Evolving a Modern Monolithic Application to a Next Generation Microservices Application

Case Study: Red Hat Aerodoc Backend Application

Final Report

Colum Foskin

20062042

Supervisor: Eamonn De Leastar

BSc (Hons) in Applied Computing

Plagiarism Declaration

Unless otherwise stated all the work contained in this report is my own. I also state that I have not submitted this work for any other course of study leading to an academic award.

1. Introduction

1.1. Background

Software applications that are built on a monolithic architecture can be either single-tiered or multi-tiered applications. The components of such systems are often very tightly coupled. This commonly leads to very large and complex codebases, which can become hard to manage, maintain and can often slow down deployment times. It is widely thought that using a microservices architecture helps to address many of these issues (Richardson, 2014). A correctly designed microservices application should be built with the aim of being loosely coupled and as cohesive as possible. Using a microservices architecture will also help to ensure that the code base will be easier to maintain (Fowler, 2014).

1.2. Objective

The objective of this project is to take an existing open source, monolithic application and re-architect its backend using a microservices architecture. The project will investigate and analyse the application in its current state, and then compare it to a version which will be rewritten using microservices. In addition, the project will investigate opportunities to enhance the implementation using emerging patterns, frameworks and new language features. This project will be developed using an Agile methodology, in an open source community project supported by Red Hat Mobile (Redhat.com, 2016).

1.3. The Subject Application

The application chosen for this project is called Aerodoc (Aerogear.org, 2016 A), which is part of a Red Hat open source software framework called Aerogear (Aerogear.org, 2016 B). Aerogear provides components for bringing mobile and the enterprise together. It offers a set of flexible and extensible cross-platform libraries, as well as various server side components that will simplify the typical mobile development cycle and the required infrastructure setup.

2. System Overview

There are three components that are required to complete the workflow of the Aerodoc application. The main component is the Aerodoc admin application, which is a Java EE backend system with an Angular.js web client. This Java backend system also provides a RESTful Application Programming Interface (API) which may be consumed by the systems mobile clients. The Aerodoc application also communicates with the mobile clients via the Aerogear Unified Push Server (UPS), which is the final component of the system. This section will describe each of these components in detail.

2.1. Unified Push Server (UPS)

One of the AeroGear projects main components is the Aerogear Unified Push Server (UPS) (Aerogear.org, 2016 C). This provides a unified API for sending push notifications to many different mobile devices across multiple platforms such as Android, IOS and Cordova. The messaging platforms that the server currently supports are Apple's APNs (Developer.apple.com, 2016), Google Cloud Messaging (GCM) (Google Developers, 2016) and Mozilla's SimplePush (Wiki.mozilla.org, 2015). The diagram in Figure 1 provides an overview of how the UPS interacts with the other components of the overall system.

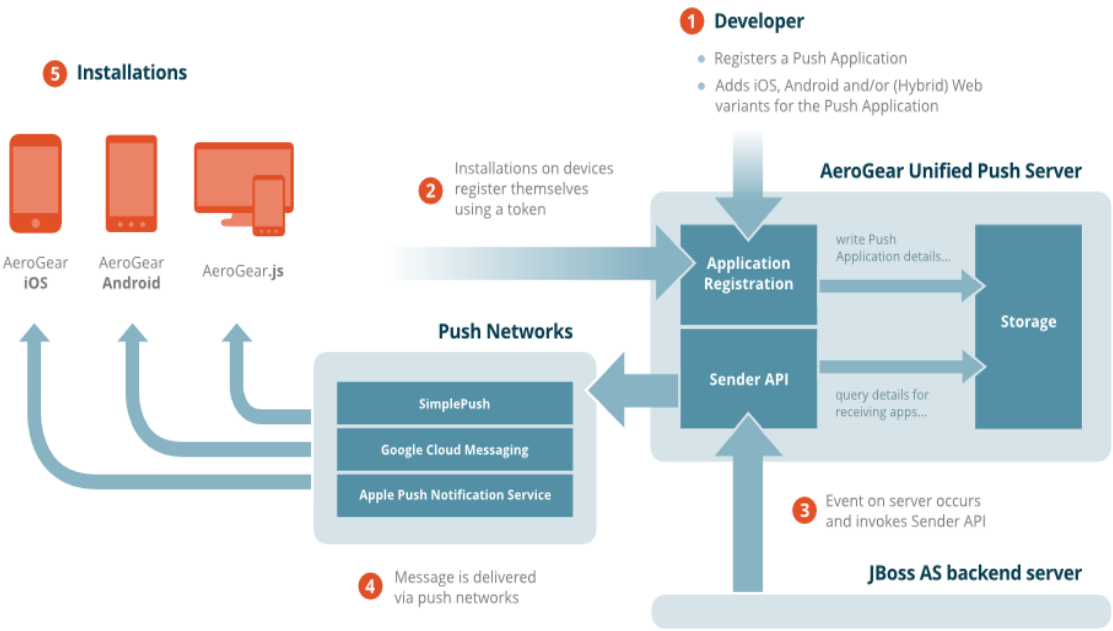


Figure 1 Aerogear Unified Push Server workflow (Aerogear.org, 2016)

The diagram shows the following steps:

1. The developer may create a new push application on the UPS using the admin web console. The next step to be taken is to add a variant for the push application. A variant is a group of client devices for a specific platform. One variant may be added for each platform and the UPS generates a set of credentials for each variant.

2. The credentials for a variant can then be entered into the code base of the relevant client application. The application can then be built and launched on a mobile device, which will register the device application against the UPS.

3. This step shows a backend application which invokes a sender API, to trigger a push notification when an event occurs. In the case of this project, this application is the Aerodoc backend system.

4. The UPS sends a push message to a client application via the relevant push network.

5. The final step shows the different mobile platforms which may have a client application installed to receive push messages.

The UPS instance which is being used for this project is hosted on the Red Hat OpenShift cloud platform, and can be found in Appendix A.

2.2. Aerodoc Application

2.2.1. Overview

The Aerogear Aerodoc application was created to showcase the capabilities of the Aerogear Unified Push Server. The application demonstrates how the Unified Push Server can interact with a real life business application. The business model of the system is based on a fictional company in the healthcare industry. The company has a particular healthcare product which they wish to sell. They try to do so by using a team based in a call centre which makes cold calls to doctors all over America – who may be potential customers. The company also have a team of sales agents, who are on the road daily. Once the team secure an appointment (referred to as a "lead") with a potential customer, they need to notify the sales agents in the area that a new lead is available to visit. In an industry where there are many companies competing with one another to sell a similar product, the ability to convert cold calls into a lead, and then process that lead, is a huge advantage over the company's competitors.

2.2.2. Application Flow & Usage

An admin user can create a new lead on the system by entering the leads name, location and phone number. Once this is done they can then view the lead and their location on a map. The admin user can then use the applications maps feature to filter out the sales agents who are in the same area as the lead. The can also filter sales agents by other criteria such as the sales agent's status. They can then choose to send a sales agent a push notification about the new lead in their location, see Figure 2.

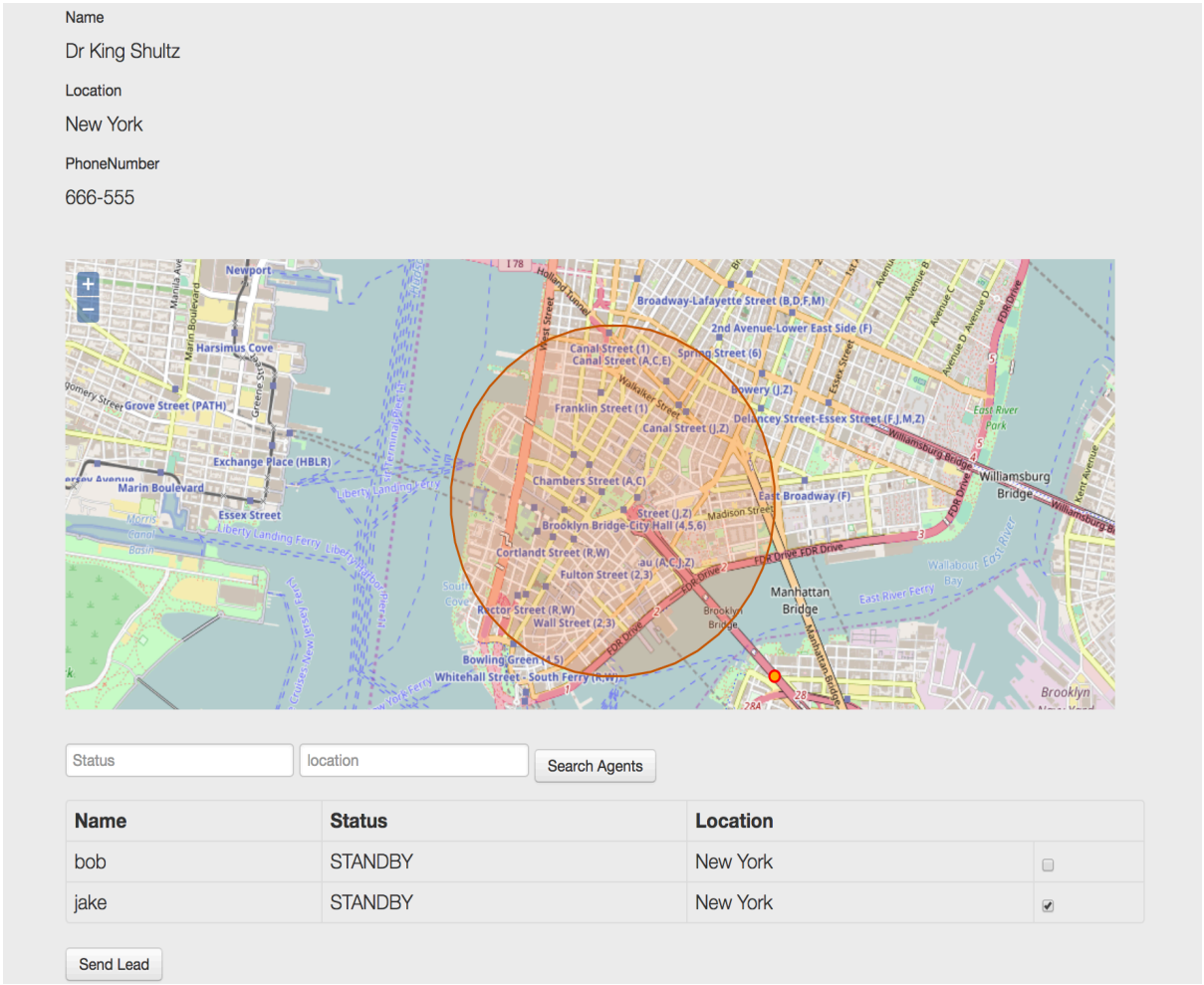


Figure 2 Sending a lead to a sales agent in the leads area

The sales agent can then access the system, while on the road, through one of the mobile client applications. When a new lead is created on the admin system, the sales agents will receive a push notification to their mobile device. It may be the case that the lead was sent to multiple sales agents in the area. Once one sales agent accepts the lead, the other sales agents will get a push notification to update them that the lead has been accepted. The lead will be removed from the admin system and from the other sales agents list of available leads.

The geolocation functionality of the admin application currently uses Hibernate search. Hibernate search provides search by geolocation and text for objects that are stored by Hibernate Object Relational Mapping (ORM) (Hibernate.org, 2016). The sales agent's location is obtained from the geo data of the client device's operating system (see section 2.2.5). The client devices communicate to the backend server using HTTP RESTful calls. The current Aerogear Aerodoc application is a monolithic Create Read Update Delete (CRUD) application which is based on Java EE API's.

The sequence diagram in Figure 3 shows the sequence of steps that an admin user of the system performs to create a new lead and send it to the sales agents.

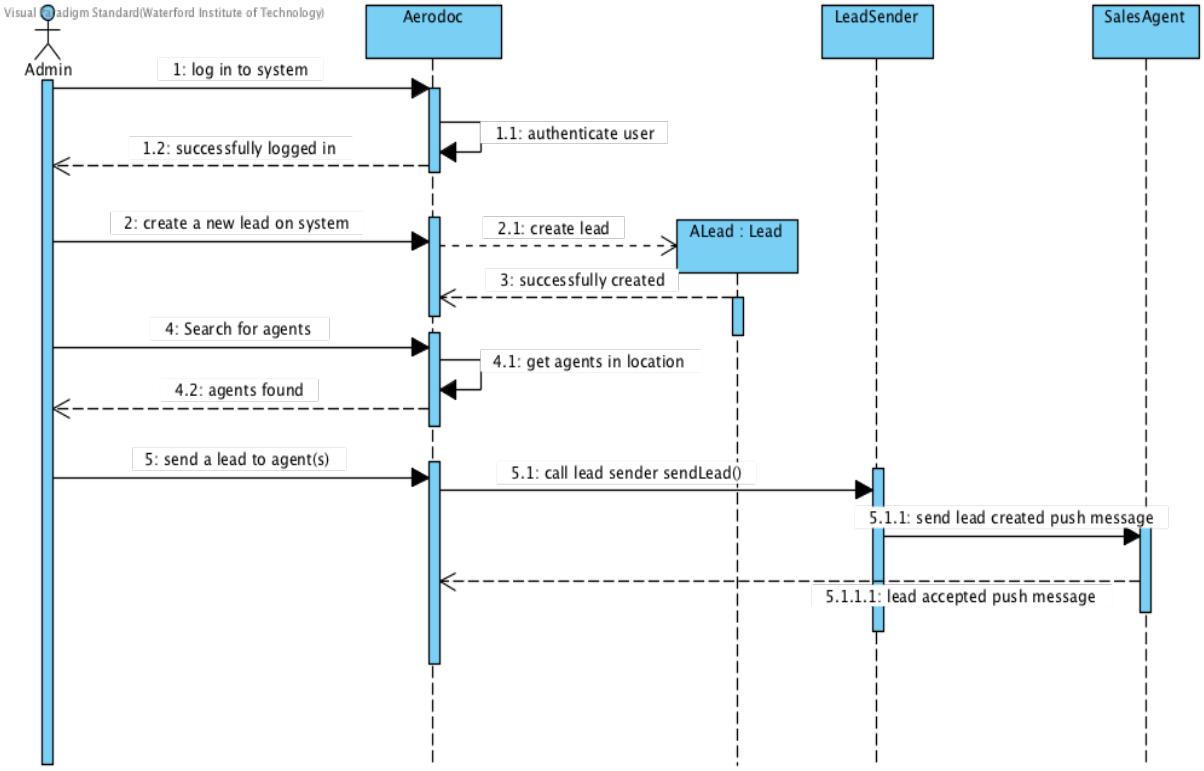


Figure 3 Aerodoc backend sequence diagram

The steps are as follows:

1. The user logs into the Aerodoc admin system.
   1.1. The system authenticates the user.
   1.2. The user is logged into the system.

2. The user chooses to create a new lead on the system.
   2.1. The system creates a new lead object from the inputted lead data.

3. The lead is created on the system.

4. The user searches for agents on the system using the search feature.
   4.1. The system retrieves all agents in the location.
   4.2. The agents are returned to the user.

5. The user chooses to send a lead to one or more agents.
   5.1. The send lead function is called.
   5.2. A push message is created and a notification is sent to the agent(s).
   5.3. The agents accept the lead and a broadcast push notification is sent to the system.

2.2.3. Aerodoc Use Case Diagram

The use case diagram in Figure 4 shows the Aerodoc systems use cases and actors.

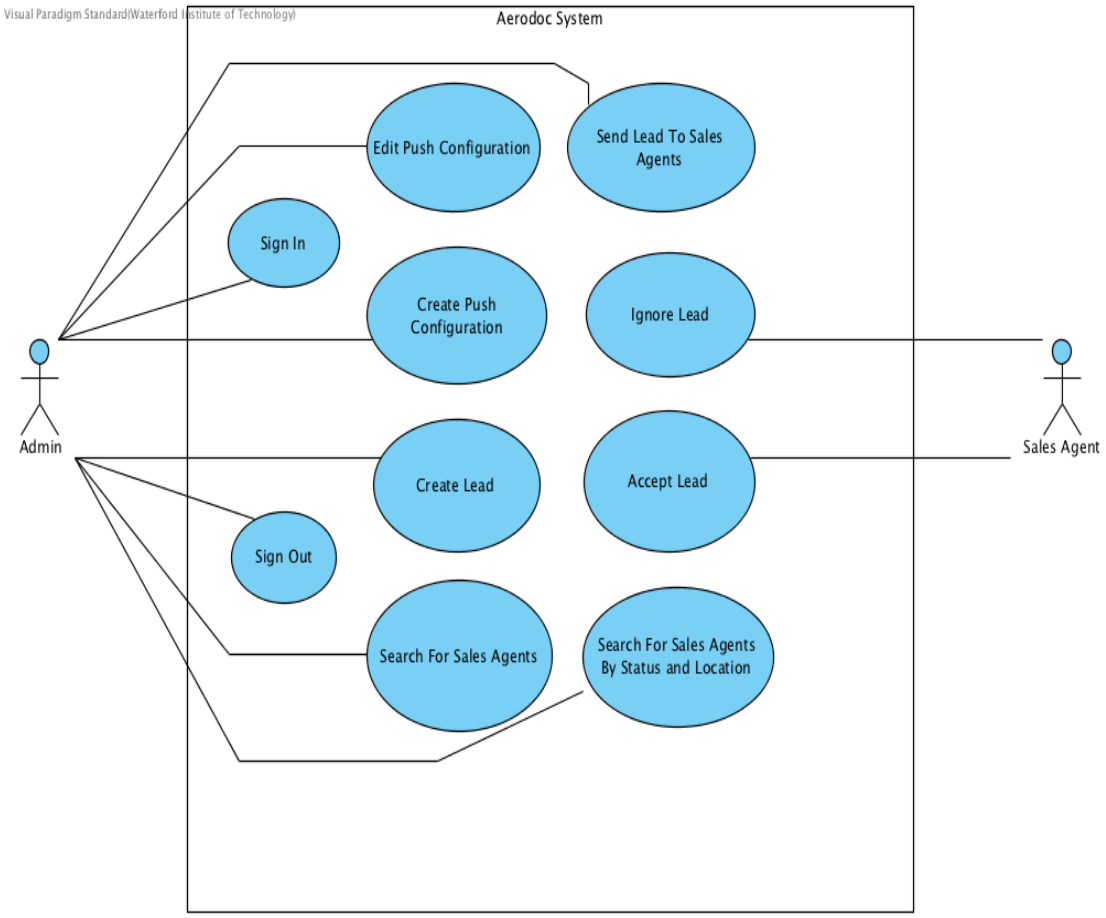


Figure 4 Aerodoc system use case diagram

The use case diagram in Figure 4 shows the following use cases:

- **Sign in –** An admin user can sign in to the system.
- **Sign Out –** An admin user can sign out of the system.
- **Create Lead –** An admin user can create a new lead on the system.
- **Create Push Configuration –** An admin user can create a push configuration on the system.
- **Edit Push Configuration –** An admin user can edit a push configuration on the system.
- **Search for Sales Agents –** An admin user can search for sales agents using a maps feature on the system.
- **Search for Sales Agents by Status and Location –** An admin user can search for sales agents by status and location on the system.
- **Send Lead to Sales Agents –** An admin user can send a lead to one or more sales agents on the system.
- **Accept Lead –** A sales agent may accept a lead that is sent to them from the system.
- **Ignore Lead –** A sales agent may ignore a lead that is sent to them from the system.

