



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

1	Overview	1
2	Requirements	2
3	Development Methodolgy	3
3.1	Initial Project Plan	3
3.2	Supporting Tools	5
3.2.1	GitHub & TravisCI	5
3.2.2	Swagger	8
3.2.3	Portainer	9
3.2.4	Docker Hub	9
3.2.5	Slack & Deployment	9
4	Design	12
4.1	System Overview	12
5	Implementation	14
5.1	Deployment	14
5.2	Java EE	14
5.2.1	Common Components	14
5.2.2	GLaDOS	16
5.2.3	Fitbit Ingest Service	17
5.3	.NET Core	17
5.3.1	17
5.3.2	Booking Facilities	17
5.3.3	Challenges	17
5.3.4	Communications	17
5.3.5	Gatekeeper	17
5.3.6	Health Data Repository	17
5.3.7	Health Dashboard	17
5.3.8	Ladders	17
5.3.9	User Groups	17
6	Testing	18
7	Status	19
8	Evaluation	20
	Appendices	21

LIST OF FIGURES

3.1	An initial design diagram which was used to break the project down into smaller microservices and their interactions with each other	4
3.2	Initial plan for microservices priorities and allocation to developers	5
3.3	A screenshot showing a pull request on the <i>GLaDOS</i> repository being peer reviewed. The continuous integration checks run on <i>TravisCI</i> have completed allowing the branch to merge into an upstream branch.	6
3.4	A screenshot from <i>TravisCI</i> demonstrating a pull request being merged into the <i>development</i> branch. This runs the unit tests and then building the <i>Docker</i> image	7
3.5	The Swagger interface for the <i>Booking Facilities</i> microservice	8
3.6	The Portainer interface for our staging / development Docker host, <code>docker2-m56.dcs.aber.ac.uk</code>	9
3.7	A screenshot of the <i>Slack</i> channel <code>#dev-booking-facilities</code> demonstrating the integrations between <i>Slack</i> , <i>GitHub</i> and <i>TravisCI</i>	10
3.8	An example situation where the execution of <code>docker-compose pull</code> caused a large amount of confusion amongst <i>Aber Fitness</i> developers	10
3.9	A demonstration of the <code>/deploy</code> command being used to re-deploy <i>Aber Fitness</i> onto the staging host	11

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, it's a start for now. Not 100% sure what Neil's 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**

