

# A FRAMEWORK FOR THE EFFECTIVE DEVELOPMENT OF MICRO-SERVICE ARCHITECTURES

by

MATTHEW COOPER

URN: 6425639

A dissertation submitted in partial fulfilment of the  
requirements for the award of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

May 2021

Department of Computer Science  
University of Surrey  
Guildford GU2 7XH

Supervised by: Paul Krause

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

Matthew Cooper  
May 2021

© Copyright Matthew Cooper, May 2021

# Abstract

A micro-service architecture is a modern software application architecture in which the application is broken down into a set of loosely coupled services. These services need to communicate with one another, and the usage of RESTful APIs are a typical solution.

The aim of this project was to create a framework that facilitated the rapid development of both a set of micro-services and the RESTful API platform for handling their communication whilst maintaining development and API best practices and stability.

The project resulted in a framework being built, that being a curated suite of tools and libraries designed to build these RESTful micro-services. The use of this framework was showcased by the development of an exemplar micro-service.

Alongside the framework, modifications were made to the OpenAPI Code Generator, allowing an API client to be fully generated on demand for a micro-service, allowing a consumer of the generated library, typically another micro-service, to integrate communication with the service effortlessly.

# Acknowledgements

Special thanks to my supervisor Paul Krause for all of his assistance and support over the last two years. Additional thanks go out to my friends and family for helping support me through this project and the last few years at university and all of my coworkers at Hindsight Software for allowing me to take on this project.

# Contents

<b>1</b>	<b>Introduction</b>	<b>16</b>
1.1	Project Motivation and Background . . . . .	16
1.2	Aims and Objectives . . . . .	17
1.3	Summary . . . . .	18
<b>2</b>	<b>Literature Review</b>	<b>19</b>
2.1	Introduction . . . . .	19
2.2	The Problem: Communication between micro-services . . . . .	19
2.3	Types of Software Architectures . . . . .	21
2.3.1	Monolithic Architecture . . . . .	21
2.3.2	Micro-service Architecture . . . . .	22
2.3.3	Monolith vs Micro-services . . . . .	22
2.4	RESTful API Architecture . . . . .	23
2.4.1	REST Requests . . . . .	24
2.5	Alternatives to REST APIs . . . . .	26
2.5.1	SOAP APIs . . . . .	26
2.5.2	GraphQL APIs . . . . .	26
2.6	RESTful API Documentation & Specification . . . . .	27
2.6.1	Documentation . . . . .	27

2.6.2	Specification . . . . .	28
2.7	RESTful API Best Practises . . . . .	28
2.7.1	Correct Assignment of HTTP Status Codes . . . . .	28
2.7.2	Rate Limiting . . . . .	29
2.7.3	Pagination of Large Results . . . . .	29
2.7.4	Well Formatted and Designed URL Endpoints . . . . .	30
2.7.4.1	Using Nouns for all Resources . . . . .	30
2.7.4.2	Using Plurals for Resources . . . . .	30
2.7.4.3	Using Identifiers for Individual Resources . . . . .	30
2.7.5	Semantic Versioning of APIs . . . . .	31
2.7.6	Query Parameters for Filtering and Sorting . . . . .	32
2.8	Conclusion . . . . .	32
<b>3</b>	<b>Requirements and Specifications</b>	<b>33</b>
3.1	Introduction . . . . .	33
3.2	Requirement Gathering . . . . .	33
3.2.1	Speeding up the Development of RESTful Node.js Micro-services with the OpenAPI Specification . . . . .	33
3.2.2	Creating a Reliable Template TypeScript API Client . . . . .	34
3.2.3	Injecting the new API Client Functionality into Auto-generated API Clients from the OpenAPI Generator . . . . .	34
3.2.4	Summary . . . . .	35
3.3	System Requirements . . . . .	35
3.3.1	Node.js Micro-service & its API . . . . .	35
3.3.2	The ideal TypeScript API Client Library . . . . .	36
3.3.3	Analysis of OpenAPI Generator and Modifications Required . . . . .	37