2.2.4. Push Notification Sequence Diagram

The sequence diagram in Figure 5 shows sequence of steps that occur when the aerodoc application triggers a push notification to be sent to a client device.

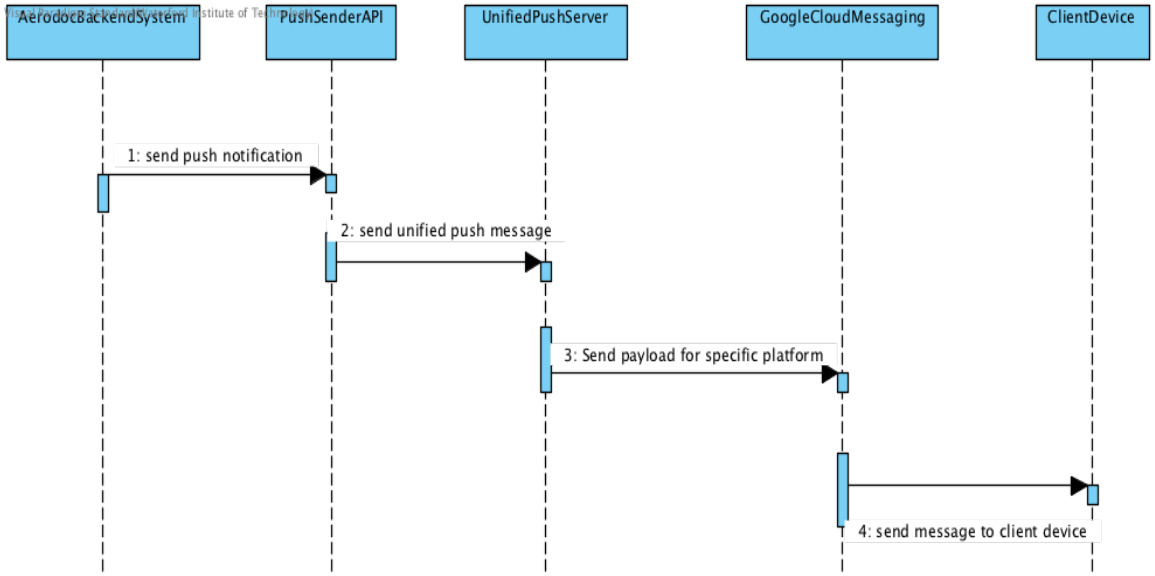


Figure 5 Sequence diagram which shows the flow of the entire system – including external components

The steps are as follows:

1. The Aerodoc system sends a push notification to the Push Sender API.
2. The Push Sender API sends a unified push message.
3. The Unified Push Server sends a platform specific payload to the Google Cloud Message platform.
4. The Google Cloud Message platform sends the message to the client device.

2.2.5. Aerodoc Clients

There are client applications for many different devices and platforms such as IOS, Android & Cordova, which use the Aerogear libraries. These are used to communicate with the Aerodoc admin application, via the Unified Push Server and HTTP RESTful calls. The client that will be used for interacting with the Aerodoc backend system for this project will be the Aerodoc Android application.

3. Monolithic System Description

This section will discuss the architecture of the current monolithic system. This will provide an overview of the components of the current system. This section will also include a UML class diagram for the monolithic system which will provide a visual representation of the Java classes for this system. The code base of the monolithic system will then be analysed using an open source technology. A critique of this system will be performed using the outputted results of this analysis.

3.1. System Architecture

The diagram in Figure 6 shows the architecture of the current Aerodoc Java system. This architecture diagram was reverse engineered using Structure101 (Structure101.com, 2016) which is a software architecture and dependency management tool. Structure101 can be used for modelling the complexity of a Java system, this architecture diagram will aid the design of the new system.

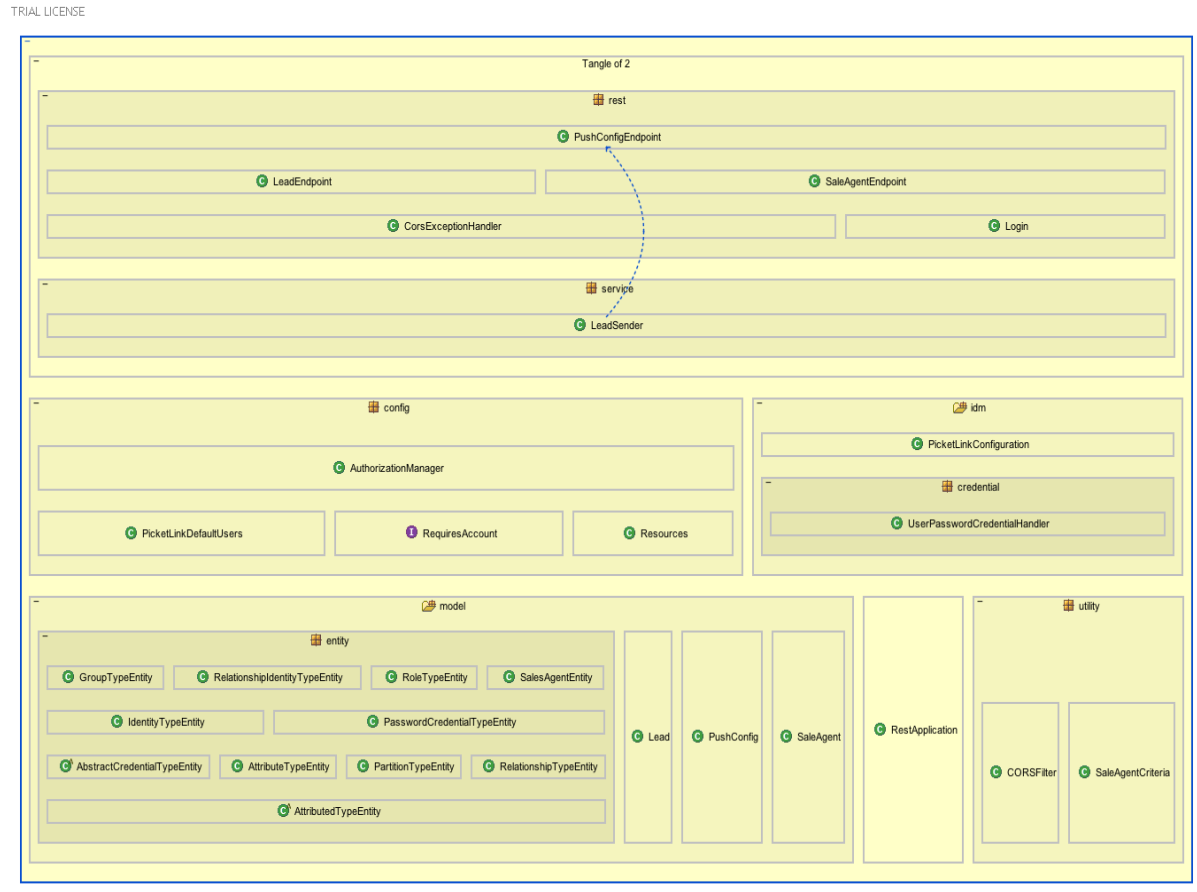


Figure 6 The current Java Architecture

3.2. Class Diagram

The diagram in Figure 7 shows the class diagram for the current monolithic Java system.

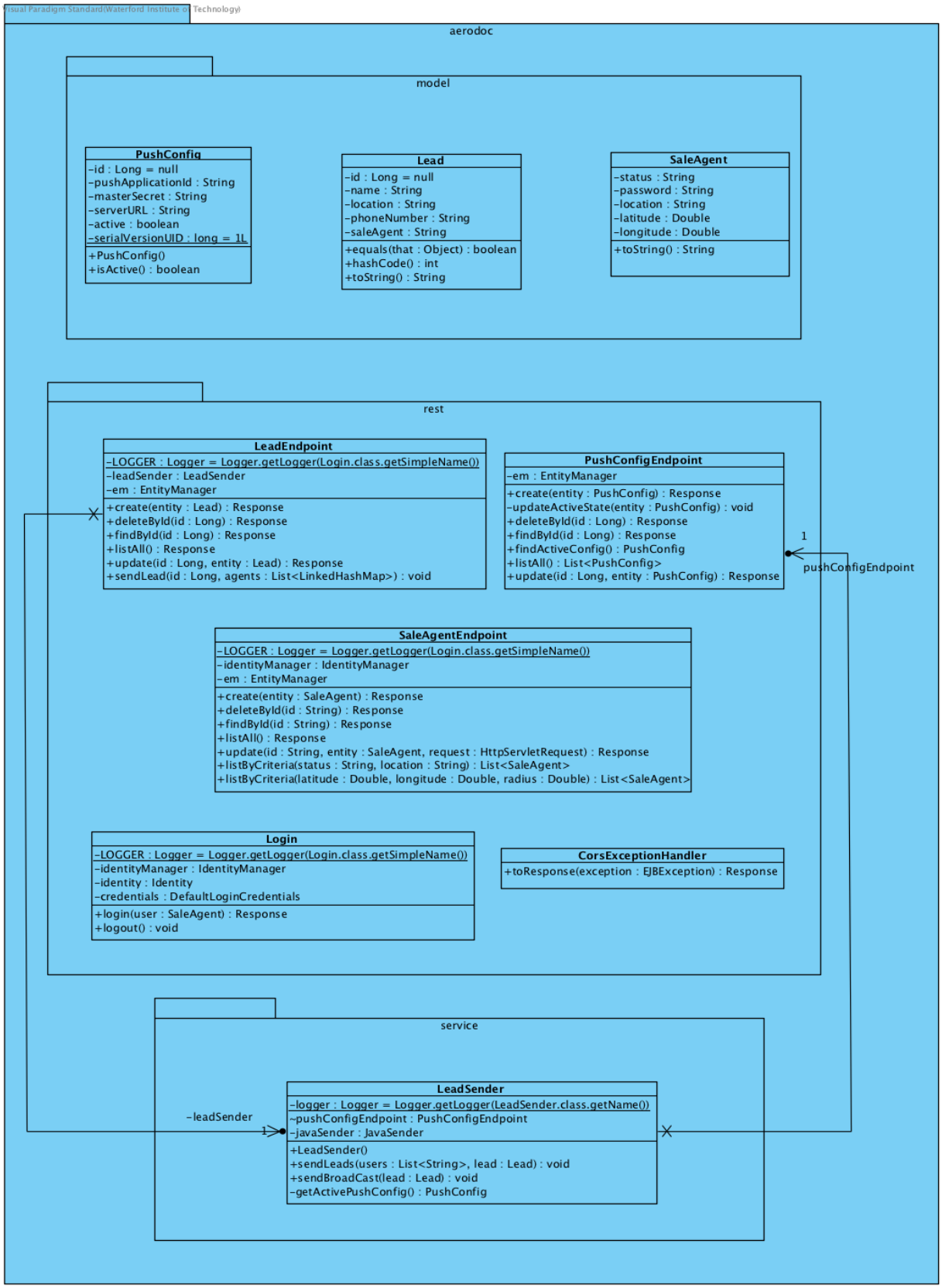


Figure 7 Aerodoc Java model class diagram

This class diagram consists of the following models:

- **Lead -** Represents a person on the system which a sales agent can visit.
- **SaleAgent** – Represents a person on the system which visits a lead.
- **PushConfig –** Represents a configuration for a UPS which can be set to active or inactive by the user. This allows the system to be configured with any UPS instance.

The class diagram consists of the following service classes:

- **LeadSender –** Class which is responsible for sending leads via push notifications.

The class diagram also consists of the following RESTful web service classes:

- **LeadEndpoint –** The RESTful endpoints for the Lead model.
- **SaleAgentEndpoint –**The RESTful endpoints for the SaleAgent model.
- **PushConfigEndpoint** – The RESTful endpoints for the PushConfig model.
- **Login** – The RESTful endpoints for logging in and out a sales agent.
- **CorsExceptionHandler –** Class which manages Cross-Origin Resource Sharing (CORS) information and REST handles exceptions that are thrown.

3.3. System Critique

This section will discuss the key metrics that can be measured when performing an analysis of the current Java monolithic system.

An analysis of the existing code base was performed using SonarQube. SonarQube is an open source technology for managing code complexity and code quality for software applications (SonarQube.org, 2016). A SonarQube analysis will provide an overview of the application in its current form. If weak points are identified in the application, this will allow them to be to architecturally considered when rewriting the application using microservices in the next phase. The diagram in Figure 8 shows the key areas that SonarQube analyses in a system.

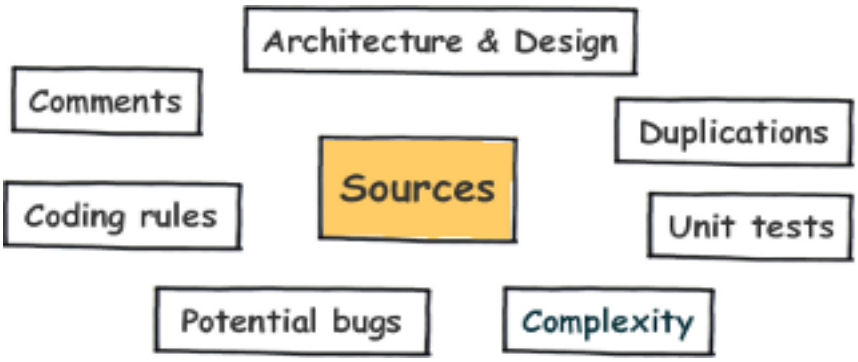


Figure 8 The areas which SonarQube helps to identify issues in (Sonarqube.org, 2016)

3.3.1. Lines of Code

This metric is the total number of line of code that exist in a code base. This is one of the key metrics that may be taken into consideration when analysing the code base of any software system. The analysis of the existing system found the total number of lines of code to be 2,016.

3.3.2. Code Complexity

Sonarqube measures code complexity by calculating the number of possible paths that can be taken through a function. The complexity increments by one, each time that the control flow of the code splits (Docs.sonarqube.org, 2017 A). The generated complexity measurement is known as the cyclomatic complexity. It is recommended that the cyclomatic complexity should not exceed a threshold of 10 (Chambers.com.au, 2017).

The code complexity metric was measured at three different levels. The output for the code complexity metric for each function was found to be 3.3. The total output for each file was found to be 9.3. The overall average code complexity metric for the monolithic system was found to be 271.

3.3.3. Technical Debt

In software development, a feature may be implemented using one of two methods. Method one consists of implementing a feature as quickly as possible, even if it may not be robust or easy to maintain. Method two consists of spending time designing a feature well, and implementing it in a way that will ensure that it is robust and easy to maintain.

The key goal of method one, is to ensure that the software just needs to work, with the intention to come back to improve this at a later date. The key goal of method two, is to get it done correctly, and is considered a permanent solution. Technical debt describes a measurement of time that is needed to be spent working on a piece of software that was implemented using method one, to a solution that was implemented using method two (Fowler, 2003).

When Sonarqube detects an issue, it also provides an estimate of the time that is needed to address this issue. Sonarqube calculates the technical debt metric by summing up the total time that is needed to address all of the issues that an analysis has picked up (Docs.sonarqube.org, 2017 B). The output for the technical debt metric was found to be 1 day and 6 hours.

3.3.4. Code Duplication

This metric details the percentage of lines of code that are duplicated in the code base. This is often a sign of inefficient coding practices. The output for the code duplication metric was found to be 3.9%. The constraints of using the Java EE framework to build the current system may be a factor in this level of code duplication.

3.3.5. Code Smell

Code smell is a term that is used to describe a piece of code that may indicate a more serious underlying issue in the code base. It may not always be the case that there are underlying issues, as it may just be weakly written code (Fowler, 2006). The output for the code smell metric was found to be a total of 97, meaning that there were 97 instances of code smell in the current system.