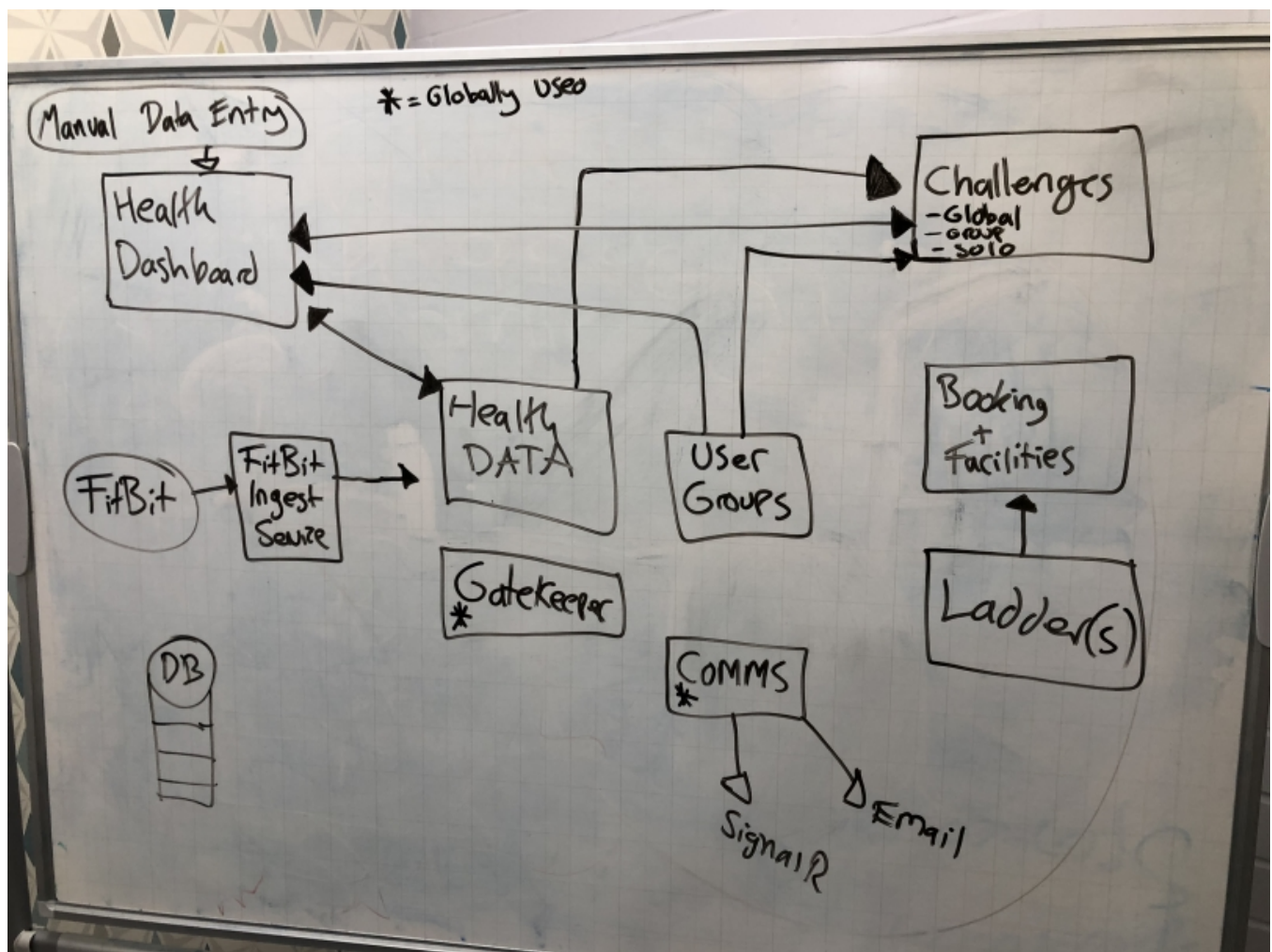


Figure 3.1: An initial design diagram which was used to break the project down into smaller microservices and their interactions with each other

