Aber Fitness Project

Final Report for SEM5640 Developing Advanced Internet Based Applications

Authors: Adam Lancaster [arl4], Andrew Edwards [ane18], Charlie Lathbury [ckl2], Daniel Monaghan [dkm2], David Fairbrother [daf5], Jack Thomson [jat36], James Britton [jhb15], Robert Mouncer [rdm10]

December 10, 2018

Version: 0.1 (Draft)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

We confirm that:

- This submission is our own work, except where clearly indicated.
- We understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- We have read the regulations on Unacceptable Academic Practice from the University's Academic Registry and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work we understand and agree to abide by the University's regulations governing these issues.

Name	**User ID**
Britton, James	jhb15
Edwards, Andrew	ane18
Fairbrother, David	daf5
Lancaster, Adam	arl4
Lathbury, Charlie	ckl2
Monaghan, Daniel	dkm2
Mouncer, Robert	rdm10
Thomson, Jack	jat36

Date: December 2018

CONTENTS

LIST OF FIGURES

Chapter 1

Overview

Aber Fitness is a web application developed using Microsoft's *.NET Core* and Oracle's *Java Enterprise Edition* (henceforth referred to as Java EE). The project aims to provide a service to encourage fitness and promote engagement with sporting activities amongst the users of the application, offering functionality such as graphing fitness data gathered by owners of *Fitbit* devices, the ability to challenge other users to competitions and a sport ladder system with tight integration into a bespoke facility booking system. *Aber Fitness* aims to offer everything that would be needed by a sporty and active person in order to bring their sporting activities into a digital platform and also to enhance their use of devices they already own, such as *Fitbit* devices or smart watches such as the *Apple Watch*.

At launch the system will ingest activity data automatically from *Fitbit*, with the capability of easily implementing other health data provider services at a later date due to the modular nature of the data ingest system. Once normalised this activity data will be used throughout the various subsystems of *Aber Fitness*, providing users with functionality such as a dashboard overview of their activity over the last hour, day, week, etc. as well as integrating tightly into the challenges system to add a competitive aspect to the system in to keep users engaged with both the platform itself and keeping fit in general.

TODO: Possibly add more here? GDPR, Docker, Microservices, Auditing

Chapter 2

Requirements

Chapter 3

Development Methodolgy

TODO: Possibly restructure this, itś a start for now. Not 100% sure what Neilś looking for here.

3.1 Initial Project Plan

At the start of the project the group met and decided on a standard style of development which would be most suitable for the project. After some discussion, we agreed to adopt the *Scrumban* [1] methodology.

As this project has a relatively short deadline with a team consisting of only eight developers we adopted Scrumban, which focuses on flexibility and adaptability for both the project's plan and sprints. The team had no prior development experience with the application stacks we were required to use, Java EE and .NET Core. *Scrumban* is tailored for the difficulties around estimating each sprint or the current velocity.

The application's requirements were initially broken down into nine distinct microservices. The group then discussed which language to use for each service. **TODO: Link to figure below**

