

---

# Exploring the Microservice Architecture to develop a flexible web-service of some description

---

Jonathan James Mitchell  
- 40311730

Submitted in partial fulfilment of  
the requirements of Edinburgh Napier University  
for the Degree of  
BEng (Hons) Software Engineering

School of Computing

January 5, 2019

### **Authorship Declaration**

I, Jonathan James Mitchell, confirm that this dissertation and the work presented in it are my own achievement.

Where I have consulted the published work of others this is always clearly attributed;

Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

I have acknowledged all main sources of help;

If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

I have read and understand the penalties associated with Academic Misconduct.

I also confirm that I have obtained informed consent from all people I have involved in the work in this dissertation following the School's ethical guidelines.

*Signed: J.Mitchell*

*Date: January 5, 2019*

*Matriculation no: 40311730*

### **Data Protection Declaration**

Under the 1998 Data Protection Act, The University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name below one of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.

### **Abstract**

This project is designed to explore the Microservice Architecture (MSA) to develop a flexible E-commerce web system. The architecture will be explored by researching and discussing various principles, tenets, architectures and technologies that can be used to develop web systems using MSA. Due to the architecture being relatively new; only being discussed and implemented in the last few years. There is no official consensus on how this architecture can be implemented. There are several suggestions and ideas available that are discussed and explored by members of the profession. These include suggestions such as: incorporating current principles like single-responsibility principle, that is part of the SOLID principle. Enforcing Domain Driven Design (DDD) to create bounded context for microservices to operate within and define where communication between microservices may be needed, automated testing and deployment pipelines and DevOps.

Modern technologies such as containerisation (Docker and Kubernetes) and the Cloud (Azure and Amazon Web Services(AWS)) are being suggested for use with MSA. These forms of Virtualisation aid in promoting a distributed system design - suggested for MSA. Using the Cloud for deployment purposes has contributed to the development of web-based systems. Integrated Development Environments (IDE's) such as Visual Studio, Eclipse and IntelliJ are available for this.

## Contents

<b>1</b>	<b>Chapter 1: Introduction</b>	<b>9</b>
1.1	Background . . . . .	9
1.2	Motivation . . . . .	10
1.3	Aims, Objectives and Scope . . . . .	10
1.4	Outline . . . . .	11
1.5	Summary . . . . .	12
<b>2</b>	<b>Chapter 2: Literary Review</b>	<b>13</b>
2.1	Microservice . . . . .	13
2.1.1	Services Are independently deployable . . . . .	14
2.1.2	Services are built around business capabilities . . . . .	14
2.1.3	Services are fine-grained Interfaces . . . . .	15
2.1.4	Services are designed to fail . . . . .	17
2.1.5	Microservice Architecture has a natural modular Struc- ture . . . . .	17
2.1.6	Encourages Infrastructure Automation and Testing . . . . .	17
2.1.7	Introduces the concept of you built it you own it . . . . .	19
2.2	Domain Driven Design . . . . .	20
2.2.1	The Domain . . . . .	20
2.2.2	Bounded Context . . . . .	20
2.3	Single-Responsibility Principle . . . . .	22
2.4	RESTful . . . . .	23
2.4.1	Communication . . . . .	23
2.4.2	API . . . . .	24
2.4.3	CRUD Implementation . . . . .	24
2.5	Polyglotism . . . . .	25
2.5.1	Polyglot Programming . . . . .	25
2.5.2	Polyglot Persistence . . . . .	25
2.6	Virtualisation . . . . .	27
2.7	Asp.Net . . . . .	28
2.7.1	Razor Pages . . . . .	28
2.8	Java Web Development . . . . .	29
2.9	Conclusion . . . . .	30
<b>3</b>	<b>Chapter 3: Proposed Design</b>	<b>31</b>
<b>4</b>	<b>Chapter 4: Prototype</b>	<b>39</b>
4.1	Analysis . . . . .	39
4.2	Requirement Specification . . . . .	39

4.2.1	Purpose . . . . .	39
4.2.2	Project Brief . . . . .	39
4.2.3	Scope . . . . .	40
4.2.4	Resources . . . . .	40
4.2.5	User Requirement . . . . .	41
4.2.6	System Overview . . . . .	41
4.2.7	System Architecture . . . . .	41
4.2.8	Functional Requirements . . . . .	42
4.2.9	Non-Functional Requirements . . . . .	44
4.2.10	Performance . . . . .	44
4.2.11	Glossary . . . . .	44
4.3	Design . . . . .	44
4.4	Development . . . . .	45
4.4.1	Prototype Development with separate databases . . . . .	45
4.4.2	Catalogue Microservice . . . . .	45
4.4.3	Basket Microservice . . . . .	48
4.4.4	Prototype Development with a single centralised database . . . . .	49
4.4.5	Front-End website development . . . . .	49
4.5	Testing . . . . .	50
4.6	Evaluation . . . . .	51
<b>5</b>	<b>Chapter 5: Conclusion/Evaluation</b>	<b>52</b>
	<b>Appendices</b>	<b>56</b>
<b>A</b>	<b>Project Initiation Document</b>	<b>56</b>
A.A	Example sub appendices . . . . .	56
<b>B</b>	<b>Second Formal Review Output</b>	<b>56</b>
<b>C</b>	<b>Diary Sheets (or other project management evidence)</b>	<b>56</b>
<b>D</b>	<b>Class Diagrams</b>	<b>57</b>
D.A	Authentication Microservice Classes . . . . .	57
D.B	Basket Microservice Classes . . . . .	57
D.C	Catalogue Microservice Classes . . . . .	58
D.D	Checkout Microservice Classes . . . . .	59
D.E	Customer Microservice Classes . . . . .	59

## **List of Tables**

**List of Figures**

1	Monolithic vs Microservices Architecture . . . . .	13
2	Conway's Law . . . . .	15
3	Built around Business Capabilities . . . . .	16
4	Automated testing to deployment . . . . .	18
5	Polyglot Persistence Implementation example . . . . .	26
6	High Level Design . . . . .	31
7	Bounded Context . . . . .	33
8	Interface . . . . .	34
9	Interactions . . . . .	36
10	Azure ADO.NET Connection String for the server containing the Catalogue Microservice Database . . . . .	46
11	Visual Studio Appsetting.JSON connection string. Partial View	46
12	Visual Studio Dependency Injection for the Catalogue Mi- croservice . . . . .	47
13	Visual Studio API Controller for Catalogue Microservice . . .	48
14	Authentication Classes . . . . .	57
15	Basket Classes . . . . .	57
16	Catalogue Classes . . . . .	58
17	Checkout Classes . . . . .	59
18	Customer Classes . . . . .	59



**Acknowledgements**

Insert acknowledgements here I would like to thank my supervisor, Xiaodong, for his advice and support with my dissertation. My Degus. My parents, Maya and Graham for listening to my rambling about my dissertation and attempts to explain it to them.

## 1 Chapter 1: Introduction

### 1.1 Background

Traditionally, software applications/systems are developed using the Monolithic Architecture – a unified model of the design of a software program [24]. They are designed to be self-contained: components of the program are interconnected and interdependent. For this tightly coupled architecture to work, each component and dependent component's must be present for the code to be compiled or executed. If any component is missing, there is a high chance the program will not work correctly. The rapid increase of software systems has changed the shape of how businesses function in today's world. As businesses have expanded and added new products/services, problems arose where the Monolithic code base would become difficult to maintain and any future changes could result in problems arising when implemented. For the past thirty years, the software industry has been moving ever closer to a service-orientated approach [12]. This evolution has resulted in a closer bond between businesses and IT. Instead of businesses making decisions controlled or constrained by software, they now make decisions supported by software.

Service Orientated Architectures (SOA) were the first realisation of transforming monolithic systems into small building blocks – components – that work together to create applications that are easier to maintain and expand upon. A service is defined as: a function that is well-defined, self-contained, and does not depend on the context or state of other services [1]. Around seven years ago, at a workshop of architects in Venice, the participants saw a common architectural style they had all been recently exploring. The term "Microservices" was created [9]. It describes a particular way of designing software applications as suites of independently deployable services. That can be maintained and modified effectively to meet the demands of the business world in the present day.

By describing each function, of a software system, as an individual service; they can be separated into their own components, contained within their own domain model. Doing this entails each service has a single responsibility to perform and no more. This would also entail each service to have its own database with which to manipulate data. There would be no shared relational database for the entire system. Although there may be a need for different services to communicate with each if they share specific information in order to maintain the validity of data their databases. The development of containerised software has increased the capability of polyglot programming. Were each Microservice can be written in a different language, and when deployed via containers, they combine, like building blocks, to form the software system. This form of modularity allows for each Microservice to

function independently and improves the stability of the system as a whole - if one Microservice was to fail, it would not result in the entire system from failing.

With the recent boom of the internet, particularly in the last decade, Microservices are beginning to be used by retail companies with a large online presence such as: Amazon, Netflix and Ebay. Although retail is not the only sector to use this architecture: Uber, the guardian and Capital One. As the internet has proven to be an excellent platform to deploy systems. More specifically web-based systems.

This project investigates and evaluates: designs, principles and technologies related to Microservices used too implement this architecture in developing an E-Commerce web system.

## **1.2 Motivation**

why are you doing this? needed?

## **1.3 Aims, Objectives and Scope**

The aim of this project is to explore how a Microservice architecture (MSA) can be used to develop a flexible web based E-commerce system. Finding out the flexibility such an architecture provides and how this can be used to allow developers too maintain/update such systems in the fast-paced environment of today's world. This project will concentrate on developing a web-based system as the internet has been one of the fastest growing resources available to businesses to expand and meet the ever growing demands of customers.

These aims are achieved by examining the literature available on Microservices, the principles involved to enforce the Microservice Architecture and the technologies available to create this architecture for developing software to define research questions. A software development environment is then chosen to implement this architecture in a prototype system. Literature research will include: academic white papers, professional journals, lectures, on-line articles and software books. The collected/researched information will be used to answer the questions and provide the necessary understanding to develop a prototype using the Microservice architecture. Following this, the project is evaluated, considering the original aims and objectives.

As MSA is a relatively new architecture; there is still a broad view for successful implementation. Due to time and cost constraints, this project will focus on web-based design patterns and protocols. As well as software and hardware to design a web system. There will be a limit to the exploration of applicable principles, design patterns, hardware and software available that cna be used to implement a Microservice architecture.

## 1.4 Outline

This dissertation is structured as follows:

- **Chapter 1: Introduction**

Introduces the topic of Microservices, names the aims and objectives and outlines the scope and constraints for this project.

- **Chapter 2: Literature Review**

Encompasses all the subjects and terms that are related/representative of a microservice architecture. It will go into detail about why Microservices are being used, why it is preferred over Service-orientated architecture and the technologies currently in use and those being developed. This chapter will also provide a critical and objective analysis of these subjects.

- **Chapter 3: Project Planning**

Provides an overview of the management of the project. With use of a Gantt chart. The methodologies used during the development cycle will also be described. With justification. This chapter will also include a description of the high level functionality needed for the project using the project management tool MoSCoW.