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**

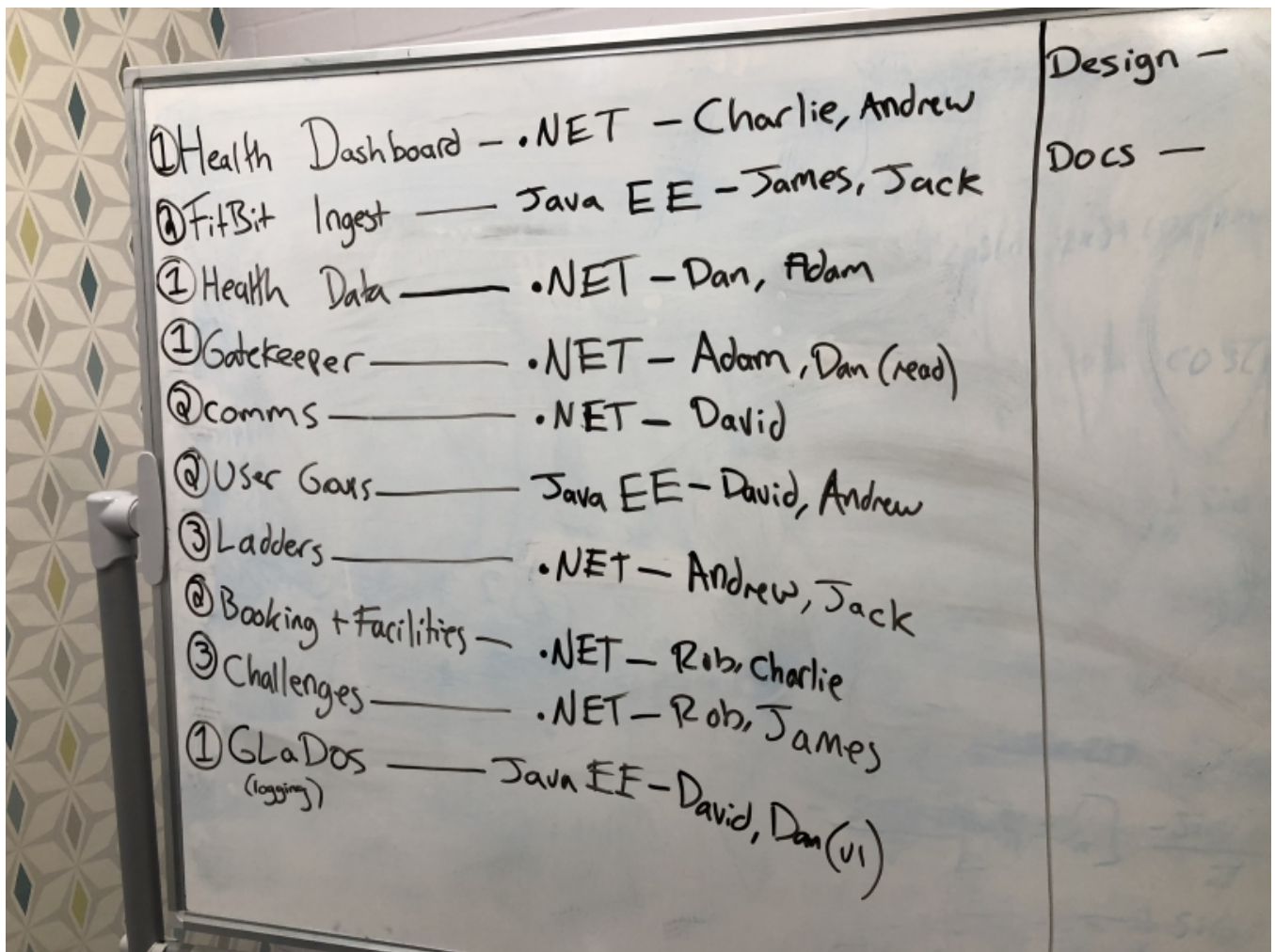


Figure 3.2: Initial plan for microservices priorities and allocation to developers

