Microservices, A future for software design

Nicholas Murray

BSc Software Design 2021

Declaration

I hereby certify that the material, which is submitted in this thesis towards the award of BSc. Software Design is entirely my own work and has not been submitted for any academic assessment other than part fulfilment of the above named award.

Future students may use the material contained in this thesis provided that the source is acknowledged in full.

Signed…………………………………………….

Date………………………………………………

Abstract

Acknowledgements

Table of contents

[Declaration ii](#_Toc65930753)

[Abstract iv](#_Toc65930754)

[Acknowledgements v](#_Toc65930755)

[Table of contents vi](#_Toc65930756)

[List of tables and figures vii](#_Toc65930757)

[Figures vii](#_Toc65930758)

[Tables vii](#_Toc65930759)

[Chapter 1: Introduction 1](#_Toc65930760)

[1.1 Introduction 1](#_Toc65930761)

[1.2 Conclusion 1](#_Toc65930762)

[Chapter 2: Background Research 2](#_Toc65930763)

[2.1 Introduction 2](#_Toc65930764)

[2.2 Microservices 2](#_Toc65930765)

[2.3 Containers 4](#_Toc65930766)

[2.4 API’s and REST 5](#_Toc65930767)

[2.5 Inbound/Reverse Proxies 6](#_Toc65930768)

[2.6 Conclusion 8](#_Toc65930769)

[Chapter 3 – System Design 8](#_Toc65930770)

[3.1 Introduction 8](#_Toc65930771)

[3.2 Requirements 9](#_Toc65930772)

[3.2.1 Requirement one 9](#_Toc65930773)

[3.2.2 Requirement two 10](#_Toc65930774)

[3.2.3 Requirement three 10](#_Toc65930775)

[3.3 Architecture 11](#_Toc65930776)

[3.3.1 Overview 11](#_Toc65930777)

[3.3.2 Architecture Breakdown 11](#_Toc65930778)

[3.3.3 Architecture Justification 12](#_Toc65930779)

[3.3.3.1 Reliability 13](#_Toc65930780)

[3.3.3.2 Availability 13](#_Toc65930781)

[3.3.3.3 Performance 13](#_Toc65930782)

[3.3.5 Platform justifications 13](#_Toc65930783)

[3.3.5.1 Spring Framework / Spring Boot 14](#_Toc65930784)

[3.3.5.2 React 14](#_Toc65930785)

[3.3.5.3 NGINX 14](#_Toc65930786)

[3.3.5.4 MongoDB 14](#_Toc65930787)

[3.3.5.5 AWS 15](#_Toc65930788)

[3.4 Design 15](#_Toc65930789)

[3.4.1 Microservices 15](#_Toc65930790)

[3.4.2 APIs and REST 15](#_Toc65930791)

[3.4.3 Continuous Integration and Continuous Delivery 15](#_Toc65930792)

[3.5 Implementation 15](#_Toc65930793)

[3.6 Conclusion 15](#_Toc65930794)

[Chapter 4 - Testing and Evaluation / Discussion of Results 16](#_Toc65930795)

[4.1 Introduction 16](#_Toc65930796)

[4.2 Conclusion 16](#_Toc65930797)

[Chapter 5 – Conclusions / Recommendations 17](#_Toc65930798)

[5.1 Introduction 17](#_Toc65930799)

[5.2 Conclusion 17](#_Toc65930800)

[References 18](#_Toc65930801)

[Glossary 19](#_Toc65930802)

[List of Abbreviations 20](#_Toc65930803)

[List of Appendices 21](#_Toc65930804)

List of tables and figures

Figures

[Figure 1: SOA and Microservices architectural comparison](#Figure_2_2_1)

[Figure 2: Inbound and Outbound proxy comparison](#Figure_2_5_1)

[Figure 3: Final system architecture diagram](#Figure_3_3_1)

[Figure 4: Entity Relation](#Figure_4)

[Figure 5: Image upload process](#Figure_5)

Tables

[Table 1: System services and properties](#Table_1)

Chapter 1: Introduction

1.1 Introduction

“By 2022, 90% of All New Apps Will Feature Microservices Architectures” [1] . A microservice architecture is an approach to application development whereby the application is broken down into a set of small independent services, each of these services provides a specific piece of functionality to the overall application.