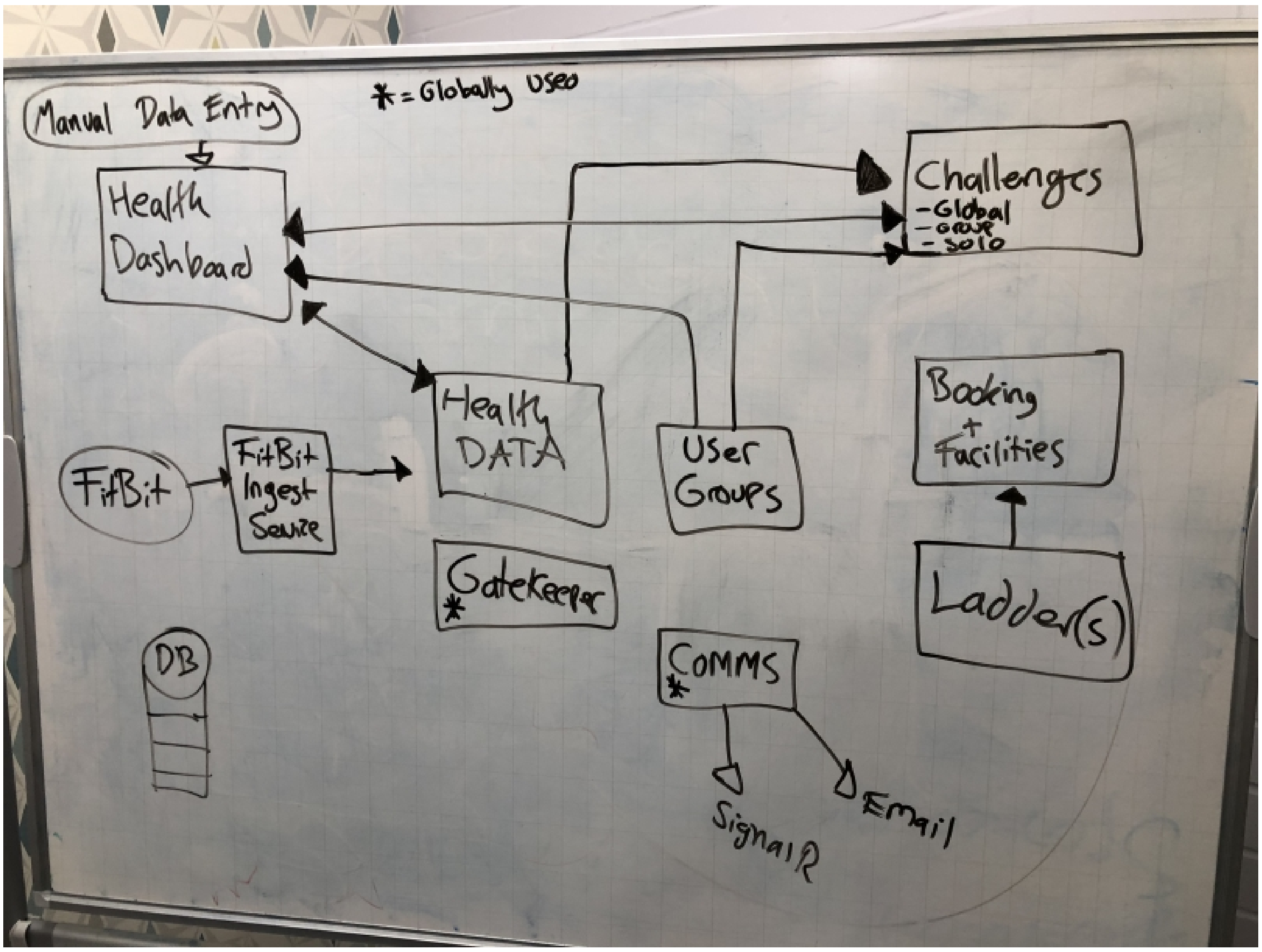
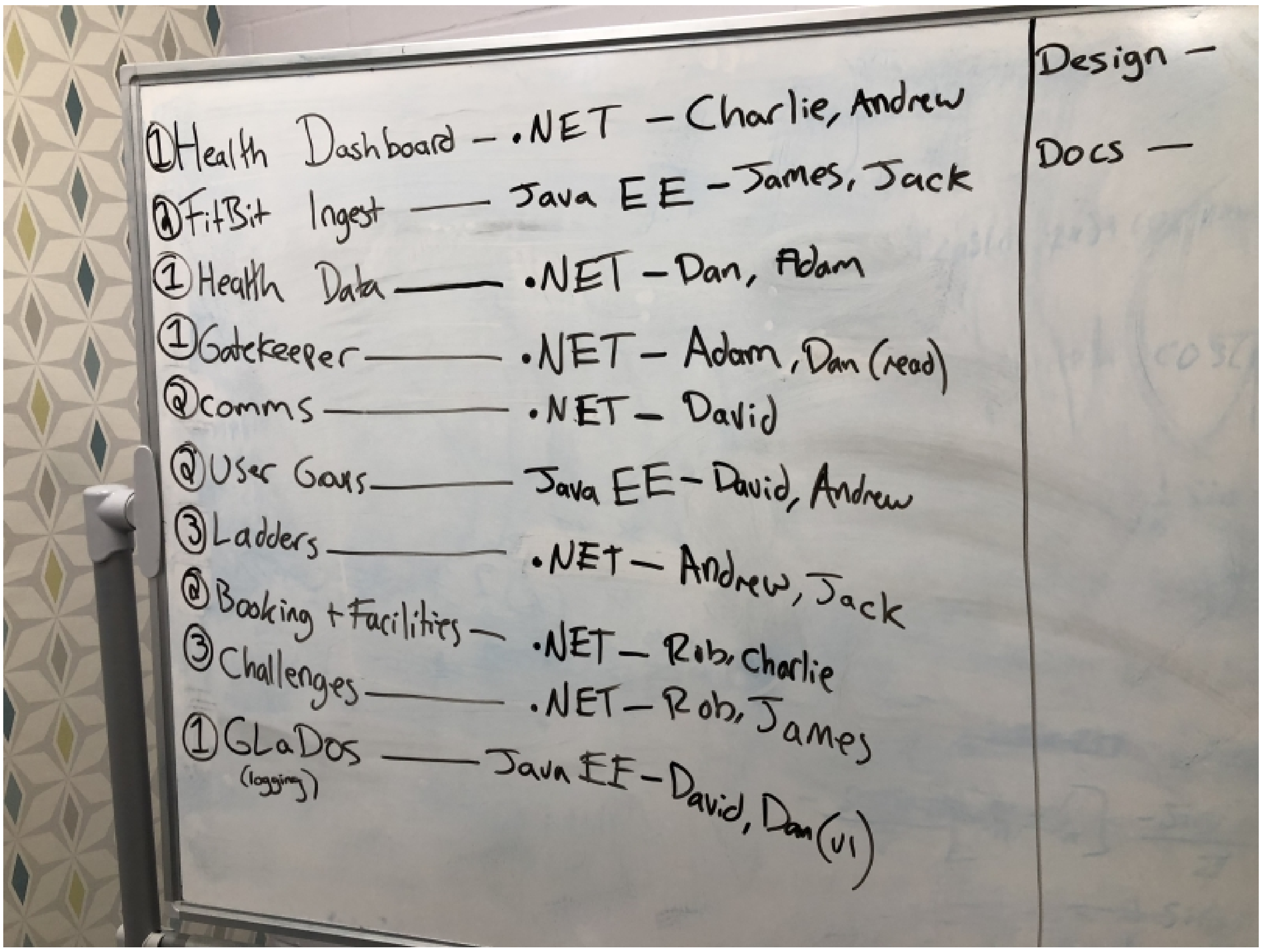


Figure 3.1: An initial design diagram which was used to break the project down into smaller microservices and their interactions with each other

We proceeded to allocate each microservice to two developer teams, then assigned each service a priority ranking between 1 and 3. Core services were marked with a priority of 1, as many other parts of the *AberFitness* infrastructure heavily relied on their APIs in order to function correctly. One examples is the *Health Data Repository* which centrally stores users' activity data. **TODO: Link to figure below**



① Health Dashboard – .NET – Charlie, Andrew
② FitBit Ingest —— Java EE – James, Jack
① Health Data —— .NET – Dan, Adam
① Gatekeeper —— .NET – Adam, Dan (read)
② comms —— .NET – David
② User Goals —— Java EE – David, Andrew
③ Ladders —— .NET – Andrew, Jack
② Booking + Facilities – .NET – Rob, Charlie
③ Challenges —— .NET – Rob, James
① GLaDos (logging) —— Java EE – David, Dan (v1)

Design –

Docs –

Figure 3.2: Initial plan for microservices priorities and allocation to developers