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>

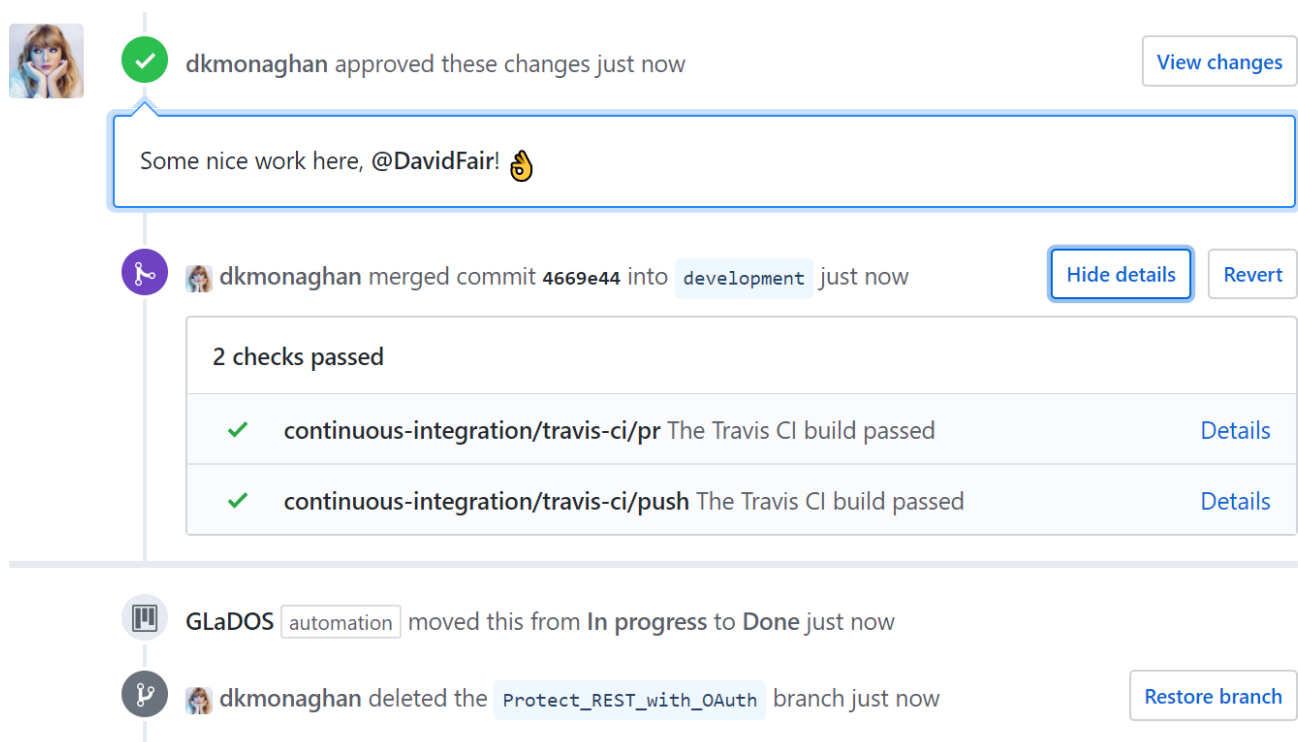


Figure 3.3: A screenshot showing a pull request on the *GLaDOS* repository being peer reviewed. The continuous integration checks run on *TravisCI* have completed allowing the branch to merge into an upstream branch.