With the massive growth and adaptation of cloud services and infrastructure there is a new wave of cloud native applications being developed which need to be highly scalable, easy to maintain and efficient in terms of development and cost. Some examples of these applications can be seen today in the form of Netflix and Amazon which migrated from a traditional monolithic architecture to a microservices based one [2] [3]. While Amazon and Netflix are different in terms of the services they provide, they both faced the same issue; how to maintain the scalability and maintainability of a massive online service while also providing the same or better levels of quality to their customers. In both situations the adaptation of a microservice architecture and migrating to a cloud infrastructure was key to their success, apart from Amazon who instead took the opportunity to create their own cloud infrastructure to migrate to, seen today in the form of Amazon Web Services (AWS).

1.2 Conclusion

In this thesis I will attempt to examine what microservices are and why there is a new drive in many areas of software industry to create new or redesign monolithic applications using the new architecture. As part of this explanation, I will be using my implementation of a microservices architecture as used by my online booking system; for context it is a booking system for appointment based services such as a haircut. I will use this system to examine my experience during development and any issues I faced relating to the architecture itself and any of the technologies used by it. Additionally, I will examine some of the development processes that I incorporated into the development process which are seen to go hand in hand with microservices such as continuous integration and continuous delivery. Finally, I will examine the overall result of the system and how it would compare to an equivalent system built using a more traditional monolithic architecture.

Chapter 2: Background Research

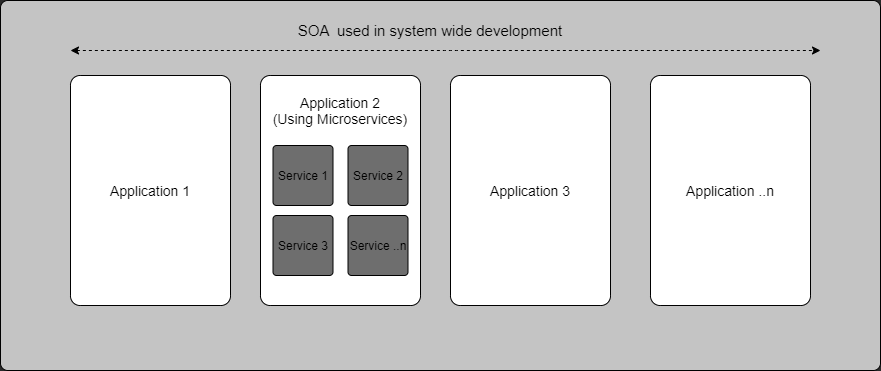
2.1 Introduction

In this chapter I will introduce the main technologies and principles that I will be incorporating in the design and implementation of my online booking service. I have curated this list of technologies and design styles based on my research into currently available systems that adhere to a microservice architecture such as Netflix and Amazon [2] [3]. In my introduction of these technologies, I will attempt to explain their origins, purpose and the functionality that they will provide/fulfil within my system. The topics I am going to cover are microservices; the main design principle of the system, containers; a key building block for microservices, RESTful API’s; the method of communication between the microservices and reverse proxies; a key component in microservices networking. A point to note is that in some cases there are alternatives to the technology or principle that I am explaining however based on my research those that I have chosen either offer a better user experience for me the developer or I have simply chosen them based on the information I could find about them.

2.2 Microservices

“Microservices are small, autonomous services that work together.” [4] A microservices architecture breaks a large complex system down into small independent applications that each fulfil a single service or functionality. Although this architectural style has gained a lot of popularity in recent years under the guise of microservices it is not a new idea, the origin of the architecture is the service-oriented architecture (SOA). SOA emerged in the late 1990s [5] as an approach to enterprise-wide system development in which a large system was broken down into smaller independent applications, which fulfilled a single business function. These independent applications contain the code required to carry out their specific business functionality including data integrations.

Microservices can be seen as taking SOA one step further and breaking down these independent applications again into small applications that focus on a specific part of the overall business functionality.

Figure 1: SOA and Microservices architectural comparison

Microservices when fully leveraged can offer many benefits to developers such as independent scalability, elasticity and cost reduction. What this means in plain language is that each service instance can be scaled up and down to meet demand and with a cost reduction as compared to scaling a monolithic application or even a single application in terms of a SOA system. There are many other benefits to using microservices which will be covered in further chapters so I will not cover them here however one that is not covered is the ability to diversify your technology stack. By this I mean each service can be built using a different programming language or framework once it has support for your chosen communication protocols such as HTTP/REST. This opportunity to diversify the technology stack between services allows developers to choose a language or framework that best meets the requirements of the service, without having to metaphorically fit a square peg in a round hole by using a language that does not natively support the required features.

2.3 Containers

A Container is an isolated environment that contains all the resources it needs to run a piece of software. [6] These environments are run using operating system (OS) level virtualisation rather than hardware level virtualisation, which would traditionally be used when creating a virtual machine (VM). The main advantage that containers have over VMs is how light weight they are in terms of system resources needed; this is down to the differences in how they are virtualised.