- **Chapter 4: Prototype**

This chapter contains the following sections:

- **Analysis**

This chapter will describe the requirements for the development of a prototype. And an analysis of the high-level design of the prototype. Including a diagram of the service interaction.

- **Design**

This chapter will provide a detailed design of the structure of the prototype – UI designs, Class diagrams, E-R models etc.

- **Implementation**

This chapter will detail the actual development of the prototype. Each staged will be documented. All issues/problems will also be documented along with the solutions. Also, providing a reference for all sources used. Screen shots of the development stages will be provided via appendices.

- **Testing**

This chapter will consist of all testing documentation and testing conducted on the prototype. With a description of the testing methodologies used.

– **Evaluation**

This chapter will provide a critical and objective analysis of the developed prototype. Providing a detailed description of its success/failure.

- Chapter 5: Conclusions  
Discusses the project, how effective Microservices architecture is etc. And going forward what further research etc. will be done/required.
- Chapter 6: References  
Contains all references used throughout this document.
- Chapter 7: Appendices  
Houses all the appendices. This includes Gantt charts, UML diagrams, testing documentation and screen shots of: development progress at various stages and working prototype.

## 1.5 Summary

This project will review the following principles, design patterns and technology:

- Single Responsibility Principle
- Model-View-Controller (MVC) Architecture
- RESTful Architecture
- Domain Driven Design (DDD)
- Docker for Containerisation
- Kubernetes??
- Visual Studio using ASP.net
- IntelliJ web development

Being a prototype, and a university project, the system will not implement a payment system nor will the login/authentication Microservice require any more than a login and password. No personal details will be requested. Nor will any anonymous logging for user interaction be logged/recorded.

## 2 Chapter 2: Literary Review

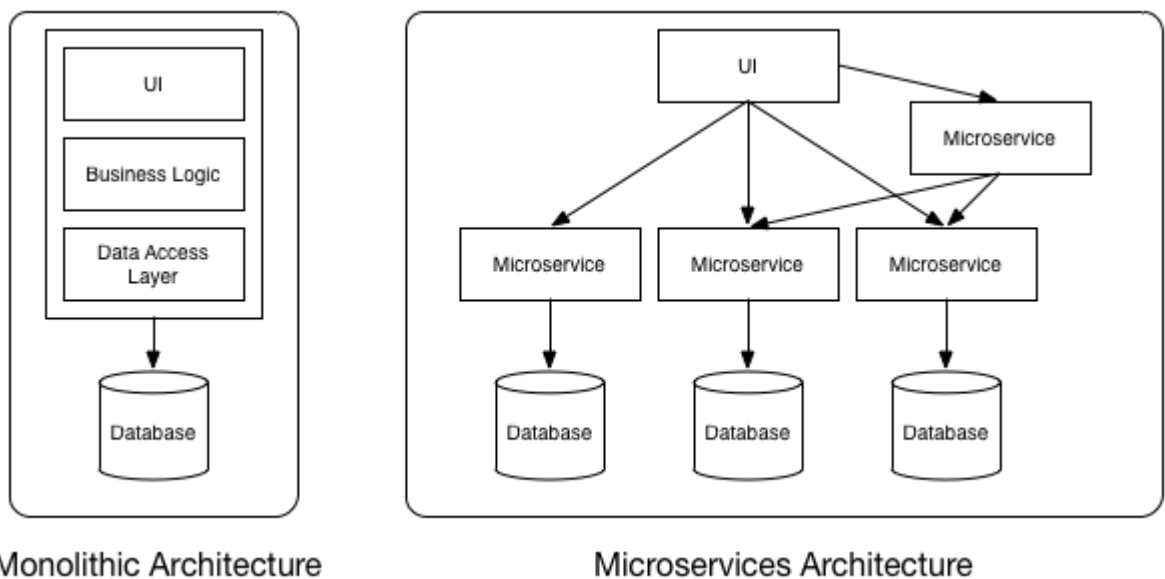
### 2.1 Microservice

This chapter discusses the Microservice architecture. It also discusses why this architecture is being preferred over the service-orientated architecture.

The term “Microservice” was first used at a software architects workshop held in Venice in May 2011 as stated by Martin Fowler [9]. Participants began to realise they were using and exploring the same architectural style. It was a year later it was decided as the more appropriate name [9]. As Microservices Architecture (MSA) is a relatively new architecture, there is no official industry consensus regarding the properties nor definition of MSA.

According to Sam Newman [20] Microservices are: “small, autonomous services that work together”. Martin Fowler [9] continues this definition by saying “Microservices are a way of designing software applications as suites of independently deployable services”. Microservices help break the boundaries of large applications and create smaller systems (the services) that are used to build applications as seen in figure 1. Tomas Cerny et al [2] have described

Figure 1: Monolithic vs Microservices Architecture



Microservices as being based on three Unix ideas:

- A program should fulfil only one task and do it well
- Programs should be able to work together

- Programs should use a universal interface

The implementation of MSA is open to interpretation. Though there are some defining characteristics that are commonly cited [15] [9] [2]. Although Zimmerman in his paper [30] explores the argument that the Microservice architecture itself is in fact not an Architecture but an implementation of the Service Orientated Architecture. But does state there is, currently, no consensus on the relationship between MSA and SOA.

### **2.1.1 Services Are independently deployable**

Unlike traditional monolithic applications that contain several modules that may be dependent on several libraries, environment etc. Each service, within a Microservice architecture, can be deployed independently. All dependencies: database, library dependencies and execution environments such as web servers or Virtual Machine (VM) are contained within each service. This ability is what enables each service to deploy independently and be essentially autonomous as described by Gupta [15]. Therefore, each individual service will only contain the dependencies it needs. And can be deployed for use and perform its intended function. Even if no other service is available, the deployed services will perform their intended function, barring failure: hardware, software etc.

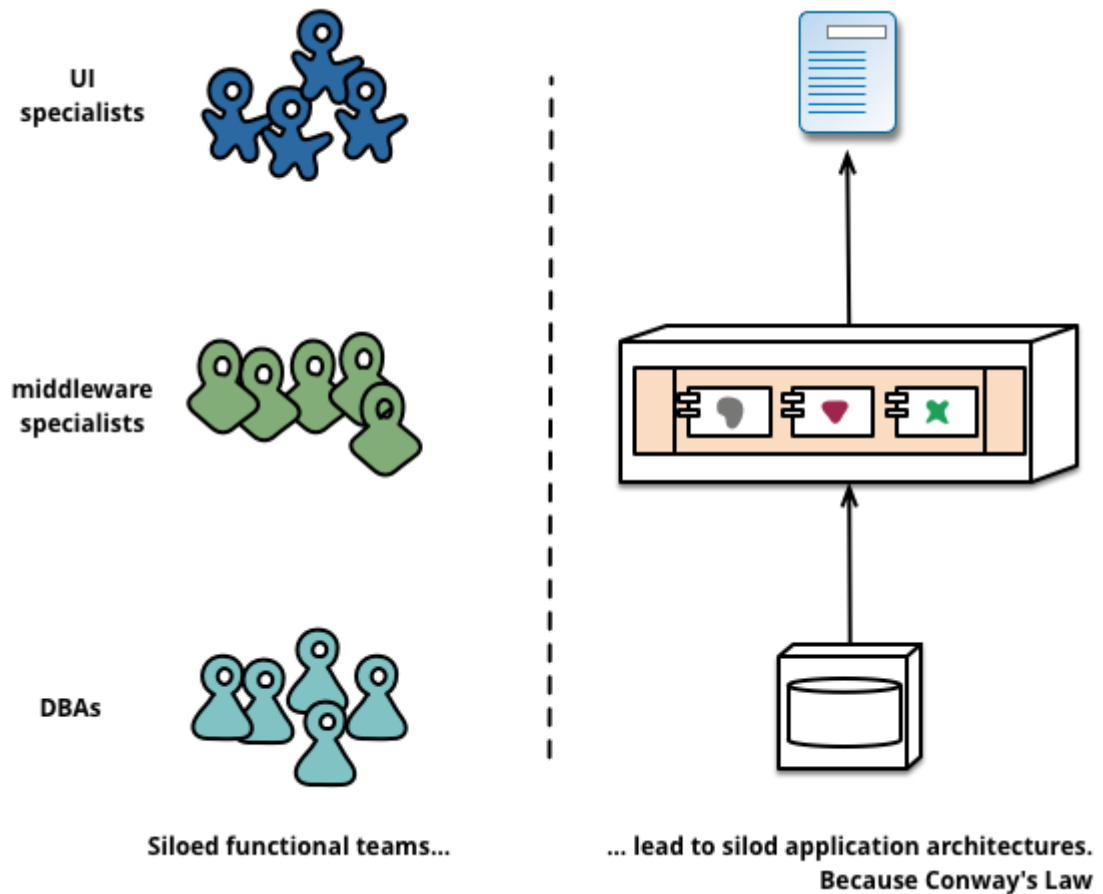
As services can be deployed on different machines, a distributed system of services, this can help with fault tolerance. If only a single service is on a machine that fails for whatever reason. Then it will not affect the rest of the services increasing the fault tolerance of the system or program.

Each of the independent services deployed combine to make the intended application(s). Such as an E-Commerce web application, news web application etc.

### **2.1.2 Services are built around business capabilities**

The more recognised model of focusing on the technology layer of applications results in creating different development teams for: UI, server-logic, database etc. Having such separated teams separated like this, simple changes to any aspect can result in delays of development due to cross-team communication, budgets etc. This results in an application created that follows Conway's law [3]: "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." See figure 2. The Microservice approach changes the view of division with development teams. This approach involves organising teams around developing services based on the businesses capabilities – accounts service,

