating data, it calls directly into the domain services using the already-defined protobuf services.

### Domain Services

The domain services contain all the complex application logic. They are called by the API services, other domain services and the team weighting service.

## Database

The MySQL database is used to store any persisted data. The database is accessed through the domain services using an Object mapping library called gORM (Golang object-relational-mapping) [10]

## Data structures

The data for the project is spread across eight tables, each identifying different components of the system.

The **users** table contains account details, hashed passwords and their role. Each user is related to **ticket\_comments**, **sprint\_comments** and **tickets** as both creator and assignee.

The **projects** table is central to the remainder of the system, almost everything is directly or indirectly related to a project. The **projects** table directly related to **tickets** by one-to-many as well as **sprints** similarly. The project is simply the name of the project along side a short identifier. The short identifier is used when identifying tickets (e.g TST-1).

The **tickets** table has one-to-many to the **ticket\_histories** table, as well as **ticket\_comments**. The ticket contains data such as the title, description and priority of a ticket, alongside assignees, creators and weightings. A **ticket\_history** denotes a change within the ticket (such as the title changing) and a **ticket\_comment** is a comment on a single ticket, which has a **user** who posted it. The ticket may also be assigned a **sprint**.

A **sprint** exists as a period of work, which initially has a start time, and has an **end\_time** assigned once the sprint closes. A sprint may contain one or more **tickets**. During the retrospective stage of the sprint, **sprint\_comments** may be made, so a sprint may have one or more comments. The type field denotes whether it's a positive comment or an improvement comment when displayed in the interface.

Finally, **workflow\_rules** exist to define the sprint board, the columns and where valid ticket moves are. These have a many-to-one relation to tickets since a ticket will be located in a given **workflow\_rule**



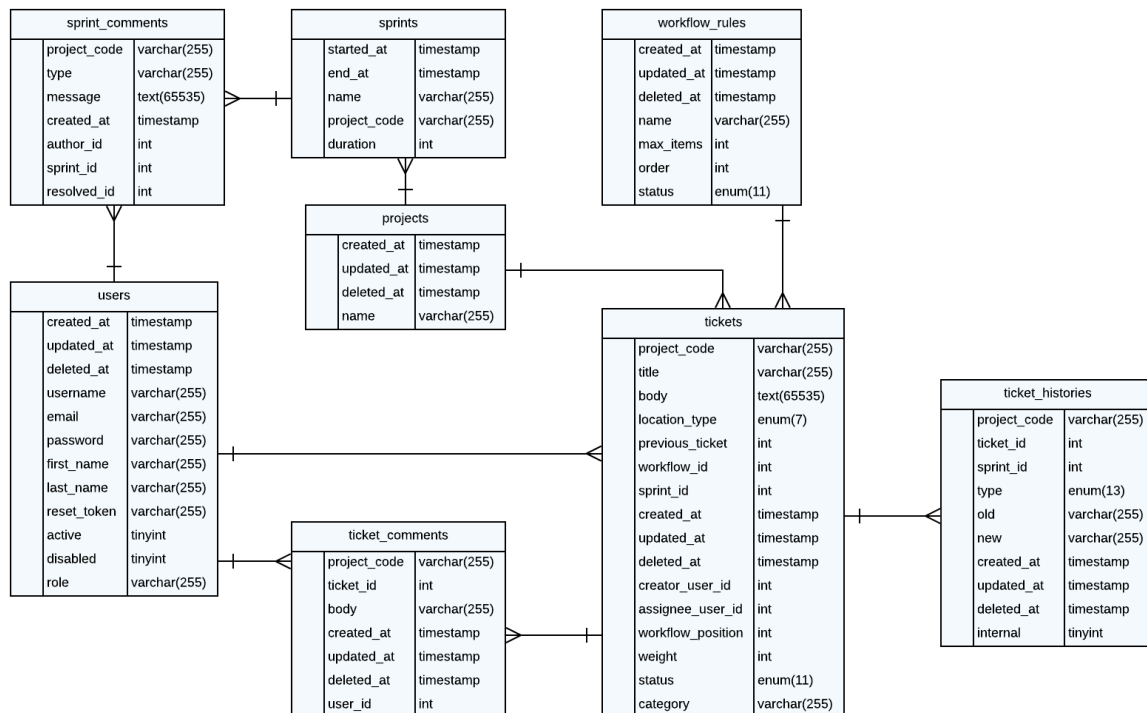


Figure 3 ERD of database

## Implementation

This section will cover the implementation of the solution, process and reasoning behind the decisions made. At a high level it will cover the overall architecture, surrounding infrastructure then explain the backend and frontend independently.

### Architectural Overview

The software solution developed can be split into three packages, firstly **app** which contains all of the microservices. UI contains all the React/Typescript code and **realtime** contains the realtime websocket server. The directory structure is as follows:

```

src/
  ui/ - The React web interface
  realtime/ - The javascript websocket service
  app/ - The golang API, containing multiple services:
    api/
      admin/
        workflow/ - The administrative workflow API (ticket.api.admin.workflow)
  
```

```
    user/ - The administrative user service (ticket.api.admin.user)
auth/ - Authentication API (ticket.api.auth)
project/ - Project API (ticket.api.project)
sprint/ - Sprint API (ticket.api.sprint)
ticket/ - Tickets API (ticket.api.ticket)
user/ - Workflow API (ticket.api.user)
workflow/ - Workflow API (ticket.api.workflow)
router/ - API Proxy (ticket.router)
services/
    auth/ - Authentication Store Service (ticket.svc.auth)
    history/ - Ticket History Service (ticket.svc.history)
    mail/ - Mailer Service (ticket.svc.mail)
    project/ - Project Service (ticket.svc.project)
    sprint/ - Sprint service (ticket.svc.sprint)
    ticket/ - Ticket management service (ticket.svc.ticket)
    user/ - User management service (ticket.svc.user)
    workflow/ - Workflow Serviced (ticket.svc.workflow)
```

## Docker

The software solution was developed using Docker [11]. Docker is a solution that runs light-weight virtual machines (containers) which run an incredibly light-weight linux operating system. This makes them ideal for executing 17 golang services, the ui, the realtime javascript server and dependencies. Docker Compose allows all the services, the interface and any dependencies in a single file (/docker-compose.yml) and the execution of the command starts entire environment, no configuration. This means the entire build environment is reproducible, regardless of the host operating system.

This simplified development by not having to keep track of all the services running locally, one command starts the entire development environment. Stopping a single container to restart it locally is also possible, making it perfect for development.