VMs virtualise physical hardware, because of this each VM contains a full copy of the guest OS and all the libraries and dependencies needed to run an application. Containers on the other hand virtualise the operating system, this results in the container only needing the specific dependencies of the application it will be running. This is because the container uses the underlying resources and features of the host OS. [7]

Containers are a key enabler for creating a microservices based application as each microservice is run in a container. The lightweight and hardware agnostic nature of containers allow them to be run on a wide range of environments from a low powered device such as a Raspberry Pi to a cloud-based system with a vast number of resources available to it. Containers are also the reason that microservices are so easily scalable as a new container instance can be created to meet high demand and shut down when demand reduces; the same could be achieved with a full VM however would require more resources and more time to create a new instance.

To run a container, you need to have container engine and an image. A container engine is a system that creates and runs your containers in a virtualised environment, the most popular of these is Docker and will be what I am going to use for my booking service. An image is template which can be used to create a container. It contains the list specifications for what a container will run and how it will run them. Images can be easily created locally using a Dockerfile or pulled from a remote repository. A simple explanation of an image is to think of an image as a cookie cutter and the container as a cookie; cut out and defined by the cutter.

2.4 API’s and REST

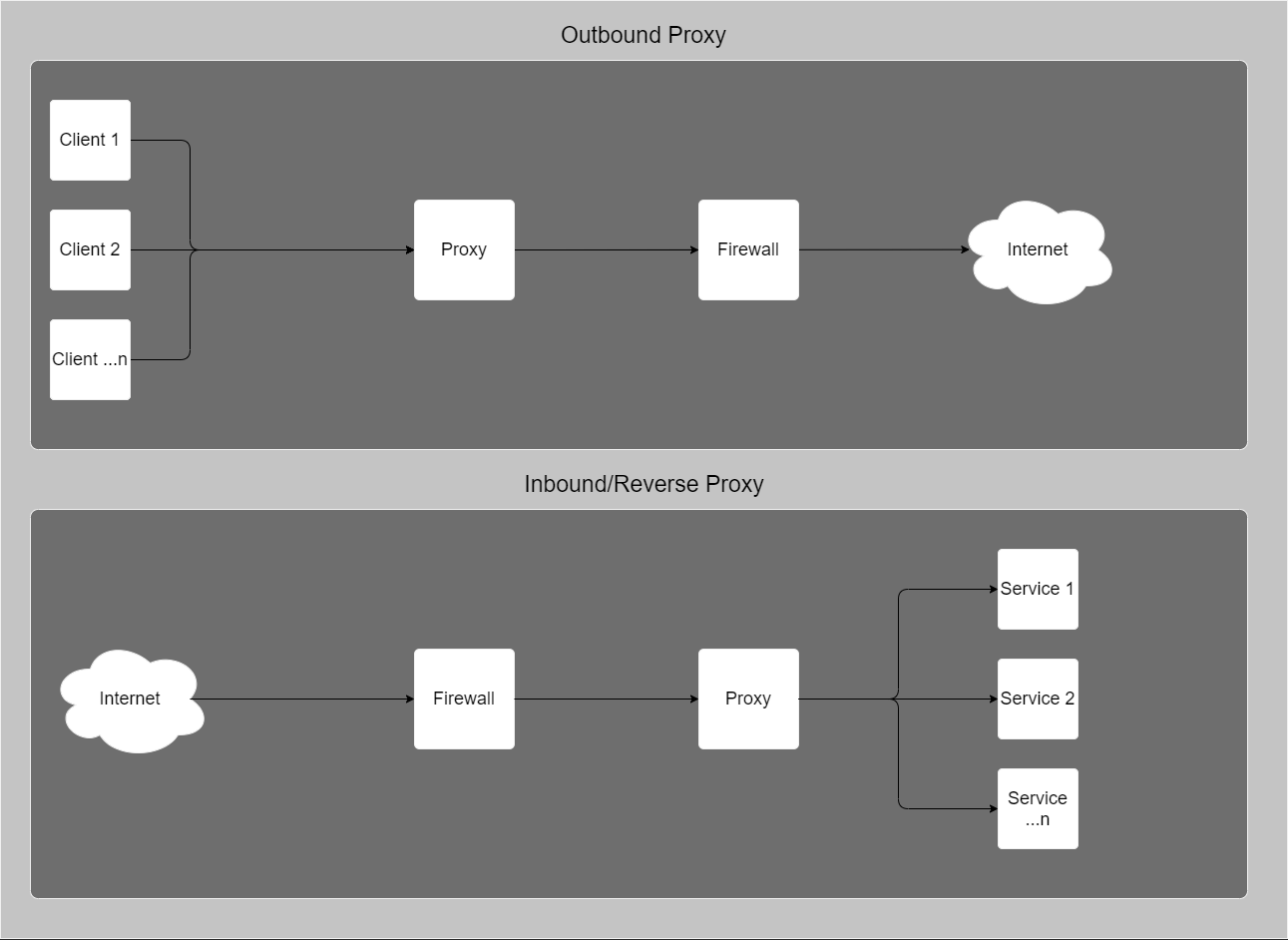
An application programming interface (API) is a software defined interface that allows multiple pieces of software to communicate with one another. The idea of an API was described as way to abstract the functionality and database access of a program away from an independent graphical user interface [8]. By creating an API you can abstract the implementation of a piece of software away from the consumer. This can allow developers to take a black box approach to utilising a piece of software as they do not need to understand the implementation behind the interface, but rather only the inputs and outputs of it. For this to work in practice there needs to a contract defined by the API that describes the inputs and outputs of the interface. An important point to note about APIs is that they are not restricted to machine to machine or software to software communication but can also allow human consumption depending on the implementation.