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

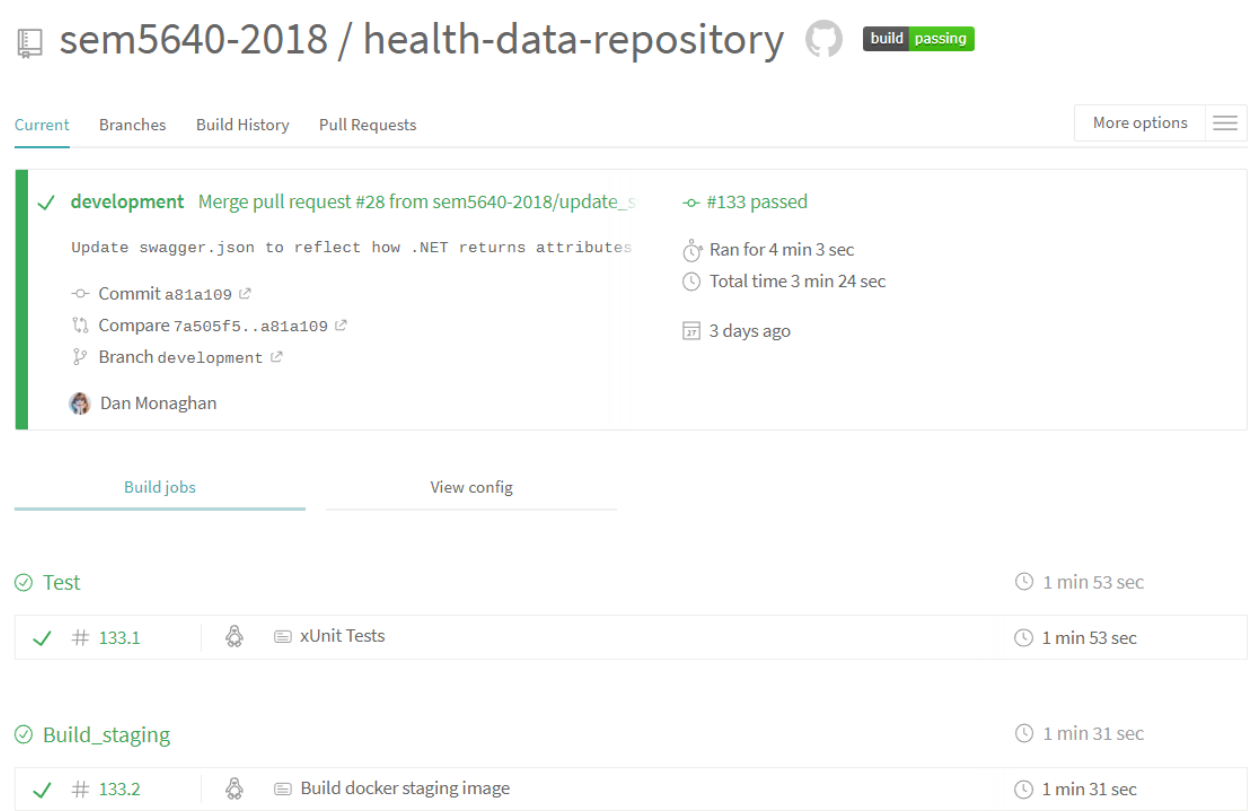


Figure 3.4: A screenshot from *TravisCI* demonstrating a pull request being merged into the *development* branch. This runs the unit tests and then building the *Docker* image