## Service Discovery

In order to be able to identify where all the services are, I need a dynamic mechanism to do so. Hard coding each service IP address and port wouldn't be sustainable and wouldn't scale to multiple replicas of the same service. Service discovery enables each service to register itself and other services to look it up. This is done using an open-source utility called Consul [12] This acts as a service registry, a lookup table, which other services query to find out where other services are. This is automatically started within the docker-compose configuration.

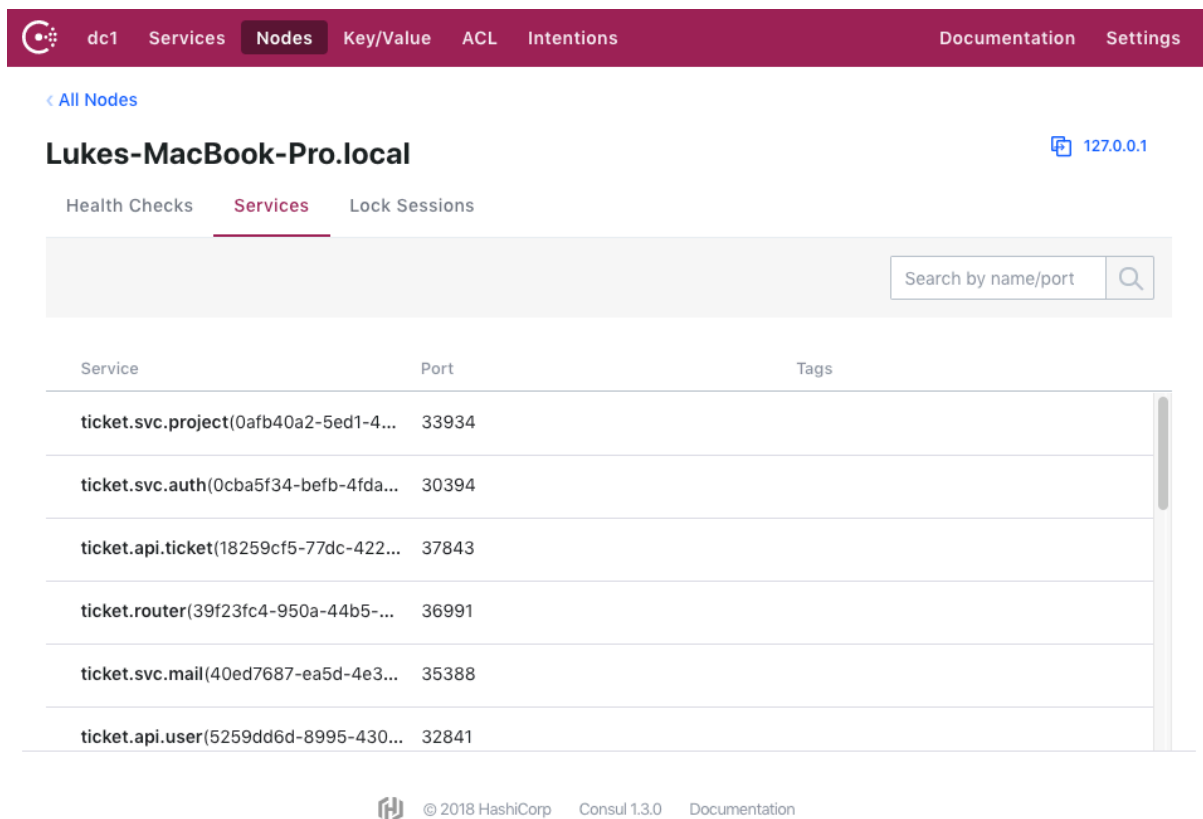


Figure 4 A screenshot of consul, displaying the registered services.

## Backend

### Language

The backend is written in a language called Go. Go's syntax is similar to that of C++ but provides an incredibly flexible standard library, making it incredibly suitable to web-driven applications. It's heavily used within Microservice architectures due to the speed and portability of the binaries it produces. Go has enabled me to make incredibly simple micro services which all have a single dedicated purpose. For example, the mailer service is simply responsible for formulating and sending emails to a given recipient. The entire service is around 140 lines of code and is fully featured. Due to go's simplicity components such as routing, service discovery, etc required implementation, this is all found within the **app/common/** folder in the project. All microservices use this shared codebase to run.

Go has extensive documentation and guides [13] which I used heavily in the early stages of development of this software.

### Communication between services

In order for services to communicate in a reliable and consistent way, I've opted to use Google RPC protocol (GRPC) which uses Protocol buffers to transmit messages between services. This is preferable to that of XML or JSON as it provides fixed types which compile down to bindings for any given language. Protocol buffers use a custom DSL to define all of the messages and any services that may be exposed.

An example of this is the authentication service. This defines the authentication service and the methods that may be called on it. It also defines all of the data structures that are required to be transmitted.

```
syntax = "proto3";

package ticket.svc.auth;

message SessionState {
    int32 userId = 1;
    string role = 2;
}

message Session {
    string token = 1;
}

message Status {
    bool status = 1;
}

service Auth {
    rpc Create(SessionState) returns (Session);
    rpc Validate(Session) returns (SessionState);
    rpc Destroy(Session) returns (Status);
}
```

Protocol buffers then provide utilities for automatic code generation, which produces the clients to communicate with the service. Throughout this project I use a combination of go bindings and Javascript bindings between the API and the websocket server. Communicating with the service looks like this:

```
authClient := ticket_svc_auth.GetAuthClient(req.Service)

// Generate and return the session token
token, tokenErr := authClient.Create(context.Background(), &ticket_svc_auth.SessionState{UserId: a.Id, Role: a.Role})

if tokenErr != nil {
    return errors.HttpInternalServerError(res, tokenErr)
}
```

This is also used within the websocket portion of the application. The use of fixed data structures means ensuring data coming in is correct and valid is easy, as it's the core concept of a protocol such as Protocol buffers. In order to transmit these over websockets, they must be encoded down to raw bytes. One of the issues with this is the resulting bytes has no way to identify the type of message that is being transmitted. To solve this, I appended an extra byte to transmitted message containing a numeric identifier of the message. This is then used when encoding and decoding:

```
export function sendMessage(data): Uint8Array {
  // Encode message to bytes
  const msg = data.constructor.encode(data).finish();

  // Build byte array
  const arr = new Uint8Array(msg.length + 1);
  arr[0] = messages.findIndex(i => i.name === data.constructor.name);
  arr.set(msg, 1);

  return arr;
}

export function getMessageType(bytes: Uint8Array): { type: string, message: any } {
  return { type: messages[bytes[0]].name, message: messages[bytes[0]].decode(bytes.subarray(1)) };
}
```

## REST API

The REST API is the API that is used by the web interface to control the software. The UI is simply the visual aspect, while the API parses, validates and manages the data itself. Due to limitations of the web, JSON is the easiest way to transfer data between a browser and a server. Internally all the services use RPC, so the API proxy acts as the interpreter between the two. All internal services *could* use JSON, but that opens the system up to bugs due to JSON not being typed by nature. JSON is dynamic, so there's no contracts on what any given service will return.