The type of API I will be using for my online booking service is a RESTful API. A RESTful API is an API which is built upon a Representational State Transfer (REST) protocol [9]. The REST architecture is made up of servers and clients, the clients can change state depending on the resources received from requests to the servers. The servers themselves are not stateful and treat each request from a client as a new independent request. This allows for better service performance as the resources are not locked to an individual client. REST relies upon HTTP for its operation and uses URLs to identify and represent its interfaces resources. RESTful web services use four major procedures for carrying out operations, these are Create, Read, Update and Delete (CRUD) these operations can be directly mapped to the HTTP verbs POST, GET, PUT and DELETE respectively; this is by design as Roy Fielding one of the authors of HTTP 1.0 defined representational state transfer in his doctoral dissertation. REST architecture does not prescribe any resource type or representation and therefore its resources can be any type of data, however commonly use JSON, XML or HTML.

2.5 Inbound/Reverse Proxies

A proxy or proxy server in terms of software development is an intermediary server that channels the requests from a network to the wider internet usually from behind a firewall [10], this can provide increased security as the network clients to not directly access the internet. There can also be benefits to performance as the proxy can carry out load balancing on the network and in some cases provide a cache to minimise outbound requests for frequently accessed resources. Proxies are a critical component of microservice applications as they provide a single point of access to the multiple services, however proxies as explained above do not fulfil the needs of a microservice application as this type of proxy focuses on outbound instead of incoming traffic. The type of proxy needed for a microservice application is a reverse or inbound proxy, this type of proxy receives requests from the wider internet and forwards them the appropriate service. It still provides the same benefits as an outbound proxy as all traffic can be routed through a firewall and can also be load balanced.

Figure 2: Inbound and Outbound proxy comparison



The reason a proxy is needed for a microservice application is because without it each service would need to be contacted by a client using a separate IP addresses which would add increased complexity to the communication process. Adding a proxy allows all the separate services to be represented on the network by a single IP, it also allows for load balancing between multiple instances of the same services to increase network performance. An added benefit of using reverse proxies when developing microservice applications is that you can access all of your containers on a single port, whereas without the proxy you would not be able to expose multiple applications on the same port. This is achieved by using the Docker network itself and allows inter container communication without exposing any ports other than that of the proxy. This requires that your proxy itself is also running in a Docker container, however given the context of which I am explaining the reverse proxy this should not be a problem.

2.6 Conclusion

Using the aforementioned technologies and principles, I hope to design my online booking systems architecture using design queues from existing applications that are considered to be at the fore front of cloud native computing. I also hope to demonstrate the advantages of using microservices and by extension the advantages taking a cloud native approach to developing and deploying modern software applications. In conclusion I will be using Docker to create multiple independent containers that will play host to my microservices, I will also be using RESTful APIs to communicate between my services and running a reverse proxy so that all of my services will be accessible via a single endpoint.

Chapter 3 – System Design

3.1 Introduction

In this chapter I will discuss the system requirements, architecture, design and implementation of the online booking service that I briefly mentioned in chapter one. In each of the listed topics I will attempt to give as much explanation as possible about the choices and justifications I have made. For context and understanding I will give a broader explanation of the booking system here.