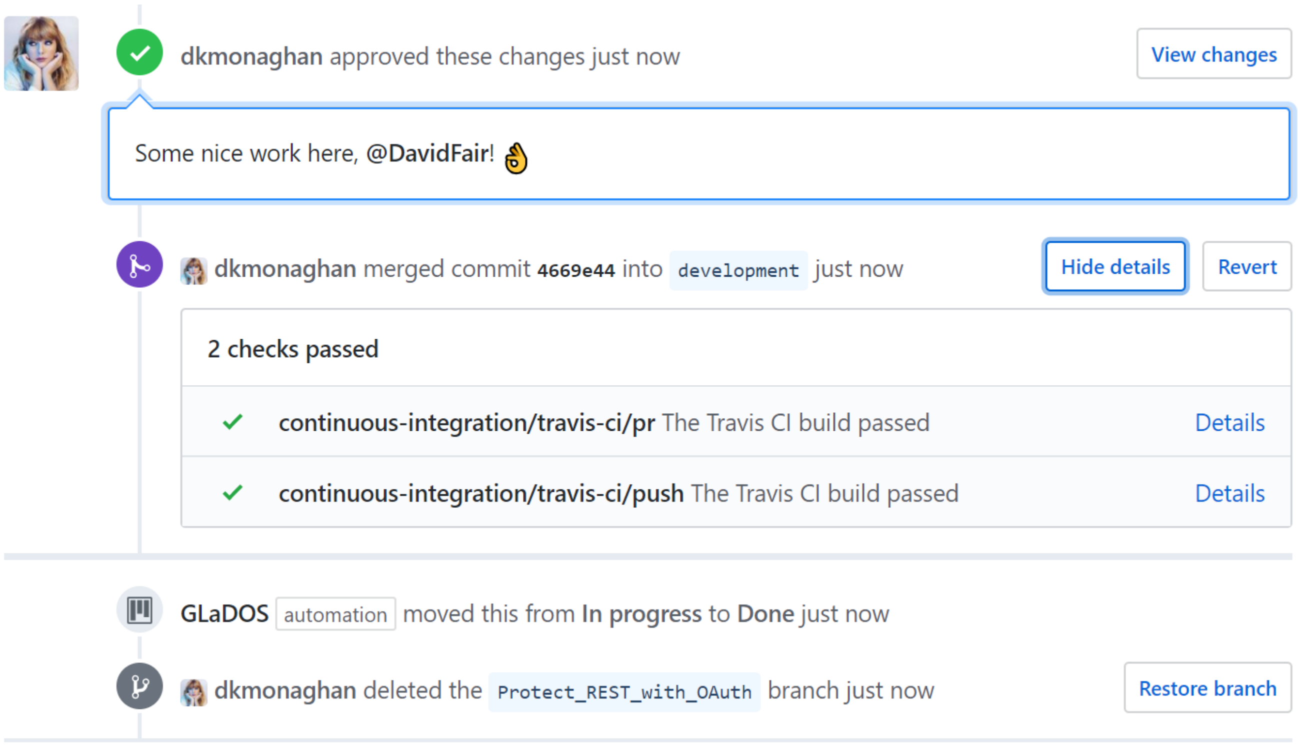
3.2 Supporting Tools

3.2.1 GitHub & TravisCI

The source control for *Aber Fitness* is hosted on *GitHub*[1]. *GitHub* provides multiple features that were incredibly useful during the development phase. This included native integration with *Slack* for notifications straight to the respective development channels. The git flow was complemented with *TravisCI* to automatically trigger unit tests and *Docker* image builds. We also developed a development pattern of requiring all code to be peer reviewed through the use of pull requests and branch protection.

Branch protection is a collection of conditions which must be met before a pull request can be merged into *development* or *master* branches. We configured branch protection in order to ensure that only tested, peer reviewed code would be committed. This reduced the likelihood of new bugs being introduced and ensured code quality was maintained.

[1]https://github.com/sem5640-2018


dkmonaghan approved these changes just now

Some nice work here, @DavidFair! 👌

dkmonaghan merged commit 4669e44 into `development` just now

2 checks passed

✓ continuous-integration/travis-ci/pr The Travis CI build passed

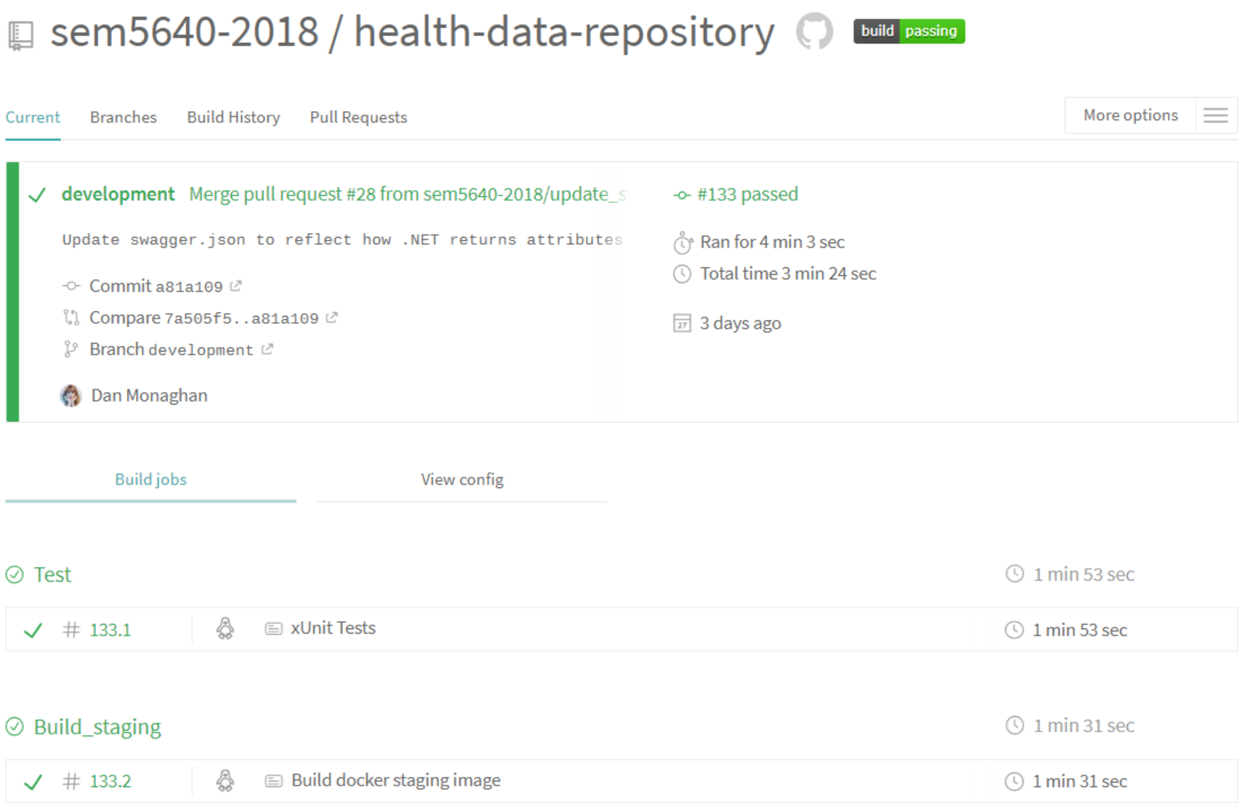
✓ continuous-integration/travis-ci/push The Travis CI build passed

GLaDOS automation moved this from **In progress** to **Done** just now

dkmonaghan deleted the `Protect_REST_with_OAuth` branch just now

Figure 3.3: A screenshot showing a pull request on the *GLaDOS* repository being peer reviewed. The continuous integration checks run on *TravisCI* have completed allowing the branch to merge into an upstream branch.