3.2.2 Swagger

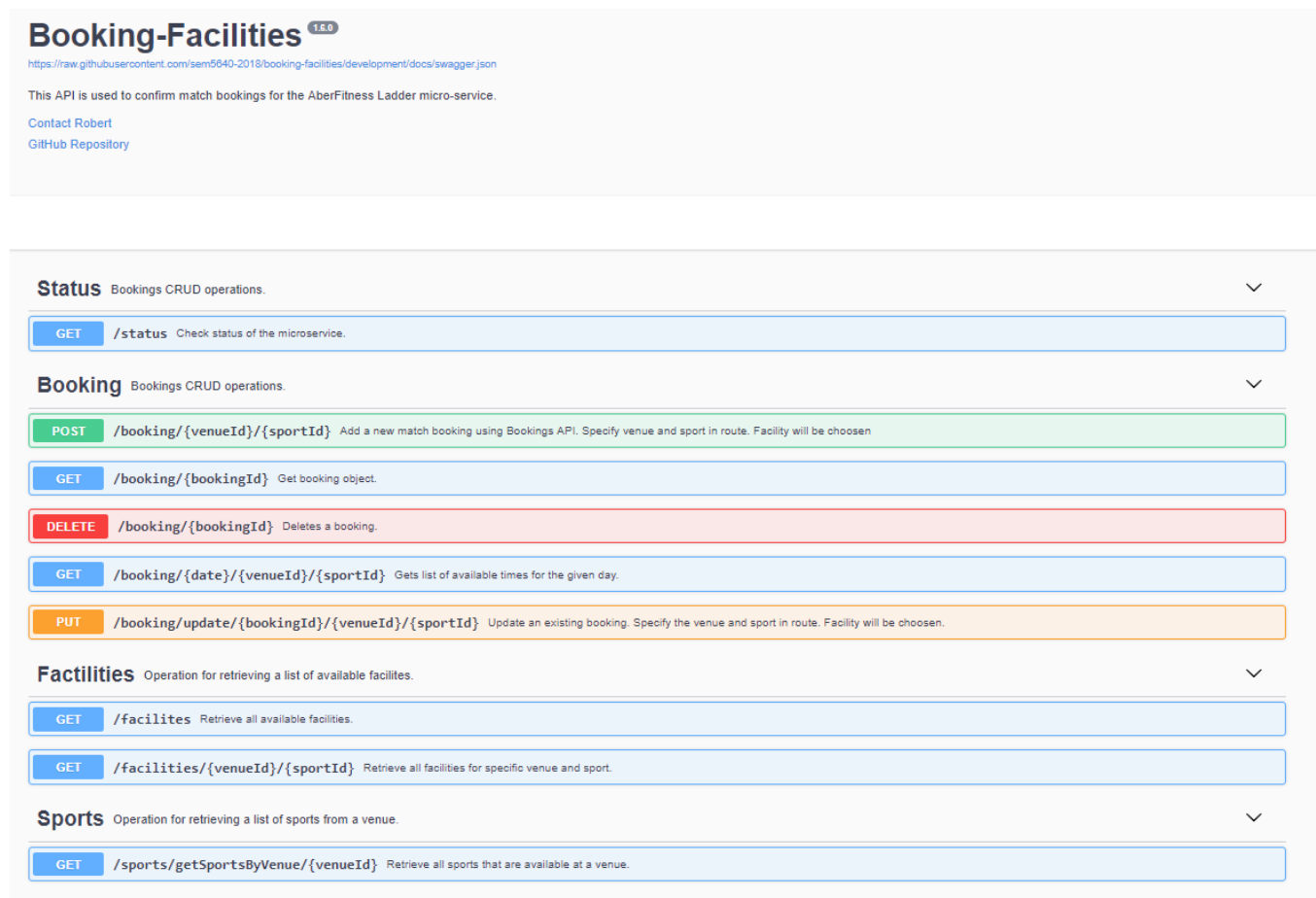


Figure 3.5: The Swagger interface for the *Booking Facilities* microservice

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

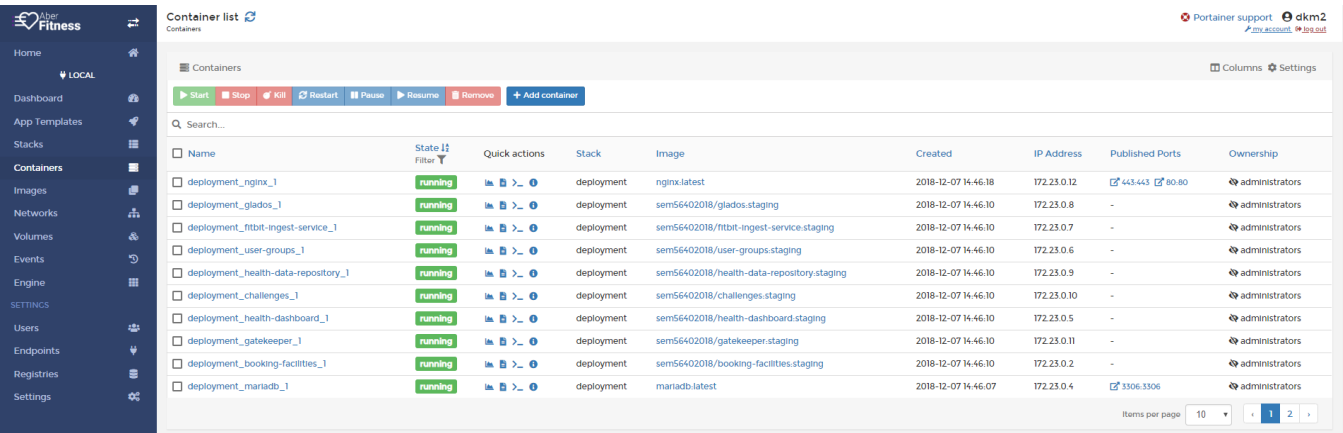


Figure 3.6: The Portainer interface for our staging / development Docker host, `docker2-m56.dcs.aber.ac.uk`

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.