The results of the Java monolithic system's analysis can be found in Table 1, while an overview of the analysis can be found in Appendix B.

Metric	Result
Lines of Code	2016
Code Complexity	Function: 3.3, File: 9.3, Overall: 271
Technical Debt	1 day 6 hours
Code Duplication	3.90%
Code Smell	97 instances

Table 1 Results of monolithic system analysis

4. Microservices System Description

This section will discuss some of the key issues that need to be considered when building a microservices system.

4.1. Microservices System Architecture

A microservices architecture consists of a suite of well-defined services, where each service runs as its own process. These services may be independently deployable, and are often distributed services. This brings a whole new set of challenges when building a system using this approach.

4.1.1. Microservices Architecture Pattern Choice

As the goal of this project is to develop a microservices system functionally equivalent to the existing monolithic system, the system must appear as a single CRUD, REST application to the clients. Therefore, each service must expose a REST API that the clients can consume. The are several design patterns that may be followed when decomposing a monolithic system into a suite of services. One of these patterns is to decompose by the business capabilities of the system. It may also be decomposed by defining the system use cases and identifying the services from that point. There are other also many other patterns, but the end goal for each is the same. It is to ensure that each service has a very specific set of responsibilities, i.e. each service should be cohesive, while the overall system should be loosely coupled. To help to achieve this loose coupling, each service should also have its own database (Richardson, 2017).

The final microservices architecture pattern choice for this system will adhere to all of the key issues that were discussed in this section.

4.1.2. Methodology for Building the Microservices System

One of the main considerations for this project, was to decide what approach would be taken to build a microservices version of the current Java system. This is a topic that divides opinion among many of the top software experts. There are two methodologies that were considered for doing this. These consisted of, the microservices first methodology, and the monolithic first methodology, both of which will be discussed in this section.

4.1.2.1. Microservices First Methodology

This methodology involves developing a system, as a suite of microservices, one at a time. This approach ensures that the developer is thinking about how the system will function as a microservices system, from day one. It is often the case that a monolithic system is not suitable for splitting into a microservices system. This may be due to building a monolithic system that is too tightly coupled, which makes splitting it up an extremely difficult task. Therefore, there is an element of risk involved in choosing to develop a monolith first. This methodology would involve designing the microservices systems architecture, at the beginning of the project. This was something that would also need to be considered, when choosing a methodology (Tilkov, 2015).

4.1.2.2. Monolithic First Methodology

This methodology involves developing a monolithic version of a system, and then splitting this monolith into a suite of services. This approach ensures that the developer can first gain an understanding of the systems boundaries and functionality. This can also help to avoid making any poor key decisions, that may prove costly, when designing the microservices system. This methodology can be further divided into sub-methodologies. These provide different approaches to implementing a system, when using the monolithic first approach.

One approach involves extracting the services from the monolith one at a time, while still keeping the monolith as a main service. This process could be iterated over a number of times, until the system consists of a small monolith, and a number of well-defined services.

Another possible approach when working with an existing monolith is to develop an entirely new microservices system, while using the knowledge gained from developing the monolith. This would ultimately mean decommissioning the monolithic system, when the microservices system is complete. The knowledge gained from developing the monolith would be used to build the new microservices system, quickly and efficiently (Fowler, 2015).

4.1.2.3. Choosing a Methodology

The monolith first methodology was chosen for building the microservices system for this project. The current Aerodoc system has no developer documentation, and it is not a system that the developer of this project is familiar with. Therefore, building a functionally equivalent system using Node.js, would help develop a strong understanding of how the existing system functions. It would also be beneficial to the developer, to help to improve any existing JavaScript and Node.js skills.

Developing a microservices system is an advanced software development challenge. It is essential that the developer of such a system possesses a strong set of development skills in the chosen development language, or that they have a strong knowledge of the system. Therefore, building an equivalent Node.js monolithic system would provide an environment to improve the developer's knowledge in both of this areas. The flexibility that this methodology provides was also key factor in taking this decision.

4.1.3. Microservices Deployment

There are some key goals to be achieved when creating a deployment strategy for a microservices system. Each service should be independently deployable, and the other services must not be affected if one service fails. The services must also be deployable quickly and efficiently. Once the system is deployed, the services should be able to scale out or in, independently of the other services. Therefore, if one service is receiving a significant amount of traffic, that service should have the ability to scale out, while the other services remain the same (Indrasiri, 2016).

As the main deliverable will be a back end consisting of microservices, each of these microservices will essentially be applications that are packaged in isolated containers. The technology chosen for this was Docker, which is an open source technology for provisioning lightweight Linux containers. Docker allows the user the ability to run multiple independent containers on a single server. Docker containers also guarantee the applications which they

contain are isolated from other containers running on the same server, as well as the hosts infrastructure (Docker, 2015). The diagram in Figure 9 shows how multiple Docker containers can live on a single host.

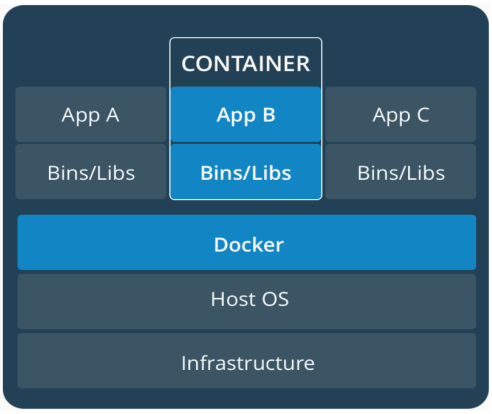


Figure 9 Docker containers (Docker, 2017)

One of the key benefits of using Docker to deploy your applications is scalability. A correctly built Docker container is ephemeral, and therefore can be built, destroyed, and rebuilt in minutes, this is a core requirement of a scalable architecture (Docker, 2016). Using Docker will help to achieve the goals outlined in this section.

The aim is to package each of the systems services in a Docker container, where they will then be deployed on a container management Platform as a Service (Paas), such as Openshift. There are two versions of Openshift, one is Openshift Enterprise, and the other is Openshift Origin. Openshift Origin is an open source upstream project for Openshift Enterprise, and is the version that will be used for deploying this project to.

Openshift is a container management PaaS that is built on top of Kubernetes. It allows users to manage containerised applications across a group of containers. Openshift allows users to manage application deployment and application scaling, amongst many other container management tasks. (Docs.openshift.org, 2017). There are multiple components that need to be configured to deploy a containerised application to Openshift. The core components that are needed for a deployment on Openshift are as follows:

- **Pod** – A pod is a group of one or more containers
- **Replication Controller** – This defines how many pods will be running in the deployment.
- **Service –** This is an end point which acts as an internal load balancer. This distributes load across one or more pods.
- **Router** – This allows for the creation of public routes which can be used to access a service.

It was discussed section 4.1.1 how each service for this system must expose a REST API to allow the various clients to access the services resources. However, the services will not be publicly accessible, and will only be reachable via an API gateway. An API gateway is essentially a server that is the only entry point to a microservices system. An API gateway is responsible for routing incoming HTTP requests to the correct service. It can also be

responsible for other tasks such as load balancing, authentication, caching etc. (Richardson, 2015). The API gateway for this project will be a Nginx server, which will act as a reverse proxy server. The main purpose of a reverse proxy server is to sit in front of one or more back end servers and route requests to the correct server, see diagram in Figure 10.

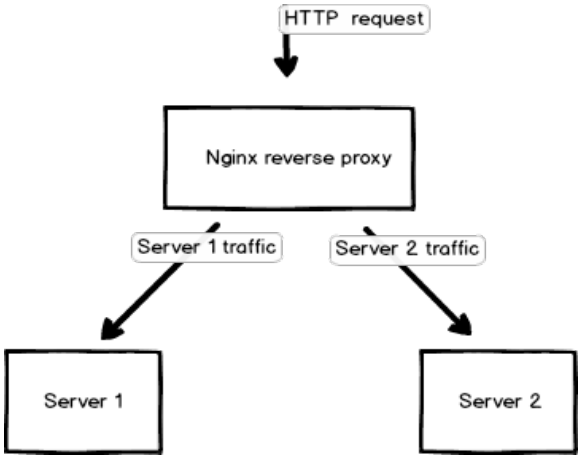


Figure 10 Nginx reverse proxy

The sole responsibility of the API gateway for this system, will be to route incoming HTTP requests to the correct service. The API gateway will run in a Docker container which will allow it to be deployed on Openshift, and managed as part of the microservices system. Openshift will be deployed on a Red Hat Linux server on the Amazon Web Services (AWS) cloud infrastructure. Using Openshift to manage the containerised microservices will help to deliver on the key goals outlined in this section.

4.1.4. Service Resilience

In a microservices system, a service may fail due to an issue that may not be affecting other services in the system. This is especially common where the services are distributed across multiple servers. This could occur because of networking issues or resources availability on a server. Therefore, it is important to design your microservices system to be fault tolerant such that, if a service fails, the rest of the system must not be affected. This can be achieved by designing a microservices architecture correctly, as described in section 4.1.1 (Indrasiri, 2016). Fault tolerance can be achieved either by implementing service replication on the system, or by implementing service redundancy.

Openshift allows for both of these methods to be applied to a deployment. Once the microservices system is deployed on Openshift, the deployment can be configured to ensure a minimum number of containers will always be running. This will ensure that in the event of a service failing, a new container will be launched immediately. This will follow the redundancy model in order to achieve a fault tolerant microservices system.

In addition to this, the deployment will be configured to scale out the number of containers, if CPU usage on a container, which runs a service exceeds a predefined threshold. This will ensure that the service remains reachable, even if the CPU resources for the container running the service are under extreme pressure. When the CPU usage for the container drops below a predefined threshold, the number of containers will revert to normal levels.

4.1.5. Service Discovery

In a microservices system, it is important for the system to know when it can begin sending traffic to a new service. The system needs to become aware that a service is now available to receive traffic. This is a concept that is known as service discovery. The system also needs to know when to stop sending traffic to a service, if it is taken off line or fails. This can be a very difficult issue to address in a microservices system. However, Openshift has a simple, yet robust mechanism that can manage this issue.

Openshift provides a simple feature which allows the user to add a label to the components that are deployed. When the user is creating a pod, a label can be applied to the pod on creation, this same label can then be applied to the service component, that is associated with that pod. The service component now knows that all pods with the same label are associated with it. If a pod that is running a containerised application for the system fails, and a new pod is launched to replace it, the service component will discover the new pod and begin sending traffic to it (Colman, 2015).

4.1.6. Microservices Logging

A system built using microservices is essentially a distributed system, which appears to be a single system to the systems clients or end users. However, this often means that the services are distributed over multiple physical servers. A more modern approach to deploying these services is to deploy each service in its own container. This can be done using a containerisation technology such as Docker, as discussed in section 4.1.3.

On a monolithic system, logging can be a reasonably manageable issue to address. However, logging for a system becomes more difficult when a system consists of multiple services that may be on different servers or in multiple Docker containers. Troubleshooting a system that only consists of a small number of services may not be too much of an issue. However, if that system scales to a large number of services, the system logs may be the only realistic way to troubleshoot an issue. A correctly implemented logging strategy is a fundamental part of designing a microservices architecture system.

4.1.6.1. Developing a Microservices Logging Strategy

This section will discuss the microservices logging strategy for this project.

4.1.6.1.1. Request Identification

In a microservices architecture system, a request may be received by a gateway service. This service may need to perform some business logic, before sending the request to another service to perform another task. In a complex microservices system, this request may need to travel through multiple services, before returning a result to the client. It may be the case, that a request is received that causes an issue in the system. It can be a difficult task to troubleshoot this issue when a system consists of multiple services. A simplified example of a situation like this can be seen in the diagram in Figure 11.

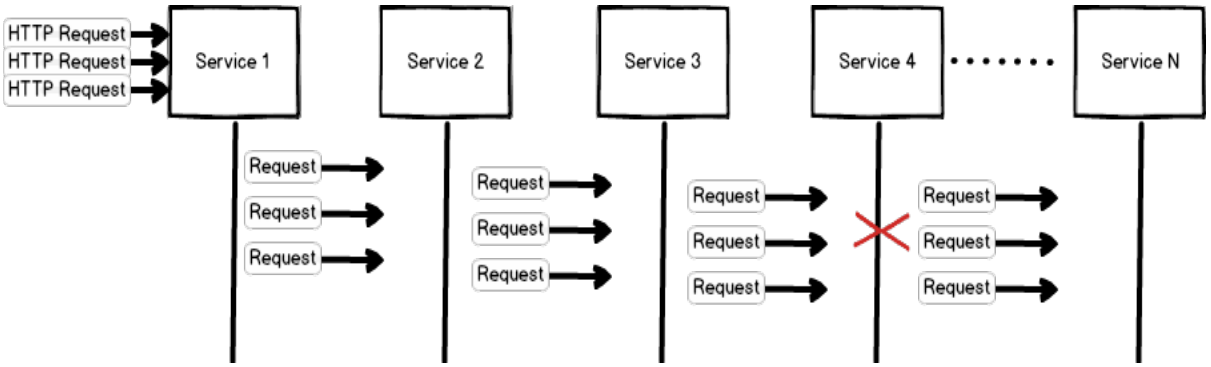


Figure 11 An issue occurring in one service

This diagram shows multiple requests travelling through a number of services, and an issue occurs at service number four. When troubleshooting the issue, it can be very difficult to determine which request caused the issue at service number four.

A solution to this problem is to introduce additional attributes to the service logs, which may help to identify the request that caused the issue. The attribute can be a request identifier, which is a unique identifier that can be generated when the request is first received from a client. It can be appended to the HTTP header, where it can then be propagated through each function or service. Figure 12 shows an example of an attribute such as this, appended to a HTTP request header.

```
_headers:
 { 'x-powered-by': 'Express',
   'x-request-id': 'ddcb7ad2-f35c-46ea-9e18-76135fb47ae4',
   'access-control-allow-origin': '*'
```

Figure 12 The header is appended to the HTTP request

This unique identifier will be used in all log entries, until the request has been completed. The diagram in Figure 13 shows how this could be implemented.

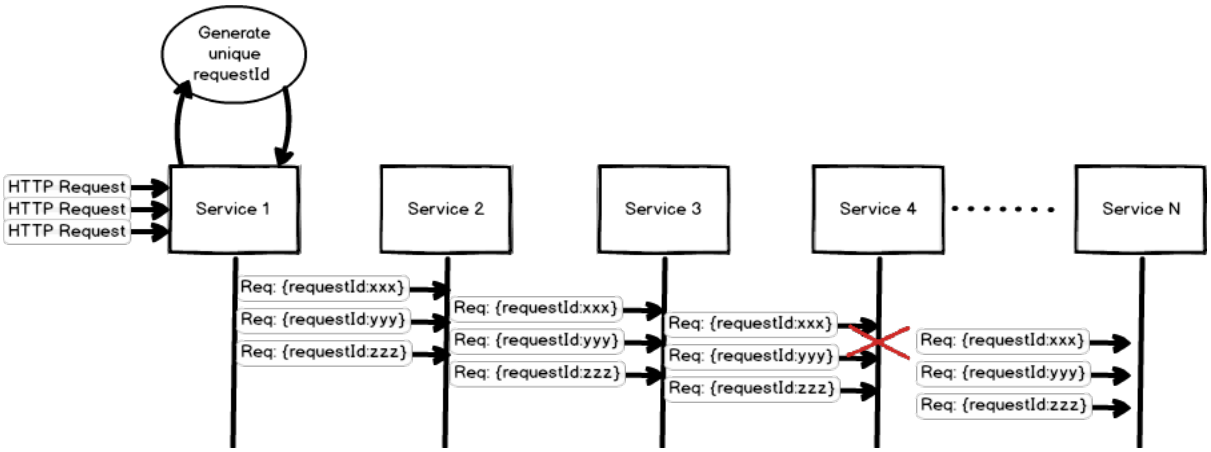


Figure 13 Generating a unique request identifier

The logs for each request will now contain this unique request identifier. When an issue occurs at service number four, the logs will contain the request identifier of the request that caused the issue. This can be traced back through each of the services and functions that have performed work on the payload of this request (Saldahna, 2016). Each service will also be tagged with an attribute which identifies that service. These additional attributes will make the troubleshooting of any issues a manageable process.

4.1.6.1.2. Microservices Log Aggregation

In a monolithic system, it is usually sufficient to send all logs to a file. This simple method ensures that the systems logs are in the one location. However, this would not be suitable for a microservices architecture based system. The services may not be on the same physical machine, or in the case of this project, the services may run in several Docker containers. As a result of this, each service would have its own log file, and it would prove very inefficient to troubleshoot this system.

A solution to this problem is to aggregate the microservices logs to a single centralised service. This allows a developer to manage the microservices logs, as if they were sent from a single monolithic system. There are many free, or open source log aggregation technologies available. The log aggregation technology that was chosen to for this project will be discussed in section 5.4.1.4.

The key issues that need to be addressed when building a microservices system have been discussed in this section. The next section will discuss the methodologies that were used throughout this projects lifecycle.

5. Project Methodologies and Technologies

This section will discuss the methodologies that were followed, when building the microservices architecture system.

5.1. Development Methodology

5.1.1. Choosing a Model

The project was developed using a form of Agile methodology, there are multiple models that follow this methodology. The main models that were considered for this project were Kanban, Extreme Programming (XP) and Scrum. After examining all three models, it was decided that the Scrum model would be best suited for this project. The role of Scrum-Master was the project supervisor, while the Product Owner was Red Hat. As this project is a fully open source project, the open source community would also be project stakeholders and potential contributors, given the nature of Open Source development. The team used JIRA (Atlassian, 2016) software to manage this project. JIRA is a project management tool for use by agile teams.