Figure 2: Conway's Law



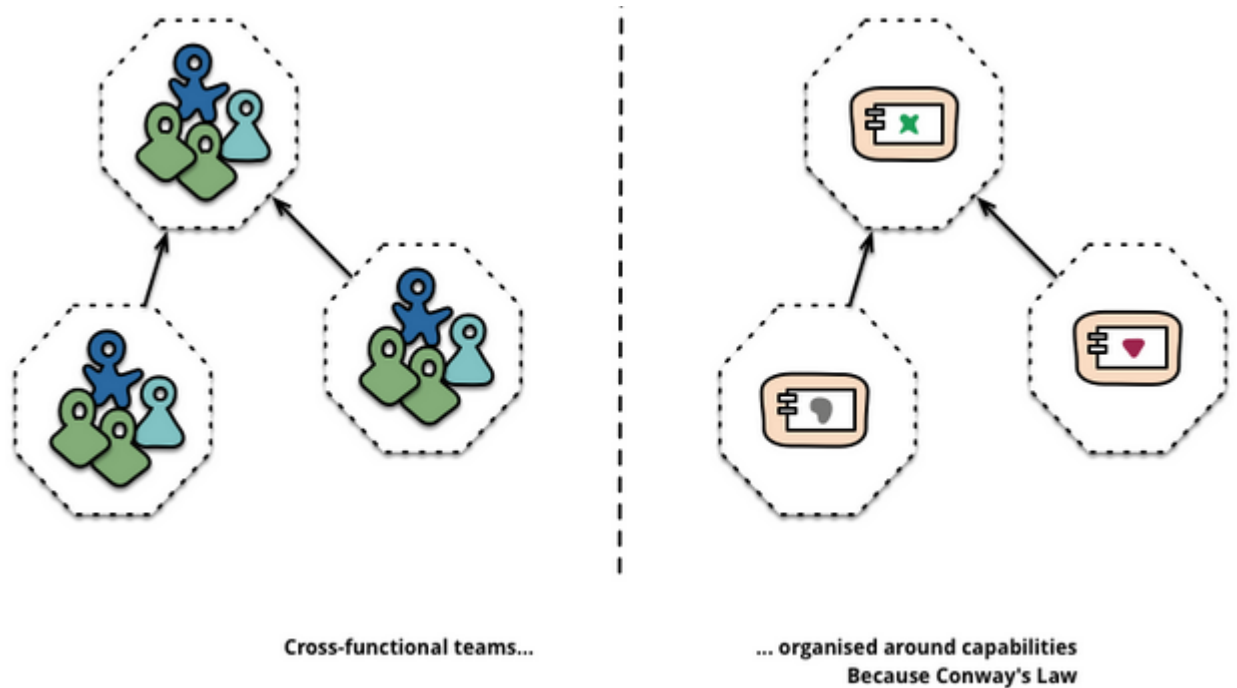
basket service, payment service, catalogue service, car insurance service etc. These services implement the software needed for that business area: UI, storage, external collaborations etc. The teams become cross-functional consequently, including the full range of skills required for development: database, project management, UI etc. As described by Martin Fowler [9] in figure 3.

### 2.1.3 Services are fine-grained Interfaces

Small, performing only one function – each service is designed to have only one responsibility, perform a single functionality of the business. This single responsibility principle also encompasses the database in use. Instead of having a single database used in a monolithic system, each Microservice can have its own database to store the data. This is a change from the Monolithic style where a single database would be used to store the systems



Figure 3: Built around Business Capabilities



data. Although, several Microservice could access the same database, again, depending on the context of the software system.

The services are described as “fine grained” referring to the granularity of the Services themselves. The actual size of the service varies depending on the context of the business. How small or fine grained a service must be is open for discussion. A general agreement is that the codebase of the service should be manageable by a small team of people. For example, Amazon [10] has coined the phrase “the two-pizza teams” – each team numbers around 8-10 people. The number you can feed off two pizzas . Meaning the code base for the Microservice would only be “big enough” for this number of people to handle. Similarly, Jon Eaves, of RealEstate.com.au, characterizes a Microservice as something that could be rewritten in two weeks as stated by Sam Newman [20]).

Zimmerman [30] reinforces this definition of: “Services that can be deployed, changed, substituted and scaled independently of each other” and states fine-grained services as one of the seven common tenets cited in introductory literature and case studies on Microservices.

#### **2.1.4 Services are designed to fail**

By creating services as components, applications that use these services need to be designed so they tolerate any failure of services [9]. This introduces another layer of complexity when designing microservices. Services can and will fail, but the effect on users must be limited.

It is important to detect failures as quickly as possible and restore them, automatically if possible. The MSA puts emphasis on real-time monitoring of the application; checking architectural (database requests per second) and business relevant (orders per minute received etc) metrics [9]. Monitoring can lead to early warning signs of a failure, that will allow development teams to proactively investigate.

This will likely lead to complex, and expensive, monitoring systems in place – The Microservice teams need to know which services, running in different processes, have failed. This is generally achieved by having separate, sophisticated monitors and logging set-ups for each service available. Monitoring such things as: up/down status and operational/business relevant metrics [9].

Famously, Netflix created a set of tools dubbed the “Simian army” that were developed solely to generate various kinds of failures and test the robustness of their system. [13].

#### **2.1.5 Microservice Architecture has a natural modular Structure**

As described previously, each service is a component of the system and, thus, enforces a modular structure where each service is seen as a module. This implementation of Modular Programming [29] encompasses many of its benefits:

Each service can be used many times by many users. This benefit is adhered too as the code written, for each service, is not repeated. Only the service needed is called upon, through events etc., and executes the required task(s).

This modularity allows for each service to be used in conjunction with other applications. As service independence is a requirement of the MSA, the services can be used in the creation of multiple applications.

A disadvantage with the modular structure of the Microservice architecture occurs in the debugging of the system.

#### **2.1.6 Encourages Infrastructure Automation and Testing**

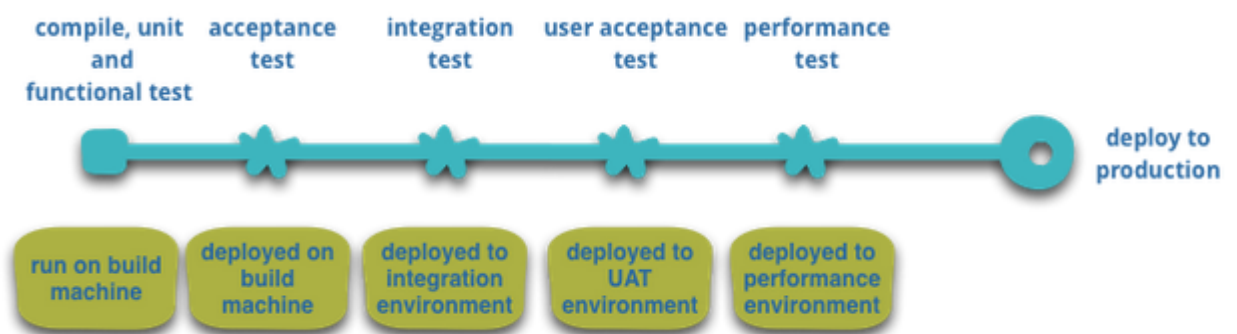
As different businesses have different needs, the number of microservices developed can differ greatly. As the Microservice architecture adheres to

the distributed systems model; microservices will be deployable on multiple servers/machines and it can become difficult, almost impossible, to develop, test, deploy, monitor each Microservice manually. Regardless of the size & number of software development teams. The answer to this is to introduce automation.

The development of the Cloud and Amazon Web Services (AWS) have reduced the operational complexity of building, deploying and operating microservices [9]. These infrastructure automation techniques have evolved extensively over the last few years.

Current systems built using microservices are development by teams with extensive Continuous delivery (discussed later) experience [9]. Software built using this principle make extensive use of the above infrastructure automation techniques. This helps to streamline and automate the testing of builds. See figure 4. Current systems built using microservices are development

Figure 4: Automated testing to deployment



by teams with extensive Continuous delivery (discussed later) experience (Fowler, 2014). Software built using this principle make extensive use of the above infrastructure automation techniques. This helps to streamline and automate the testing of builds.

Testing and deployment automation brings several advantages:

- allows teams to choose when they want to deploy the microservices they are in control of. Potentially deploying new builds multiple times in a short period of time.
- Newly developed/updated features are in the hands of customers faster
- Allows for frequent testing of new features

- Teams can experiment new processes, algorithms etc. Lokesh Gupta [15] describes this as Enterprises being given the ability to experiment and fail fast.

But there are disadvantages. A potential risk to continuous deployment/integration is “moving fast and breaking things” [11]. If backward compatibility is not diligently adhered too, for versioning deployments, a change can be released that results in the Microservice breaking and affecting consumers.

### **2.1.7 Introduces the concept of you built it you own it**

Depending on the scale of the services provided by companies, there can be hundreds of microservices within a company’s software system structure. For example, Amazon, one of the first companies to embrace MSA, has over 150 services and can up to 100 of these services to build a single web page [10]. Services include: Search, register, account, catalogue, refine, recommendations etc.

## 2.2 Domain Driven Design

This section covers Domain Driven Design. Software development is made complex by multiple factors. One of these being the problem domain.

### 2.2.1 The Domain

Eric Evans [5] describes the domain, of any software program, as the representation of a real-world domain. Such as an airline booking program booking people onto an air plane, a finance software system dealing with money and finance. He goes on to explain that the domain is formed from models that are a simplification of aspects of a real-world domain.

### 2.2.2 Bounded Context

The bounded context of Domain Driven Design is described as the boundary of a coherent part of the business by Andreas et al [4]. Martin Fowler [8] provides a similar description when he says that the bounded context is necessary when creating a software system based on a large business domain. As it is difficult to build a single, unified model. Eric Evans [5] has a similar view point to this. Each bounded Context defines what is to be included within it, what is not and its relationship with other bounded contexts, this includes communication. Each bounded context, in a software system, contains a model that is a representation of a real world aspect of the business domain. The model would contain classes, attributes and methods that describe domain concepts. Martin Fowler explains: "To be effective, a model needs to be unified - that is to be internally consistent so that there are no contradictions within it." [8] For example (should I include this?), In the context of this project. A bounded context for the software system could be a customer support/services Context. The Context-model of this business Domain could contain the following classes:

- Customer  
Contains the attributes that represents a customer - name, contact details etc.
- Ticket  
Attributes including ticket number, type of issue, description, order number related to it etc.
- Ticket communication  
Contains information on what has been done to resolve the issue

- Product

Provides a description of the product - name, type, price etc.

From this example, it can be deduced that other bounded contexts will be created, to model the other aspects of the business domain and these contexts would then combine to model the actual business domain itself. Some of the bounded contexts modelling will contain similar classes, with similar attributes. Such as the above Customer and Product Classes. These would be similar to, for example, a context of the sales context. This model would contain a class for customer and product. The former being the customers identity and the latter the description of the product. There would be some form of communication between these contexts to ensure the data help in each model is the same, were shared. So both contexts are consistent within themselves and as models of the domain.

As each Bounded Context is conceptualised, the individual Microservices can be discovered. There are issues with using this design for discovering Microservices. Florian Rademacher et al[22] discusses how using Domain Driven Design omits the following information when deducing the Microservices and their characteristics:

- Interfaces and operations
- operational parameters and return types
- endpoints, protocols and message formats

### 2.3 Single-Responsibility Principle

Single Responsibility Principle (SRP) is one of the SOLID Principles described by Robert Martin [16]. He proposes this principle follows on from Conway's Law: the structure of a software system mirrors the communication design of the business that uses the software and that each module should only have one reason to change. And each software module should only be responsible for a single actor, not do just one thing. Each Microservice should be its own module, within MSA, and have a single responsibility to provides its intended service a single actor - be that for a user, stakeholder, internal software process, business policy etc. This design principle ensures each microservices is easy to understand and maintainable: allowing for changes to occur based on the requirements of the users. Another benefit provided is each service can be used as the basis of components to be used in many different software systems. Creating the notion that services themselves are modular in design. Using this principle in developing microservices enforces cohesiveness. Each class, function etc of a Microservice combine to form the Microservice. Microservices take the single responsibility a step further by also including the data storage (database) as part of the principle. Monolithic software systems share a single, large database for data storage. The Microservice Architecture is designed so each Microservice has its own database, purely to store the data it needs. Nothing else.