### API Proxy

At the forefront of the API is an API Proxy. The proxy runs on port 8080 and listens for HTTP requests from the web interface. This is effectively the gateway into the microservice architecture.

All of the API services must implement the protocol buffer of the router:

```
service Router {
  rpc Call(Request) returns (Response) {}
}
```

This proxy service interprets all of the HTTP data and encodes it to meet the structure of Request. When the correct API service has completed work, the API service will formulate a response which is then decoded and returned as JSON via the API.

In order to identify what service should be contacted, the API URL is split, taking the first two, falling back to the first segments into account. This means that while /admin/user will contact the ticket.api.admin.user service /ticket/get will only contact the ticket.api.ticket service. Calls to the service discovery agent are cheap and fast, so it's sustainable to call twice. Caching could be carried out here, but this is a micro-optimisation that wasn't needed in the scope of this project. Caching would need to have a TTL assigned to ensure stale results aren't stored for long periods, ensuring the API proxy is entirely dynamic.

```
func (cache *RouterCache) Find(url string) (*discovery.DiscoveredService, string, error) {  
    // Split the URL to calculate parts  
    urls := strings.Split(url, "/")[1:]  
  
    // If the URL contains 2 parts, the service is grouped  
    if len(urls) >= 2 && urls[1] != "" {  
        urlPrimary := fmt.Sprintf("ticket.api.%s.%s", urls[0], urls[1])  
  
        if discovered, err := cache.Agent.Find(urlPrimary); err == nil {  
            return discovered, strings.Join(urls[2:], "/"), nil  
        }  
    }  
  
    // The service isn't grouped.  
    urlSecondary := fmt.Sprintf("ticket.api.%s", urls[0])  
  
    if discovered, err := cache.Agent.Find(urlSecondary); err == nil {  
        return discovered, strings.Join(urls[1:], "/"), nil  
    }  
}
```

### Routing

With the request forwarded to the correct service, the service must now parse, execute and return the correct data. Each api service will control multiple routes related to it.

Go doesn't provide any framework for routing, so I opted to implement a radix tree to store the routing table. Each node of the tree corresponds to a "segment", which are made by splitting the resulting URL by all forward-slashes. It's fast to lookup in a radix-tree due to it simply being a case of the next node existing or not. Due to some URLs requiring wildcard parameters, some special casing was required to check if any given segment is allowed to be a dynamic value.

The resulting solution means any number of routes can be registered, which accepts a function that is provided metadata about the route. The response to this function is simply encoded and transmitted back to the browser.

Authentication and authorisation are ideal be done at the routing layer. It means the application code doesn't have to repeatedly implement checks for authentication and authorization. This is done through middleware, which is a function that accepts the route callback, and returns a new callback in it's place. It's then free to carry on and call the resulting route handler or halt and return it's own value (e.g an error). Registering a route looks like this:

```
svc.RegisterRoute(api.MakeHandler(api.GET, "/", middleware.IsAuthenticatedAs(routes.List, "admin")))
```

You can see it's using middleware to ensure the user is authenticated, but also an administrator. In the case of the `IsAuthenticatedAs` middleware, it parses the Authorization header passed by the browser and makes a call to the Authentication service, validating the session token for the role.

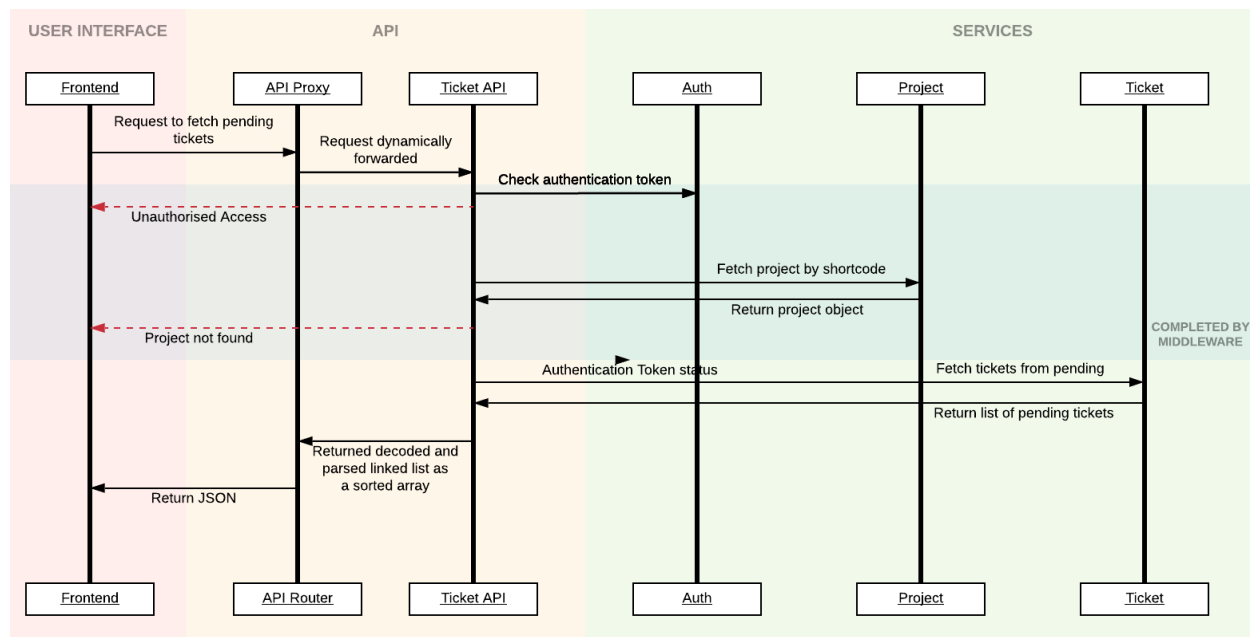


Figure 5 Interaction between services and middleware

## Session Tokens

In order to authenticate the user with the system, the user needs to be given a token that will correctly identify them for subsequent API calls. To do this I used JSON Web Tokens (JWT) [14]. JSON Web Tokens are tokens that store JSON encoded data and are

cryptographically signed so we can verify the contents of them was created by us. An example of a token for user a given user **2** with role **user** is:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1NTkzNDkyNjYsImluc3Npb24iOi0nsidXNlcklkIjoyLCJyb2x1IjoidXNlciJ9fQ.40UXN1vP7YecRMDu8AKAjdYnNhS1BcBi0vz1YcVXE8
```

The first two segments are base64 encoded, which makes them safe for transmission over the internet without formatting of text becoming an issue. The first segment contains the header which defines how the token has been signed. In the case of this it will signed using HS256 since that's what the implementation mandates:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

The second segment is what we're interested in, it contains arbitrary data that we include in the token. In the case of this application it contains a Session object, which includes the users internal id and their role in the system. This data is used when authenticating and authorising the user across various API routes. Having this data within the token saves extra database calls as it's entirely within the token. We're able to verify this data hasn't been tampered with since it's signed with a secret only we know. The 3<sup>rd</sup> part contains the signature, which is used when verifying the token.