Once a pull request had been approved and merged, *TravisCI* would then build and push the *Docker* image to *Docker Hub*.


sem5640-2018 / health-data-repository

build passing

Current Branches Build History Pull Requests More options

✓ development Merge pull request #28 from sem5640-2018/update_s

Update swagger.json to reflect how .NET returns attributes

Commit a81a109

Compare 7a505f5..a81a109

Branch development

Dan Monaghan

#133 passed

Ran for 4 min 3 sec

Total time 3 min 24 sec

3 days ago

Build jobs View config

Test 1 min 53 sec

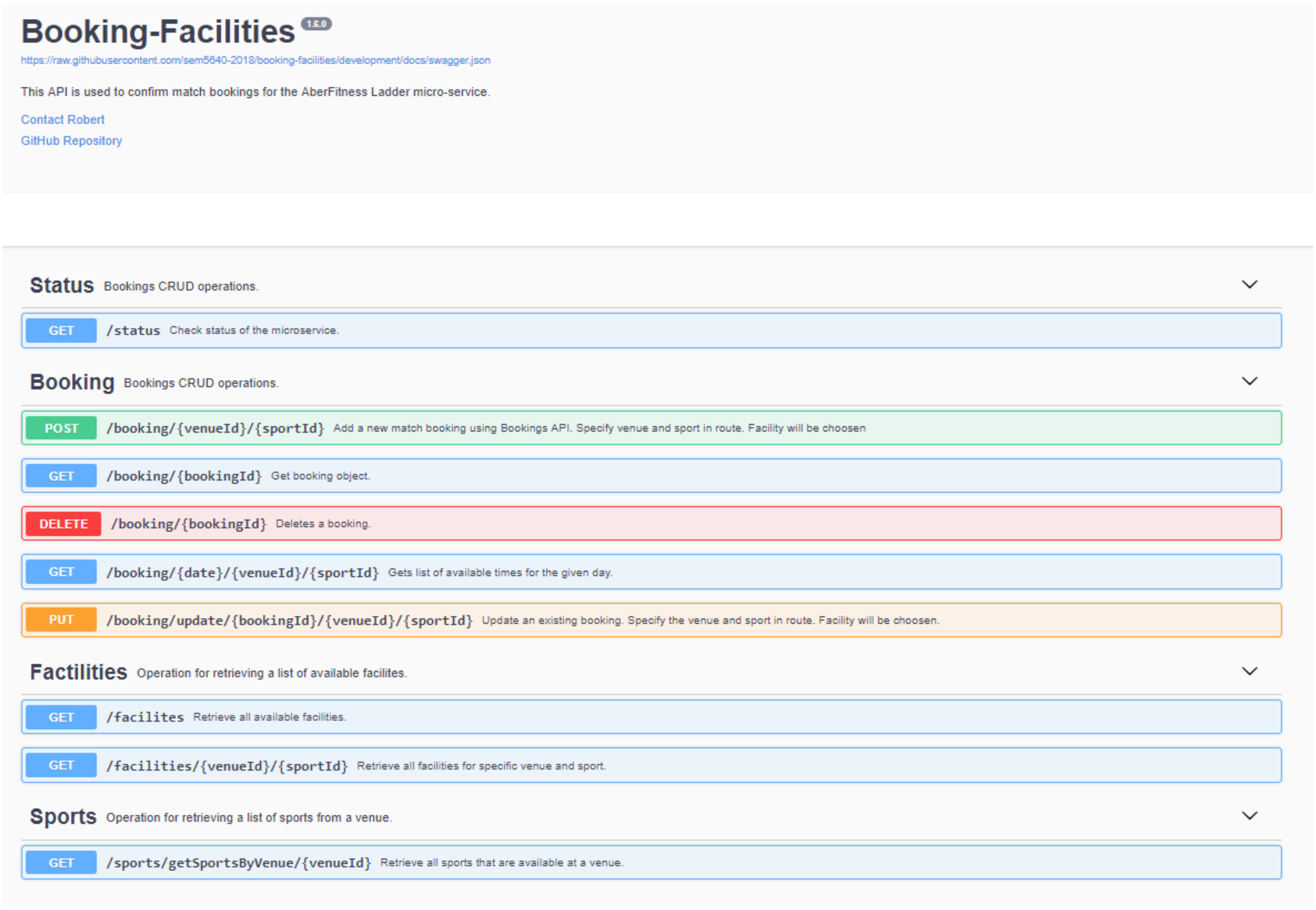
✓	# 133.1	xUnit Tests	1 min 53 sec

Build_staging 1 min 31 sec

✓	# 133.2	Build docker staging image	1 min 31 sec

Figure 3.4: A screenshot from *TravisCI* demonstrating a pull request being merged into the *development* branch. This runs the unit tests and then building the *Docker* image

3.2.2 Swagger



Booking-Facilities 1.6.0

https://raw.githubusercontent.com/sem5640-2018/booking-facilities/development/docs/swagger.json

This API is used to confirm match bookings for the AberFitness Ladder micro-service.

Contact Robert

GitHub Repository

Status Bookings CRUD operations.

GET `/status` Check status of the microservice.

Booking Bookings CRUD operations.

POST `/booking/{venueId}/{sportId}` Add a new match booking using Bookings API. Specify venue and sport in route. Facility will be choosen

GET `/booking/{bookingId}` Get booking object.

DELETE `/booking/{bookingId}` Deletes a booking.

GET `/booking/{date}/{venueId}/{sportId}` Gets list of available times for the given day.

PUT `/booking/update/{bookingId}/{venueId}/{sportId}` Update an existing booking. Specify the venue and sport in route. Facility will be choosen.

Factilities Operation for retrieving a list of available facilites.

GET `/facilites` Retrieve all available facilities.

GET `/facilities/{venueId}/{sportId}` Retrieve all facilities for specific venue and sport.

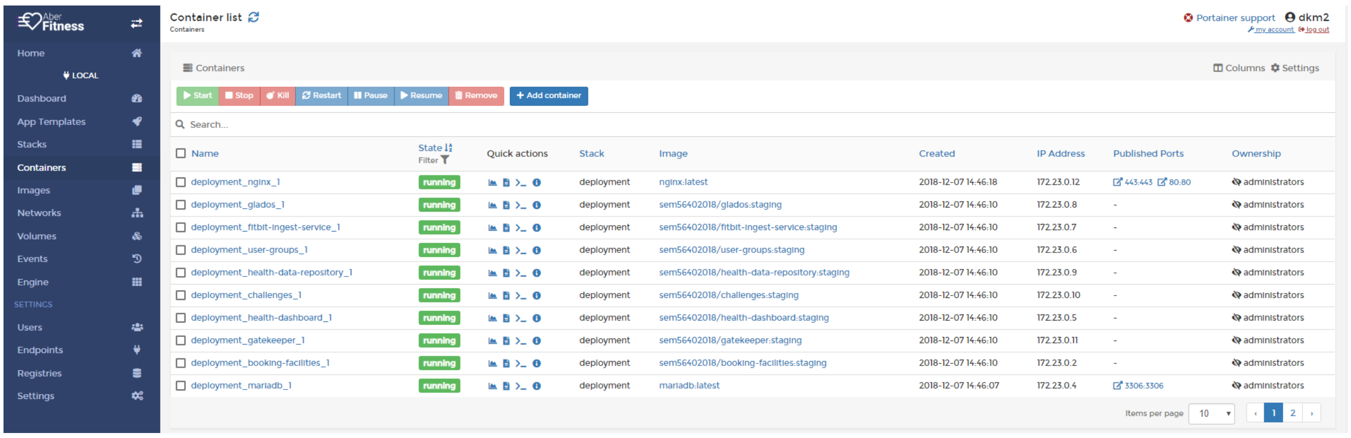
Sports Operation for retrieving a list of sports from a venue.