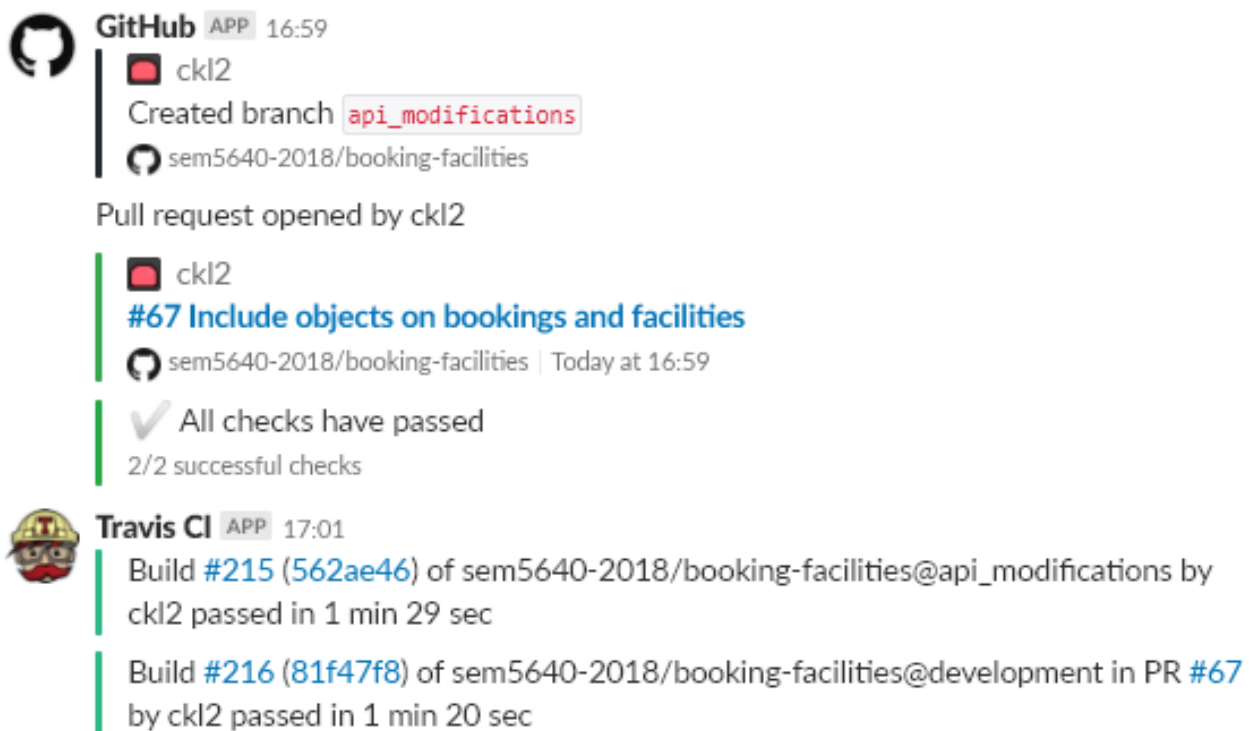


Figure 3.7: A screenshot of the *Slack* channel `#dev-booking-facilities` demonstrating the integrations between *Slack*, *GitHub* and *TravisCI*

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.**

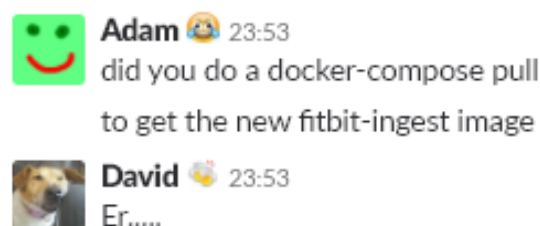


Figure 3.8: An example situation where the execution of `docker-compose pull` caused a large amount of confusion amongst *Aber Fitness* developers