Developing software following this model involves gathering initial requirements for the project at the beginning. The requirements for this project were composed of three types of requirements. These include the requirements from the Product Owner (Red Hat) and requirements from myself, as the developer, i.e. what I want to gain from the project. The last set of requirements were from the project supervisor and included other possible solutions to the project, while also representing the requirements of Waterford Institute of Technology with respect to formal hand ups as part of the project lifecycle. Once the phase of gathering requirements was completed, the team could organise a sprint planning meeting where a backlog of items was created. These items could then be story pointed to give a measure of complexity, which later on would help inform the ordering of the backlog. This would allow the team to prioritise the items and decide what would go into each sprint of the project.

5.1.2. Why JIRA?

All development on the Red Hat Aerogear project is currently managed using JIRA software, therefore it is beneficial to both Red Hat and the developer, that JIRA be used to manage this project. Using JIRA would allow the complexity of the system to be examined, and broken up into tasks and subtasks, this would help to achieve efficient time management. The JIRA project board can be found in Appendix A.

5.2. Testing Methodology

5.2.1. Behaviour Driven Development (BDD)

The methodology which was followed for the testing of this system is the Behaviour Driven Development (BDD) methodology. BDD has evolved from Test Driven Development (TDD) which is was a fundamental part of the agile methodology. The naming convention is much more intuitive and explains what the test should do, as opposed to just putting the word “test”

in the testing function's name. The term "specification" or "scenario" is preferred to the traditional name of "test". The team will often refer to the specifications of the behaviour of a feature, instead of referring to the tests for a given feature. BDD also encourages better communication between the developers and the testing team (Agilealliance.org, 2016). BDD scenarios also map closely with User Stories providing an additional means to communicate requirements and acceptance criteria with the customer.

5.2.2. Continuous Integration

To ensure that the code is robust and of deliverable quality, a Jenkins continuous integration build server would be used. Jenkins is an open source continuous integration technology, which allows for the continuous testing and integration of a software application (Wiki.jenkins-ci.org, 2016). This is hosted on a Ubuntu 14.04 Linux server on Amazon Web Services (AWS). The Jenkins server is configured to run a build each time it detects that a change has been made to the GitHub project repository. This build consists of running the existing integration tests, and any new integration tests that may have been added with the changes. If a build fails, the server is configured to email the developer with the details of the failed build. This would allow any bugs that arise to be fixed quickly and the changes committed. The diagram in Figure 14 shows the continuous integration set-up for developing on this project.

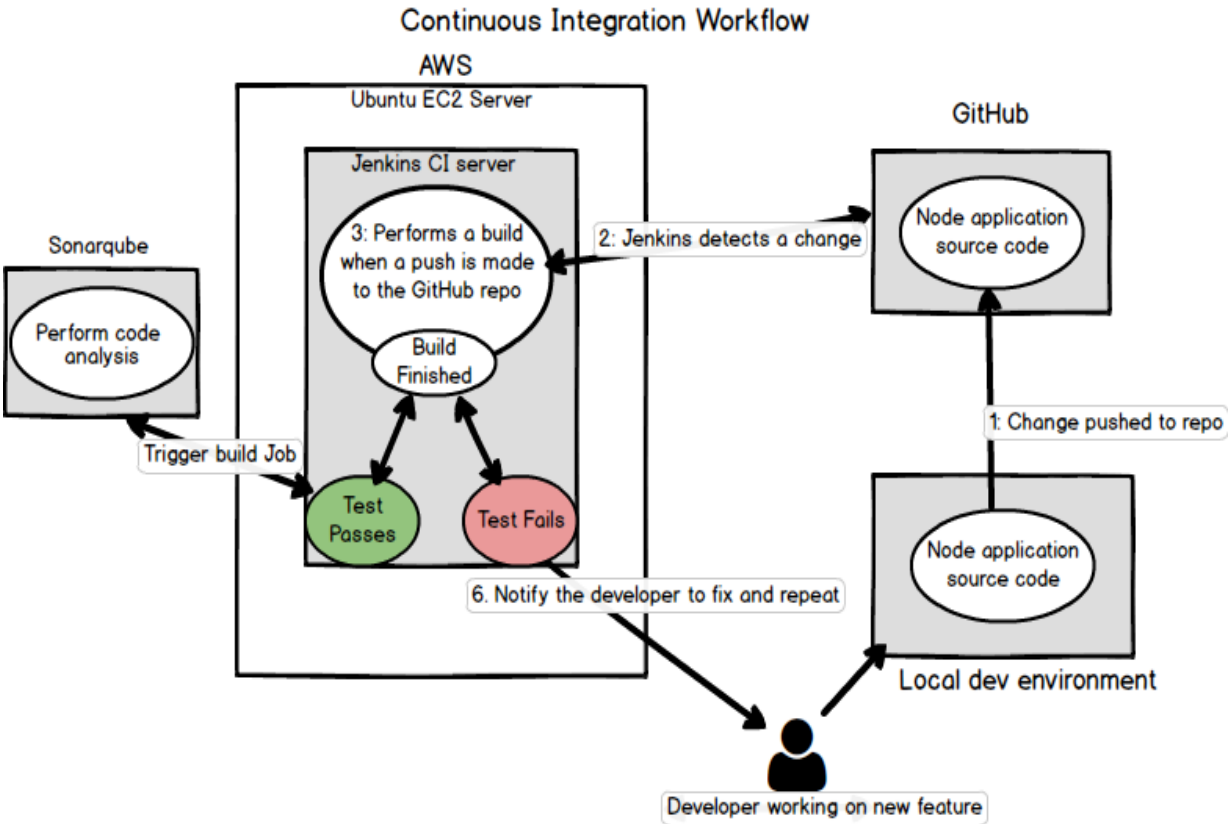


Figure 14 Continuous Integration Setup

5.3. Code Quality Management

SonarQube was used for code quality management on this project. It was used to analyse the code base as new features are added to the project. Note that the previous diagram shows that SonarQube is part of the continuous integration process. SonarQube was integrated with Jenkins continuous integration pipeline to automatically perform a code analysis if the systems integration tests pass.

5.4. Technologies Used

This purpose of this section is to discuss the possible software and hardware technologies that were chosen to build this system. Each technology examined was done so with consideration to its suitability to a system designed using a microservices architecture. The alternatives that were considered will be discussed, along with the reasons for deciding on the chosen technologies.

5.4.1. Software

5.4.1.1. Development Language

There were several languages that were considered for rebuilding the backend for this system using microservices. Considering that the existing monolithic backend was built using Java, this was the first language that was considered. However, after researching which language was most suitable for a microservices architecture, the decision was taken to use JavaScript and Node.js. Node.js is a runtime environment, this was built on Google Chrome's V8 JavaScript engine (Nodejs.org, 2016 A). JavaScript is a single threaded, asynchronous language and is one of the fastest growing programming languages, therefore learning it is also an opportunity to keep abreast with industry.

Node.js was designed to build event driven and scalable network systems, and given the event-driven nature of this system, it was the correct choice (Nodejs.org, 2016 B). There have been some studies which compare the performance of a Node.js backend system against the same system built with Java. One such study was conducted by IBM, this case study featured a REST API that could be consumed by other applications – a scenario which was similar to the one that will be implemented in this project. The Node.js system outperformed the Java system in all of the tests and it was built in 40% of the time it had taken to build the Java equivalent (Ibm.com, 2013). This case study helped me to justify my decision to use Node.js.

5.4.1.2. Framework

Using a Node.js framework allows the developer to get a server running in a relatively short time, this was important to me due to the time constraints of the project. The frameworks that were considered for the project were Hapi (Hapijs.com, 2016), Express (Expressjs.com, 2016) and Koa (Koajs.com, 2016). Hapi and Express were both found to be suitable for building the system. Koa was also considered, as it provides support for next generation JavaScript features, an area of interest to me. However, the product owner (Red Hat) had a specific recommendation that Express be used to give us more opportunities for extensions within the system. This was due to ensuring the system can be easily integrated with other open source authentication systems such as KeyCloak (Keycloak.org, 2016), which currently only supports Express.

5.4.1.3. Logging Technology

There are many open source technologies that can be used for system logging. The logging technology that was chosen for this project is Winston. Winston is a multi-transport logging library that was designed for application logging in asynchronous systems. A transport is the method of storage that will be used for the systems logs. Winston decouples the storage of logs, from the gathering of logs. The developer can configure a Winston transport to send system logs to a file, a database or any third party logging service (Johnson, 2016).

5.4.1.4. Log Aggregation

The log aggregation technology that was chosen for this project is Loggly. Loggly is a feature-rich, enterprise grade, log management Software as a Service (SaaS). It is free for applications where log traffic is under a certain threshold. As Aerodoc is a demonstration application that was built for showcasing the Aerogear UPS's capabilities, the logging traffic is quite low for this system. Therefore, Loggly can be justified as a permanent log aggregation solution for the microservices system (Loggly, 2017).

One of the main reasons for choosing Loggly, was due to its integration with open source logging technologies such as Winston.js. The Loggly team have developed an open source Node.js package, which acts as a client for Loggly. The developer can configure a Node.js application to send all application logs to the Loggly service using a unique token. Each service will also be tagged with an attribute that will identify which service the logging has been received from. The Loggly web console can be used to monitor the logs of the overall system. It provides customisable search functionality, which allows the user to search by service name, in a time period or using regex. Using Winston with Loggly will provide a robust logging solution for a microservices system.

It must be noted that one of the key benefits of using Winston, is the ability to only have to make a minor change in the code base to change the log aggregation technology.

5.4.1.5. Database

The suitability of using a relational database such as MySQL was examined, but research pointed at using a NoSQL database such as MongoDb when working with Node.js. MongoDB was the chosen database technology, as it has the ability to adapt and change quickly along with the ever changing and evolving requirements of an Agile project (MongoDB, 2016). The main aim of this project is to have a functionally equivalent backend, so the database choice would not impact this much, but MongoDB complements Node.js and therefore justified my decision.

5.4.1.6. Version Control System

The version control system that was used for this project is GitHub. The Red Hat Aerogear team also use GitHub for version control of their projects, this was another reason for choosing GitHub. All GitHub repository links for this project can be found in Appendix A.

5.4.1.7. Testing

The main testing frameworks that were considered were Mocha (Mochajs.org, 2016) and Jasmine (Jasmine.github.io, 2016). The suitability of both was examined, and Mocha was chosen. Although it is slightly more complex than Jasmine, it provides more flexibility when it comes to picking libraries to use with it (Marco Franssen, 2015). The assertion library that was chosen was Chai (Chaijs.com, 2016). Chai provides multiple assertion styles that integrate well with modern testing methodologies.

5.4.2. External Software Components

In order to have a fully functional development environment setup, the following software components also needed to be configured:

- **Aerogear Unified Push Server (UPS**) – Push notification integration.
- **Aerodoc Client** – Android application.

5.4.3. Other Development Tools Used

- **Red Hat JBoss Wildfly 10 Application Server** – for serving up existing monolithic system.
- **Maven** – for building existing monolithic system.
- **Android Studio** – To build the client device to test functionality.
- **SonarQube** – used to analyse the characteristics of the code base, in its current form and later after the microservices rebuild using Node.js.
- **NPM** – Node.js package management.
- **Mongoose** – MongoDB object data modelling.
- **Morgan** – HTTP request logging middleware.
- **Postman** – API testing.

6. Microservices System Implementation

This section will discuss the final microservices system implementation, including describing the final architecture. It will describe each of the components that exist in the microservices system before performing an analysis of the new systems code base.

6.1. System Architecture

It was discussed in section 4.1.2.3 how a monolithic first approach would be taken to building this system. This would help to improve the developer's knowledge of the existing Java system. As this project had very strict requirements, development on the Node.js monolith involved very strict user testing. It was necessary to test every feature, using both the Angular web client and the Android client. This was a relatively slow process, but it provided the developer a lot of exposure to every detail of the overall systems workflow.

During the development of the Node.js monolithic system, the developer began to identify what the microservices for the evolved system would be. The models for the monolithic system consisted of a model for a Sales Agent, a model for a Lead, and a model for a Push Configuration. Each of these models had full CRUD functionality via RESTful API endpoints. Each model was analysed, to decide if all functionality and CRUD behaviour for each one, could be extracted to a cohesive microservice. The full functionality for each potential microservice was analysed, to verify that it would be independently deployable. Each potential microservice was also analysed, to decide if the other microservices would be affected, if the service in question failed. Lastly, the overall planned microservices system was analysed to verify that system would be loosely coupled. The final microservices architecture can be seen in Figure 15. The diagram shows a set of well-defined, cohesive, independently deployable, and loosely coupled microservices which sit in behind a Nginx API gateway server. Each of these services will be packaged in a Docker container, and will communicate with their own MongoDB database.

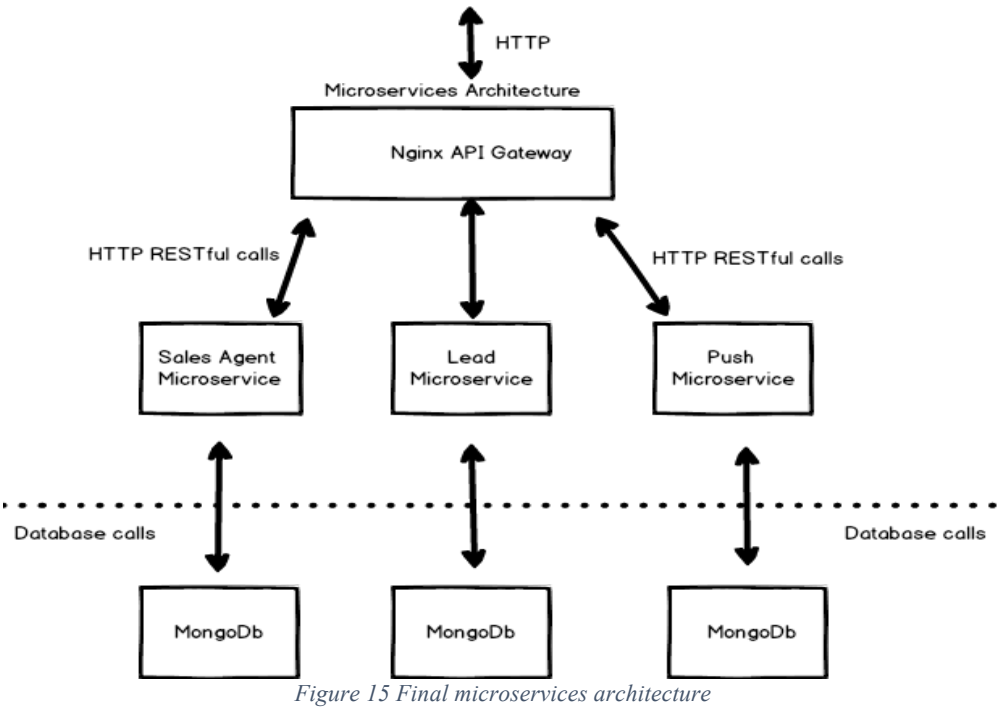


Figure 15 Final microservices architecture

6.2. System Components

The components of the microservices system are described in this section.

6.2.1. Lead Microservice

This microservice consisted of the following components:

CRUD RESTful API – A RESTful API encapsulates all functionality for the lead model. The core responsibility of this API is UPS integration, which is achieved via the Aerogear Node.js unified push sender library. This UPS integration consists of sending leads to the sales agents via push notifications. It also consists of sending broadcast push notifications to all sales agents, when a sales agent accepts a lead. This component also encapsulates full CRUD behaviour for the lead model.

Integration Tests – The lead service has its own suite of integration tests which test the API endpoints of the microservice. The tests are run independent of any other microservice.

Microservice API Documentation – The lead microservice has its own set of API developer documentation. This documentation is fully interactive and acts as a client to the microservices REST API. It can be found at: "SERVER_URL: PORT/aerodoc/lead-service/docs". This documentation will be discussed in section 6.6.1.

6.2.2. Sales Agent Microservice

CRUD RESTful API – A RESTful API encapsulates all functionality for sales agent model. The core responsibility of this API is to accept geolocation information from the Angular client application. This information will include a latitude/longitude pair, and a radius. This information will be used by a RESTful endpoint to determine if there are any sales agents within the radius, from a location that is determined from the latitude/longitude pair. This component facilitates the searching and filtering of sales agents, by two criteria and also encapsulates full CRUD behaviour for the sales agent model.

Integration Tests – The sales agent service has its own suite of integration tests which test the API endpoints of the microservice. The tests are run independent of any other microservice.

Microservice API Documentation – The sales agent microservice has its own set of API developer documentation. This documentation is fully interactive and acts as a client to the microservices REST API. It can be found at: "SERVER_URL: PORT/aerodoc/sales-agent-service/docs". This documentation will be discussed in section 6.6.1.

6.2.3. Push Configuration Microservice

CRUD RESTful API – A RESTful API encapsulates all functionality for push configuration model. The core responsibility of this API is to provide full CRUD functionality for the push configuration service.

Integration Tests – The push configuration service has its own suite of integration tests which test the API endpoints of the microservice. The tests are run independent of any other microservice.

Microservice API Documentation – The push configuration microservice has its own set of API developer documentation. This documentation is fully interactive and acts as a client to the microservices REST API. It can be found at: "SERVER_URL: PORT/aerodoc/push-configuration-service/docs". This documentation will be discussed in section 6.6.1.

6.2.4. Nginx API Gateway Container

This component consists of a Docker container which runs a Nginx server. An open source implementation of a Nginx server running in a Docker container was used for this project, and can be found in Appendix A. The responsibility of this Nginx server is to act as a reverse proxy which will forward HTTP requests to the correct service, as defined in the Nginx configuration file.