```
{
  "exp": 1559349266,
  "Session": {
    "userId": 2,
    "role": "user"
  }
}
```

JWT's come pre-built with expiries, which means they have a finite lifespan. This means if an attacker manages to obtain this token, it will only work for a short period of time before it becomes ineffective.

The token is signed by combining the first two segments, and then uses the HMAC256 algorithm and the pre-defined key to sign it. It's then appended on to the end of the token for form the entire token.

As an extra layer of security, I've opted to track the tokens in-memory, so when a user signs out it's easy to de-authenticate users. The downside of this is re-creating a service will cause



the in-memory store to clear. Any key-value store such as redis would be ideal in this situation, but given the scope of the project, isn't necessary

### Password Security

To ensure passwords are securely stored, not in plain text, the passwords are hashed using a library called bcrypt. Internally bcrypt automatically generates a unique salt for each password generated and uses that to hash the password. When comparing the password it re-uses the salt to hash the inputted password to see if it matches the stored value. Using a simple hashing mechanism such as sha1 isn't sufficient, as it's easy to pre-calculate all the values as the input to the sha1 function is consistent with the output. Bcrypt using a random salt negates this problem.

### Frontend

The frontend is the interface that users interact with. It's developed as a single page application powered by React.

### Language and Framework

The frontend is written in typescript [15]. Javascript by nature is an untyped, dynamic language. When you define a variable, it's type is not set in stone. There is no way to tell whether it's an integer, string or an object. Typescript introduces types at compile time meaning any code written is able to type checked, reducing the bugs that can be introduced.

At compile time, typescript is compiled down to Javascript, enabling the browser to execute it.

React is currently the leading frontend development tool. Facebook developed and open sourced the project, it's designed to build complex web applications. Redux is commonly used to provide a central way to store application state.

### Structure

The interface codebase can be grouped into different functions. It leans heavily on Redux for state management, so a lot of the code make up is around that.

```
src/actions
```

Contains all of the redux actions which can be fired from the container components

`src/assets`

Contains any non-code assets that are needed across the application. This is limited to the logo and background images.

`src/components`

Contains any React components that aren't tied into state. This is strictly presentational components. For example, a component that just displays the tickets status.

`src/containers`

Contains are the React components that are connected to the Redux state. This makes up the bulk of the components since this application is mostly data driven.

`src/epics` - Contains all the epics for action side effects

Epics contains all of the Redux epics. Redux epics listen for actions and fire side-effects which translate into actions.

`src/realtime`

The realtime websocket server generates the protocol buffers and relevant javascript code. This is the same code that can be found in `/src/realtime/realtime`

`src/reducers`

Reducers contains all of the reducers that change the application state. They listen for actions and modify the state as defined. All reducers return a new copy of state, making sure not to modify the existing state.

`src/state`

State contains type definitions for the internal application state. It's a series of interfaces that are connected together to build the full state. The file **index.ts** contains the root of the state tree, which include the other groups (app, project, sprint).

`src/util/models` - Contains classes and types for accessing the REST api

This contains an array of classes around interacting with the REST API. Each classes has many associated types which help build type safety when interacting with the API.

src/index.tsx

This is the core file which enters into the application. In order to build the UI a tool called webpack is used which concatenates all the files as needed, and index.tsx is used as the entry point into this. Webpack calculates what other dependencies need to be pulled in and bundled.

## State Management

As previously mentioned, state is managed using Redux, a framework agnostic library. The reducers form the state and are executed every time an action is fired. Each defined reducer only responds to specific actions, so it's possible the state isn't changed at all.

The reducers provide the default state, and from then onwards, the state is modified by reducers. A possible reducer for authenticating the user looks like this:

```
case AuthActionTypes.LOGIN_REQUEST:
  return { ...state, loading: { ...state.loading, login: true } };

case AuthActionTypes.LOGIN_SUCCESS:
  return { ...state, loading: { ...state.loading, login: false }, state: { auth: IAuthStatus.AUTHENTICATED }, user: action.payload.user };
```

(the triple dot notation is causing a copy of that property, meaning we're not modifying existing state).

The action LOGIN\_REQUEST is dispatched by a component, for example when the user clicks the login button:

```
login(username: string, password: string): void {
  dispatch(AuthAction.loginRequest({ username, password }));
},
```

Epics listen for an action to be fired so they can fire off web requests. In order to log the user in we need to fire a web request when the LOGIN\_REQUEST action is fired. This looks like this:

```
action$ => action$.pipe(
  ofType(AuthActionTypes.LOGIN_REQUEST),
  switchMap((resp: any) => {
    return getApi(Auth).login(resp.payload.username, resp.payload.password)
      .pipe(
        tap(token => cookies.set('token', token.token)),
        switchMap(() => getApi(User).getCurrent()),
        map(user => user && AuthAction.loginSuccess({ user })),
      )
  })
)
```

```
tap(() => SCRUM_WEBSOCKET.connect()),
catchError((err: Error) => {
  toasts.error(err.message);

  return of(AuthAction.loginFailed());
}),
);
}),
),
```

You can see this is an ideal place to do any actions outside of state. In this case we set a cookie so we can login next time the user loads the page, authenticate with a websocket and fetch down the full user model. In the end we return the LOGIN\_SUCCESS or LOGIN\_FAILED action to the user which modifies the state in a way that will either display the error or log the user into the application.

### API Communication

The frontend communicates with two different APIs. The core API is accessed over REST and the ticket weighting API is called via websockets.

Communication with the core API occurs using the common HTTP Verbs:

- GET – Request a resource
- POST – Create a new resource
- PATCH – Modify a resource
- DELETE – Delete this resource

Requests may contain a payload body which is formatted as JSON, this is parsed by each API service and validated, as well as validated within the UI to ensure it is correct. The responses from the API include the origin service and the statusCode as well as a data property containing what was returned. Similar for errors, but with an error property.

Successful Response:

```
{"statusCode": 200, "source": "ticket.api.ticket", "data": [] }
```

Failed Response:

```
{"statusCode": 404, "source": "ticket.router", "error": "No such route"}
```

For the realtime aspects of the software (ticket weighting), a websocket connection is opened between the client and a small javascript power service. The listens for the users to send the Authenticate method and authenticates the user with the core Auth service.

An administrator is the only user who can create planning sessions, but all online users will receive an event if an administrator starts a planning session. This will allow the user to join. The nature of websockets being bi-direction means the server is free to send information down to the client when it sees fit and the client is able to trigger actions at any time.

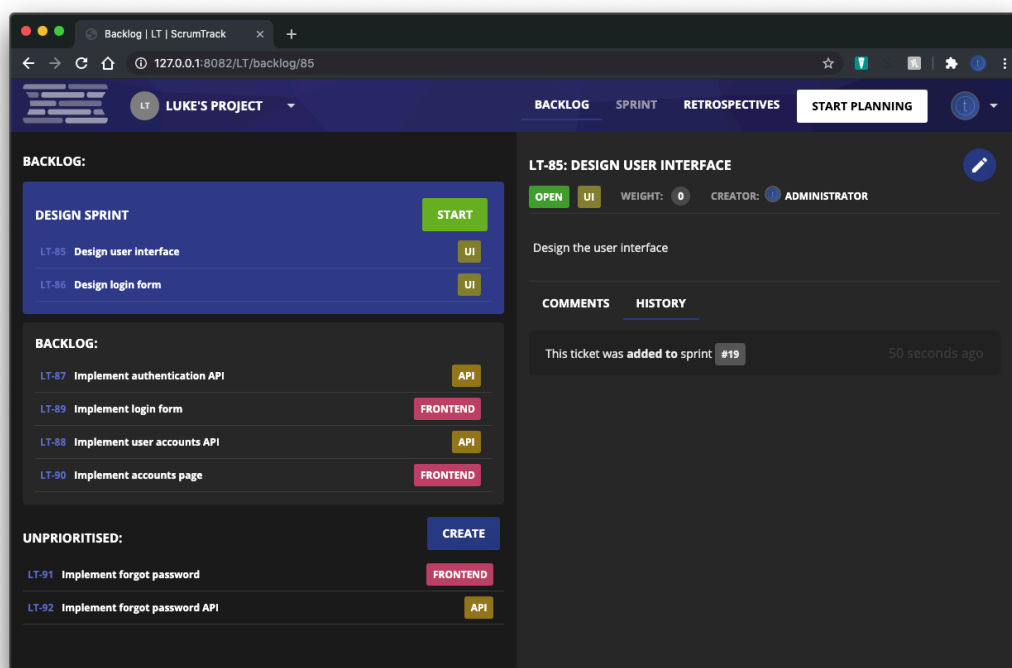
## Features

With the implementation of the core system covered, we will now take a look into the implementation of specific features and their functionality.

### Backlog Management

Backlog management is displayed as a list of tickets, grouped into pending, backlog and any possible sprints. Unprioritized tickets are un-ordered, while backlog tickets are ordered based on priority, top-down. Any sprints will be displayed along the top with the tickets within that sprint.

Tickets are free to move between backlog and sprints but may never go back into unprioritized.



Internally, pending is stored un-ordered, the tickets by default are stored in creation order (loosely based on created times). This means as you go down the list, you find newer tickets. Tickets higher in the list should be prioritised before the tickets lower down.

The backlog tickets are stored as a linked list. Each ticket has a reference to the ticket above it, with the top ticket being stored as NULL. This is preferable to the alternative, which is storing the index of each ticket, and when tickets are reprioritised, update all the tickets.

id	title	location_type	previous_ticket
88	Implement user accounts API	backlog ↕	NULL
89	Implement login form	backlog ↕	87
87	Implement authentication API	backlog ↕	88
90	Implement accounts page	backlog ↕	89

Figure 6 Fragment of the tickets table

Say we have 20 tickets, and we move the top ticket to the bottom, this will require all 20 tickets to have their order shifted by 1. This is incredibly inefficient. When moving between the backlog and sprint, this process would need to be executed for each list.

By using a linked list, we limit the number of changes to a maximum of 3 updates.

1. Updating the ticket itself, to be either NULL or reference the ticket above it.
2. Update the ticket directly below the moved ticket to have the same nextId as the ticket we're moving.
3. Update the ticket below the ticket in the original position.

When these results are fetched from the database, they are transmitted as an unsorted array. In order for these results to be usable, we need to assemble the linked list, and then flatten it in the correct order. The logic for this is within the ticket API service and not the UI since the UI shouldn't be aware of the internal implementation.

Initially we loop through the tickets, storing a mapping of id -> node, this is used later to build the structure:

```

nullCount := 0
rootNode := &Node{}
maps := make(map[int32]*Node)

// For each ticket, store it in a map
for _, b := range list {
    maps[b.Id] = &Node{Node: b}

    // If a ticket has no previous, store it.
    if b.PreviousTicket == 0 {
        nullCount++
        rootNode = maps[b.Id]
    }
}

```

Once the map has been built up, we can loop over the tickets, assigning the tickets to their relevant parent:

```

for _, b := range list {
    if b.PreviousTicket != 0 {
        maps[b.PreviousTicket].Next = maps[b.Id]
    }
}

```

Finally, the flat array is built, starting from the rootNode and recusing down, adding based on the index in the chain. This produces a correctly sorted array, based on the linked list. This is then able to be transmitted to the interface and is able to be displayed.

```

func addNode(arr []*ticket_svc_ticket.Ticket, node *Node, idx int32) {
    arr[idx] = node.Node

    if node.Next != nil {
        addNode(arr, node.Next, idx+1)
    }
}

```

```

// Add all tickets into single, flat array.
sorted := make([]*ticket_svc_ticket.Ticket, len(list))
addNode(sorted, rootNode, 0)

```

In order to be able to send the request from the frontend, the frontend provides the ticketId of the ticket we're moving as well as the previousId of the destination. Using this combination we're able to update it accurately.