GET `/sports/getSportsByVenue/{venueId}` Retrieve all sports that are available at a venue.

Figure 3.5: The Swagger interface for the *Booking Facilities* microservice

Swagger is a web based application for documenting API specifications. Each microservice within *Aber Fitness* has a file located in `docs/swagger.json` which defines its API endpoints and any associated data models. *Swagger* was a crucial part of the development process as it allowed us to draft API specifications. Other members of the group could give feedback and identify issues before commencing development.

3.2.3 Portainer



Name	State Filter	Quick actions	Stack	Image	Created	IP Address	Published Ports	Ownership
deployment_nginx_1	running		deployment	nginx:latest	2018-12-07 14:46:18	172.23.0.12	443:443 80:80	administrators
deployment_glados_1	running		deployment	sem56402018/glados:staging	2018-12-07 14:46:10	172.23.0.8	-	administrators
deployment_fitbit-ingest-service_1	running		deployment	sem56402018/fitbit-ingest-service:staging	2018-12-07 14:46:10	172.23.0.7	-	administrators
deployment_user-groups_1	running		deployment	sem56402018/user-groups:staging	2018-12-07 14:46:10	172.23.0.6	-	administrators
deployment_health-data-repository_1	running		deployment	sem56402018/health-data-repository:staging	2018-12-07 14:46:10	172.23.0.9	-	administrators
deployment_challenges_1	running		deployment	sem56402018/challenges:staging	2018-12-07 14:46:10	172.23.0.10	-	administrators
deployment_health-dashboard_1	running		deployment	sem56402018/health-dashboard:staging	2018-12-07 14:46:10	172.23.0.5	-	administrators
deployment_gatekeeper_1	running		deployment	sem56402018/gatekeeper:staging	2018-12-07 14:46:10	172.23.0.11	-	administrators
deployment_booking-facilities_1	running		deployment	sem56402018/booking-facilities:staging	2018-12-07 14:46:10	172.23.0.2	-	administrators
deployment_mariadb_1	running		deployment	mariadb:latest	2018-12-07 14:46:07	172.23.0.4	3306:3306	administrators

Figure 3.6: The Portainer interface for our staging / development Docker host, `docker2-m56.dcs.aber.ac.uk`

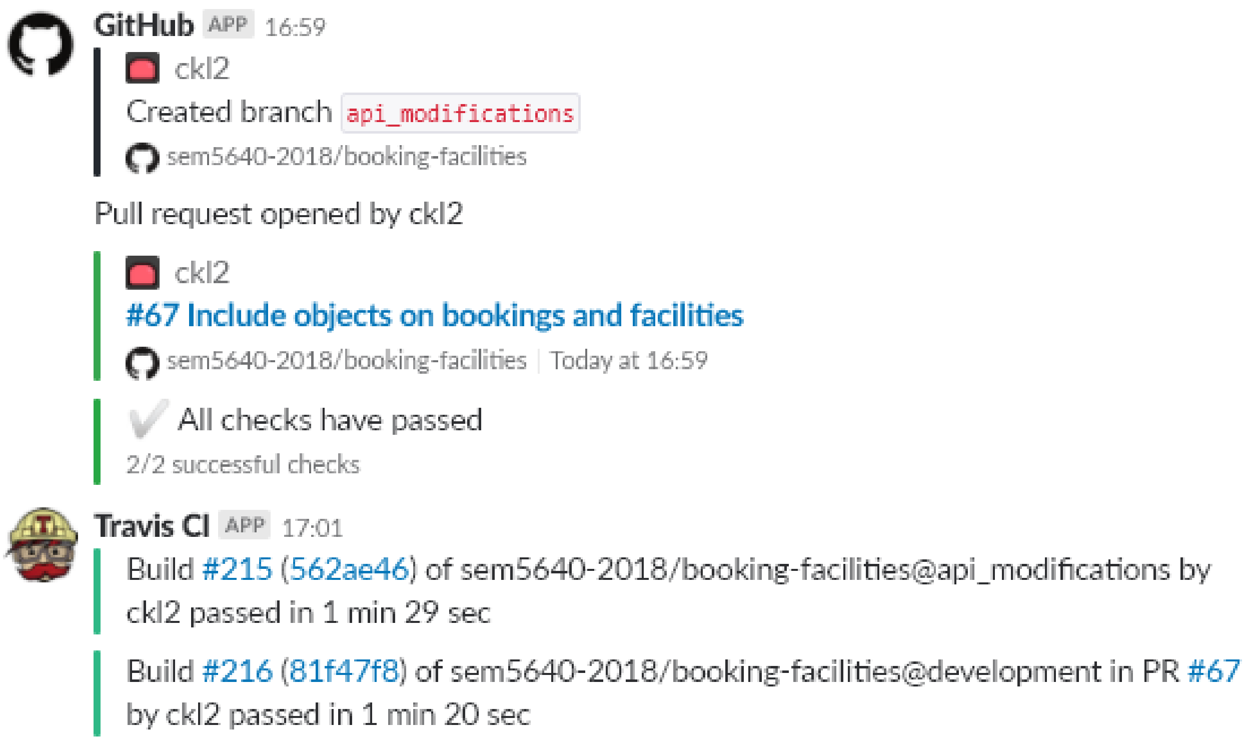
Portainer provides a dashboard for managing *Docker* volumes, networks, images and containers. Whilst completing the initial configuration of the *Docker* images, *Portainer* proved invaluable, it provided rapid visual feedback allowing developers to quickly and easily understand what the host was running. **TODO: More here probably.**

3.2.4 Docker Hub

Docker Hub is an online platform provided by *Docker* which allows *Docker* container images to be uploaded and hosted. The image full system stack is defined in the `docker-compose` file, which can be updated and re-deployed . As part of our build process (**TODO: reference build pipline diagram here**), images are built by *TravisCI* and then pushed to *Docker Hub* before being pulled down onto the *Docker* hosts.

3.2.5 Slack & Deployment

Slack is a hosted chat service designed for offices and teams, and particularly suits itself to the development of software. The group used *Slack* extensively throughout the development of *Aber Fitness* not only to communicate and discuss progress, ideas and troubleshoot problems, but also made extensive use of *Slack*'s integrations with services such as *TravisCI* and *GitHub*.