The online booking system will allow partners to create a listing for their venues, that will allow their potential customers to find out more information about the venue and allow them to make bookings for the services offered by venue. The purpose of the system is to create a uniform way for small businesses and chains to bring their business online so that they can provide a better and more convenient service for their customers. Currently many small businesses use non uniform and informal methods to allow customers to make bookings for any services they provide. An example of these methods that I have had personal experience with is businesses using messaging apps and social media profiles to accept bookings. This relies on the business owner or employee constantly monitoring the online profile or messenger service.

3.2 Requirements

At a high level the requirements for the online booking system can be stated as one core requirement from which all requirements can be derived, this requirement is as follows:

*“The online booking system should make it easier for consumers to find a service or service provider and make a booking in a standard and uniform way. The system should make this process as simple and short as possible for the user with a focus on QoS so that the system does not reflect badly upon the partners”.*

When examined at a more detailed level this core requirement can be broken down further into multiple more focused requirements, of which I will list the three which I think capture the system requirements most accurately and which can be derived further to a user story level for system development.

3.2.1 Requirement one

*“A consumer should be able to find available venues and services that meet their requirements, once found the consumer should be able to view the availability of the service and make a booking at an available time of their choosing”.*

From requirement one we can begin to break down the main requirements for the customer as a user of the system. Below I will list some of these requirements in no particular order:

1. A search functionality that will allow the customer to find services based on a set of criteria.
2. A database to persist customer bookings.
3. A visual representation of the availability of services offered by a venue that will allow customers to view available times to make bookings at.

3.2.2 Requirement two

*“A partner should be able to create and manage listings for their venue(s) and the service(s) offered by them. From which the partner should be able to receive and manage bookings by potential customers”.*

From requirement two we can identify the main requirements of the partner as user of the system. Below I have listed some of these requirements in no particular order:

1. A user interface to create and manage venues and services offered by them.
2. A database to persist the venues and services.
3. A user interface for viewing and managing bookings made by customers.

3.2.3 Requirement three

“*The system should provide a high QoS for all types of users, with a focus on reliability, availability and performance*”.

In terms of system design and architectural considerations I believe that this requirement is the most important and from it we can surmise some of the requirements that can help define the system and its architecture. Below I will list a set of these derived requirements:

1. The system design and development processes should aim to provide a reliable service where possible.
2. The system architecture and deployment should allow for and enable high availability.
3. The system architecture should allow for high performance in relation to user-based interactions and functionalities.

3.3 Architecture

3.3.1 Overview

As should be obvious considering all previous sections and chapters up to this point, I will be using a microservices based architecture for the system. The decision to use this type of architecture was based heavily off both my background research into cloud native services mentioned above and requirement three from the previous section. In [figure 3](#Figure_3_3_1) you can see that the I have split the system into five microservices, below in [table 1](#Table_1) I will give an overview of the roles that each of these services will fulfil. In this chapter I will also give a brief justification for each of the platforms I have chosen to utilise in my system.

3.3.2 Architecture Breakdown

Graphical user interface, diagram, application

Description automatically generatedFigure 3: Final system architecture diagram

Table 1: System services and properties

|  |  |  |
| --- | --- | --- |
| **Name** | **Platform** | **Role** |
| Authentication | Spring Boot  (Java) | This service will be responsible for the logging in and registering of users of the system. |
| Venue | Spring Boot  (Java) | This service will be responsible for carrying out operations relating to partner venues. This will encompass all the relevant CRUD operations required for the venues and its dependent entities the services and bookings. |
| Images | Spring Boot  (Java) | This service will be responsible for the upload and deletion of images used in venue listings. In future updates this may include any editing required by the images. |
| Web Client | React  (JavaScript) | This service will be responsible for serving the physical web page to the users. |
| Reverse Proxy | NGINX | The reverse proxy service will be used to direct inbound traffic to the relevant services. It can also be configured as a load balancer. |

3.3.3 Architecture Justification

I believe that the described microservices architecture will allow me to fulfil the three derived requirements listed under requirement three. I will discuss my justifications for this belief below for each of the derived requirements.

3.3.3.1 Reliability

Using microservices I will be able to incorporate a continuous integration and delivery into the system development [4]. By using this process, testing is easy to incorporate into the development process and can be easily automated. It also allows for quick deployment of new features or fixes for bugs that may be identified in production. The combination of testing and the ability to roll out new features and fixes at ease can result in a high level of reliability for the users.

3.3.3.2 Availability

High availability can be realised using microservices as it allows for multiple independent instances of a service to run concurrently, this means when one instance of a service is experiencing down time traffic can be directed to a different instance. Where another instance is not available to receive redirected traffic, a new instance can be easily created to minimise downtime.