This addition to the SRP ensures there is very little, if any, coupling between Microservices. And prevents any actions from a Microservices affecting any other Microservice of the system.

## 2.4 RESTful

Representational State Transfer (REST) is an architectural style for systems built on the web [25] & [20]. REST is used to build on protocols and standards like HTTP. It uses HTTP, and similar protocols, to do more than static web content. Unfortunately there is limited information available in regards to RESTful implementation. Possibly due to academic papers/books/research being unavailable at the time of writing this report or there is relatively little to write about it.

Using the verbs available through HTTP, the REST architecture style prevents the need to create a multitude of different methods to do the same thing [20]. For example, an object of a new customer created would only need to call the verb POST to request the server create a new resource and perform the request. GET would only need to be called to retrieve the representation of a resource.

REST can also make use of the large HTTP ecosystem: caching proxies, load balancers, monitoring tools, security protocols etc. These allow REST architectures to handle large volumes of HTTP traffic and route them in a transparent and fair way [20]. And handle the security of communications from basic authentication to client certificates.

It uses the HTTP verbs to manipulate resources. This can be used to implement the CRUD (Create Read Update Delete) principle when using databases.

Newman [20] describes the most important concept of using REST is resources. Resources are viewed, by services, as a "thing" it knows about. The resources can have different representations, depending on the server request. This flexibility allows complete decoupling from the external representation of a resource to how it is stored.

### 2.4.1 Communication

Each service can be deployed on different machines or they can all be deployed from a single machine. Depending on the quantity of microservices created and the hardware/software availability of businesses. Even though no service should have a dependency on another service, it is accepted that, at times, services may need to communicate with each other to update information stored in the services' database. This type of communication can be done in numerous ways:

**2.4.1.1 Synchronous** This type of communication involves communicating with a remote server and transferring blocks of data in a continuous and consistent timed manner [26]. Primarily designed for transmission of



large blocks of data, it is real-time, bi-directional communication between the client and server.

Microsoft [18] describes the main disadvantage of this form of communication is the remaining code to execute, from the client side, must wait for a response to this communication before the rest of the thread can execute. This could result in delays and high levels of Latency.

**2.4.1.2 Asynchronous** Sam Newman [20] describes this form of communication is inherently event driven. The client does not request an initiation for things to be done. Instead the client states something has happened and then assumes the other parties involved (server side etc.) know what to do. Asynchronous communications are inherently decoupled. Microsoft developers [18] add to this by stating The client code sender does not wait for a response to continue with code execution.

## **2.4.2 API**

Using Application Program Interfaces in a web-based MSA system allows decoupling of the microservices themselves. By this, each Microservice can be only accessed through its corresponding API only. Providing a means to decouple the services as the only, potential, interaction between each would only be done through the API, not actually accessing the service directly. This style of implementation can be achieved with use of an API gateway.

The Front-end user interface will pass requests, from the user, to the gateway which will then implement the required Microservice through its API.

## **2.4.3 CRUD Implementation**

CRUD, an acronym for Create, Read, Update and Delete. These represent the four basic functions for Persistent storage. Often used with SQL to manipulate data in relational databases, and more recently NoSQL database implementation. CRUD is synonymous with the HTTP verbs: PUT, GET, SET and DELETE. Both go hand in hand.

## 2.5 Polyglotism

### 2.5.1 Polyglot Programming

Believe to be first introduced in 2006 by Neal Ford [23]. Polyglot programming is the practice of writing code, for applications, in multiple languages [6] as applications are becoming more complex, in design and functionality, different types of problems occur. This is done to add functionality, efficiency and to tackle specific problems [7], to applications, when a single language can't provide everything required.

For example, building an E-commerce website would likely involve developing the system using: C#/Java, Python, JavaScript and HTML5. The document formatting language (CSS) and data query language (SQL) could also be used. JavaScript, HTML5 & CSS would also be used for user Interface design. C# and/or Java can be used for Object-orientated functionality etc. SQL would be used for database manipulation/interrogation. Python could be used to provide security protection for applications.

The microservice architecture, being a distributed system, can incorporate polyglot programming extensively. Besides the above mentioned, various microservices could be written in different programming languages, to take advantage of functionality only present in them.

However, including Polyglot Programming into an application is likely to add complexity to the system. Developers will need to learn the range of languages present or teams will need developers proficient in the required languages.

### 2.5.2 Polyglot Persistence

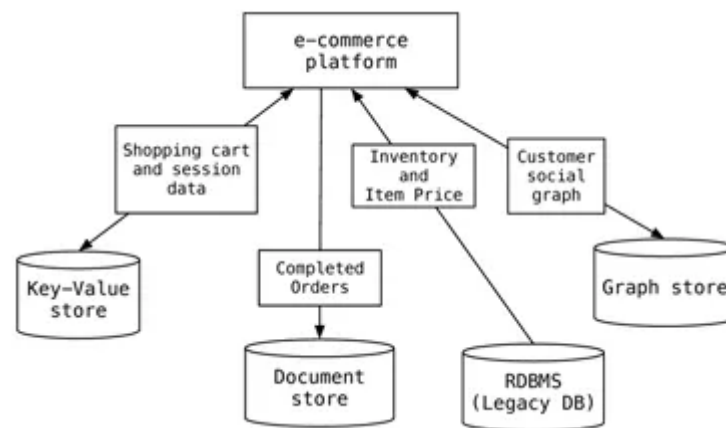
Believed to have been first conceived by Scott Leberknight [14]. Polyglot Persistence is the practice of having various database technologies to store persistent data. With the introduction of NoSQL Databases, and other non-traditional data stores, the move away from using only a Relational Database Management System (RDBMS) to store data began.

Similar to polyglot programming, the increased complexity of applications also incurs problems when it comes to storing data. As the variety of data to be stored increases, such as big data, the traditional structured RDBMS is not always suitable. Applications may need data to be stored regularly, if there is multiple users using its services at any one time.

NoSQL databases allows for a variety of unstructured data to be store successfully. Data, keeping in context of this project, such as a customer's basket would not be consistent in what it stores - each item in the basket will possibly have different information stored about it. Adding the fact multiple

customers will be using an E-commerce web system at the same time, the velocity of data inserts will be potentially be fast and NoSQL Databases (DB's) can handle this well. They provide Horizontal and vertical scaling. Providing greater benefit than most RDBMS which are limited to vertical scaling As seen in figure 5 [27]:

Figure 5: Polyglot Persistence Implementation example



As with Polyglot Programming, Polyglot Persistence adds complexity as each data storage technology added involves learning how they work [27].

## **2.6 Virtualisation**

blah

## **2.7 Asp.Net**

blah

### **2.7.1 Razor Pages**

blah

## **2.8 Java Web Development**

blah

## 2.9 Conclusion

In Summary, the literary review has provided material enough to describe the benefits associated with the Microservice Architecture and the disadvantages.

The Microservice Architecture is an effective implementation of the Service-orientated Architecture. There is evidence to support it is its own architecture aswell.

The separation of services into individual, distributed modules does have its benefits in terms of being able to identify where failures/bugs arise and fixing them quickly. The notion of each service representing a different business capability allows for team

Of note is the different take on the use of databases. This project will develop a prototype with both styles of database available. Each will be tested individually for all available CRUD implementation for each Microservice. Recording the response times for each and presenting this data, in a visual form, to show which is faster and which style should be preferred for Microservice Architectures. This could be done, internally in the code, using the stopwatch class or using the performance monitor in Azure cloud services.

The following will be used to evaluate this project:

- The proposed design of the system  
involving class diagrams, Microservice interactions, domains etc.
- A successful prototype developed  
Developed based on the proposed design, with an evaluation including the difficulties and easiness in adhering to the design. With a discussion on the advantages/disadvantages of the design. Looking into potential improvements.
- An evaluation of both database styles: centralised and separate  
This will be conducted by simulating use of the website, using testing software described previously, and displaying the gathered data of when the different database styles are queried. Displaying response times and comparing them to help determine why the separated database design is chosen over a centralised one.

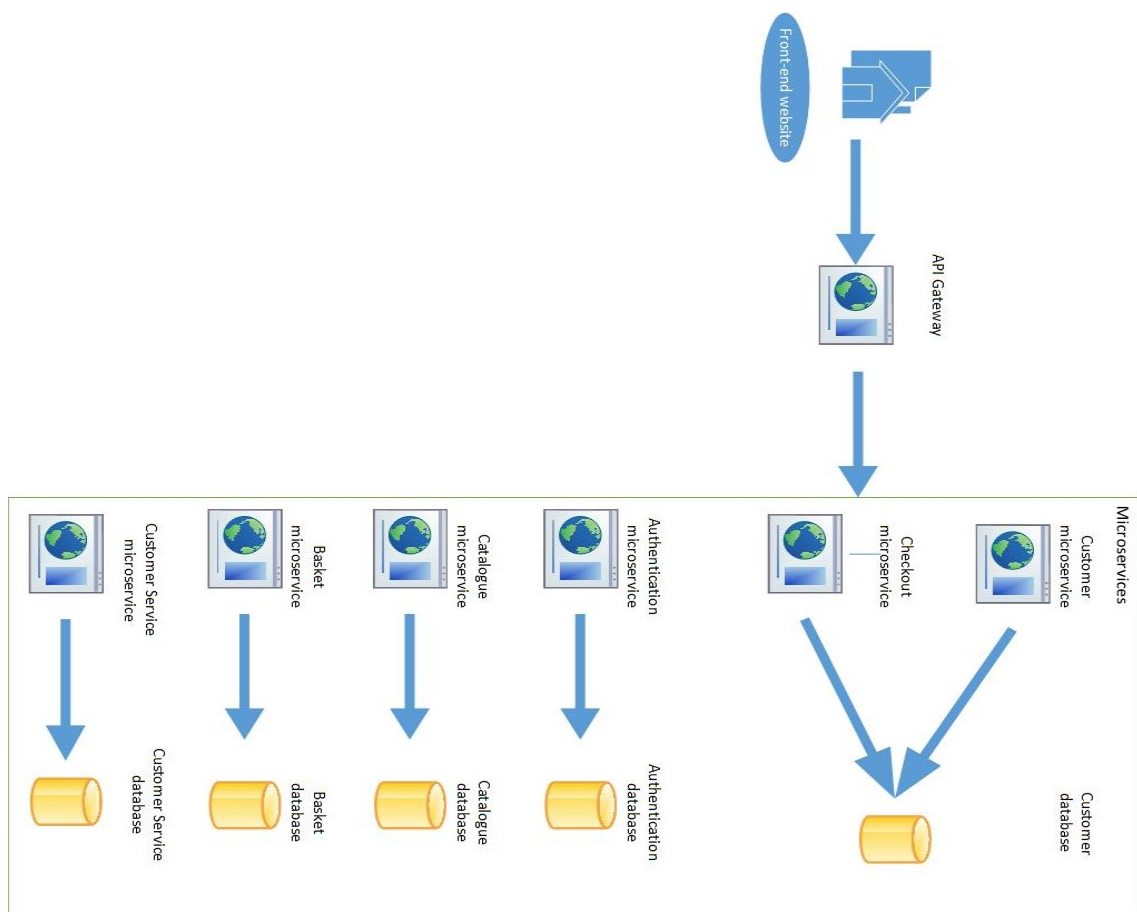
### 3 Chapter 3: Proposed Design

The proposed design for this project will consist of the following:

- A Front-end web application
- An API Gateway for the web app to access the Microservices
- Microservices are accessed through their own API
- The system created will consist of two different styles of database: a centralised DB and the "traditional" separated Db style associated with Microservice Architecture

The high level design of this system See figure 6.