```
// Fetch the ticket we're moving
targetTicket, _ := common2.FindTicket(t.Service, req.ProjectCode, req.TicketId)
// Fetch the ticket below the current
sourceTicketAfter, err2 := common2.FindTicketByPrevious(t.Service, req.ProjectCode, targetTicket.LocationType, targetTicket.Id,
targetTicket.SprintId.Int32)
// Fetch the ticket below us in the new location
targetTicketAfter, err3 := common2.FindTicketByPrevious(t.Service, req.ProjectCode, targetTicket.LocationType, req.PreviousTicket, req.SprintId)
```

With our 3 tickets, we're then able to swap the locations:

1. We start by setting the sourceTicketAfter's previousTicket to the previousTicket of the target
2. We then set the targetTicket's previousTicket to that of targetTicketAfter's previousTicket.
3. Finally, targetTicketAfter's **previousTicket** is set to be targetTicket.

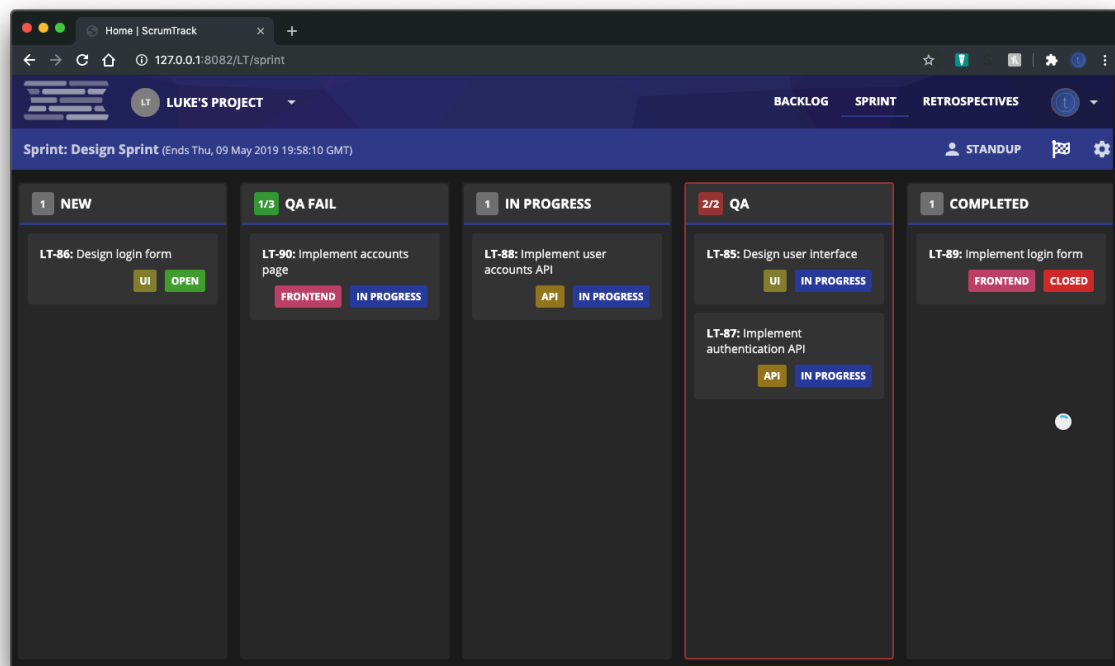
```
sourceTicketAfter.PreviousTicket = targetTicket.PreviousTicket
targetTicket.PreviousTicket = interfaces.MakeNullInt32(req.PreviousTicket)
targetTicketAfter.PreviousTicket = interfaces.MakeNullInt32(targetTicket.Id)
```

We're then able to persist the 3 tickets to the database. If they don't exist, we're able to safely drop the update since the PreviousTicket's will correctly set to NULL due to Go's default values being that.



## Active Sprint

The sprint workflow displays a set of columns enabling the user to move a ticket through them as the ticket progresses in its lifecycle. It's common for tickets to have Peer reviews, and with workflow rules you can define that a ticket can only go into the completed column via peer review. Workflow rules are only able to be configured by administrators, meaning anybody is unable to change it as they see fit.



The implementation of this is centred around tickets and workflow columns. Each ticket is assigned to a workflow column. A workflow column can have many dependencies defined, which is a many-to-many relationship with itself. A ticket with no dependencies defined assumes it's free to move into any column.

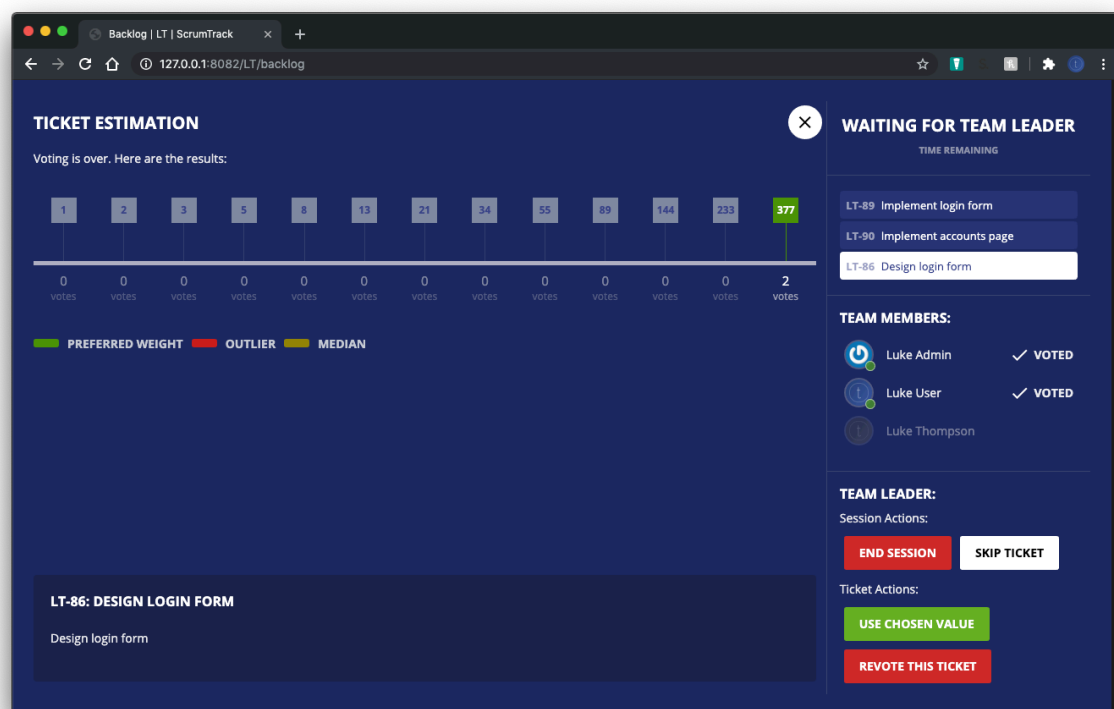
Ordering is stored as the numerical index. It was mentioned previously that storing based on order is bad performance wise, but a sprint is likely to have a limited set of tickets. It's reasonable to assume that a sprint will have fewer tickets than that of a backlog, where a backlog may contain hundreds, a sprint will only contain a small fragment of that.

The logistics of dragging between columns is done by an open-source library, which handles browser compatibility and detecting which column the ticket is moved to. The library doesn't handle storing, sorting and logically moving. It just tells us when something has

moved. If moving wasn't handle ddata-wise, the ticket would just revert back to it's initial position. I deemed building my own solution to be an ineffective use of time, something of this nature is prone to bugs, compatibility issues and unexpected edge cases.

### Team weighting

Team weighting is a collaborative mode which allows multiple people in a team to join a session and weight tickets. This is typically done at the sprint planning stage. By default it will loop through all tickets in a given backlog and one-by-one weight them. The team is displayed a set of cards (the Fibonacci sequence) and should pick one within the time limit. For the purposes of demonstration there is a time limit of 15 seconds, but this is customisable within the server configuration.



Administrators are the only role able to start the planning session. Since either a Scrum master or product owner leads the meetings, it's safe to assume they would want to lead a planning session. The team leader is given extra options around the control of the session. At any time they're able to end the session or skip the current ticket.

When a ticket is voted on, they're given different options based on the result of the ticket.

The ticket can fall into one of 3 categories:

1. If the team unanimously votes for one ticket, the following will be presented:
  - a. Continue to the next ticket with the chosen weight
  - b. Revote on the current ticket
2. If the team agrees on a value, but not everyone, based on a threshold percentage (configurable, but default at 66% of votes)
  - a. Any outliers of the 66% will be marked, and the team leader will be able to choose the remaining value
  - b. The option to take the median of the weights
  - c. The option to revote
3. If we're unable to resolve the weighting this way the team leader will be able to:
  - a. Revote
  - b. Accept the median

In each of these steps it's safe to assume discussion between the team would occur, but this is outside of the tool. This is why the time while waiting for team leader isn't time boxed.

Every 15 seconds, a tick() event is called within the websocket server. At this point we evaluate that at least 2 votes have been taken. If not, we restart the timer.

We calculate the results, and they get sent down to the interface with the following structure:

```
message ResultWeight {
    int32 weight = 1; // The weight we're representing
    bool isOutlier = 2; // Whether this is a possible outlier
    bool preferred = 3; // Whether this is the preferred weight
    bool isMedian = 4;
    repeated int32 voters = 5; // All voters
}

// All weights for a voting round
message ResultSet {
    repeated ResultWeight weights = 1;
}

// SERVER -> The available results for the last round.
message Results {
    bool isFinal = 1;
    repeated ResultAction options = 2;

    ResultSet current = 3; // Votes for the current session
    ResultSet previous = 4; // Votes for the last round if possible
}
```

The logic for calculating who should be voted on is simplified into three steps.

We initially calculate whether only 1 value was voted for:

```
let votedNumbers = [];

// Calculate the weights with votes.
Array.from(map.keys()).forEach(weight => {
  if (map.get(weight).length >= 1) {
    votedNumbers.push(weight);
  }
});

// Everyone voted for the same thing!
if (votedNumbers.length === 1) {
  return new Results({
    isFinal: true,
    options: [
      ResultAction.CONTINUE,
      ResultAction.REVOTE,
    ],
    current: buildSet(map, [], votedNumbers[0]),
  });
}
```

If this is not the case, we then have to calculate whether at least the defined percentage voted for the same value.

```
// See if we can get 75% of votes on 1 number.
const voterCount = Array.from(map.values()).map(i => i.length).reduceRight((p, c) => p + c);
const outliers = [];
let primary = null;

Array.from(map.keys()).forEach((weight: number) => {
  const votes = map.get(weight).length;

  if (votes > 0 && (votes / voterCount >= config.suitablePercentage)) {
    primary = weight;
  } else if (votes > 0) {
    outliers.push(weight)
  }
});

// We've found some form of weighting
if (primary !== null) {
  return new Results({
    isFinal: false,
```

```
        options: [
            ResultAction.STRIP_OUTLIER,
            ResultAction.AVERAGE,
            ResultAction.REVOTE,
        ],
        current: buildSet(map, outliers, primary),
    });
}
```

If neither of these cases are met, we will return just the revote and median options:

```
// No consistent data can be gathered, give some basic options.
return new Results({
    isFinal: false,
    options: [
        ResultAction.AVERAGE,
        ResultAction.REVOTE,
    ],
    current: buildSet(map, [], null),
});
```

## Critical Appraisal

### Critical Analysis

Throughout this section I will discuss each of the previously mentioned objectives and whether the provided solution was a success.

#### Users

The provided solution enables users to be assigned two roles and the administrative role is granted permission to manage users, sprints and workflows, this meets the defined criteria.

Users are able to authenticate with the system using their username and password successfully, although email doesn't. I opted to not implement email authentication since it is used for password resets and is another security measure against fraudulent password resets. Users are able to receive password reset emails, and correctly change it. An improvement here would be to notify the user their password has been changed.

Administrators are the only account type to be able to create users. It might have been useful for users to be able to request accounts, with administrator's approval, but this wasn't necessary in the requirements.

### Ticket Management

Tickets are correctly able to create new tickets, with titles, descriptions and categories. Priority is defined implicitly in the backlog but an extension of this would have been to implement a visual indicator for severity/priority levels (e.g LOW/MEDIUM/HIGH). This would aid the product owner when weighting the backlog.

The description and comment editors are implemented the same. They support the use of headings, links and images (among other features). I opted to not implement tables due to the added complexity, but given further time and scope, I would likely have implemented this functionality.

It's possible to view the revision history of any given ticket, including sprint, title, weighting and assignee changes. An improvement in this functionality would have been to include the main ticket body in this. Although displaying the body comparison would be harder due to it just being HTML internally.

Tickets are correctly divided into pending, backlog and sprints. If a new ticket is created, it is added to the bottom of the pending section and is unable to be prioritised within pending. The user is then able to drag the same ticket into backlog where it is prioritised.

An improvement here could be to display some sort of visual indicator of priority. An idea would be to have a border along the left, which fades as you go down in priority. It's possible it's not clear that order alone is priority.

### Sprint planning

Sprints are able to be created from the backlog view. To better meet Scrum, an improvement would be to allow multiple parallel sprints, allowing different teams to work under the same project. This would have required many changes that underpin the core system, but a likely change I would make given the time and resources.

Permissions are correctly handled and only administrators are able to create, start and stop sprints.

A team weighting solution was implemented but there is still scope for improvement. Any administrator is currently able to start this mode and all users will see the button to join display in real time. An adaptation to this would be to use Service Workers to send real-time notifications to offline users allowing them to quickly jump into the session without having to go straight to the interface.

The weightings are not displayed for each user, but an aggregate is. This point doesn't meet the set-out objectives, and a necessary feature at that. Upon reflection, displaying the weights dispersed among team members is more valuable than the aggregates due to different team members having more/less influence (more, less experience). Although displaying user experience would be hard to factor into the weightings.

If the entire team votes for the same value, it is correctly identified and displayed to the user. While I set out to automatically select the option, I feel giving control to the team leader was an important move. If the system just assumes, it could become a pain point where the system is over-assuming. Allowing the team to revote or accept the vote was an important solution to this.

If the weightings of the team are slightly dispersed, it correctly identifies the outliers. It would be useful. A configurable percentage is defined for identifying outliers. An improvement to this would be to base the percentage on how many are planning. In larger teams it may be harder to meet a quorum.

If the values are incredibly dispersed, it prompts the team leader to revote or take the median. The median is always displayed throughout the process alongside outliers and the preferable value.

An improvement to the team weighting solution would be to implement a way to discuss or prompt specific people to discuss their reasoning. Currently we assume that the team is in the same room to be able to discuss votes. I made it clear in the requirements that this wasn't something I would factor in, but I think it's important to mention.

The software is unable to forecast sprints. This is failure on my part due to over-estimation of planned work and not factoring in other tasks during semester-2. A solution to this would have been to ask for a simple good-bad-maybe feedback when each ticket is closed based on how long they feel it took them. We would then use this data to identify whether to increase or decrease the impact that ticket has on the overall sprint forecast.

## Sprint

The sprint board enables you to move tickets between defined columns. Administrators are able to toggle into a mode which lets them configure what columns exist and what the valid transitions are between the columns.

The sprint simply displays the projected end time of the sprint, but not the days remaining as initially set out.

A stand-up mode was implemented, which shows what tickets have changed over the last 24 hours. A Possible improvement to this would be to allow the duration that is displayed be extended.

## Retrospectives

Retrospectives have been implemented for previous sprints. When you view a retrospective you are presented with a graph of the burn-down based on story points. An extension to this would have been to collect, and then display more metrics to the user. Things like average completion time over the sprint, tickets remaining, tickets injected, etc.

The team is able to display what went well, and what could be improved. Each comment is signed by the user. Any unfinished points are displayed in the “could improve”, prompting them to reflect and discuss it. Each unfinished point can be checked to make it as completed, in which it is removed from this view. An improvement to this would have been to also display the completed tickets in the resolved retrospective. This would give them a full view of what a team has accomplished.

Another extension to retrospectives would be the ability to see the retrospective for the live sprint. This way the burn-down chart would have been exposed to the team, and they could see the points they need to work on to make this sprint successful.

## Miscellaneous

In order for this software to be useful, I feel it should have included some extra functionality. These were excluded on grounds of keeping scope limited and timescales.

- Integration with other software



Development teams use a host of other software in their process. Having their issue and sprint tracking tool integrate with this would help improve workflows. Tools such as Jenkins for CI

- Integration with version control systems

On the history tab, the ability to see commits made, code changes, etc against this ticket would opt to provide full visibility of what's happening. While not specifically scrum, a lot of existing tools support this

- Deeper integration with the product owner

The software mostly focuses on the development teams experience. Extra functionality for the product owner would be useful. Indicating business importance, directed comments to the scrum master, etc.

### Discussion of context

In recent months, Atlassian has released a modernized version of Jira which sets out to solve a lot of the same issues as I did. This shows there is a real need for a well-structured, simple project tracking tool.

With further development, this project would be usable within a real software development company. Due to time limitations, some functionality is missing that would make it viable for use. This project would have little impact within an academic context due to the nature of Scrum being software oriented.

### Personal Development

This project has helped me develop and extend my knowledge in several ways. The biggest learning curve was building a full web application with Go. With minimal experience of Go going into this project, I was required to learn the language to sufficient level within a relatively short period of time. Go, being developed by Google is being picked up across the industry, so knowledge of the language will help accelerate my career.

Managing a large project such as this required me to develop my time management skills, estimating how long features should take, while not perfect throughout the project, these skills will continue to develop. Underestimating tasks at times caused a knock on effect of

causing other tasks to get further behind and being overly ambitious and over-estimating the feature set I would be able to complete in the early stages was clear, this improved as time went on.

React is used industry wide, developing a real-world web application using it and related technologies was useful. Experience with the pitfalls of such the technologies as well as the benefits will help me in future projects.

## **Conclusion**

To conclude this project, the software built solves most of the objectives set out initially. Time limitations resulted in reduced scope, as with a project of this nature, the scope is infinite and with further development of features there is the possibility for this software solution to be usable in real workflows. Features such as:

- Integration with 3<sup>rd</sup> party platforms for Continuous Integration
- Integration with version control systems, providing code reviews and the like within the sprint workflow
- Development of sprint forecasting based on past sprints

## Bibliography

- [1] StackOverflow, “Stack Overflow Developer Survey,” 2018. [Online]. Available: <https://insights.stackoverflow.com/survey/2018#development-practices>.
- [2] Atlassian, “Jira,” [Online]. Available: <https://www.atlassian.com/software/jira/core>.
- [3] JetBrains, “Team Tools in 2018,” 2018. [Online]. Available: <https://www.jetbrains.com/research/devecosystem-2018/team-tools/>.
- [4] Trello, “Trello,” [Online]. Available: <https://trello.com>.
- [5] K. Schwaber and J. Sutherland, “Scrum Guide,” 2017. [Online]. Available: <https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf#zoom=100>.
- [6] B. Hartman, “An Introduction to Planning Poker,” DZone, [Online]. Available: <https://dzone.com/articles/introduction-planning-poker>.
- [7] JetBrains, “The State of Developer Ecosystem in 2018,” 2018. [Online]. Available: <https://www.jetbrains.com/research/devecosystem-2018/>. [Accessed 27 04 2019].
- [8] Facebook, “React,” [Online]. Available: <https://reactjs.org/>.
- [9] C. Richardson, “Pattern: API Gateway,” [Online]. Available: <https://microservices.io/patterns/apigateway.html>.
- [10] “gORM,” [Online]. Available: <http://gorm.io/>.
- [11] “Docker,” [Online]. Available: <https://www.docker.com/>.
- [12] “Consul,” Hashicorp, [Online]. Available: <https://www.consul.io/>.
- [13] “Golang Documentation,” [Online]. Available: <https://golang.org/doc/>.
- [14] “Json Web Tokens,” [Online]. Available: <https://jwt.io/>.
- [15] “Typescript,” [Online]. Available: <http://typescriptlang.org>.
- [16] “grpc Faq,” [Online]. Available: <https://grpc.io/faq/>.

## **Glossary**

API – Application Programming Interface

Base64 - Encoding function

CI – Continuous Integration

DSL – Domain Specific Language

GRPC – gRPC Remote procedure call [16]

HTTP - Hyper text transfer protocol

HMAC256 - hash-based message authentication code

REST – Representational state transfer

RPC – Remote procedure call

JSON - Javascript Object Notation

JWT – JSON Web Token

XML – Extensible Markup Language