3.3.3.3 Performance

While perceived performance can be affected by multiple factors outside of the systems control such as a user’s network connection, high performance can be achieved using microservices for the similar reasons to that availability. Using multiple instances of a service can allow for system traffic to be load balanced between them. This can reduce the amount of time a request is waiting to be processed by a service and therefore result in better perceived performance by the user.

3.3.4 Platform justifications

From [figure 3](#Figure_3_3_1) you can see that I have included the platforms that each of the services will be implemented in and the overall platform host and database. I have chosen each of these platforms based on my research into similarly designed systems but also based on the ease of use of each of the platforms, this is because development time must be a consideration so that can insure the completion of the system before my project deadline.

3.3.4.1 Spring Framework / Spring Boot

The Spring framework is an opensource enterprise level Java framework that can be used for creating standalone web applications [11]. Spring provides multiple services such as spring-data and spring-security which simplify data access and security configuration for Java applications [12] [13]. I have chosen to use Spring Boot; a project by the Spring which has an embedded Tomcat web container along with multiple starter dependencies, which makes it easier to get up and running with the Spring framework [14]. The framework also has a large opensource community and is used by many large online web applications such as Netflix who replaced their internally developed systems with it as their core Java backend framework [15].

3.3.4.2 React

React is an opensource JavaScript library that can be used to create dynamic and interactive user interfaces for web applications. React is used by many large companies including Facebook who originally created the library as a replacement for their existing framework [16]. React encourages the creation of re-usable components wrote with a combination of JavaScript and JSX. Making re-usable components speeds up development time and reduces code duplication across the system. According to a survey carried out in 2020 by Stack Overflow, React is the most wanted web framework and the second most loved framework to work with [17].

3.3.4.3 NGINX

NGINX is an opensource web server that can be used for many different applications such as load balancing and reverse proxying. It is the second most popular web server and increasing in popularity as the current most popular web server Apache HTTP server is decreasing in popularity [18]. For the systems purposes either of these web servers would have sufficed so I chose based on my personal preference.

3.3.4.4 MongoDB

MongoDB is a NoSQL database that uses BSON based documents to store data. In my initial testing and prototypes I chose to MySQL as the database for the system as I was most familiar with it from previous projects, however after my first couple of design iterations I made the decision to make the switch to MongoDB. I made this decision as the schemaless nature of the database system made it much easier to make changes to or add elements to documents in the database, without the need for any updates to the schema which can be time consuming. Using MongoDB Atlas; my database hosting provider, I can view and edit elements of my database in BSON without having to make and database queries.

3.3.4.5 AWS

I chose AWS because it offered two of the services that I wanted to utilise for my system. These services are Elastic Container Service (ECS) and Simple Storage Service (S3), these are for hosting the microservices and storing pictures that will be used by the web client. This choice was motivated by personal preference for having one less service to sign up to and configure but also because AWS offers free credit to students, so I was able to use the system for free.

3.4 Design

3.4.1 Overview

In this section I will discuss various design aspects of the online booking system such as the microservices and how I chose to split the services and what each of them is responsible for. I will also give an overview of my REST interfaces and how I designed them also how I set up GitHub Actions to use it as continuous integration and delivery pipeline for the whole project. As can be seen in [figure 3](#Figure_3_3_1) the architecture will be designed to include a mobile app, which I will not be implementing as part of this project; for the purposes of design I will make design considerations that will allow the theoretical app to interact with the services the same way as the web client using the same REST interfaces.

3.4.2 Microservices

Microservices are small autonomous services that encompass an independent unit of work [4], to achieve this they need to be designed in such a way that each service fulfils a separate piece of functionality, while disregarding the size or complexity of the functionality. The latter half of that statement is crucial as it becomes tempting to split a piece of functionality further if it contains a large codebase or high complexity, but this could harm performance by trying to split the functionality into multiple services [4]. With that in mind, I divided my system up into easily identifiable pieces of functionality which I will describe below.

3.4.2.1 Authentication service

The authentication service as the name suggests is responsible for the authentication of users. For the purposes of my system this includes both the registering of new users and then subsequently the logging in of users, these could be seen as separate functionality however I kept both of them together as they both rely on many of the same services for the encryption of and storing or retrieving of user information.

3.4.2.2 Venue service

The venue service contains the biggest piece of functionality which is responsible for controlling the management of the Venue database entities and its related entities for the Services and Bookings. I chose to keep the management of these three entities together under the umbrella of the Venue service because there would have been a lot of duplication of code and functionality if they were broken into separate microservices.