GitHub APP 16:59

ckl2

Created branch `api_modifications`

sem5640-2018/booking-facilities

Pull request opened by ckl2

ckl2

#67 Include objects on bookings and facilities

sem5640-2018/booking-facilities | Today at 16:59

✓ All checks have passed

2/2 successful checks

Travis CI APP 17:01

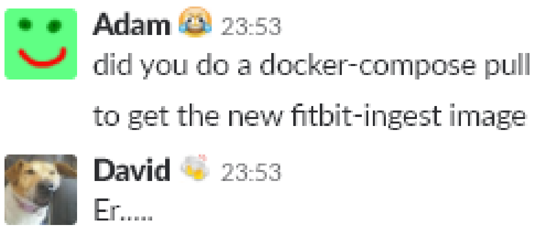
Build #215 (562ae46) of sem5640-2018/booking-facilities@api_modifications by ckl2 passed in 1 min 29 sec

Build #216 (81f47f8) of sem5640-2018/booking-facilities@development in PR #67 by ckl2 passed in 1 min 20 sec

Figure 3.7: A screenshot of the *Slack* channel `#dev-booking-facilities` demonstrating the integrations between *Slack*, *GitHub* and *TravisCI*

Slack also played a major role in our deployment strategy when rolling out updated *Docker* images to our staging host. On multiple occasions we ran into permission issues whilst trying to deploy on the two *Docker* hosts we had been provided by the Computer Science department. Each member of the team had their own individual login to the hosts, so permissions errors would occur after performing commands like `git pull`.

Another issue we ran into was developers forgetting the specific command sequence to update the application images. This would lead to confusion when the upstream changes were not deployed, wasting valuable development time resolving bugs. **TODO: figure no.**



Adam 23:53

did you do a docker-compose pull

to get the new fitbit-ingest image

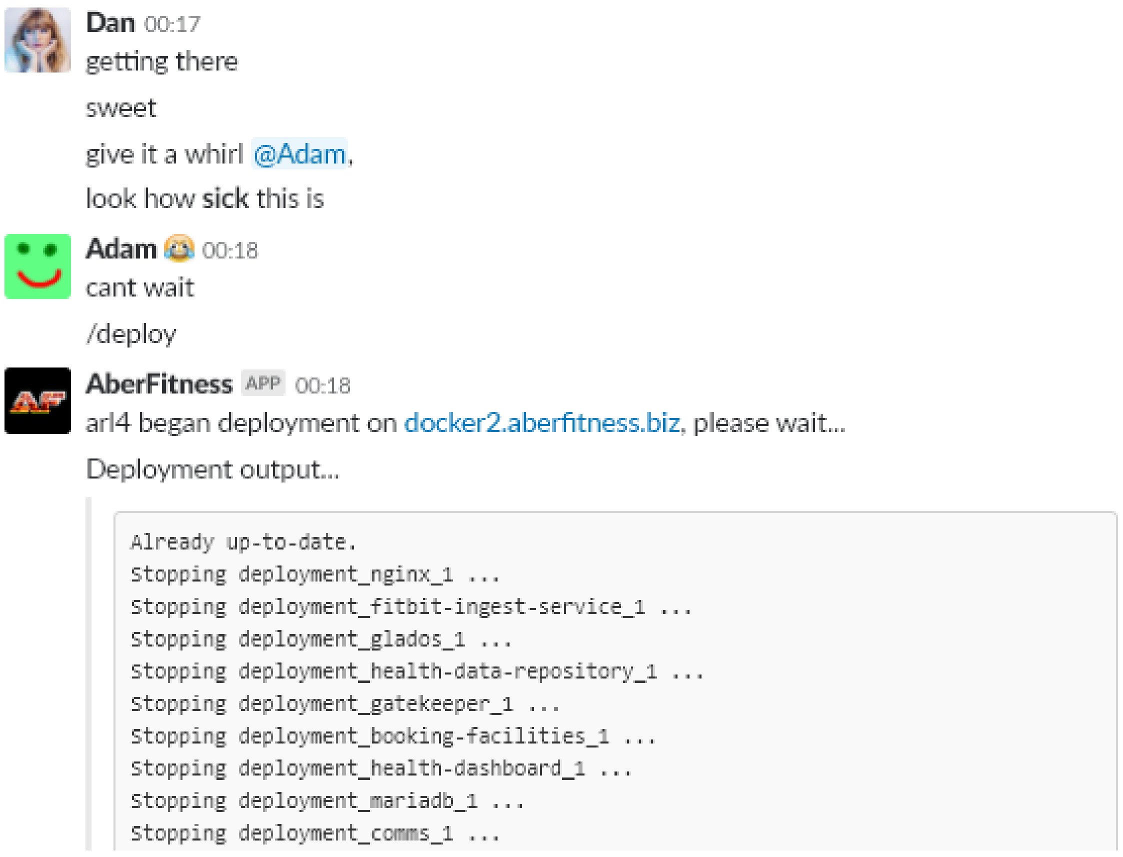
David 23:53

Er.....

Figure 3.8: An example situation where the execution of `docker-compose pull` caused a large amount of confusion amongst *Aber Fitness* developers

Slack ended up providing us with an elegant solution to this, users could call a custom webhook by entering a specific command in a chat channel. A *Slack* application was put together to automatically pull the latest `docker-compose.yml` file from GitHub, as well as updating all the *Docker Hub* images, then re-deploy the stack.

This could all be done from within *Slack* itself through the `/deploy` command. **TODO: Figure no.**



Dan 00:17
getting there
sweet
give it a whirl @Adam,
look how sick this is

Adam 00:18
cant wait
/deploy

AberFitness APP 00:18
arl4 began deployment on docker2.aberfitness.biz, please wait...
Deployment output...

```
Already up-to-date.
Stopping deployment_nginx_1 ...
Stopping deployment_fitbit-ingest-service_1 ...
Stopping deployment_glados_1 ...
Stopping deployment_health-data-repository_1 ...
Stopping deployment_gatekeeper_1 ...
Stopping deployment_booking-facilities_1 ...
Stopping deployment_health-dashboard_1 ...
Stopping deployment_mariadb_1 ...
Stopping deployment_comms_1 ...
```

Figure 3.9: A demonstration of the `/deploy` command being used to re-deploy *Aber Fitness* onto the staging host

Chapter 4

Design

4.1 System Overview

The *Aber Fitness* system is broken down into a number of microservices in order to aid portability, scalability and promotes a more maintainable codebase. After reviewing the initial project specification, the following microservices were created:

- **Booking & Facilities** - The *Aber Fitness* offers functionality for users to be able to schedule bookings at sports venues, such as swimming pools and squash courts. This microservice is called used by the *Ladders* service to create bookings for competitions.
- **Challenges** - The system offers the ability to give users activity challenges, for example completing a number of steps in a specific timeframe. These challenges can also be 'group' challenges, where a number of users can compete against each another to achieve goals such as furthest distance walked in a week, etc.
- **Communications** - This microservice provides an API for other services to send email notifications to users. It does not present any form of web UI, and users do not directly interact with it. This system could also be easily expanded to send out text messages, push alerts, etc. depending on future requirements.
- **Fitbit Ingest Service** - At launch, the *Aber Fitness* platform allows a user to link their *Fitbit* accounts to the system in order to import their activity data. The service periodically polls the Fitbit API for new data on the users' behalf, then stores this into the *Health Data Repository*.