3.4	Software Development Life-cycle Methodology . . . . .	38
3.4.1	Agile and Kanban . . . . .	38
3.5	Conclusion . . . . .	38
<b>4</b>	<b>Legal, Social, Ethical and Professional Issues</b>	<b>39</b>
4.1	Introduction . . . . .	39
4.2	Legal Considerations . . . . .	39
4.2.1	Computer Misuse Act . . . . .	40
4.2.2	Data Protection Act . . . . .	40
4.3	Social Considerations . . . . .	40
4.4	Ethical Considerations . . . . .	41
4.4.1	Privacy and Consent . . . . .	41
4.4.2	Agency and Identity . . . . .	41
4.5	Professional Considerations . . . . .	41
<b>5</b>	<b>System Design</b>	<b>42</b>
5.1	Introduction . . . . .	42
5.1.1	Client-Server API Interactions . . . . .	43
5.2	Premise of the micro-service . . . . .	43
5.3	RESTful API Design . . . . .	43
5.3.1	The API . . . . .	44
5.3.1.1	The shoppingItem Response Object . . . . .	44
5.3.1.2	API Endpoints . . . . .	44
5.4	Technologies and Libraries . . . . .	47
5.4.1	Node.js . . . . .	48
5.4.2	TypeScript . . . . .	48
5.4.3	OpenAPI Specification . . . . .	49



5.4.4	Micro-service specific libraries . . . . .	49
5.4.4.1	Express.js . . . . .	49
5.4.4.2	TSOA . . . . .	49
5.4.4.3	Swagger-UI-Express . . . . .	50
5.4.4.4	Express-Rate-Limit . . . . .	51
5.4.4.5	Mongoose . . . . .	51
5.4.4.6	Mongoose-Pagination-V2 . . . . .	51
5.4.5	Client specific libraries . . . . .	52
5.4.5.1	Creating the Client as an NPM Library . . . . .	52
5.4.5.2	Axios . . . . .	52
5.4.5.3	Axios-Retry . . . . .	53
5.4.6	OpenAPI Code Generator . . . . .	53
5.4.6.1	The typescript-axios templates . . . . .	53
5.5	Microservice . . . . .	54
5.5.1	data-layer . . . . .	54
5.5.1.1	models/ShoppingItemModel.ts . . . . .	54
5.5.1.2	data-agents/ShoppingItemAgent.ts . . . . .	55
5.5.2	middleware . . . . .	57
5.5.2.1	enums/ShoppingItemCategories.ts . . . . .	57
5.5.2.2	types/ErrorLibrary.ts . . . . .	57
5.5.2.3	ShoppingItem.ts . . . . .	58
5.5.2.4	ErrorHandler.ts . . . . .	58
5.5.2.5	ErrorWrapper.ts . . . . .	58
5.5.2.6	RateLimiter.ts . . . . .	59
5.5.3	service-layer . . . . .	59

5.5.3.1	controllers/ShoppingItemController.ts . . . . .	59
5.5.3.2	routes/Routes.ts . . . . .	62
5.5.4	Root Files . . . . .	62
5.5.4.1	App.ts . . . . .	62
5.5.4.2	Server.ts . . . . .	63
5.5.4.3	tsoa.json . . . . .	63
5.5.5	Commands Used . . . . .	63
5.5.5.1	Generating routes through TSOA . . . . .	63
5.5.5.2	Generating an OpenAPI Specification from the Controllers . . .	63
5.5.5.3	Compiling & building the Service . . . . .	63
5.5.5.4	Starting the Service . . . . .	63
5.6	API Client . . . . .	63
5.6.1	utils . . . . .	64
5.6.1.1	Http.ts . . . . .	64
5.6.1.2	HttpVerbs.ts . . . . .	65
5.6.1.3	ErrorWrapper.ts . . . . .	66
5.6.2	microservice-client . . . . .	66
5.6.2.1	MicroserviceClient.ts . . . . .	66
5.6.3	Root Files . . . . .	67
5.6.3.1	Index.ts . . . . .	67
5.6.4	Building and Publishing the Library . . . . .	67
5.7	Modifying the OpenAPI Code Generator . . . . .	68
5.7.1	Generating the Client using the Base Templates . . . . .	68
5.7.2	Determining the differences between the generated client and my client . .	68
5.7.3	Modifying the generated client . . . . .	69