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.**

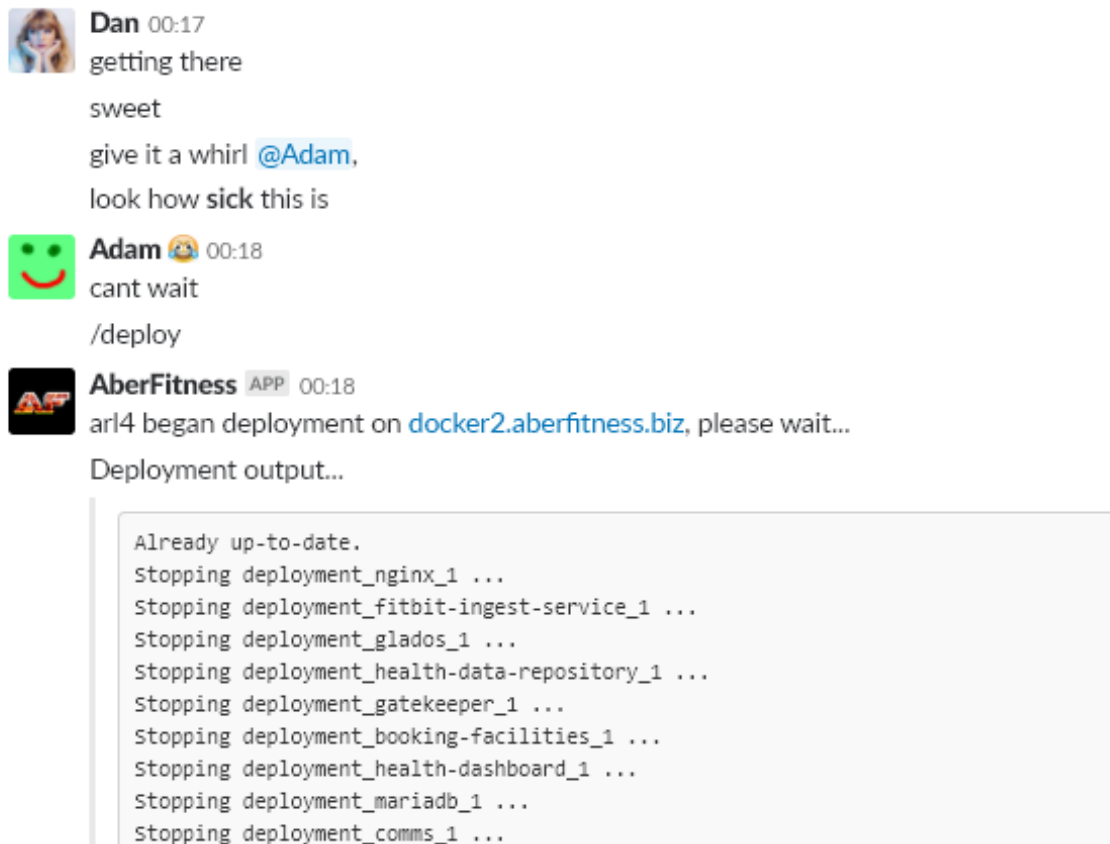


Figure 3.9: A demonstration of the `/deploy` command being used to re-deploy *Aber Fitness* onto the staging host

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 *Gatekeeper*. *Gatekeeper* 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* 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 initialising 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 GLaDOS

5.2.2.1 REST API

- Using JAX-RS

5.2.2.2 Unit Testing and Mocks

- Mockito - Database Connection Wrapper

5.2.2.3 Integration Testing with Arquillian

- Problems injecting correct set of deps - Maven could use reflection, but did not support zip - Switching to manual tests

5.2.2.4 Entity Management

- Using the Javax persistence layer instead of ORM - Writing adaptors to marshall / unmarshall Instant objects

5.2.2.5 Troubleshooting CDI

- Description of problems - Trying other micro-profile servers - Dropping servlet version - Removing mvn-war dependency from deployment

5.2.2.6 Developing Facelets

- Adding backing beans and OAuth - Generated URLs and Nginx reverse proxy

5.2.3 Fitbit Ingest 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]