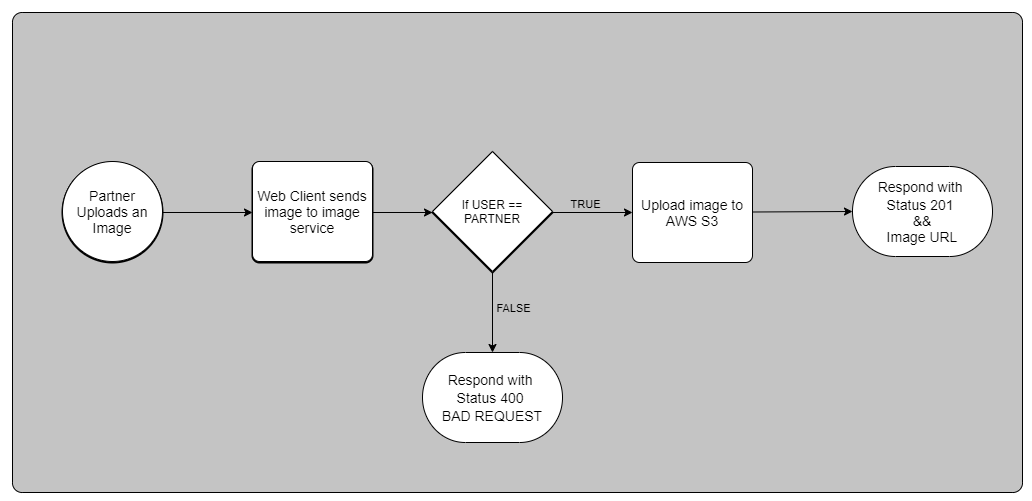
This is because of the relation between them ([figure 4](#Figure_4)), for example when a Venue is created or deleted all related Services or Bookings need to also be updated accordingly, likewise when a Service is updated in anyway all its related Bookings need to be updated. This should not break the single responsibility principle of microservices, which states you should house together things that change for the same reason [19]. Even though I have decided to house the three entities together each entity would have had its own REST interface regardless of how they are broken down, so it would have been possible to divide the service into three individual services and still achieve the same overall functionality. There is an argument to be made for splitting the service into individual services based on increasing the individual scalability of each. If this system was to be deployed commercially, I would definitely have to weight that argument much heavier in decision making as this service will be the most heavily used part of the system and will experience much greater traffic than other services.

Diagram

Description automatically generatedFigure 4: Entity Relation

3.4.2.3 Image service

The image service will be responsible for the uploading and deletion of images from the AWS storage bucket, originally this functionality was going to be fulfilled directly on the web client as Partners uploaded images for their Venues, however in my initial prototyping I quickly realised that this would lead to exposed API keys for the AWS S3 API as users could retrieve them by simply inspecting the web page in their browser. To prevent this security issue, I moved this functionality to image service. Now the web client will upload the image to the image service which will check their account permissions and then conditionally upload the image, once uploaded the image service returns a URL linking to the stored image so that the Venue entity has reference to it when being viewed. I have included [figure 5](#Figure_5) below which is a diagram of this process for a better explanation.

Figure 5: Image Upload process

3.4.2.4 Web Client

The web client service will host the React web app. This will be the website / user interface that Customers and Partners will be able use to interact with the booking system. The users should be able to carry out all proposed user functionality via the web client.

3.4.2.5 Proxy service

The proxy service will be set up as a reverse proxy as explained in chapter 2, it will be configured to forward all requests to the appropriate service based on the pathname in the request. It will forward all requests on the Docker network by sending the requests to the containers themselves and not an associated URL, this restricts access to the REST interfaces via the proxy service only. The configuration of NGINX to use as a proxy service is made very simple by using a single configuration file.