5.7.4	Modifying the template files . . . . .	69
5.7.5	Generating a client with the new templates . . . . .	70
5.7.6	Conclusion . . . . .	70
<b>6</b>	<b>Testing and Evaluation</b>	<b>71</b>
6.1	Introduction . . . . .	71
6.2	Unit Testing with SuperTest, Jest and Nock . . . . .	71
6.2.1	SupertTest . . . . .	72
6.2.2	Jest . . . . .	72
6.2.3	Nock . . . . .	73
6.2.4	Code Coverage Target . . . . .	73
6.2.5	Unit Testing the Micro-service . . . . .	74
6.2.5.1	Testing the Controllers . . . . .	74
6.2.5.2	Testing the Data Agents & Models . . . . .	74
6.2.6	Unit Testing the Handcrafted Client . . . . .	75
6.2.6.1	Intercepting HTTP Requests to the micro-service with Nock . . . . .	75
6.3	End-to-End Testing the Auto-generated Client . . . . .	78
6.3.1	Migrating the Unit Tests from my Client . . . . .	78
6.4	Results of the Tests . . . . .	79
6.4.1	Reading the Code Coverage Reports . . . . .	79
6.4.1.1	Statements . . . . .	79
6.4.1.2	Branches . . . . .	79
6.4.1.3	Functions . . . . .	80
6.4.1.4	Lines . . . . .	80
6.4.2	The Micro-service . . . . .	80
6.4.2.1	Statements . . . . .	81

6.4.2.2	Branches . . . . .	81
6.4.2.3	Functions . . . . .	81
6.4.3	The Handcrafted Client . . . . .	82
6.4.3.1	Statements . . . . .	82
6.4.3.2	Branches . . . . .	82
6.4.3.3	Functions . . . . .	83
6.4.4	The Auto-generated Client with the modifications . . . . .	83
6.5	Evaluation against Project Requirements and Specifications . . . . .	83
6.5.1	The Exemplar Micro-service . . . . .	84
6.5.1.1	The Objectives for the Micro-service . . . . .	84
6.5.1.2	Generate an OpenAPI Specification file . . . . .	84
6.5.1.3	Use well-formed URLs . . . . .	84
6.5.1.4	Provide endpoints for each of the major HTTP verbs . . . . .	85
6.5.1.5	Assign the correct HTTP status codes to both Success and Error Responses . . . . .	86
6.5.1.6	Validate both request and response payloads . . . . .	86
6.5.1.7	Provide useful documentation for the API . . . . .	87
6.5.1.8	Utilise Rate Limiting . . . . .	88
6.5.1.9	Make use of Pagination on large responses . . . . .	89
6.5.1.10	Provide access to a database to support CRUD usage of different HTTP Verbs . . . . .	90
6.5.2	The Handcrafted Client . . . . .	90
6.5.2.1	The Objectives for the Handcrafted Client . . . . .	90
6.5.2.2	Support all endpoints across the API . . . . .	90
6.5.2.3	Provide details of the responses types returned by endpoints . . . . .	91
6.5.2.4	Provide details of all input parameters for the endpoints . . . . .	92

6.5.2.5	Ensure all important error information is correctly is passed on .	92
6.5.2.6	Automatically retry requests that fail with certain HTTP status codes . . . . .	93
6.5.3	The Modified Auto-generated Client . . . . .	94
6.5.3.1	The Objectives for the Auto-generated Client . . . . .	94
6.5.3.2	Support all endpoints across the API . . . . .	94
6.5.3.3	Provide details of the responses types returned by endpoints . .	95
6.5.3.4	Provide details of all input parameters for the endpoints . . . . .	95
6.5.3.5	Ensure all important error information is correctly is passed on .	96
6.5.3.6	Automatically retry requests that fail with certain HTTP status codes . . . . .	96
6.6	Conclusion . . . . .	96
<b>7</b>	<b>Evaluation of the New Framework</b>	<b>97</b>
7.1	What is the Purpose of this Project . . . . .	97
7.2	The Framework . . . . .	98
7.2.1	Creating the API for a Micro-service . . . . .	98
7.2.2	Modular Client Generation using API Specification Files . . . . .	99
7.3	Using the Framework at Hindsight . . . . .	100
7.4	Conclusion . . . . .	101
<b>8</b>	<b>Conclusion</b>	<b>102</b>
8.1	Overview of the Project . . . . .	102
8.2	Enhancing the Framework in the Future . . . . .	103
8.2.1	More advanced Rate-limit support for the micro-services . . . . .	103
8.2.2	More intelligent mechanism for retrying rate-limited requests . . . . .	104
<b>A</b>	<b>Ethics Self-Check Form</b>	<b>105</b>