  With the possibility of adding future platform support the "ingest service" concept was created. This would allow us to support services such as Apple's *HealthKit* and other fitness tech providers. This architectural design means that activity data can be normalised by a number of "ingest services" before being passed through to the *Health Data Repository* service for storage.
- **Gatekeeper** - *Gatekeeper* is *Aber Fitness*'s OpenID Provider, and handles all authentication within the system. User credentials and account metadata is stored within *GatekeeperGatekeeper* uses the OAuth 2.0 flow and is responsible for providing a single sign-on service for all of the various microservices. Microservices also contact *Gatekeeper* to obtain and verify tokens when calling internal APIs.
- **GLaDOS** - *GLaDOS* is the centralised auditing mechanism for *Aber Fitness*. It presents a REST API which is used to store audit data; such as when a user's data was accessed, modified, or deleted. *GLaDOS* is provides a Status page which displays the availability of all the other microservices.

- **Health Dashboard** - *Health Dashboard* is the first interface users will encounter after logging in, or navigating to *Aber Fitness*. It provides the user with an overview of their recent activity as well as providing updates on any challenges or ladder competitions the user may be involved in.
- **Health Data Repository** - The *Health Data Repository* service is responsible for providing an API for accessing and storing activity data. It receives normalised activity data from the Ingest Services, and provides multiple API endpoints for other microservices to access user activity data.
- **Ladders** - *Ladders* is responsible for organising and managing ladder style competitions among users of the system. Users can compete in sporting championships for a variety of competitive sports such as tennis, running or cycling etc. The *Ladders* also automatically books venues for upcoming competitive events, which is managed by the *Booking Facilities* microservice.
- **User Groups** - *Challenges* can also be turned into a competition amongst users of a group. For example, a group may consist of a few friends or an entire office department. Users within a group compete to complete goals, such as who can achieve the most steps in a single day. The *User Groups* service is responsible for managing users into groups, and allowing users to leave and join other groups.

Chapter 5

Implementation

5.1 Deployment

5.2 Java EE

5.2.1 Common Components

5.2.1.1 Handling Dependencies and Static Analysis

At the start of the project the JavaEE development teams agreed to use *Maven* [**?**] to handle their dependencies. This allows the entire build process to be ran at the commandline rather than within the IDE, which is essential for *TravisCI* integration.

Maven uses an XML file called *POM.xml* to define the dependency names, versions and additional options. These are fetched from Maven Central and cached locally for future builds. To add a new dependency a developer simply visits the hosting website and copies the XML provided alongside each package. Additionally developers can use the *Scope* option which instructs the packaging plugin the associated *JAR* files are provided by the Payara runtime.

IntelliJ natively supports reading from Maven's *POM.xml* file, which defines the list of dependencies and compilation steps. Maven also contains a plugin which allows developers to package the built files into a web archive (WAR). This did have some caveats however; running '*mvn war:war*' would result packaging artefacts from the previous compilation, without compiling the current application. This could lead to confusion as new changes would no be reflected without '*mvn compile war:war*'.

Whilst working locally developers could still rely on IntelliJ's built in mechanism for packaging and deploying to a local Payara instance. This allowed for rapid development feedback and iteration without the overhead imposed by Maven.

5.2.1.2 Creating Docker Images

By utilising the ability to run the entire build process on the command-line with Maven we were able to quickly add Travis CI integration. This would clean compile the entire application, run the full *JUnit* test suite and lastly package a WAR file.

Initially we used *CheckStyle* to also enforce coding style conventions. However the rules were over-specified. Additionally, unlike other formatting tools there was no option to emit a patch-file which allows developers to automatically fix-up many errors such as incorrect whitespace or formatting.

The Java teams migrated to the *PMD* [**?**] static analysis tool: this warns about unused fields and variables, unsafe constructs,and dangerous coding patterns. The Maven test target was modified to run the *PMD* [**?**] static analysis tool before any unit tests. If any violations were detected the build would immediately stop, ensuring all modifications were inspected.

The groups proceeded to add a *Dockerfile* to build a docker image for later deployment. Payara provide multiple images on *DockerHub* [**?**], which range from the full server to an embedded instance.

Initially we copied the local web archive, which was built on Travis, into the full Payara deployment folder. This allowed us to use the web administration console to resolve various deployment issues without having to dig through log files. After moving changing the default source code layout, which resolved *resources* not being included in the final archive, we successfully deployed GLaDOS to the full instance.

However we could not get Fitbit Ingest to successfully deploy, they had already started implementing the OAuth flow which is required by both the Fitbit API and our own internal API. The internal SSL implementation had changed with JDK update 191, which prevented the *CDI* layer from initalising the associated beans.

As the bundled JDK version is controlled by Payara in the base image we proceeded to look for an alternative solution. We decided against modify the image to transplant a specific JDK at build time. Instead we that upstream had opened an issue [2] and used reflection to resolve the problem internally. As this fix was not released into the latest stable we had to switch to running pre-release images for all JavaEE applications.

5.2.1.3 Migrating to Micro Image

The Payara Microprofile image is designed specifically to be used in Docker deployments. Whilst this does not implement the full EE, the memory usage is ten times lower at 90MB. The profile still provided the required set of services for the application so there was a significant benefit to completing this migration. However trying to deploy to the micro instance resulted in an exception being thrown whilst completing the implicit bean discovery phase.

The CDI 1.1 specification and above requires the server to automatically scan for injection points and pair it with matching enterprise Java beans (EJBs). This can fail with older dependencies, so we initially suspected one of the dependencies did not correctly use the *beans.xml* annotation to turn this off.
Switching off bean discovery and manually annotating them allowed the ingest service to correctly deploy. However bean discovery is required for Facelets 4.0 and an exception will be thrown if it is not enabled. Adding to the confusion we created a new project and removed all but the essential dependencies to discover the exception was still thrown.

By looking at the package namespaces associated with the injection points and running ‘mvn dependency:tree’ we could see that ‘*org.sonatype.guice*’ was the culprit. Walking up the tree we discovered that the WAR plugin was being included in our packaged dependencies. At deployment the microprofile server tried to start a Maven instance and failed, thus removing this dependency correctly allowed the application to start. The development teams realised that the plugin was included in our installed copies so we could still continue to use it.

5.2.1.4 Setting up JTA Targets

Both services started off by instantiating the single JDBC connection through the driver manager as required. This had some caveats however: Firstly, the JDBC driver had to be distributed and managed within the web archive. Secondly, there was no form of connection pooling to allow for scaling at the persistence layer.

Traditionally a connection pool is created using the administration web GUI. However this would require a developer setup the pool each time continuous deployment finished on a full instance. The microprofile we had just switched to did not provide a web GUI at all.