6.2.5. MongoDB Container

This component consists of a single MongoDB instance running in a Docker container. Each of the system microservices will have its own database on this MongoDB instance.

6.3. System Critique

This section will discuss an analysis of the microservice system. As the system is now composed of several microservices. These microservices are essentially stand-alone applications. Therefore, the metrics for each microservice can be analysed individually using Sonarqube. This characteristic alone highlights a benefit of using microservices to build a system. Each service can now be analysed as a small application. It can also be tested as small application, as each service now has its own set of integration tests. Each service now also has its own set of API documentation.

This output of the analysis for each of the metrics, for each of the microservices will be combined. The combined total for each metric will be compared to the total for the monolithic Java system. This will allow for a like for like comparison of each version of the system.

6.3.1. Lead Microservice

The results of the lead microservice analysis can be found in Table 2.

Metric	Result
Lines of Code	239
Code Complexity	Function: 2.2, File: 4.5, Overall: 27
Technical Debt	0
Code Duplication	0.00%
Code Smell	0

Table 2 Results of lead microservice analysis

The table shows the lines of code metric was found to be 239 for the lead microservice. This output is a low total for this metric, and even before combining it with the same metric from the other microservices, it is clear that this is a very small code base for what is essentially a small application. The code complexity metric for the lead microservice has three outputs. The average complexity for each function was found to be 2.2. The average complexity for each file was found to be 4.5. The overall average code complexity for this microservice was found to be a total of 27. The technical debt metric for the lead microservice was found to be a total of 0. This output shows that there have been no poor development practices which may have incurred a level of technical debt while developing this microservice. The code duplication and the code smell metrics for the lead microservice, were both found to be a total of 0.

An overview of the full lead microservice analysis can be found in Appendix C.

6.3.2. Sales Agent Microservice

The results of the sales agent microservice analysis can be found in Table 3.

Metric	Result
Lines of Code	303
Code Complexity	Function: 2.3, File: 6.2, Overall: 37
Technical Debt	0
Code Duplication	0.00%
Code Smell	0

Table 3 Results of sales agent microservice analysis

The table shows the lines of code metric was found to be 303 for the sales agent microservice. The benefits of a low output for this metric have been discussed in section 6.3.1. The code complexity metric for the sales agent microservice has three outputs. The average complexity for each function was found to be 2.3. The average complexity for each file was found to be 6.2. The overall average code complexity for this microservice was found to be a total of 37. The technical debt, code duplication and the code smell metrics for the sales agent microservice, were each found to be a total of 0. The benefits of low results for each of these metrics have been previously discussed in section 6.3.1.

An overview of the full sales agent microservice analysis can be found in Appendix D.

6.3.3. Push Configuration Microservice

The results of the push configuration microservice analysis can be found in Table 4.

Metric	Result
Lines of Code	218
Code Complexity	Function: 2.2, File: 6.0, Overall: 30
Technical Debt	0
Code Duplication	0.00%
Code Smell	0

Table 4 Results of push configuration microservice analysis

The table shows the lines of code metric was found to be 218 for the push configuration microservice. This output is a similarly low total for this metric, as with the other microservices. The benefits of a low output for this metric have been discussed in section 6.3.1. The code complexity metric for the push configuration microservice has three outputs. The average complexity for each function was found to be 2.2. The average complexity for each file was found to be 6.0. The overall average code complexity for this microservice was found to be a total of 30. The technical debt, code duplication and the code smell metrics for the push configuration microservice, were each found to be a total of 0. The benefits of low results for each of these metrics have been previously discussed in section 6.3.1.

An overview of the full push configuration microservice analysis can be found in Appendix E.

6.3.4. Final System Metrics

Table 5 shows the final metrics for each of the systems microservices, along with the total for the overall microservices system.

Analysed System	Lines of Code	Code Complexity	Technical Debt (Days)	Code Duplication	Code Smells
Lead Microservice	239	27	0	0	0
Sales Agent Microservice	303	37	0	0	0
Push Configuration Microservice	218	30	0	0	0
Total Node.js Microservices System	**760**	**31.33333333**	**0**	**0%**	**0**

Table 5 Analysis results for all systems

The results for the microservices systems analysis will be discussed further in section 7.

6.4. Development Sprints

This section describes the work that was completed during each development sprint for the project in semester two. A sprint planning meeting was attended by the developer and the Scrum team before each sprint began. The backlog was examined and each of the remaining items were prioritised at each sprint planning meeting. A sprint retrospective meeting was held at the end of each sprint. These meetings would help to identify what could have been done better, and what was done well in the previous sprint. This section does not discuss the development work that was completed in semester one. This work involved completing three development sprints. They mainly consisted of setting up each of the existing systems components, and the development required to build a Node.js prototype of the system.

6.4.1. Sprint One

6.4.1.1. Sprint Planning

The goal of this sprint was a Node.js monolithic backend, which would be functionality equivalent to the Java version. The output of this sprint could then be analysed, to identify the most suitable approach to re-architecting it using microservices.

6.4.1.2. Sprint Review

During the course of this sprint there were some issues encountered. The main issue faced was in relation to the geolocation feature of the system. The work to implement this geolocation feature was encapsulated in the ticket AGPUSH-1928. However, an issue was discovered during the period of initial setup to work on this ticket. It was discovered that the corresponding functionality for the Android client was recently removed from the Android application. This was due to an incompatibility between the Android application, and the latest messaging technology which the application was recently upgraded to. The Aerogear Android team were consulted about the missing feature, and they confirmed it had been removed. A JIRA feature request AGDROID-599 was subsequently created to address this issue.

Aerogear consists of multiple teams of developers, who are currently working on improving and maintaining multiple projects. Therefore, it was unknown at this point if this feature request would be prioritised by the Android team inside the timeframe of this college project. The Aerodoc geolocation feature JIRA ticket was de-prioritised from this sprint, as the discovered issue was deemed to be a blocker. The term blocker describes an issue that is preventing some work being done. This issue highlighted some of the difficulties that may be faced when working on an open source project. The other issues that were faced in this sprint were minor and were overcome by the developer communicating with the team or the open source community.

The sprint items that were completed can be seen in Figure 16. This sprint was scheduled to last two weeks and was completed on schedule.

Status Report

* Issue added to sprint after start time

Completed Issues

View in Issue Navigator

Key	Summary	Issue Type	Priority	Status	Story Points (27 → 29)
AGPUSH-1916	Create an endpoint to update a lead on the system.	Task	Major	RESOLVED	3
AGPUSH-1917	Create an endpoint to send a lead on the system to one or more sales agents	Task	Major	RESOLVED	3
AGPUSH-1927	Create an endpoint for searching for all agents based on two criteria.	Task	Major	RESOLVED	2 → 3
AGPUSH-1945	Configure existing angular application to talk to new API	Task	Major	RESOLVED	2
AGPUSH-2006	Create an end point for login to application	Task	Major	RESOLVED	2 → 3
AGPUSH-2008 *	Implement Swagger for documenting API	Task	Major	RESOLVED	2
AGPUSH-2010 *	Add functionality to pull the active push configuration and use it for sending push	Task	Major	RESOLVED	3
AGPUSH-2012 *	Modify sales agent model to have id field other than mongo id	Task	Major	RESOLVED	2
AGPUSH-2013 *	Update tests failing after model updates	Task	Major	RESOLVED	3
AGPUSH-2014 *	Create documentation for new endpoints	Task	Major	RESOLVED	2
AGPUSH-2016 *	create script to load up sales agents	Task	Major	RESOLVED	3

Figure 16 Sprint one JIRA items

6.4.1.3. Sprint Retrospective

The output of the sprint retrospective meeting for this sprint can be seen in Appendix F.

6.4.2. Sprint Two

6.4.2.1. Sprint Planning

It was decided at this sprint planning meeting that a version would be created on JIRA for the microservices implementation. The developer, with the help of the team, could then examine all of the items in the backlog that would need to be completed to release this version. This would allow the version progress to be tracked on JIRA. JIRA also displays a predicted completion date for a particular version. JIRA calculates this date using statistics gathered from the velocity of the work completed since the time that the version was created (Confluence.atlassian.com, 2017).

The main goal of this sprint was to identify a suitable approach for extracting microservices from the monolith. This would involve a number of JIRA items which are known as “spikes”. A spike is where a developer will perform a predefined amount of work on something, that may be an unknown in a project. This often relates to a feature which will be implemented in a later sprint. It often consists of a technology or concept, that is new to the developer or the development team.

To perform a spike, a period of time is agreed upon by the team, and the time spent working on the spike must not exceed this predefined time. Once this time period has passed, the developer will have some artefact as a result of work done on the spike. The team will then analyse this artefact, to determine a definitive approach to work on the feature that depended on this spike. It is vital to perform spikes as early as possible to eliminate any unknowns in the project, and reduce the risk of a technology or a feature not integrating well with the system.

6.4.2.2. Sprint Review

During the course of this sprint there were no issues encountered that the developer could not overcome. This was likely due the sprint consisting of multiple JIRA items that were exploratory and not definitive features. The sprint items that were completed can be seen in Figure 17. This sprint was scheduled to last two weeks and was completed on schedule.

Status Report

Completed Issues View in Issue Navigator

Key	Summary	Issue Type	Priority	Status	Story Points (17)
AGPUSH-1951	Spike - Investigate the best approach to break up the system into microservices	Task	Major	RESOLVED	2
AGPUSH-2022	Deploy Openshift on AWS for microservices	Task	Major	RESOLVED	2
AGPUSH-2023	Spike - Test out some logging approaches to find one suitable for distributed logging	Task	Major	RESOLVED	3
AGPUSH-2024	Extract the Push configuration CRUD functionality as a microservice	Task	Major	RESOLVED	4
AGPUSH-2034	Spike - deploy an application on openshift and explore how to set it up	Task	Major	RESOLVED	3
AGPUSH-2036	Document sprint one for final report	Task	Major	RESOLVED	3

Figure 17 Sprint two JIRA items

The version release report at the end of this sprint can be seen in Figure 18. It shows a predicted completion date of 18th of April. Although this was inside the college project deadline date, it was planned that this version would be completed sooner, to allow for any unforeseen issues that may potentially arise. This issue was to be discussed in the sprint retrospective meeting.

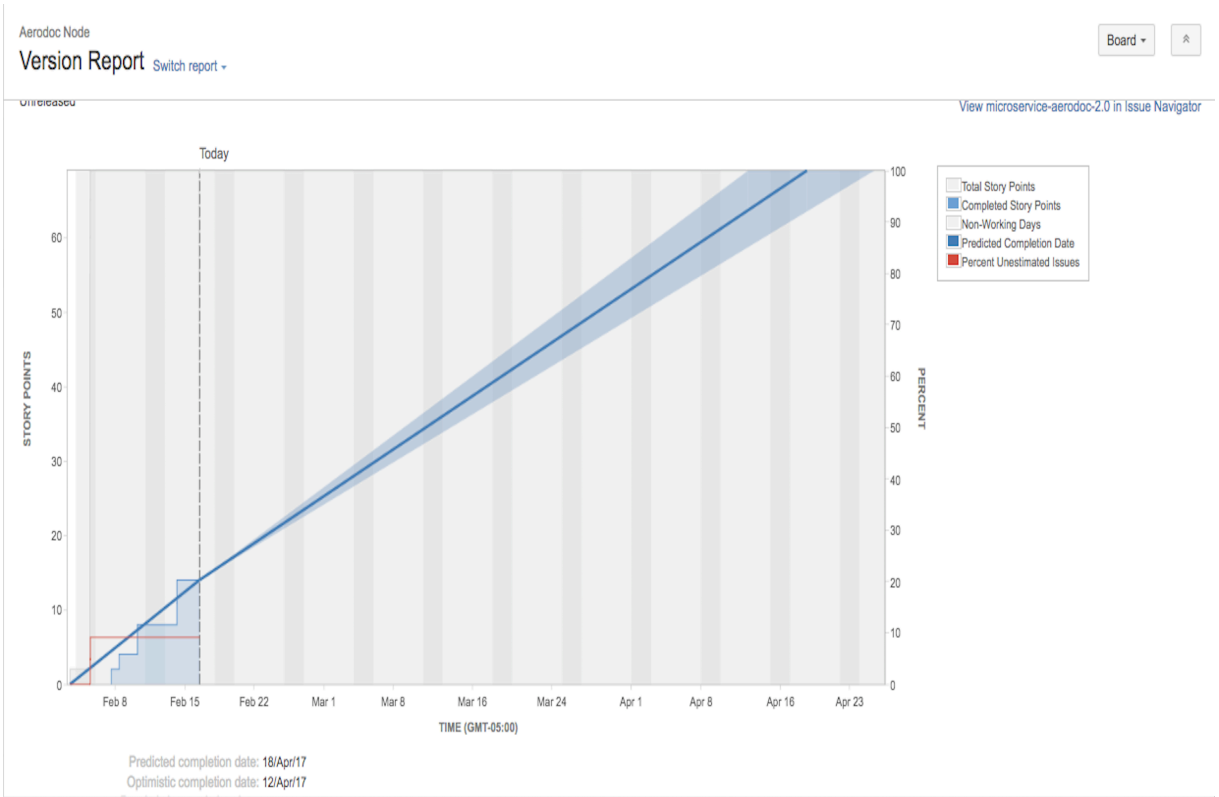


Figure 18 Version report after sprint two

6.4.2.3. Sprint Retrospective

A sprint retrospective meeting was held at the end of the sprint. The output of this meeting can be seen in Appendix G.

6.4.3. Sprint Three

6.4.3.1. Sprint Planning

This sprint was to consist of more story points than any of the previous sprints. This was due to action that was to be taken, as a result of the previous sprint retrospective meeting. This sprint coincided with a week-long college professional development period. This would allow the developer more time to work on the project than on a normal college week. Therefore, it made sense to take on extra work at this time, to bring the release date for the microservices version forward. Another factor which contributed to the developer taking on a large number of story points, was due to the number of potential unknowns which were removed by completing several spikes in the previous sprint.

The end goal for this sprint was to have all of the systems microservices deployed on the Openshift container management platform. However, the microservices would not yet be publicly accessible as a single system at the end of this sprint. The tasks for this sprint addressed many of the core issues that need to be solved when splitting a monolithic system into a microservices one.

To reach the desired end goal, each microservice would need to be extracted from the monolith and containerised using Docker. The logging for the microservices was also to be implemented in this sprint. This included the logging functionality itself, and the aggregation of the microservices logs to a centralised location. The sprint also included a spike to investigate API gateways for a microservices system. This was a new concept to the developer, and it was important to remove any unknowns in this sprint to allow for its implementation in the next sprint.

6.4.3.2. Sprint Review

There were some issues surrounding the JIRA item to aggregate the microservices logs in this sprint. It was discovered that the chosen technology for log aggregation was not as suitable as first thought. It was only when the microservices were running for a prolonged period of time that this came to light. It was not something that would have come to light in a spike. After some initial troubleshooting and contact with the technology developers, it was decided that this technology would be dropped. This decision was taken due to the time constraints for this project. The technology that was to be used for microservices log aggregation was changed in this sprint. The log aggregation technology choice discussed in section 5.4.1.4, details the backup technology that was eventually used.

This backup technology was tested during the spike in the previous sprint. This was a good example, that highlighted the benefits of performing spikes. It ensured that a backup technology was previously analysed, which allowed for it to be implemented at short notice, thus resolving the issue. The sprint items that were completed can be seen in Figure 19. This sprint was scheduled to last two weeks and was completed on schedule.

Status Report

* Issue added to sprint after start time

Completed Issues

View in Issue Navigator

Key	Summary	Issue Type	Priority	Status	Story Points (41 → 40)
AGPUSH-2025	Extract the lead functionality as a microservice	Task	Major	RESOLVED	4
AGPUSH-2026	Extract the sales agent functionality as a microservice	Task	Major	RESOLVED	4
AGPUSH-2027	Spike - look at API gateways and decide on a method for implementation	Task	Major	RESOLVED	4
AGPUSH-2028	Implement logging functionality into the lead service	Task	Major	RESOLVED	3
AGPUSH-2029	Implement logging functionality into the push configuration service	Task	Major	RESOLVED	3
AGPUSH-2030	Implement logging functionality into the sales agent service	Task	Major	RESOLVED	3
AGPUSH-2031	Dockerise the lead microservice	Task	Major	RESOLVED	3
AGPUSH-2032	Dockerise the push configuration microservice	Task	Major	RESOLVED	3
AGPUSH-2033	Dockerise the sales agent microservice	Task	Major	RESOLVED	3
AGPUSH-2035	Implement centralised logging to aggregate all logs	Task	Major	RESOLVED	3
AGPUSH-2037 *	Document sprint two for final report	Task	Major	RESOLVED	3
AGPUSH-2039 *	Deploy Dockerised Aerodoc services on Openshift	Task	Major	RESOLVED	5 → 4

Figure 19 Sprint three JIRA items

The version release report at the end of this sprint can be seen in Figure 20. It shows a revised predicted completion date of 12th of March. This is a significantly improved predicted completion date and it is reflective of the large number of story points that the developer completed in this sprint (40), in comparison to the previous sprint (17).