# List of Figures

6.1	Code Coverage Report for the Exemplar Micro-service . . . . .	80
6.2	Code Coverage Report for the Handcrafted Client . . . . .	82
6.3	Code Coverage Report for the Generated Client with my Modifications . . . . .	83
6.4	Documentation snippet from the micro-service documentation URL . . . . .	88
6.5	Example of Rate Limit response headers provided by the API . . . . .	89
6.6	Example of Paginated Response provided by the API . . . . .	90
6.7	Example Response Type Casting . . . . .	91
6.8	Example Usage of the Response Typings . . . . .	92
6.9	Example Usage of the Request Typings . . . . .	92
6.10	Example Unit Test for Error Handling . . . . .	93
6.11	Example Unit Test for Retrying Rate Limited Requests . . . . .	94

# List of Tables

6.1	Method name to URL mapping for the Handcrafted Client . . . . .	91
6.2	Method name to URL mapping for the Generated Client . . . . .	95

# Abbreviations

API	Application Programming Interface
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
SOAP	Simple Objects Access Protocol
DoS	Denial of Service



# Chapter 1

## Introduction

### 1.1 Project Motivation and Background

Micro-services are a relatively new style of software architecture which has been gaining increasing popularity over the last few years. Rather than having a single monolithic application, a micro-service architecture breaks the application down into loosely coupled services which communicate with one-another.

Micro-services solve a lot of the issues which monolithic architectures have. Code-base changes are isolated to the micro-service responsible for the feature being modified, so changes in one part of the system will have no impact on other parts. The same goes for bugs, which are likely to be isolated to single services at a time. However, micro-services do introduce issues that monoliths do not face.

In order for the application to work, the micro-services must establish communications with each other. Data will be passed and processed between micro-services, with each micro-service carrying out its own niche functionality. These communications are typically handled through the use of APIs. An API defines the interface through which communication with the service can occur, specifying the shape of both requests and responses as well as any data formats that should be used.

Each micro-service details not only its own unique API, but must also have knowledge of how to consume the APIs of all the other services that it needs to communicate with. A typical large application may have over twenty micro-services, and often a service will interact with at

least four or five additional services. Ensuring that communication is happening as expected between all these services is paramount if the application is going to operate reliably. Outside of an application's own micro-services, it may also communicate with external APIs for services such as GitHub, Bitbucket, Gitlab and Jira.

During my placement year, and subsequent two years of employment, I have worked on the transformation of a large monolithic Java application into a set of micro-services at the company Hindsight Software. During this architectural shift we encountered a number of problems, mainly when handling the communication between the individual services. Each service needs to have code acting as an API client for each and every other service that it needs to communicate with and so code can be quite heavily duplicated across the entire architecture. Additionally, we have encountered issues handling error responses between the services, as well as transferring knowledge of API request responses from the server to client micro-service. The aim of this project is to alleviate these issues through code-generation of API clients, with these new clients solving the problems we have been facing and being able to be installed as dependencies of each micro-service to reduce duplication and code overhead.

## 1.2 Aims and Objectives

Keeping the project within scope is key to its success, and part of this is defining a set of concrete Aims and Objectives that it needs to fulfil. These are as follows:

1. Explore and evaluate the different types of APIs
2. Explore and evaluate different specification formats for REST APIs
3. Explore and evaluate expected functionalities of a modern REST API
4. Explore and compose a list of best practises for consuming REST APIs
5. Evaluate and choose a suite of TypeScript and Node.js tools and libraries for creating a well-documented RESTful API
6. Evaluate and choose a suite of TypeScript and Node.js tools and libraries for creating a RESTful API client
7. Create a Node.js micro-service with a well-documented and standardised REST API

8. Create functionality for exporting the API specification from the micro-service
9. Create an API client library consuming the micro-service and handling expected error cases
10. Explore and evaluate technologies for automatically generating API client libraries
11. Create or modify a code generator in order to generate API clients from an API specification

## 1.3 Summary

This project aims to help alleviate the issues faced by both the client and the server services. Over its course, I will create an example Node.js micro-service with its own REST API. I will then create a handcrafted client library consuming the API, before exploring ways to automatically generate client libraries for APIs. Finding a way to automatically generate a client library from an API specification will completely remove all of the issues related to maintaining code consuming the services. If information related to the request and response payloads can be provided as TypeScript types, then the consumers will have all of the information that they need in order to effectively interact with the micro-service.