Figure 6: High Level Design



This diagram shows how the front-end web system will interact with the Microservices. The website will be created using the MVC pattern. With



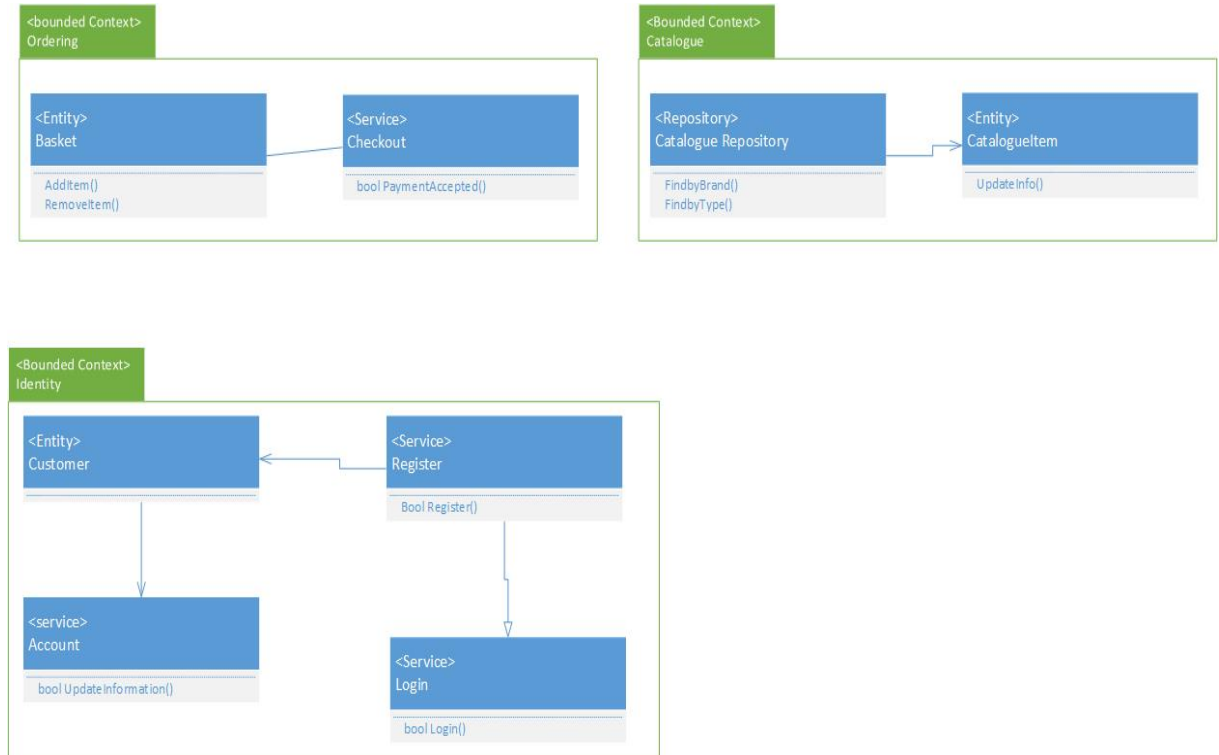
each section of the UI needing to interact with a service(s) is done through the API Gateway to the microservices own API. An API Gateway will be chosen as the entry way for access to each Microservice. An API Gateway has been chosen as it seen as the best option to incorporate expansion - such as adding more Microservices, introducing different User-interfaces (Mobile version/Single Page Website etc.) this way the connectors, to each Microservice, are removed from the website. This prevent unnecessary complexity in the website instead having this present in the Gateway which will be more manageable.

From here Each Microservice will be accessed through its own API. That only the API Gateway will be able to access. Adding a level of protection and ensuring each Microservice is not coupled with the website. By incorporating API controllers, The RESTful implementation for each Microservice can be controlled. Removing any redundant "verbs" for each service. Although this Prototype will run mostly form a single machine. It is entirely possible for each Microservice to be running on a separate machine,, fully embracing the distributed system style. Also, the project will look into incorporating Docker to containerise each Microservice, API Gateway and the website. To explore this option and emphasise the requirement for a microservice architecture-system to be distributed.

The bounded Context for this system will be: see figure 7 There was three bounded contexts identified for this prototype:

- Ordering  
This context covers the Basket Entity and Checkout Service. The Basket is the representation of a literal shopping basket/trolley that a customer will "fill-up" with selecting items. Naturally allowing the user to add, remove and update items they have added to their basket The Checkout is the representation of the user buying the items in the basket. As in the physical world, a customer will go to the checkout to buy the items they want. Here is where the transaction will occur in the system.
- Catalogue  
The Catalogue context will contain all the information for each item available for purchasing. Representing a warehouse, the repository was identified to be used here that store all catalogue items. Each individual Item is also recognised and represented as an individual entity. Each item has its own Entity. And the Repository consists of these Entities.
- Identity  
This is the representation of the customer and all associations with

Figure 7: Bounded Context



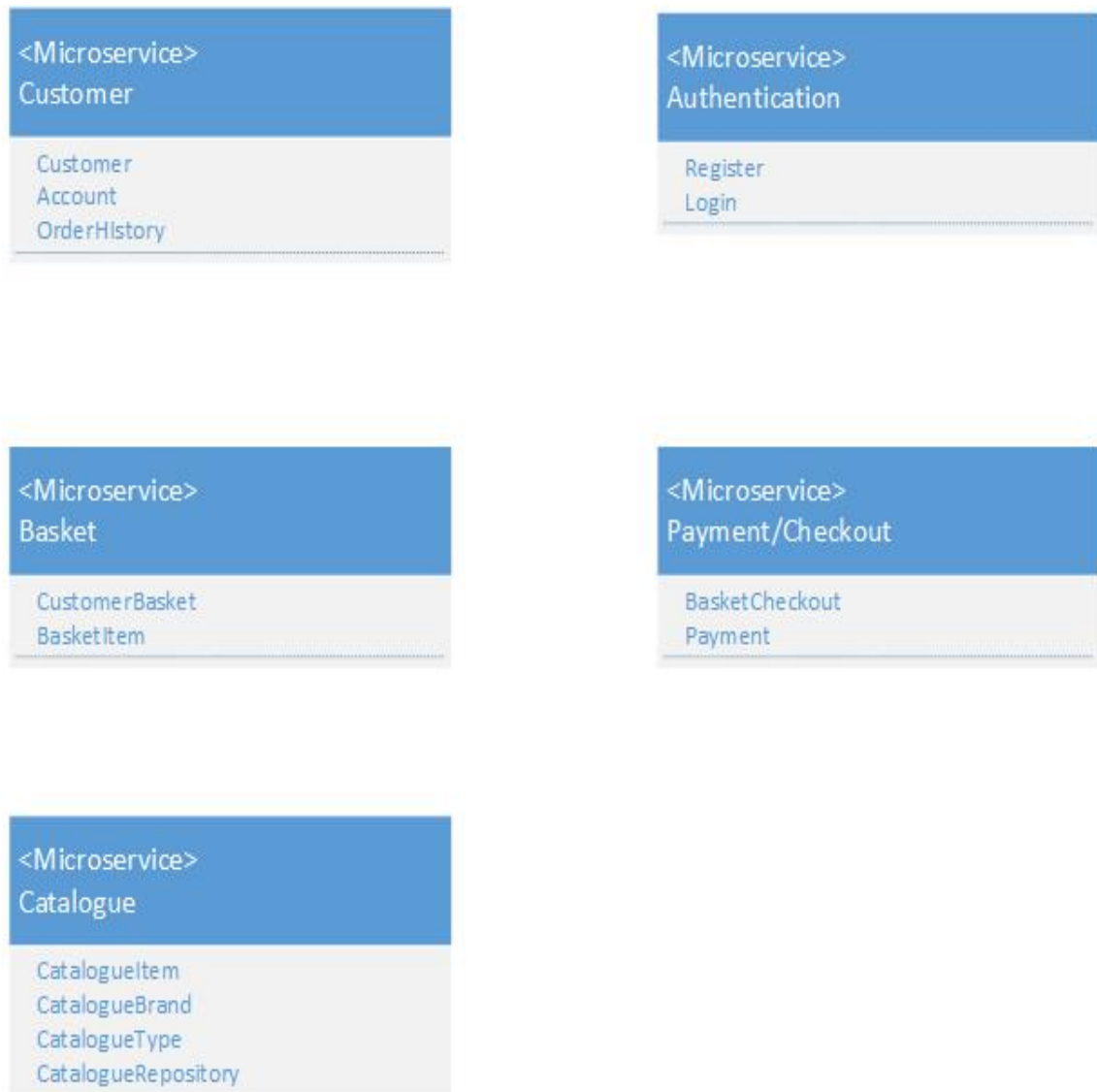
the customer. Where the customer will be an entity stored within the database that will contain the personal information of each person who has registered with the website. This will also be what the service Account mostly comprises of: the ability for customers to edit their information as when required. The login and register services are self explanatory: the register service is the portal for users to register as "customers" with inputting the necessary information, this is where the user's customer entity is created and stored in the customer database. Login service is the portal for users to login using the password they created and Email address they registered with.

NOTE: For the purposes of this prototype the registers users information will be generic.

The chosen interfaces for each Microservice: see figure 8

This diagram shows the composition of each microservice

Figure 8: Interface



- Customer Microservice  
This contains all the classes needed to represent this microservice. Customer class that stores all the users personal information, the customer ID being the primary key of the record automatically generated when the user registers. The order history class is the representation of each

order placed by the user. Note to be taken that each order may comprise multiple items. Account class is the collection of all previous orders placed and the user's information. Allowing the user to view their previous orders and edit their information.

- Authentication Microservice

This contains the classes for a user to login and register. The basis for the authentication for this system. The register Class forms the entity that will compose the customer object from the respective class.

- Basket Microservice

This interface's classes are linked with each other. Each item added to the basket will be created as an object of the BasketItem class and the customer basket will contain all the BasketItems every time one is added/removed/edited.