3.4.3 APIs and REST

3.4.4 Continuous Integration and Continuous Delivery

3.4.5 Implementation

3.4.6 Conclusion

Chapter 4 - Testing and Evaluation / Discussion of Results

4.1 Introduction

4.2 Conclusion

Chapter 5 – Conclusions / Recommendations

5.1 Introduction

5.2 Conclusion

References

|  |  |
| --- | --- |
| [1] | International Data Corporation (IDC), “IDC FutureScape: Worldwide IT Industry 2019 Predictions,” IDC Corporate USA, Framingham, 2018. |
| [2] | Netflix, “Completing the Netflix Cloud Migration,” Netflix, [Online]. Available: https://about.netflix.com/en/news/completing-the-netflix-cloud-migration. [Accessed 06 02 2021]. |
| [3] | S. M. F. III, “The New Stack,” The New Stack, 08 10 2015. [Online]. Available: https://thenewstack.io/led-amazon-microservices-architecture/. [Accessed 06 02 2021]. |
| [4] | K. Bakshi, “Microservices-based software architecture and approaches,” in *2017 IEEE Aerospace Conference*, Big Sky, MT, USA, 2017. |
| [5] | IBM Cloud Team, “SOA vs. Microservices: What’s the Difference?,” IBM, 02 09 2020. [Online]. Available: https://www.ibm.com/cloud/blog/soa-vs-microservices. [Accessed 13 02 2021]. |
| [6] | D. V. N. R. S. David Jaramillo, “Leveraging microservices architecture by using Docker technology,” in *SoutheastCon 2016*, Norfolk, VA, USA, 2016. |
| [7] | IBM Cloud Team, “Containers vs. VMs: What’s the Difference?,” IBM, 02 09 2020. [Online]. Available: https://www.ibm.com/cloud/blog/containers-vs-vms. [Accessed 12 02 2021]. |
| [8] | J. IRA W. COTTON & FRANK S. GREATOREX, “Data structures and techniques for remote,” in *Managing Requirements Knowledge, International Workshop*, San Francisco, 1968. |
| [9] | N. S. Williams, Java for Web Applications, Indianapolis: John Wiley & Sons, Inc, 2014. |
| [10] | A. L. &. K. Altis, “World-Wide Web Proxies,” *Computer Networks and ISDN Systems,* pp. 147-154, 11 1994. |
| [11] | Spring, “Spring Framework,” VMware, [Online]. Available: https://spring.io/projects/spring-framework. [Accessed 6 March 2021]. |
| [12] | Spring, “Spring Data,” VMware, [Online]. Available: https://spring.io/projects/spring-data. [Accessed 6 March 2021]. |
| [13] | Spring, “Spring Security,” VMware, [Online]. Available: https://spring.io/projects/spring-security. [Accessed 6 March 2021]. |
| [14] | IBM Cloud Education, “Java Spring Boot,” IBM, 25 March 2020. [Online]. Available: https://www.ibm.com/cloud/learn/java-spring-boot. [Accessed 6 March 2021]. |
| [15] | Netflix Technology Blog, “Netflix OSS and Spring Boot — Coming Full Circle,” Netflix, 18 December 2018. [Online]. Available: https://netflixtechblog.com/netflix-oss-and-spring-boot-coming-full-circle-4855947713a0. [Accessed 6 March 2021]. |
| [16] | P. Hunt, “Why did we build React?,” Facebook Open Source, 5 June 2013. [Online]. Available: https://reactjs.org/blog/2013/06/05/why-react.html. [Accessed 6 March 2021]. |
| [17] | Stack Overflow, “2020 Developer Survey,” Stack Overflow, February 2020. [Online]. Available: https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved. [Accessed 6 March 2021]. |
| [18] | SimilarTech, “Apache vs NGINX,” SimilaTech Ltd, [Online]. Available: https://www.similartech.com/compare/apache-vs-nginx. [Accessed 6 March 2021]. |
| [19] | R. C. Martin, Agile Software Development, Principles, Patterns, and Practices, Pearson Education, 2003. |

Glossary

|  |  |
| --- | --- |
| Partner | In the context of the online booking system a Partner is a business who is using the system for managing their service bookings, |
| Customer | In the context of the online booking system a Customer is a user who will be using the system to create and manage bookings of services offered by Partners. |
| Venue | In the context of the online booking system a Venue is the physical store front or premises where a partner is offering a set of services from. A single Partner can have multiple venues. Venues are represented in the database as the Venue entity. |
| Service | In the context of the online booking system a Service is an offering from a Venue that a user can make a booking for. This can be things such as haircuts or car services. Services are represented in the database as the Service entity. |
| Booking | In the context of the online booking system a Booking is a reservation made by a customer for a Service offered by a Venue. Bookings are represented in the database as the Booking entity. |

List of Abbreviations

|  |  |
| --- | --- |
| AWS | Amazon Web Services. A collection of cloud based services that vary from storage to virtual machine hosting, it can be seen as a IaaS or PaaS. |
| API | Application Programming Interface. This is a set of operations or objects that developers can use to create or communicate between software applications. |
| BSON | Binary JSON. |
| OS | Operating System. |
| QoS | Quality of Service. |
| REST | Representational State Transfer. A design standard for web based APIs or web resources. |
| SOA | Service-Oriented Architecture. The division of a large application into individual components that focus on a single business value or feature. |
| VM | Virtual Machine. A virtualised computer environment which can be used to replicate a bare metal system. This enables multiple environments to be run off a single bare metal server/computer. |

List of Appendices