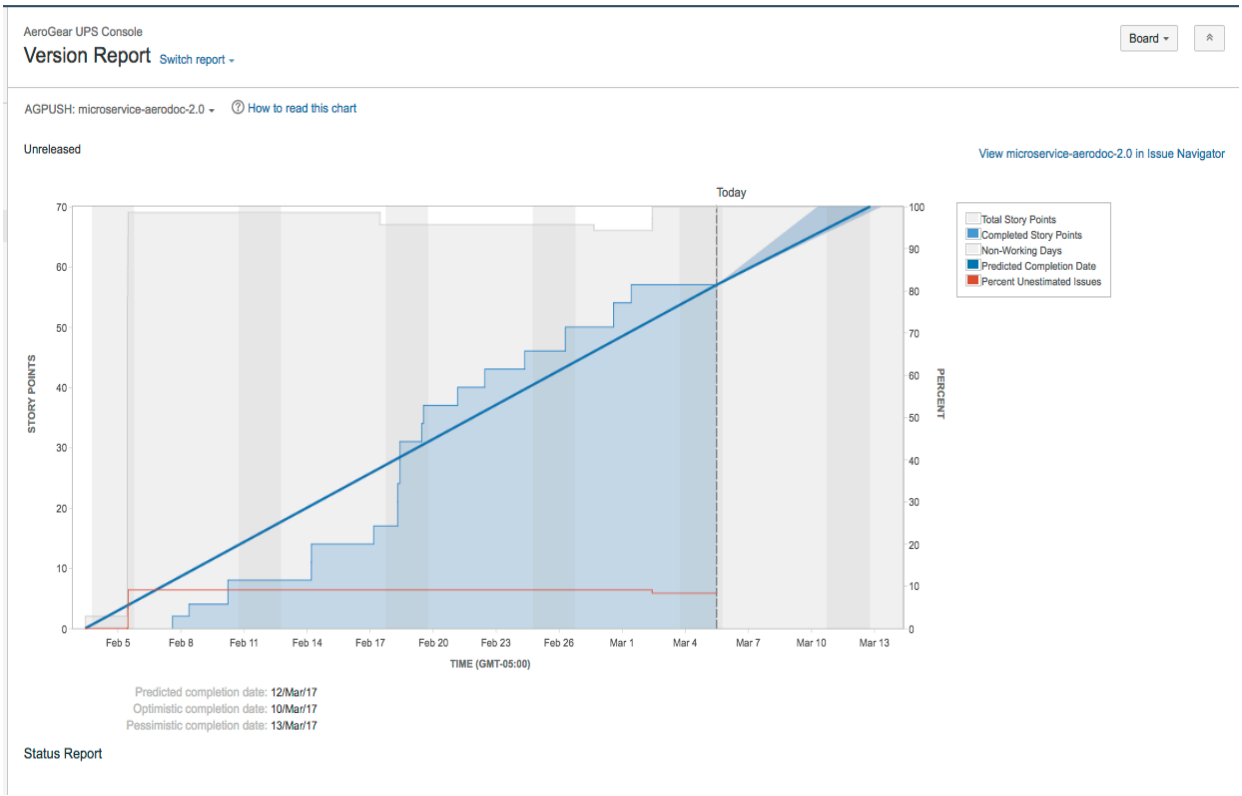


Figure 20 Version report after sprint three

6.4.3.3. Sprint Retrospective

The output of the sprint retrospective meeting for this sprint can be seen in Appendix H.

6.4.4. Sprint four

6.4.4.1. Sprint Planning

The goal of this sprint was to have a functionally equivalent, tested, and deployed microservices version of the original monolithic system. There were two complex items in this sprint, which would be the last features that needed to be implemented, in order to complete the functionally equivalent microservices system. The remaining items focused on testing the functionality of the system.

One of the sprint items encapsulated the work to implement the geolocation functionality for the sales agents. This JIRA item for this feature was AGPUSH-1928, which was de-scoped in sprint one of semester two. This was previously blocked by an issue that was discovered while working on the sprint in question. However, this issue was raised by the developer with the project Scrum-Master at this sprint planning meeting. The issue was then communicated with the Aerogear team, with the end result being, the JIRA item was prioritised to be in the upcoming sprint by the Aerogear team. The Aerogear Android team would work on this issue early in the next sprint This sprint would run in parallel with the planned sprint for this project. This allowed the team to re-scope this JIRA item into this sprint for the developer to implement.

6.4.4.2. Sprint Review

There were several new JIRA items created during this sprint, which was due to a number of different factors. The API gateway ticket went as planned, but it did raise a deployment issue with system. The developer was using Openshift as a platform to deploy the microservices for this project. However, this would not be practical for the Aerogear team going forward. The deployment option for the system needed to be suitable for local development, and deployment on another environment, other than Openshift. This prompted the developer to investigate a second deployment option for the microservices. A JIRA item was created for this second deployment implementation, and the developer was confident that this could be implemented in the current sprint.

There were a number of items in this sprint which were tasked to test the various clients against the new microservices backend. Therefore, there were some functionality issues detected during this testing. As a result of this, a number of new JIRA bug-related items were created mid-sprint. The JIRA items were brought into the sprint and the developer succeeded in completing these.

During the course of this sprint the developer learned how to deal with multiple unplanned challenges which arose. This was a very important learning experience and it highlighted some key challenges that can arise when working on a real world software development project. The items that were completed in this sprint can be seen in Figure 21. This sprint was scheduled to last two weeks and was completed ahead of schedule.

Status Report

* Issue added to sprint after start time

Completed Issues

View in Issue Navigator

Key	Summary	Issue Type	Priority	Status	Story Points (31 → 30)
AGPUSH-1903	Build client application to use for verifying functionality of application	Task	Major	RESOLVED	2
AGPUSH-1928	Create an endpoint for the searching of agents in a given range	Task	Major	RESOLVED	4
AGPUSH-1944	Ensure the old angular application behaves the same using the new backend	Task	Major	RESOLVED	2
AGPUSH-2040	Implement and deploy an API gateway on Openshift for the microservices	Task	Major	RESOLVED	5
AGPUSH-2054	Document sprint three for college report	Task	Major	RESOLVED	2
AGPUSH-2056	create swagger docs for each service individually	Task	Major	RESOLVED	3
AGPUSH-2060 *	Create docker compose file to launch all services and API gateway locally	Task	Major	RESOLVED	3
AGPUSH-2061 *	fix push config api route	Bug	Major	RESOLVED	2
AGPUSH-2063 *	Remove accepted leads from being displayed	Task	Major	RESOLVED	3 → 2
AGPUSH-2064 *	Test functionality of android client against node js backend	Task	Major	RESOLVED	3
AGPUSH-2065 *	Push not working for sending leads	Bug	Major	RESOLVED	2

Figure 21 Sprint four JIRA items

The version release report at the end of this sprint can be seen in Figure 22. It shows that the version has been completed successfully.

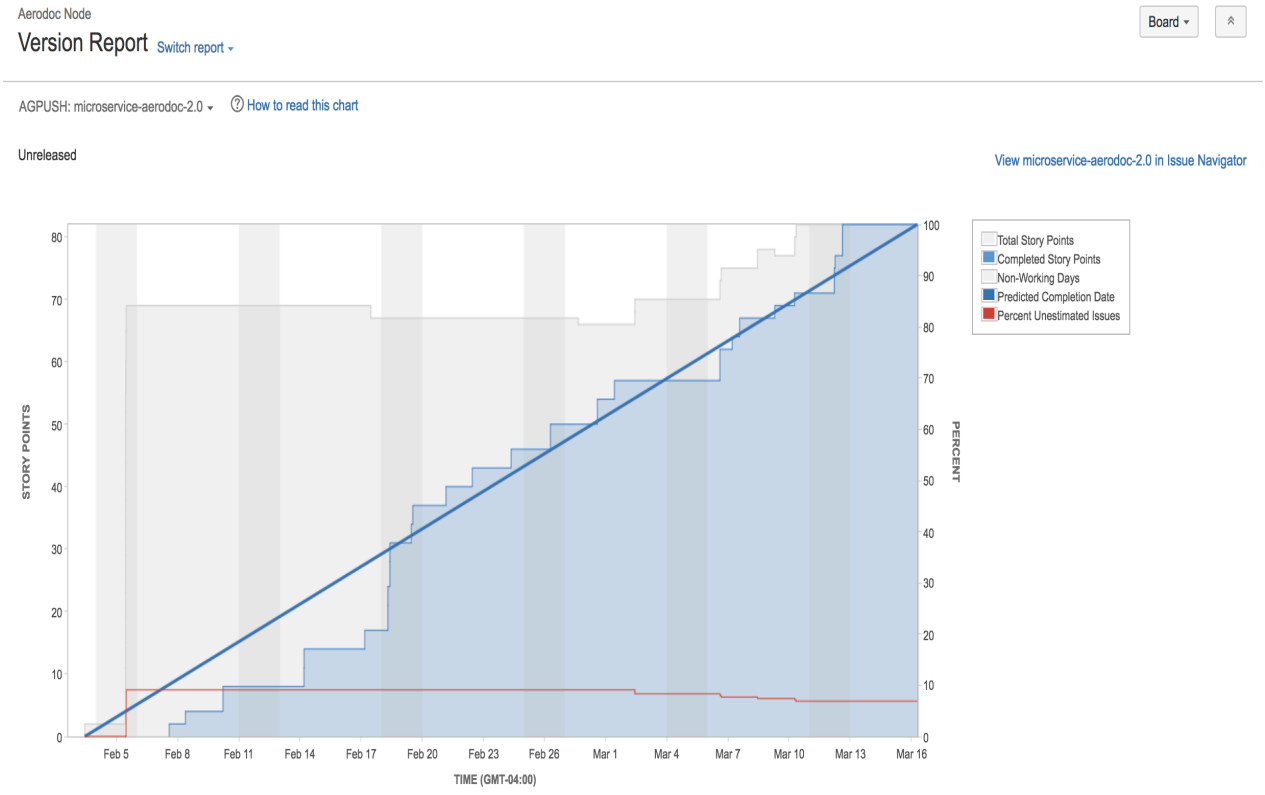


Figure 22 Final version report

6.4.4.3. Sprint Retrospective

A sprint retrospective meeting was held at the end of the sprint. The output of this meeting can be seen in Appendix I.

6.5. ECMAScript Modern Features

A small case study was conducted at the end of this project to establish if the microservices systems code base could be further simplified using some of JavaScript's more modern, and upcoming features. A function was identified in the code base, which makes several asynchronous calls, and was chosen for this case study. ECMAScript is the standard for JavaScript language and this section will discuss how this standard has evolved over recent years. It will demonstrate some of the features that were introduced in recent years, to improve the usability of JavaScript for asynchronous programming.

6.5.1. Callbacks

JavaScript is a non-blocking, single threaded, asynchronous language. In a single threaded system, if a piece of code will take a long period of time to execute, this would block anything else from happening, until the code returned. Using callbacks is just a convention for using JavaScript functions that is essentially the solution to this. The difference in functions that use callbacks and most other functions, is the callback can take an unknown amount of time to produce a result, i.e. it is non-deterministic. With an asynchronous callback, the next step in the code will be taken while the function goes off and does some other work, and thus when a result is returned some code is executed. The idea of using callbacks to solve the single threaded problem is a positive one. However, with the growing use of Node.js in larger software systems, the need to use multiple asynchronous callbacks can lead to very confusing and complex code. This situation is known as callback hell (Harter, 2014). The chosen function was rewritten using callbacks to demonstrate a case of callback hell, and how complex it can become to write code using older ECMAScript features. This can be seen in Figure 23.

```
27
28  exports.create = function (req, res) {
29      const data = new PushConfig(req.body);
30      data.id = Date.now().toString();
31      data.save(function(err, newPushConfig) {
32          if (err) {
33              return res.status(500).json({
34                  error: err.message
35              });
36          }
37          if (newPushConfig.active === true) {
38                  PushConfig.find({ active: true }, function(err, truePushConfigs) {
39                    if (err) {
40                        return res.status(404).json({
41                            error: err.message
42                        });
43                    }
44
45                    truePushConfigs.forEach(function(pushConfig) {
46                    if (!pushConfig._id.equals(newPushConfig._id)) {
47                        pushConfig.active = false;
48                        pushConfig.save(function(err, updatedPushConfig) {
49                             if (err) {
50                                return res.status(500).json({
51                                    error: err.message
52                                });
53                            }
54                            return updatedPushConfig;
55                        });
56                    }
57                  });
58                })
59              }
60              return res.status(201).json(newPushConfig);
61      })
62  };
```

Figure 23 Using Callbacks in case study

6.5.2. Promises

One of the alternatives to using callbacks is to use promises which were introduced in ECMAScript 6. A promise has a state and is either pending and waiting for a value or resolved with a value. The structure of a promise is similar to a "try catch" in Java, and the main component of a promise is the "then" function. This function can take up to two arguments which are also functions and the idea is that the first one contains the code that gets executed when the state is successful and the second in the event of an error (Mattgreer.org, 2014). A "catch" function is chained onto the "then" function, this catches the error should one occur.

Promises can be implemented using two approaches, one of these is a naive approach which nests the promises inside of each other. Each nested promise would also have a "catch" function to catch the error case. This approach can be just as confusing to read for a developer as callbacks are, hence making this approach ineffective. The chosen function was rewritten using nested promises to demonstrate how this can replace callbacks, this can be seen in Figure 24.

```
26
27  exports.create = (req, res) => {
28      const pushConfig = new PushConfig(req.body);
29      pushConfig.id = Date.now().toString();
30      pushConfig.save()
31          .then(newPushConfig => {
32              if (newPushConfig.active === true) {
33                  PushConfig.find({ active: true })
34                      .then(pushConfigs => {
35                          pushConfigs.forEach((pushConfig) => {
36                              if (!pushConfig._id.equals(newPushConfig._id)) {
37                                  pushConfig.active = false;
38                                  return pushConfig.save()
39                                      .then(updatedPushConfig => {
40                                          return updatedPushConfig;
41                                      })
42                                      .catch(err => {
43                                          return err
44                                      })
45                              }
46                          });
47                      })
48                      .catch(err => {
49                          return err;
50                      })
51              }
52              return res.status(201).json(newPushConfig);
53          })
54          .catch(err => {
55              return res.status(500).json({
56                  message: 'Error creating push config',
57                  error: err
58              });
59          })
60  };
61
```

Figure 24 Using nested promised in case study

The correct way to implement promises to replace callbacks is to use promise chaining. This means only one asynchronous call is made and the promise is returned. The next asynchronous call is then chained to the resulting promise. This method is much easier to read than either callbacks, or nested promises. This approach also only requires a single "catch" function, which makes for much easier reading. The chosen function was rewritten using promise chaining to demonstrate how this is a huge improvement on the previous approaches, this can be seen in Figure 25.

```
exports.create = (req, res) => {
    const newPushConfig = new PushConfig(req.body);
    newPushConfig.id = Date.now().toString();
    newPushConfig.save()
        .then(pushConfig => {
            if (pushConfig.active === true) {
                return pushConfig;
            }
        })
        .then(pushConfig => {
            return PushConfig.find({ active: true })
                .then(truePushConfigs => {
                    return truePushConfigs;
                })
        })
        .then(truePushConfigs => {
            truePushConfigs.forEach((pushConfig) => {
                if (!pushConfig._id.equals(newPushConfig._id)) {
                    pushConfig.active = false;
                    return pushConfig.save()
                        .then(updatedPushConfig => {
                            return updatedPushConfig;
                        })
                }
            });
        })
        .then(() => {
            return res.status(201).json(newPushConfig);
        })
        .catch(err => {
            return res.status(500).json({
                error: err.message
            });
        })
};
```

Figure 25 Promise chaining in case study

6.5.3. Generators and Yields

ECMAScript 6 also introduced the concept of generators, which are another solution to callback hell. A generator is a function is marked with a special "*" character and can be suspended and later re-started. This is achieved by using the "yield" keyword. Generators and yields make asynchronous code read synchronously. It also greatly reduces the amount of code that is written to achieve the same functionality as the previous ECMAScript approaches discussed (Harter, 2014). The chosen function was rewritten using generators and yields to demonstrate how much it can simplify a function that makes multiple asynchronous calls to read like a synchronous function, this can be seen in Figure 26.

```
exports.create = function*(req, res) {
    const data = new PushConfig(req.body);
    data.id = Date.now().toString();
    var activeConfigs = [];
    try {
        var newPushConfig = yield data.save();
        if (newPushConfig.active === true) {
            activeConfigs = yield PushConfig.find({ active: true });
        }
        activeConfigs.forEach((pushConfig) => {
            if (!pushConfig._id.equals(newPushConfig._id)) {
                pushConfig.active = false;
                return pushConfig.save();
            }
        });
        return res.status(201).json(newPushConfig);
    } catch (err) {
        return res.status(500).json({
            error: err.message
        });
    }
};
```

Figure 26 Generators in case study

6.5.4. ECMAScript Next

ECMAScript is a term that refers to future features of the ECMAScript standard. The next release of this standard will include new features, which will aim to further improve the methods of writing asynchronous code. The features that will be released for this are “async” functions and the use of “awaits” (Mozilla Developer Network, 2017). These features were not implemented in this case study but will be looked at in the future, as these features become more mainstream.

This case study was conducted to determine if the metrics for the each of the Node.js microservices could be further improved by implementing some of the latest ECMAScript features. It has shown that using some of the latest ECMAScript features such as generators and yields, can greatly reduce metrics such as the lines of code and complexity metrics. This case study has provided some useful information which may be used in for further work on this system.

6.6. Additional Work

This section will discuss some of the extra work that was undertaken by the developer in this project.

6.6.1. Swagger API Documentation

A decision was taken to implement a comprehensive API documentation solution for this project. The tool that was used for this is Swagger. Swagger is an open source API documentation tool with multiple implementation styles. This was chosen as it renders a UI for the API documentation for the end user, directly from the API specification. The Swagger UI tool essentially acts as a client for the API, for which it is configured to document. It allows the end user to perform RESTful calls to the API using provided sample payloads (Swagger.io, 2016). The API specification which is written in YAML, format can be found in Appendix A. The Swagger documentation for each microservice can be also be found in Appendix A.