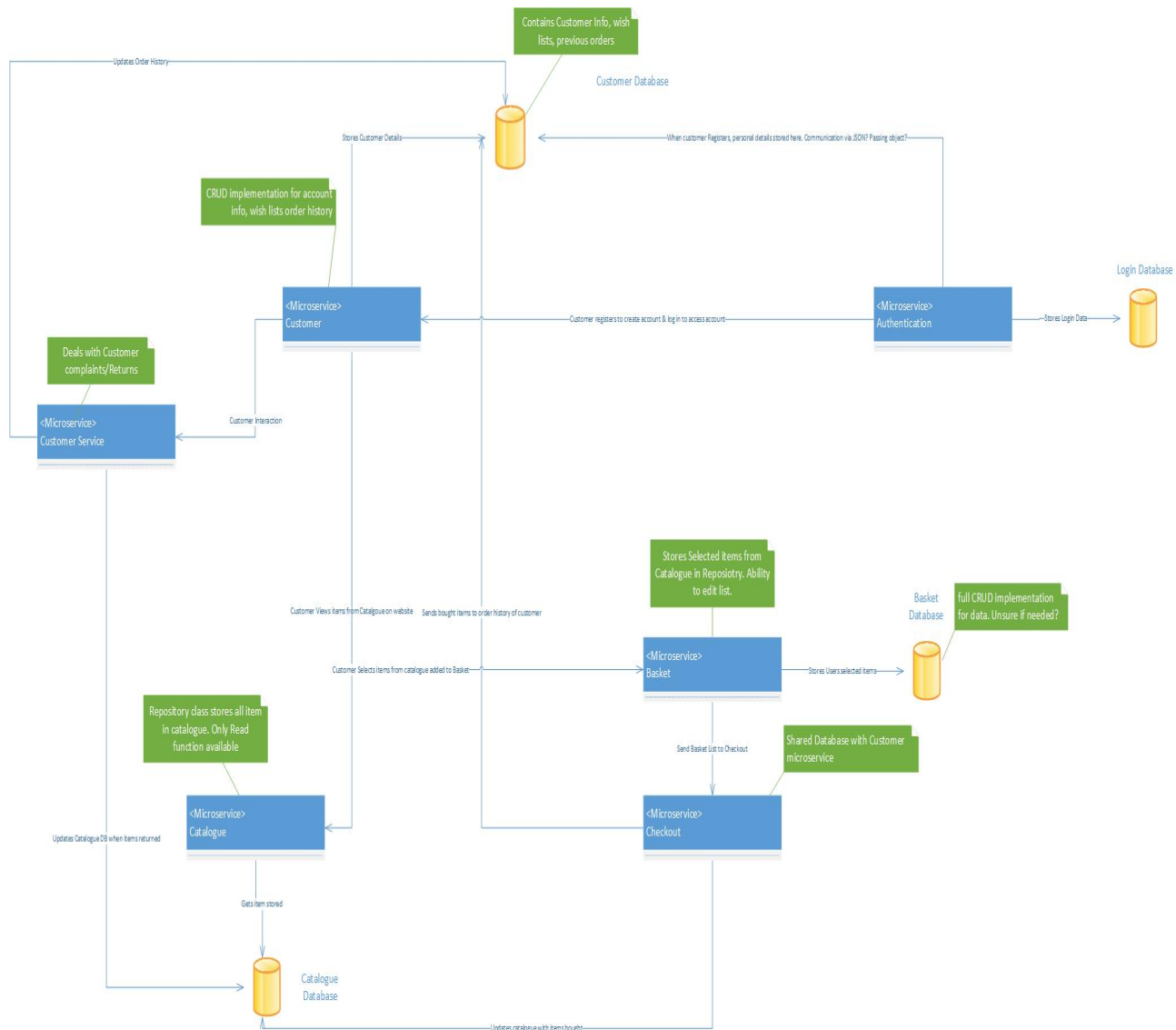
- Checkout Microservice

- Catalogue Microservice

This microservice will house the entire catalogue of items available for sale. Each Catalogue Item will be created as an individual object, each being added to the catalogue repository that will be used by the website, by consuming the Catalogue API, to display each catalogue item to the user. The CatalogueBrand and CatalogueType classes will be used to search for specific items based on brand or type.

The microservices will interact in the following manner: see figure 9  
Microservice Interaction is complicated. As can be seen by the diagram. To

Figure 9: Interactions



date, a more suitable modelling diagram for microservice interaction has not been found. Microservice Interaction is thus:

- Authentication Microservice

This microservice will store the login details within its own databases, Email and password, and sending the full set of information entered to

the customer Database. When a user enters their login credentials, this is checked with current records in the login DB and either successful or not.

- Customer Microservice  
Gets the customer information from authenticate service and stores it within its own database. Receives the object of bought items from the checkout service and stores them as part of the order history. When a user creates a ticket for the customer service Microservice this service sends the customer ID to be attached to the ticket for identification purposes.
- Basket Microservice  
Receives an updated object for an order if anything was changed with a previous order such as an item returned etc. The record in the customer DB is updated to reflect changes.
- Checkout Microservice  
Receives the repository of basket items from the Basket microservice  
Sends the basket repository, from a successful checkout process, to the customer microservice.
- Catalogue Microservice  
Gets all catalogue items from the database to be viewed. Receives updates on items from customer service and Checkout Microservices when stock numbers increases/decreases and the amount.
- Customer Service Microservice  
Receives customer ID from Customer Microservice to for customer ticket. Sends updated order to customer if anything has been changed. Sends updated stock count of items from an order to the Catalogue Microservice if anything has changed.

Class diagrams for the Microservices: figure 14, figure 15, figure 16, figure 17, figure 18.

## **4 Chapter 4: Prototype**

### **4.1 Analysis**

### **4.2 Requirement Specification**

The following Requirement Specification was devised from the analysis:

#### **4.2.1 Purpose**

The purpose of this document is too detail the requirements of the project's Web system Prototype. The Requirement specification contains the following headlines:

- Project Brief - provides a description of the web system prototype.
- Scope - details the size of the project
- Resources - What suitable resources have been sourced to implement the prototype
- User Requirements - Provides a description of the services provided to users
- System Overview - contains an overview of what the system will do
- System Architecture - provides a high-level understanding of the design structure of the web system
- Functional Requirements - Details what each Microservices function
- Non-Functional Requirements - details the design of the front-end web User Interface (UI)
- Performance - contains the expected performance of the system

#### **4.2.2 Project Brief**

The prototype to be created will be an E-commerce web system. The web system will allow users to perform actions commonly associated with E-commerce web systems.

This prototype will provide basic functions commonly found in E-commerce websites:

- A simple authentication service in the form of register and login
- Search for specific catalogue items



- Add items to Basket
- manipulate items in the basket - edit/delete items
- Deal of the week

These functions will form the bounded context of Microservices within this prototype.

Due to this being a prototype system and the constraints of being a university student, in terms of confidentiality and data protection, this prototype will not be providing a means to buy items. And the register function will only ask for a user name and password to be created by the user. no personal data will be requested.

#### **4.2.3 Scope**

The web system will allow users to register a login name and password to use the features of the site. The catalogue available for browsing will be limited to a few items under the following categories:

- clothes
- toys
- Electronics
- books
- miscellaneous

Users will also be able to add more than one item to their basket, selecting the quantity of each item they want to add.

The web system prototype must be usable for all major web browsers. The User Interface (UI) will be a simple and non-complex design so users can use it successfully.

This prototype will be finished by -Insert date xx/xx/2019-

#### **4.2.4 Resources**

This project will use the following development software:

- Microsoft Visual Studio - to develop the web system prototype using asp.net
- Programming code - C# will be used to write the coding of the software and SQL will be used for all database query command

- Microsoft Visio - To create the UML diagrams used for the Analysis & Design stages
- TeXstudio - Used to write up any documents required
- SQLite/Entity Framework - this will be used to store necessary data permanently

#### **4.2.5 User Requirement**

#### **4.2.6 System Overview**

The project will a web system that can run on the main web browsers available: Mozilla Firefox, Google Chrome, Microsoft Edge, Opera, Safari.

The Web system will provide several web pages that provide the user with different information and functionality depending on that web pages requirements.

Each Microservice will perform a single function that is derived from the business domain and bound within the context of that domain.

To simulate a distributed system, the web system and some microservices will be separated and treated as a client-server set up. Where by the Microservice(s) will be communicated with over the local host of the machine used for deployment.

Visual Studio will be used to develop the web system.

The business model, databases, will be created using the Entity framework or SQLite. And each Microservice will have its own database that it is solely responsible.

#### **4.2.7 System Architecture**

The web system will incorporate the following architectures and design principles:

- **Microservice Architecture**  
This will be used for the high-level design of the website.
- **Model-View-Controller (MVC)**  
This layered Architecture style will be used within the front-end user Interface. This layered Architecture will also be the entry point to access the API gateway to access the required Microservice.
- **RESTful API**  
Used within the MVC architecture to provide the communication between the UI and Microservices in regards to database usage. Implementing GET, PUT, DELETE and POST HTTP methodologies.

- **API Gateway Pattern**

Used to provide the access for the front-end User Interface to the individual Microservices available. The API gateway will provide the methods needed to access the required Microservice.

- **Domain Driven Design (DDD) Principle**

The model layer will be created using this principle so each model will be constructed to contain the business domain it was constructed for and each model will have its bounded context.

- **Single Responsibility Principle**

Will coincide with the above principle to be enforced and ensures each Microservice has only one main function to perform and, thusly, will only require minor changes for future expansion/maintenance

#### 4.2.8 Functional Requirements

The functional requirements will display the microservices to be developed and the functionality associated with each. The web system will provide the following Microservices:

- Authentication service
- Catalogue service
- Search Service
- Basket service
- payment service?
- Administrator Service
- Customer service service
- customer service

And provide the following functionality:

##### 4.2.8.1 Authentication Service

- Provide users the ability to register
- Provide users the ability to login
- identify type of user logging in - customer or administrator

**4.2.8.2 Catalogue Service**

- Fill the home page with a selection of available items
- Provide a detailed description of each item selected by a user
- Allow the user to select more than one of an item to buy
- Allow users to add items to their list

**4.2.8.3 Search Service**

- Allow users to search for specific items from the catalogue

**4.2.8.4 Basket Service**

- Allow users to view the items in their "basket"
- Allow users to edit their basket - delete items and increase/decrease the number ordered of individual items

**4.2.8.5 Payment Service**

- Allows users to view their items they wish to buy
- Provide Users with options on how they want items delivered
- Provide Users with multiple Payment options
- Display an invoice of the order made
- send an email confirmation of the order to the customer

**4.2.8.6 Administrator service****4.2.8.7 Customer service service****4.2.8.8 Customer service**

- Provide Customers the ability to view previous orders
- Provide Customers the ability to view lists they have created
- Allow uses to change their password if needed

## **4.2.9 Non-Functional Requirements**

### **4.2.10 Performance**

### **4.2.11 Glossary**

## **4.3 Design**

## **4.4 Development**

### **4.4.1 Prototype Development with separate databases**

Developing the prototype began with creating the Catalogue Microservice first. This appeared to be a good place to start: create a basic MVC (Model-View-Controller) front-end that would connect to the Catalogue Microservice to display the list of items in the catalogue. Initially the idea was to create the Databases, with a table with data, connect to the microservice and then, through the API Gateway, connect with the front-end. Through research, it was deemed this was a difficult task to do.