Glassfish allows a web application to specify resources found on the server using ‘glassfish-resources.xml’. This XML file can also use system environment variables allowing us to protect and specify database credentials on the deployment targets. The examples found online primarily pertained to a *MySQL* deployment, so multiple iterations of testing and deployment were required to get the pool working successfully. This switch allows an application to detect and create connection pools at deployment, allowing the images to rapidly redeploy facing another database instance.

5.2.2 Fitbit Ingest Service

5.2.2.1 To Do

5.2.3 GLaDOS

5.2.3.1 REST API

The primary role of GLaDOS is to store audit data so that users can view how their data was viewed and modified. Whilst there are native implementations, such as the Java Messaging Service, these require the calls be proxied into a local JVM instance to handle the transport component. We chose to serialise into JSON and use REST as the transport mechanism as all applications had native support for this setup.

JAX-RS is an API specification for JSON parsing in Java, additionally Payara provides *Jersey* as the implementation of this API at runtime. This avoided us having to package additional dependencies into the web archive. This also allows us to build standard response pages based on the error code internally generated, avoiding having to develop error views.

5.2.3.2 Unit Testing and Mocks

A new class abstracting the database layer allowed us to avoid writing entity handling at this. We instead focused on writing unit tests which verified the various endpoints correctly serialised or de-serialised data. *Mockito* [**?**] allows developers to inject mock objects by specifying the class which the test fixture is using.

We could not use dependency injection within the endpoint implementation, as the instantiation point is internal to the framework. This doesn’t prevent us from using mocks as Mockito allows a developer to inject them through the reflection methods built into the library. A database call such as ‘*db.getEntry(id)*’ could be tested to see which status codes would be sent and verify the data,if any, was serialised correctly.

5.2.3.3 Integration Testing with Arquillian

Arquillian [?] allows developers to specify the classes to archive and deploy within the text fixture. Payara also provides an embedded Arquillian test container which can be ran at test time to exercise sections of a full system. This would allow us to write integration tests for an end-to-end operations, such as POSTing data to an endpoint and ensuring it is written to the persistence layer.

There was extensive documentation on the methods required to pack an archive, but limited examples on handling Maven dependencies. As the embedded instance provides no runtime methods we were having problems getting the container to correctly run. Developers could specify the full class paths for their dependencies to ensure the were packaged, but this was extremely verbose and fragile.

We looked into using the Maven dependency resolver, which allows developers to get a list of runtime Jars and package them into the archive. However this would led to another set of problems where Maven was trying to export them into a *.zip* format, then throw an exception because this was not supported.

With no easy was to control the format that runtime dependencies were exported in, combined with a lack of documentation pertaining specifically to Maven we agreed to abandon this method of automated testing. This would give us more time for implementing other components within the deadline. We opted to rely on manual integration testing for the duration of the project instead.

5.2.3.4 Entity Management

This service only persists Audit Data, therefore we ultimately need to persist a single entity. Whilst many ORM solutions exists for Java,such as *Hibernate*, we decided against them due to the implementation overhead that was required.

Instead we relied on the persistence layer provided in the EE specification. Using the connection pools which were setup for both Java services (see 5.2.1.3), we could specify which JTA the injected entity manager should use.

The table schema is managed through annotations on an entity class. This also allows the implementation provided by Payara, *EclipseLink*, to create the tables required at deployment time. As the framework relies on the field types to determine which underlying storage to use there is a hidden pitfall. If the type does not have a native serialisation Java will try to persist this using binary data.

Instants are used to specify timestamps on logs, these are cross compatible as they use *ISO 8601* [?], a string specifier for absolute time points. This format allows both .Net and Java application to send timestamps in the following format *2018-11-28T12:04:14Z.* However as this type was added in JDK 8 there is no native storage conversion built into the entity framework.

Two adaptors were written based on the *AttributeConverter* interface. These provide methods for marshalling and un-marshalling Instants into String objects, which the entity manager can easily store. Annotating the field installed the converting class and correctly updated the schema. This also has the added benefit of making the stored timestamps human readable within the database, which extremely helpful for debugging.

5.2.3.5 Developing Facelets

GLaDOS also provides a page that allows a user to retrieve audit data associated with their account. Administrators can lookup any users audit data by user their unique ID too. In addition there is a status page which allows anyone to view the status of all other micro-services without logging in.

The front-end of GLaDOS use facelets to implement the MVC pattern. A backing bean for the user data uses named queries from the persistence layer to retrieve data into a list of entity objects. We switched to using *PrimeFaces* [?], which is a fork of the now deprecated *RichFaces*. This allows us to use components such as *DataTables* that are implicitly reactive rather than us developing our own supporting CSS.

The Fitbit Ingest Service had completed OAuth implementation for connecting to internal and external APIs. This was ported across to GLaDOS and further modified. As the ingest service has no front-end they don't need to persist *JSON Web Token (JWT)*. Additional methods were written specifically for this microservice. These include using the HTTP session handling built into the web framework to persist the token on the server. Another helper class was written which validates the stored JWT as the user navigates through protected pages, or POSTs any requests.

Service statuses were implemented using a singleton Java bean which is instantiated at deployment. Using the scheduling capability provided for EJBs we poll all services every 20 seconds in a seperate thread. All services implement an endpoint at */api/status* that returns a 200 or 204. This is stored until the page is retrieved where the backing bean, acting as a presenter forward the results on.

As the application stack uses an Nginx instance to reverse proxy queries the URLs must take this into account when they are generated within the facelet. .Net makes this trivial by calling '*App.UsePathBase(URL)*' at startup, however Java applications rely on the context root. This was interfering with the URL rewriting that Nginx performs resulting in the service using the '$\backslash glados \backslash glados \backslash$' base path.

Initially we used an environment variable to manually generate links between the pages of the service and set the context root to '$\backslash$'. However, this quickly proved to be untenable when using forms, as POST requests were being sent to '$\backslash$*destination*' rather than '$\backslash glados \backslash$*destination*'. Ultimately to work around this an exception was added to Nginx configuration to avoid re-writing any URLs for GLaDOS, and instead rely on the service to correctly address all requests. This allowed us to switch back to generating addresses based on the *outcome* tag and use forms on the service.

5.3 .NET Core

5.3.1

Common Components some blurb about .net core here

5.3.2 Booking Facilities

5.3.3 Challenges

5.3.4 Communications

5.3.5 Gatekeeper

5.3.6 Health Data Repository

5.3.7 Health Dashboard

5.3.8 Ladders

5.3.9 User Groups

Chapter 6

Testing

Chapter 7

Status

Chapter 8

Evaluation

Appendices

Bibliography

[1] "What is Scrumban?" Available at: https://leankit.com/learn/agile/what-is-scrumban/. [Accessed: 08- Dec-2018]

[2] "TLS fails with jdk 8u191" Available at: https://github.com/payara/Payara/issues/3284 [Accessed: 10- Dec-2018]