6.6.2. Alternative Deployment Option

Although this project is deployed on Openshift, it was decided that an alternative deployment option would be needed for the microservices system. This was to provide a method of deployment for the system, to users who wished to deploy the system in an alternative environment, especially for development purposes.

Docker Compose is an open source tool for defining and deploying multiple Docker containers. This was the exact problem that needed to be addressed for this system. This works by defining the services in a YAML file that must be named “docker-compose.yaml”. The configuration for each service can also be defined in this file. Once the file was created and each service defined, the entire backend microservices system could be deployed locally and would accessible to the systems clients. The Docker Compose file was pushed to each of the services repository. This would allow any developer to clone a single microservices repository from GitHub, and deploy the full backend microservices system. The Docker Compose file for deploying this system can be found in Appendix A.

6.6.3. Extraction of Existing Web Client from Monolith

The existing Angular.js web client was also extracted from the monolithic systems code base. This allows the web client be served up separately from the old JBoss Java application and therefore, removes all links or dependencies to the old Aerodoc system. As an additional feature, this was also containerised using Docker and added to the Docker Compose deployment option described in section 6.6.2. This deployment option would now allow for the deployment the entire Aerodoc backend microservices system, and also the web client, by running a single command.

6.7. Future Work

This section will discuss some of the planned future work for the Aerodoc system.

6.7.1. Modern Client Application

The existing web client for the system is using the first version of Angular.js. This web client could be updated to a more modern technology such as React.js or a more recent version of Angular.js. This additional improvement to the Aerodoc system was added to the JIRA backlog, for the Aerogear project.

6.7.2. Authentication

The Java monolithic system is using a Red Hat open source authentication technology called picketlink (Picketlink.org, 2013). This authentication technology was heavily tangled in the Java system. It also now deprecated, meaning its use is no longer recommended. The planned authentication technology for the new microservices system will be implemented using KeyCloak, which is the latest open source authentication technology from Red Hat. While security is an extremely important feature for any software application, it was deemed to be too much of a risk to take on the implementation of KeyCloak authentication in the latter stages of this project. This was due to several reasons which are discussed in the following section.

6.7.2.1. System Clients

In order to implement KeyCloak, a lot of research and spikes would need to be carried out to first gain an understanding of how this technology worked. However, in the event that KeyCloak was implemented on the microservices system, additional development work would also need to be done on the Aerodoc clients.

This would involve investigating what changes needed to be made to the Android client code base. The changes would then need to be converted into JIRA items, and added to the Aerogear Android projects backlog. They would then need to be prioritised and be put into a sprint, which would most likely not be within the timeframe of this project.

Implementing KeyCloak would also mean that changes would need to be made to the Aerodoc Angular client. The current authentication technology was heavily tangled in this code base, and a significant amount of development work would be needed to remove this and make this compatible with KeyCloak. In fact, the best approach here would to build a new web client

from the ground up, with KeyCloak in mind. The work that is involved in just removing it from the existing Angular client is quite significant.

This work that would be involved in implementing KeyCloak, or even another and simpler approach to authentication, was deemed to be much to significant. Therefore, this functionality was added to the project backlog for completion at a later date.

7. Reflection

Project Reflection

The table in Table 6 shows a comparison of the metrics for the Java monolithic system vs the Node.js monolithic system.

Analysed System	Lines of Code	Code Complexity	Technical Debt (Days)	Code Duplication	Code Smells
Java Monolithic System	**2016**	**271**	**1.6**	**3.9%**	**97**
Lead Microservice	239	27	0	0	0
Sales Agent Microservice	303	37	0	0	0
Push Configuration Microservice	218	30	0	0	0
Total Node.js Microservices System	**760**	**31**	**0**	**0%**	**0**
Difference	-1256	-240	-1.6	-3.9%	-97

Table 6 Comparison of the Java monolith vs Node.js microservices system

It can be seen in the table, that each of the analysed metrics for the Node.js microservices system have significantly better results than the Java monolithic system. The bar chart in Figure 27 provides a visual representation of a comparison of the metrics for both versions of the Aerodoc code base.

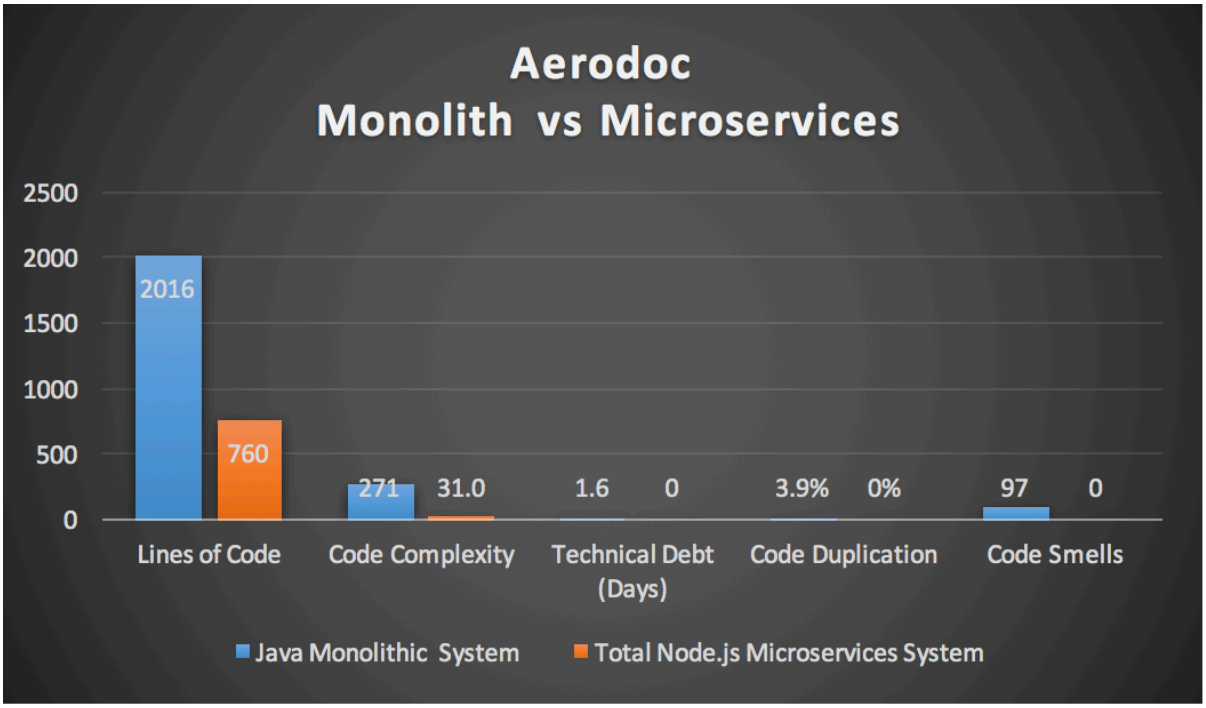


Figure 27 Bar chart showing monolith vs microservices metrics

The results show that the complexity of the system has been significantly reduced from 271 to an average of 31 across each of the microservices. The number of duplicated code blocks and cases of code smells have been reduced to 0. The technical debt has also been reduced to 0 days for each of the microservices. The results show that the microservices version of the backend system has a code base that totals 760 lines, as opposed to 2016 lines for the monolithic version. That is a decrease of 65% in the number of lines of code for the microservices system.

While this is a huge improvement in this metric for the system, it is not even realistic to compare them as a like for like system anymore. The whole purpose of a microservices architecture system, is that it is composed of a suite of microservices and it is not a single application. The microservices are now essentially mini standalone applications, which appear as a single application when deployed with the other microservices. Each microservice must now be treated as its own application. They now each have their own set of integrations tests, documentation, and are packaged in an isolated, independently deployable Docker container.

Each service also has its own GitHub repository, which means a developer who wishes to work on any of the microservices can do so without needing to learn about the entire overall system. The code base for each microservice is now an average of 233 lines of code. This is quite a small code base for any system and therefore, a developer would not need to go through a steep learning curve, to understand the system and work on it. This, along with the accompanying Swagger UI API documentation make each the microservices reasonably straight forward to work on.

The reasons discussed here are why I feel that a microservices architecture system is a vast improvement on a monolithic architecture system. The system is now a much simpler system. It is extensible, such that new microservices can be developed independently, before being deployed with the other microservices of the system. The existing system had few, if any of these attributes.

Personal Reflection

This project has thought me some key lessons in multiple areas of software development. I have gained invaluable experience of developing on an Agile software project. I have learned how to communicate with the open source community via the appropriate channels. The project has thought me a lot about how to manage an Agile project, by using software like JIRA. The exposure to industry practices such as continuous integration, has improved my knowledge of the software development process.

I have also learned how to solve a problem that is being faced by a lot of software companies today. Legacy monolithic systems have proven to be complex, tightly coupled, and very difficult to manage as they grow. They are common in most software companies, and the task of breaking them up into a more modular system is a difficult one. I have learned to analyse an existing monolithic system and identify how it can be broken into a suite of microservices. This was then converted to an microservices version of this system, which is an excellent challenge to have overcome.

I have been faced with multiple issues over the lifecycle of this project, including when first trying to understand the system to set up each of the components in semester one. I especially

found it difficult to finalise the finer grained functionality of the system. This was likely due to having no prior knowledge of the system to begin with, and also having no developer documentation for it.

It was issues like these, that were key in my decision to implement a comprehensive developer documentation solution. This solution is now part of each microservices code base, and will be available to those who wish to build clients for the system, or develop new features for it.

I have come out of this project feeling that I have greatly improved my understanding of server side development using Node.js. Node.js is one of the most popular server side development languages used today, and I hope to build on these skills upon graduating from college. The exposure that I have experienced to a concept such as microservices, will be of a great benefit to me when moving into industry after college.

8. Bibliography

Aerogear.org. (2016 A). *AeroGear - Demos and Examples*. [online] Available at: https://aerogear.org/getstarted/demos/#aerodoc [Accessed 9 Dec. 2016].

Aerogear.org. (2016 B). *AeroGear - Open Source Libraries for Mobile Connectivity*. [online] Available at: https://aerogear.org/ [Accessed 9 Dec. 2016].

Aerogear.org. (2016 C). *AeroGear - UnifiedPush Server User Guide*. [online] Available at: https://aerogear.org/docs/unifiedpush/ups_userguide/index/#_how_the_unifiedpush_server_works [Accessed 12 Nov. 2016].

Agilealliance.org. (2016). *Given – When – Then | Agile Alliance*. [online] Available at: https://www.agilealliance.org/glossary/gwt/ [Accessed 6 Oct. 2016].

Atlassian. (2016). *JIRA Software - Issue & Project Tracking for Software Teams | Atlassian*. [online] Available at: https://goo.gl/05OXUw [Accessed 9 Dec. 2016].

Chaijs.com. (2016). *Chai*. [online] Available at: http://chaijs.com/ [Accessed 11 Dec. 2016].

Chambers.com.au. (2017). *McCabe's Cyclomatic Complexity | Software Quality Metric | Quality Assurance | Complex System | Complex | Software Engineering*. [online] Available at: http://www.chambers.com.au/glossary/mc_cabe_cyclomatic_complexity.php [Accessed 30 Mar. 2017].

Chris Richardson. (2014). *Pattern: Monolithic Architecture*. [online] Available at: http://microservices.io/patterns/monolithic.html [Accessed 8 Sep. 2016].

Colman, G. (2015). *Discoverable — Microservices Architecture with Red Hat Openshift Platform (Part 4)*. [online] Microservices Practitioner Articles. Available at: https://articles.microservices.com/discoverable-microservices-architecture-with-red-hat-openshift-platform-part-4-5b63870ec12f [Accessed 10 Apr. 2017].

Developer.apple.com. (2016). *Local and Remote Notification Programming Guide: APNs Overview*. [online] Available at: https://goo.gl/7eXjMi [Accessed 9 Dec. 2016].

Docs.openshift.org. (2017). *Kubernetes Infrastructure - Infrastructure Components | Architecture | OpenShift Origin Latest*. [online] Available at: https://docs.openshift.org/latest/architecture/infrastructure_components/kubernetes_infrastructure.html#overview [Accessed 10 Apr. 2017].

Docs.sonarqube.org. (2017 A). *Metric Definitions - SonarQube Documentation - SonarQube*. [online] Available at: https://docs.sonarqube.org/display/SONAR/Metric+Definitions [Accessed 30 Mar. 2017].

Docs.sonarqube.org. (2017 B). *Technical Debt - SonarQube-5.2 - Doc SonarQube*. [online] Available at: https://docs.sonarqube.org/display/SONARQUBE52/Technical+Debt [Accessed 18 Apr. 2017].

Docker. (2017). *What is a Container*. [online] Available at: https://www.docker.com/what-container [Accessed 19 Apr. 2017].

Docker. (2016). *Docker Docs*. [online] Available at: https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/ [Accessed 5 Oct. 2016].

Docker. (2015). *What is Docker?* [online] Available at: https://www.docker.com/what-docker [Accessed 5 Oct. 2016].

Expressjs.com. (2016). *Express - Node.js web application framework*. [online] Available at: http://expressjs.com/ [Accessed 9 Dec. 2016].

Fowler, M. (2003). *bliki: TechnicalDebt*. [online] martinfowler.com. Available at: https://martinfowler.com/bliki/TechnicalDebt.html [Accessed 3 Apr. 2017].

Fowler, M. (2006). *bliki: CodeSmell*. [online] martinfowler.com. Available at: https://martinfowler.com/bliki/CodeSmell.html [Accessed 3 Apr. 2017].

Fowler, M. (2015). *bliki: MonolithFirst*. [online] martinfowler.com. Available at: https://www.martinfowler.com/bliki/MonolithFirst.html [Accessed 4 Apr. 2017].

GitHub. (2016). *esnext*. [online] Available at: https://github.com/esnext [Accessed 9 Dec. 2016].

Google Developers. (2016). *Cloud Messaging | Google Developers*. [online] Available at: https://developers.google.com/cloud-messaging/ [Accessed 9 Dec. 2016].

Hapijs.com. (2016). *hapi.js*. [online] Available at: https://hapijs.com/ [Accessed 9 Dec. 2016].

Harter, M. (2014). *Managing Node.js Callback Hell with Promises, Generators and Other Approaches - StrongLoop*. [online] StrongLoop. Available at: https://strongloop.com/strongblog/node-js-callback-hell-promises-generators/ [Accessed 12 Apr. 2017].

Hibernate.org. (2016). *Hibernate Search - Hibernate Search*. [online] Available at: http://hibernate.org/search/ [Accessed 9 Dec. 2016].

Ibm.com. (2013). *Developing mobile apps with Node.js and MongoDB, Part 1: A team's methods and results*. [online] Available at: https://www.ibm.com/developerworks/library/mo-nodejs-1/ [Accessed 16 Nov. 2016].

Indrasiri, K. (2016). *Microservices in Practice: From Architecture to Deployment - DZone Cloud*. [online] dzone.com. Available at: https://dzone.com/articles/microservices-in-practice-1 [Accessed 10 Apr. 2017].

Jasmine.github.io. (2016). *Jasmine Documentation*. [online] Available at: https://jasmine.github.io/index.html [Accessed 11 Dec. 2016].

Johnson, D. (2016). *Using Winston, a versatile logging library for Node.js | thisDaveJ*. [online] Thisdavej.com. Available at: http://thisdavej.com/using-winston-a-versatile-logging-library-for-node-js/ [Accessed 9 Apr. 2017].

Keycloak.org. (2016). *Keycloak*. [online] Available at: http://www.keycloak.org/ [Accessed 9 Dec. 2016].

Koajs.com. (2016). *Koa - next generation web framework for node.js*. [online] Available at: http://koajs.com/ [Accessed 9 Dec. 2016].

Loggly. (2017). *Overview | Loggly*. [online] Available at: https://www.loggly.com/docs/about-loggly/ [Accessed 9 Apr. 2017].

Martin Fowler. (2014). *Microservices*. [online] Available at: http://martinfowler.com/articles/microservices.html#footnote-etymology [Accessed 8 Sep. 2016].

Marco Franssen. (2015). *Jasmine vs. Mocha*. [online] Available at: https://marcofranssen.nl/jasmine-vs-mocha/ [Accessed 3 Nov. 2016].

Mochajs.org. (2016). *Mocha - the fun, simple, flexible JavaScript test framework*. [online] Available at: https://mochajs.org/ [Accessed 11 Dec. 2016].

MongoDB. (2016). *MongoDB and MySQL Compared*. [online] Available at: https://goo.gl/zvQctz [Accessed 3 Nov. 2016].

Mozilla Developer Network. (2017). *ECMAScript Next support in Mozilla*. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_Next_support_in_Mozilla [Accessed 12 Apr. 2017].

Nodejs.org. (2016). *About | Node.js*. [online] Available at: https://nodejs.org/en/about/ [Accessed 3 Nov. 2016].

Nodejs.org. (2016). *Node.js*. [online] Available at: https://nodejs.org/en/ [Accessed 3 Nov. 2016].

Redhat.com. (2016). *Mobile application platform, MBaaS*. [online] Available at: https://www.redhat.com/en/technologies/mobile/application-platform [Accessed 9 Dec. 2016].

Picketlink.org. (2013). *PicketLink*. [online] Available at: http://picketlink.org/ [Accessed 12 Apr. 2017].

Richardson, C. (2015). *Building Microservices Using an API Gateway | NGINX*. [online] NGINX. Available at: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/ [Accessed 10 Apr. 2017].