The API Gateway itself being implemented using JSON documents and using a middle-ware technology, such as Ocelot Insert Citation, to implement. This would need further researching purely for this section of the prototype.

Instead, creating a blank Database was decided upon. From here, connect it to the API for the microservice and then to the front-end. The API Gateway has been omitted, at this stage, purely as there is only one API and to see the system working, albeit in a very basic way. Future, successful additions of microservices will see the API Gateway being added.

NOTE: Docker support was added to begin with, but proved to be difficult to work with. Instead, the prototype was developed without Docker support. Which will be added later, after further researching and small prototype builds to use the features needed for this project. This decision was taken as it was necessary to provide a working prototype of the proposed system within the time-frame than being delayed by understanding and then fixing the problem incurred using Docker. If docker is not able to be added to the prototype. Then this will be documented in the evaluation section and discussed as fully as possible.

### **4.4.2 Catalogue Microservice**

Creating the database was relatively simple: creating an account with Azure and creating the database and the server for its use. From here, the connection information needed is provided by Azure. Conveniently providing string connections depending on the type of connection wanted. ADO.NET was chosen. All that would be needed is having the string connection copy and pasted into the Appsettings.JSON file of the Catalogue Microservice. By adding the string to this file, enables Visual Studio to locate and connect to the server successfully to begin Entity Framework coding for manipulation/interrogation of the database.

Figure 10: Azure ADO.NET Connection String for the server containing the Catalogue Microservice Database

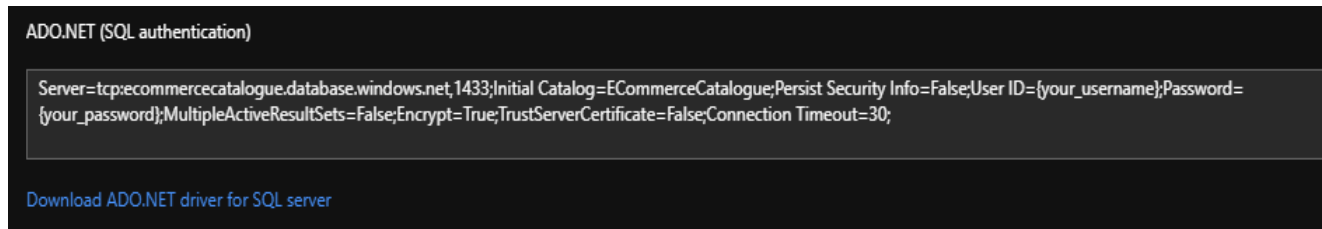
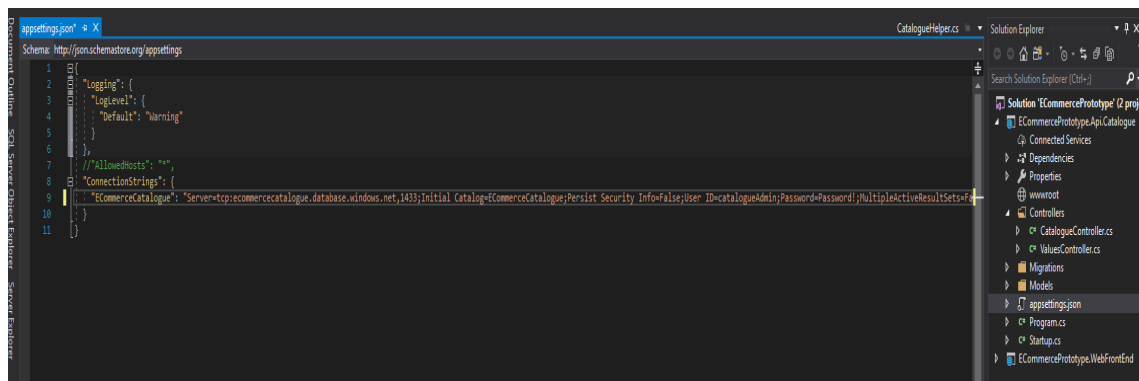


Figure 11: Visual Studio Appsetting.JSON connection string. Partial View



Using the Microsoft.Docs website, specifically the Entity Framework (EF) section, is very useful to gain an understanding, with example tutorials provided on how to use this feature for Database Manipulation [17], [19]. Also, an interesting blog on the c-sharpcorner website on this [21]. And this blog (INSERT CITATION)

After reading over these sites. The project was started. Developing the Catalogue API first. Connection string and Entity framework were first to be created and inserted. Creating a DbContext class for the system to interact with the Catalogue database. This class inherits from the DbContext class that enables a session created with the databases, getting the connection string through the Appsettings.JSON file, and allows for querying and saving entities to the database.

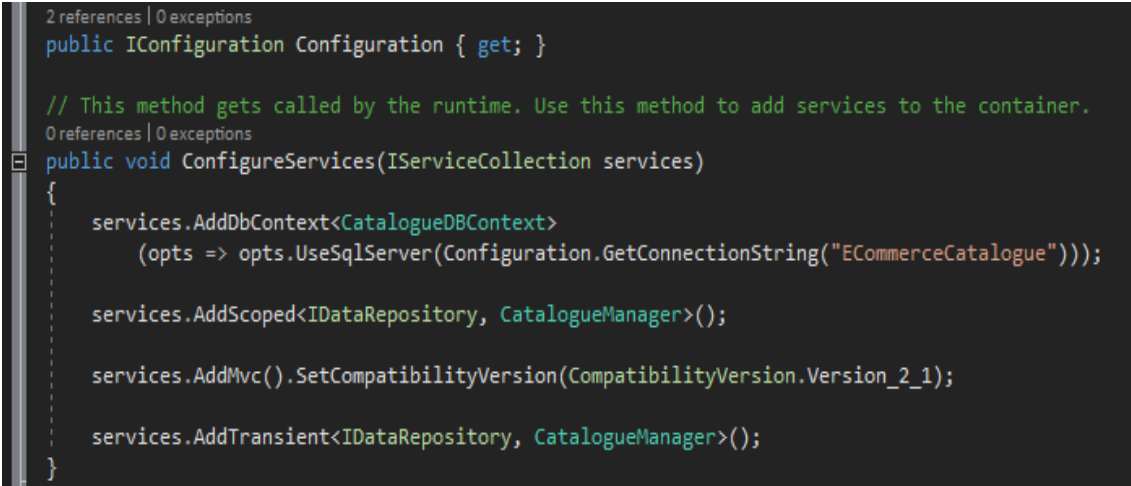
The Catalogue class was created using the system.ComponentModel.DataAnnotations.Schema to enable the properties to be used to create columns within the created table, within the CatalogueDB, that would store all catalogue item records. This was done by using Entity frameworks migration function to create the table based on the structure of the class and then updating the database with this migration. This was done successfully. Proving the system was

connected to this database, through the Azure server, successfully. Of note, the Property storing the location of the picture for the item was not created at this point. It was done later.

Based on research (INSERT CITATION) it was deemed necessary to provide an interface that would provide the methods needed, for this microservice, to query the database. Specifically GetAll() and Get(int id) methods. From here, the CatalogueManager class was created, that inherits from this interface. To provide implementation of these methods. The CatalogueManager class creates a read-only only object of the CatalogueDbContext that it uses to perform the methods upon. Returning the required entities - either a full list of all Catalogue Items or a single item based on its id in the database.

To enable this database connection to be used repeatedly, the DbContext, Data Repository interface and data manager class needed dependency injection by adding them as services to the ConfigureServices method of startup class [28]. By setting the data repository & catalogue manager as both scoped and transient, ensures each interaction with the database is created each time its requested and only once per request. Preventing duplication of requests within the one request.

Figure 12: Visual Studio Dependency Injection for the Catalogue Microservice



```
2 references | 0 exceptions
public IConfiguration Configuration { get; }

// This method gets called by the runtime. Use this method to add services to the container.
0 references | 0 exceptions
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<CatalogueDbContext>
        (opts => opts.UseSqlServer(Configuration.GetConnectionString("ECommerceCatalogue")));

    services.AddScoped<IDataRepository, CatalogueManager>();

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddTransient<IDataRepository, CatalogueManager>();
}
```

The final part of the Catalogue microservice to complete was the API Controller. This controller would be controlling how the database is interacted with, using HTTP Verbs, to expose what the front-end website can consume from this microservice. As the controller will be using a read-only object of the IDataRepository class, the only verbs being exposed will be GetAll() and get() a single catalogue item by its id in the database. Mi-



Figure 13: Visual Studio API Controller for Catalogue Microservice



```

namespace ECommercePrototype.Api.Catalogue.Controllers
{
    [Route("api/[controller]")]
    1 reference
    public class CatalogueController : Controller
    {
        private readonly IRepository _iRepo;

        0 references | 0 exceptions
        public CatalogueController(IRepository repo)
        {
            _iRepo = repo;
        }

        // GET: api/catalogue
        [HttpGet]
        0 references | 0 requests | 0 exceptions
        public IEnumerable<Models.Catalogue> GetAll()
        {
            return _iRepo.GetAll().ToList();
        }

        //GET api/catalogue/5
        [HttpGet("{id}")]
        0 references | 0 requests | 0 exceptions
        public Models.Catalogue GetCatalogueItem(int id)
        {
            return _iRepo.Get(id);
        }
    }
}

```

Microsoft SQL server management system was used to connect to the database, through its server, to insert several records to display on the website. After the inserts created, they were checked within the Azure portal dashboard to see they were present.

The Microservice was also run, as an API on the localhost, to display all the records created. And that a single record can be viewed by passing in an id to the URL.

(INSERT PICS??)

#### 4.4.3 Basket Microservice

NOTE: All API's will be created using the same design of a DbContext, IRepository and Manager interfaces/classes. Only differences being naming convention and CRUD implementation. Which will be identified

for each API.

#### **4.4.4 Prototype Development with a single centralised database**

#### **4.4.5 Front-End website development**

The website will be developed using the MVC architecture pattern. The Views applicable to each Microservice will be separated, as the standard when developing web applications using .NET Core, using folders. Each folder named after the corresponding controller that controls each group of Views. To emphasise the use of Agile throughout the development cycle, the website will be developed upon after each Microservice has been successfully implemented. The different sections being developed are discussed:

**4.4.5.1 Displaying Catalogue Items** This section of the prototype was created with the MVC template provided by Microsoft as the basis. The implemented code was replaced with the required code for views and controllers. Firstly, the models needed were created. A copy of the catalogue class from the Catalogue Microservice was created, as this would be needed to create an object of each record to be stored in another class, CatalogueItemRepository, that would be used by the View to display each catalogue Item, as per the initial class diagram design.

For the consumption of the API by the controller, a Catalogue API class was created, this would use the HttpClient class to create a connection to the URL address of the Catalogue Microservice API. Adding a header to information sent in JSON format. This Class implemented its method through the Interface, ICatalogueAPI, that it inherits from. To allow for features to be added in the future, also to Inject Dependency, in the same fashion as data repository class of the Catalogue Microservice. Without this, it would be impossible for calls, after the initial call, to be conducted as the first call would still be using the connection.

The controller for the views representing the catalogue Microservice

## 4.5 Testing

## **4.6 Evaluation**

## **5 Chapter 5: Conclusion/Evaluation**

**References**

- [1] Douglas K Barry. Service architecture. [https://www.service-architecture.com/articles/web-services/service-oriented\\_architecture\\_soa\\_definition.html](https://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html), 2000.
- [2] Thomas Cerny, Michael J. Donahoo, and Michal Trnka. Contextual understanding of microservices architecture: Current and future directions. Technical report, Baylor University and Czech Technical University.
- [3] Melvin Conway. How do committees invent? <http://www.melconway.com/Home/Committees.Paper.html>, 1968.
- [4] Andreas Diepenbrock, Florian Rademacher, and Sabine Sachweh. An ontology-based approach for domain-driven design of microservice architectures. In Maximilian Eibl and Martin Gaedke, editors, *INFORMATIK 2017*, pages 1777–1791. Gesellschaft für Informatik, Bonn, 2017.
- [5] Eric Evans. *Domain Driven Design: Tackling Complexity in the heart of software*. Addison Wesley, 1st edition, 2003.
- [6] Neal Ford. Polyglot programming. <https://searchsoftwarequality.techtarget.com/definition/polyglot-programming>, 2006.
- [7] Martin Fowler. Polyglot persistence. <https://martinfowler.com/bliki/PolyglotPersistence.html>, 2011.
- [8] Martin Fowler. Bounded context. <https://martinfowler.com/bliki/BoundedContext.html>, 2014.
- [9] Martin Fowler. Microservices. <https://www.martinfowler.com/articles/microservices.html%7Cmonolith>, 2014.
- [10] Todd Hoff. Amazon architecture. [highscalability.com/amazon-architecture](https://highscalability.com/amazon-architecture), 2007.
- [11] Thomas Hunter. *Advanced microservices*. Apress, 1st edition, 2017.
- [12] Judith Hurwitz, Robin Bloor, Marcia Kaufman, and Fern Halper. *Service Orientated Architecture for dummies*. Wiley, 2 edition, 2009.
- [13] Yury Izrailevsky and Ariel Tseitlin. The netflix’s simian army. <https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116>, 2011.

- [14] Scott Leberknight. Polyglot persistence. [www.sleberknight.com/blog/sleberkn/entry/polyglot\\_persistence](http://www.sleberknight.com/blog/sleberkn/entry/polyglot_persistence), 2008.
- [15] lokesh Gupta. Microservices - definition, principles and benefits. <https://howtodojava.com/microservices/microservices-definition-principles-benefits/#principles>, 2016.
- [16] Robert C. Martin. *Clean Architecture A craftsman's guide to Software Structure and Design*. Prentice Hall, 1st edition, 2018.
- [17] Microsoft. Asp.net core mvc with entity framework core. <https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-2.0>, 2018.
- [18] Microsoft. *.NET-microservice for containerized NET applications*. Microsoft Developer Division, 2nd edition, 2018.
- [19] Microsoft. Quickstart: Use .net core (c#) to query an azure sql database. <https://docs.microsoft.com/en-us/azure/sql-database/sql-database-connect-query-dotnet-core>, 2018.
- [20] Sam Newman. *Building Microservices*. O'Reilly Media, 1st edition, 2015.
- [21] Nitin Pandit. Entity framework with microsoft azure sql. <https://www.c-sharpcorner.com/article/entity-framework-with-microsoft-azure-sql/>, 2016.
- [22] Florian Rademacher, Jones Sorgalla, and Sabine Sachweh. Challenges of domain-driven microservice design, 2018.
- [23] Margaret Rouse. Polyglot programming. [memeagora.blogspot.com/2006/12/polyglot-programming.html](http://memeagora.blogspot.com/2006/12/polyglot-programming.html), 2006.
- [24] Margaret Rouse. Monolithic architecture. <https://whatis.techtarget.com/definition/monolithic-architecture>, 2016.
- [25] Dr.Andreas Schroeder. Microservice architectures.
- [26] Teach Computer Science. Synchronous and asynchronous data transmission. <https://teachcomputerscience.com/synchronous-and-asynchronous/>, 1999.

- [27] James Serra.
- [28] Steve Smith, Scott Addie, and Luke Latham. Dependency injection in asp.net core. <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2>, 2018.
- [29] techopedia. Modular programming. <https://www.techopedia.com/definition/25912/modular-programming>, 2018.
- [30] Olaf Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310, url=, Jul 2017.



# Appendices

## **A Project Initiation Document**

### **A.A Example sub appendices**

...

## **B Second Formal Review Output**

Insert a copy of the project review form you were given at the end of the review by the second marker

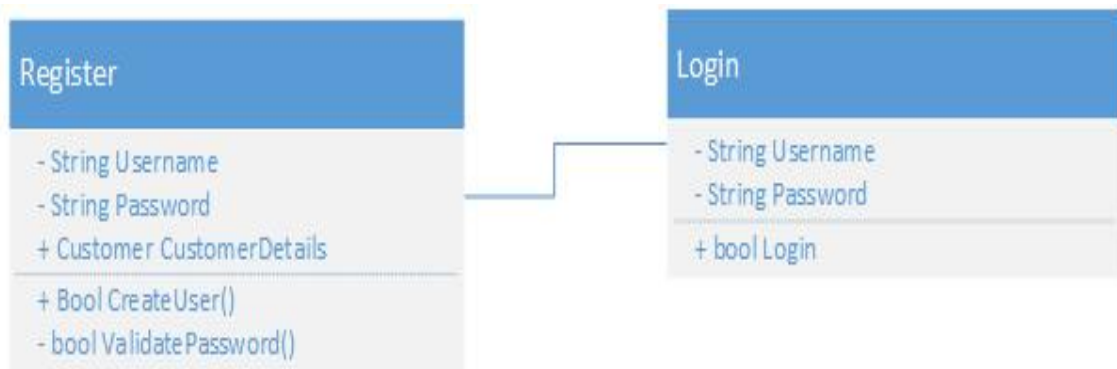
## **C Diary Sheets (or other project management evidence)**

Insert diary sheets here together with any project management plan you have

## D Class Diagrams

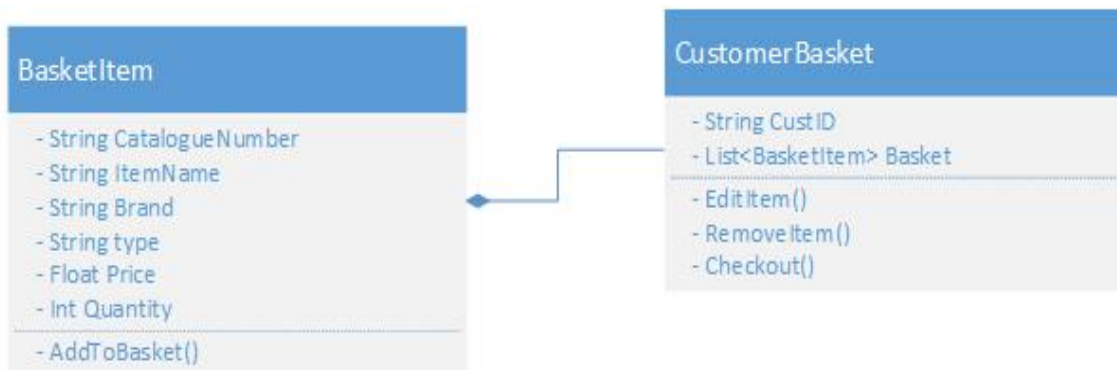
### D.A Authentication Microservice Classes

Figure 14: Authentication Classes



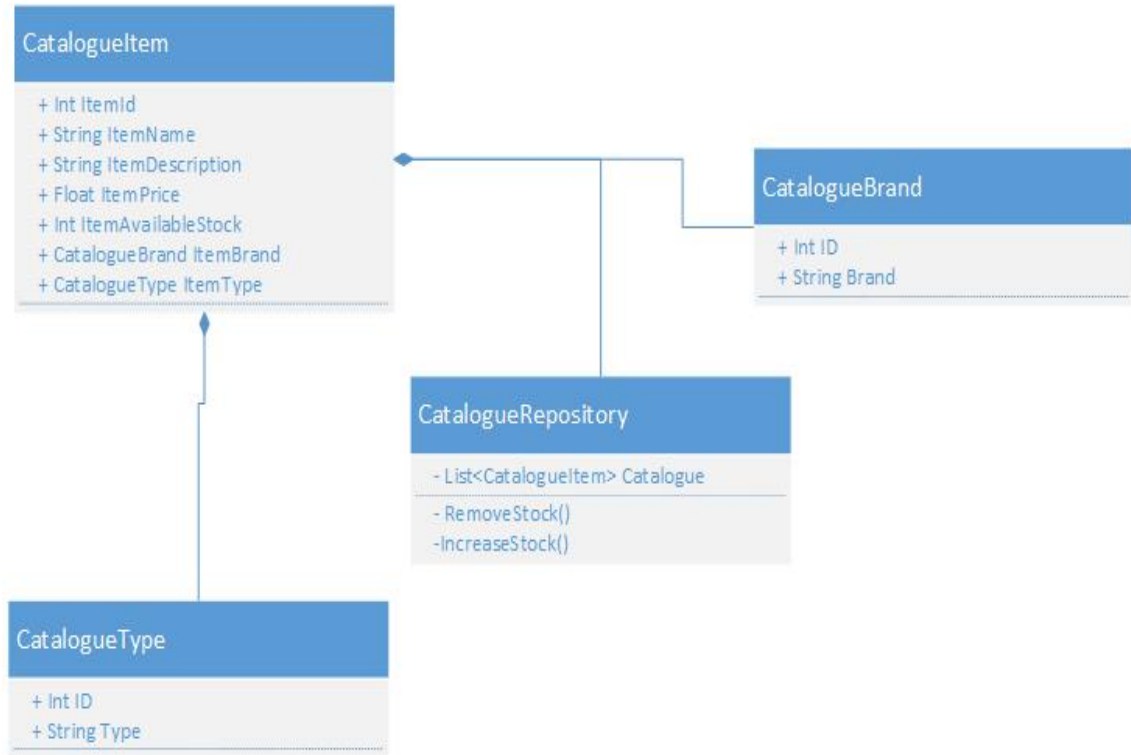
### D.B Basket Microservice Classes

Figure 15: Basket Classes



## D.C Catalogue Microservice Classes

Figure 16: Catalogue Classes



## D.D Checkout Microservice Classes

Figure 17: Checkout Classes



## D.E Customer Microservice Classes

Figure 18: Customer Classes