Richardson, C. (2017). *Microservice Architecture pattern*. [online] Microservices.io. Available at: http://microservices.io/patterns/microservices.html [Accessed 5 Apr. 2017].

Saldahna, L. (2016). *Tips on Logging Microservices - Logz.io*. [online] Logz.io. Available at: https://logz.io/blog/logging-microservices/ [Accessed 8 Apr. 2017].

Sonarqube.org. (2016). *SonarQube*. [online] Available at: http://www.sonarqube.org/ [Accessed 12 Nov. 2016].

Structure101.com. (2016). *Structure101 Software Architecture Development Environment (ADE)*. [online] Available at: http://structure101.com/ [Accessed 9 Dec. 2016].

Swagger.io. (2016). *Swagger Tools Docs*. [online] Available at: http://swagger.io/docs/swagger-tools/#usage-34 [Accessed 12 Apr. 2017].

Tilkov, S. (2015). *Don't start with a monolith*. [online] martinfowler.com. Available at: https://www.martinfowler.com/articles/dont-start-monolith.html [Accessed 4 Apr. 2017].

Wiki.jenkins-ci.org. (2016). *Meet Jenkins - Jenkins - Jenkins Wiki*. [online] Available at: https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins [Accessed 18 Nov. 2016]

9. Appendices

A. Project Resource Links

a. **Aerogear Project GitHub Repository:** https://demos-pushee.rhcloud.com/ag-push/#/

b. **Aerogear JIRA Board:** https://issues.jboss.org/secure/RapidBoard.jspa?rapidView=840&projectKey=AGPUSH

c. **Aerodoc Monolithic System GitHub Repository:** https://github.com/cfoskin/Aerodoc-Node.js

d. **Aerodoc Lead Service GitHub Repository:** https://github.com/cfoskin/lead-service

e. **Aerodoc Sales Agent Service GitHub Repository:** https://github.com/cfoskin/sales-agent-service

f. **Aerodoc Push Configuration Service GitHub Repository:** https://github.com/cfoskin/push-configuration-service

g. **Swagger API YAML Example**: https://github.com/cfoskin/lead-service

h. Docker Compose Deployment File: https://github.com/cfoskin/lead-service/blob/master/docker-compose.yaml

i. Docker Hub Project Links: https://hub.docker.com/u/cfoskin/

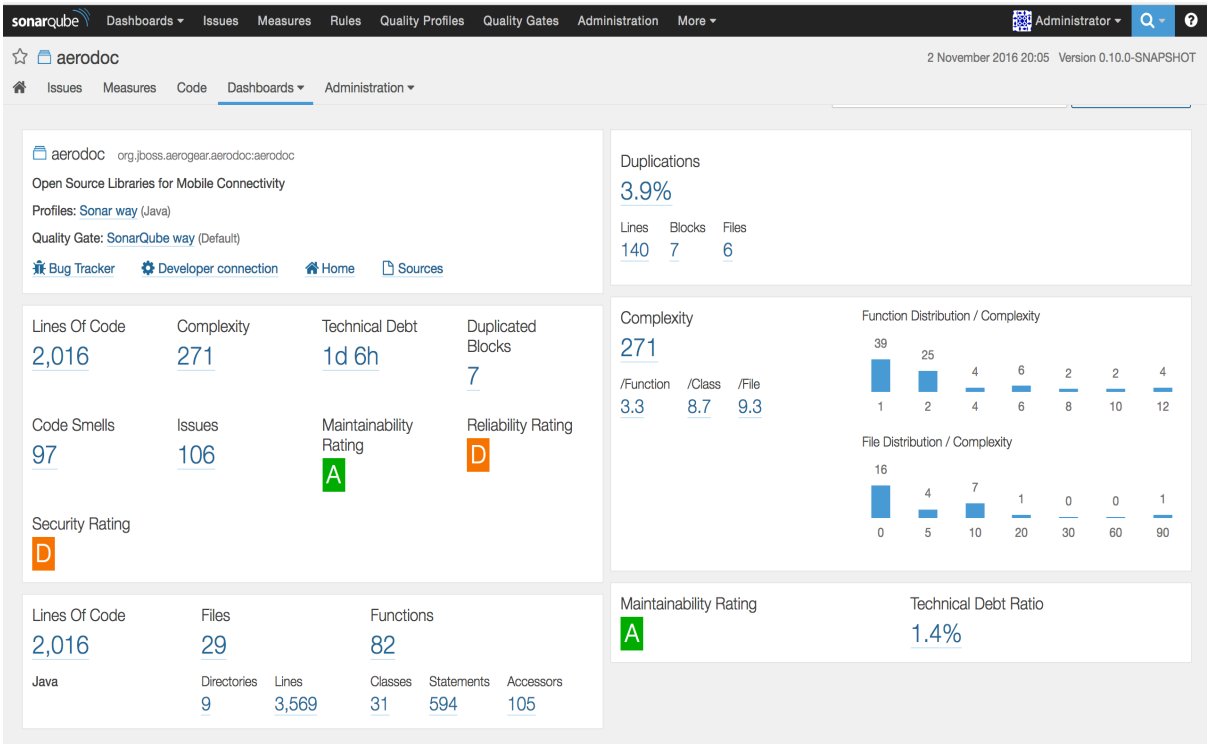
j. Nginx Application Repository: https://github.com/feedhenry/wildcard_proxy

k. **Aerodoc Lead Service Swagger Documentation:** http://api-gateway-aerodoc-node-microservices.52.208.143.88.xip.io/aerodoc/lead-service/docs/

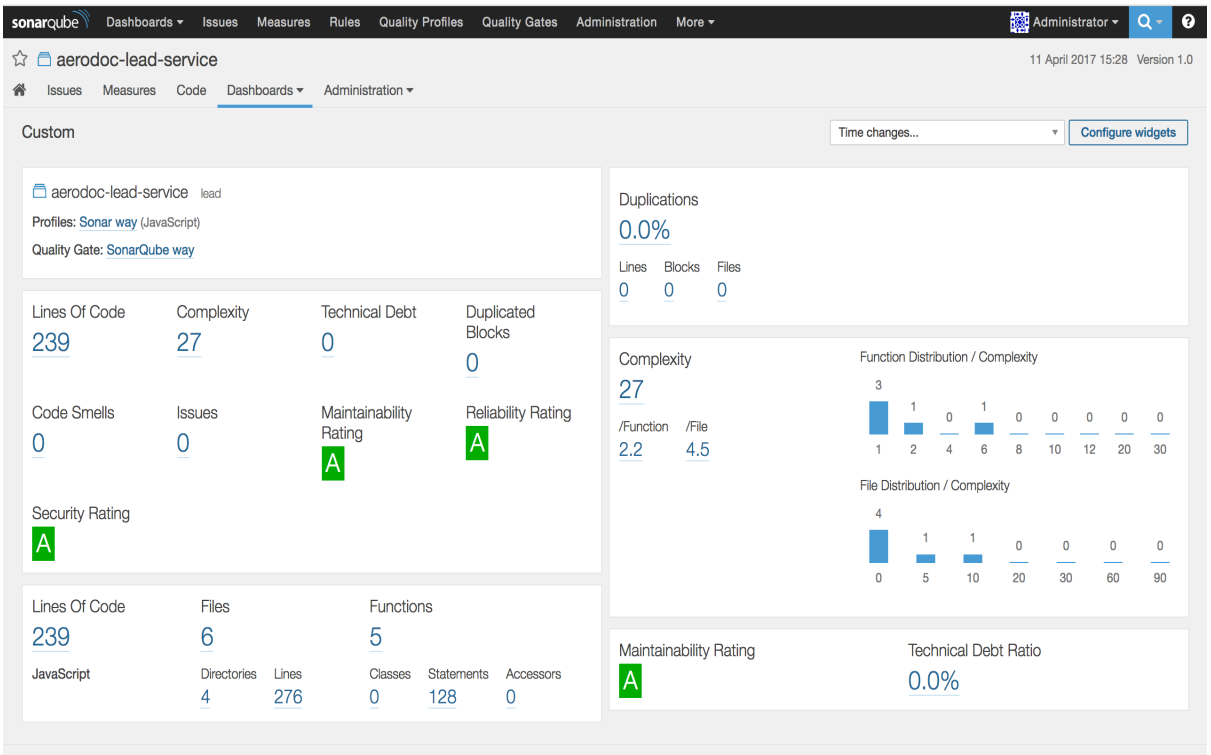
l. Aerodoc Push Configuration Service Swagger Documentation: http://api-gateway-aerodoc-node-microservices.52.208.143.88.xip.io/aerodoc/push-configuration-service/docs/

m. Aerodoc Sales Agent Service Swagger Documentation: http://api-gateway-aerodoc-node-microservices.52.208.143.88.xip.io/aerodoc/sales-agent-service/docs/

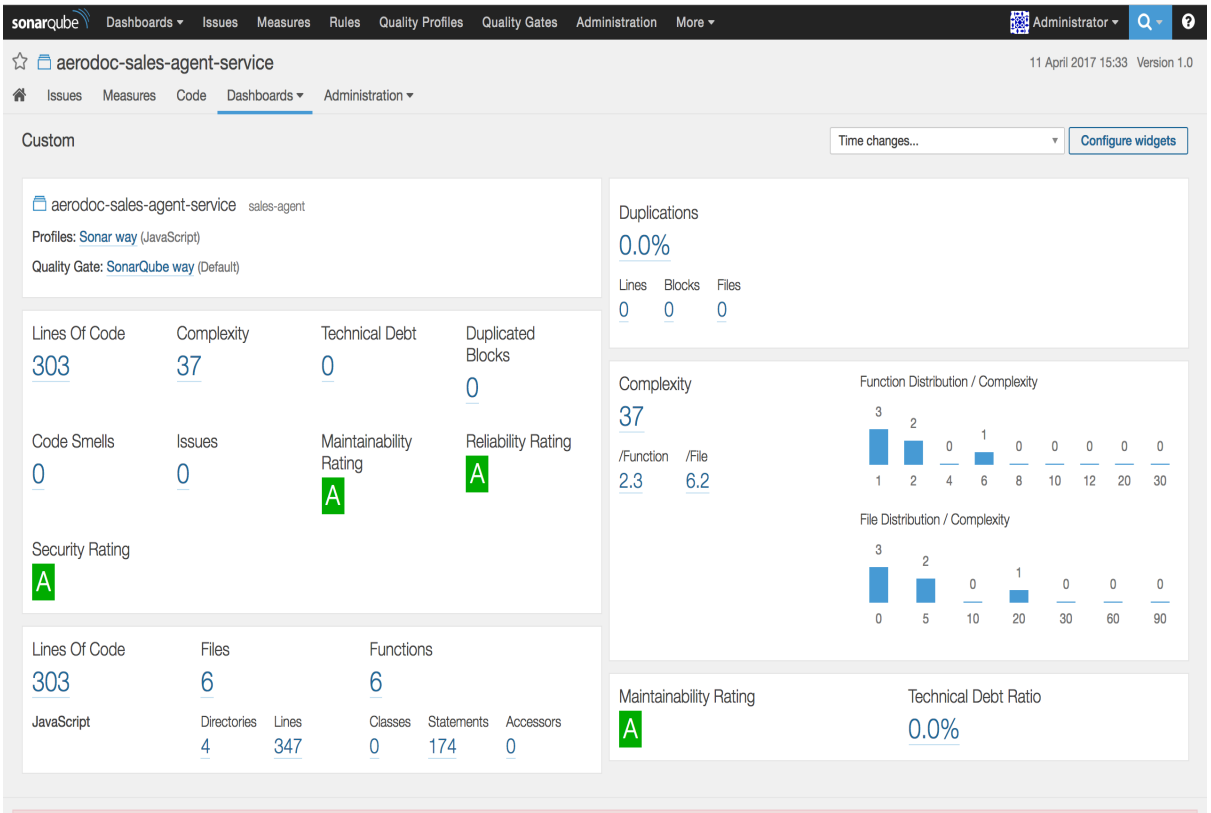
B. Monolithic System Analysis



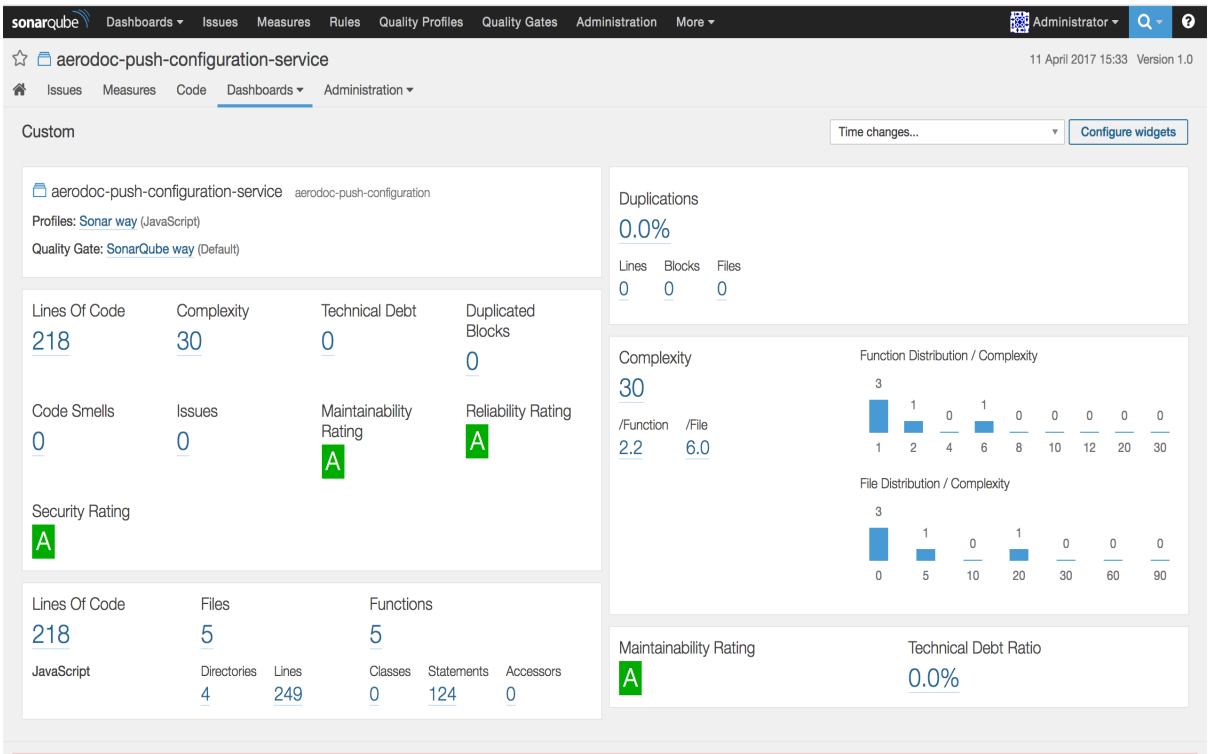
C. Lead Microservice Analysis



D. Sales Agent Microservice Analysis



E. Push Configuration Microservice Analysis



F. Retrospective One

2017-02-03 Microservices Retrospective

Date 03 Feb 2017
Participants Leigh Griffin Colum Foskin

Retrospective

What did we do well?

- Prior functionality in place (prototype) helped speed up this Sprint
- Guidance from Matthias helped clarify what needed to be done was a big help
- CI Setup and Sonarqube helped improve code quality a lot

What should we have done better?

- Poor documentation strategy
- Tickets being created mid sprint, should have been identified as in scope at start of Sprint
- Better breaking up of tickets to more granular tickets (e.g. docs and testing)

Actions

- Colum Foskin review all current tickets from a bigger picture perspective to help minimise mid sprint creation
- Colum Foskin to prune the backlog, close out unnecessary tickets

G. Retrospective Two

2017-02-16 Microservices Retrospective 2

Date 16 Feb 2017
Participants Leigh Griffin Colum Foskin

Retrospective

What did we do well?

- Sprint burndown was consistent, steps of stair dropping
- Focused on one ticket at a time
- Sprint priority of the next sprint was set and agreed upon
- Spikes went well and know we have an understanding of figuring out how to investigation technology and commit to a choice
- Logging solution was very elegant and saved a lot of time.
- Domain knowledge of Microservices improving

What should we have done better?

- Possibly could have added another ticket or two to the Sprint, finished slightly ahead of schedule

Actions

- Colum Foskin release burndown is showing 18th of April for the next Sprint lets target 25-30 Story Points to bring the expected release date in to early April
- Colum Foskin to move spikes and any other unknown JIRA items to the top of the backlog to ensure that we know what is involved in releasing our product

H. Retrospective Three

2017-03-03 Microservices Retrospective 3

Date 03 Mar 2017
Participants Leigh Griffin Colum Foskin

Retrospective

What did we do well?

- Smooth steps of stairs
- Spikes from the previous sprint really helped this
- Planning is on point for a successful 1.0 release within the next Sprint
- Velocity and maturity of planning is really emerging

What should we have done better?

- Rescope the sprint a few days in when you hit it so hard at the beginning
- API Gateway went in cold tackling it which was a big learning overhead
    - Story Points on the API Gateway were off as a result

Actions

- None

I. Retrospective Four

2017-03-13 Retrospective Microservices 4

Date 13 Mar 2017
Participants Leigh Griffin Colum Foskin

Retrospective

What did we do well?

- Interfaced with an open source project and got a change landed
- Exceptional sprint burndown, discovered issues and could bring them in confidently
- Functionality testing tickets in this sprint caught several bugs that needed to be fixed
- Implementing second deployment option using docker compose

What should we have done better?

- More intelligent testing approaches particularly with microservices in mind

Actions

- None