Chapter 2 reviews the problem in more depth, with a specific focus on identifying best practices for API design. Chapter 3 then moves on to identifying the core requirements for framework used in conjunction with a way of working that would address the problems we have identified with micro-service architectures. The Legal, Social, Ethical and Professional concerns around the project are explored and evaluated in Chapter 4. Chapter 5 moves onto the system design for both the exemplar micro-service and API client as well as how the modifications to the OpenAPI Generator templates were implemented. The process and results of the testing of the systems, alongside critical evaluation against the project objectives takes place in Chapter 6. A more general evaluation of the system and its integration at Hindsight then takes place in Chapter 7. Chapter 8 closes the report with an overview of the project before exploring any future work on the software.

Overall the aims and objectives were all met, with the framework and way of working developed already seeing integration at Hindsight.

## Chapter 2

# Literature Review

### 2.1 Introduction

In this chapter I shall give a more in-depth overview of the problem. I will then look at the different areas relating to the project, including analysis of different software and API architectures, best practises for API design as well as looking at documentation and specifications for APIs. Note that this chapter does not contain the volume of citations that is typical for a literature review. Instead, it captures practical experience obtained from working within the software community from my placement year to the present.

### 2.2 The Problem: Communication between micro-services

During our transition from a monolithic architecture to micro-services we have encountered the following issues:

- Lack of documentation for APIs: Documenting APIs is crucial, consumers need to know how to make requests, and what responses look like. Ensuring the documentation is up to date and accurate is mandatory in order for an API to be used effectively. Documentation also allows users to study how the API works without having to integrate its usage into their software.
- Duplication of code handling communication between micro-services: Every micro-service will have code responsible for communicating with all of the other micro-services it needs

to send requests to. If the API is modified or new endpoints are added, each service consuming the API is going to need to also have the modifications applied. Good API documentation will provide example response and request payloads as well as detailed information about expected error responses. Without this information any integrations will likely run into issues, whether it be processing data retrieved from the API or when errors that the server expects to be returned are not handled correctly.

- The client service lacks knowledge about the shape of the response when a request is made: In a monolithic architecture, all the code is encapsulated under the same umbrella, and so if one part of the system communicates with another part, all of the class information will be provided. This is even more of an issue with external APIs. If we make a request to GitHub, the service making the request needs to know the format and structure of the response so that we can consume the information provided as we need to. Currently we have to manually create TypeScript types detailing request and response bodies. The issue with this approach is that it is very static, any changes will need to be copied across all of the services consuming the API, and the whole process is extremely time consuming.
- Handling HTTP errors: Errors happen, and sometimes a request to a service will fail. The server might have hit an exception it could not recover from, or an underlying issue with the hosting may have temporarily knocked the service out. Alongside this RESTful APIs often make use of techniques such as Rate Limiting in order to protect their services from DoS attacks. The API will allow X number of requests over a time-frame of Y. If the number of requests exceeds this number, then errors will be thrown on subsequent requests until the allotted time-frame is exceeded. These errors usually return specific HTTP status codes in the response, informing the client what went wrong with the request. Having mechanisms in place for automatically handling these errors, and trying to send the request again after an indeterminate amount of time will help to prevent avoidable errors from bubbling up the micro-service chain.

There are two main areas that these issues fall under. One is code duplication; if three micro-services need to communicate with the same micro-service, then there will be three separate client integrations for that service. Any changes made to the client will have to be duplicated across the different services and any bugs in the implementation will also be present across the board.

The other area is related to transferring information about the API, whether that be at the documentation level, or within the code when information is requested and then processed. Handling expected errors would also fall under this domain.

The solution to the problem can be broken down into several parts.

1. Creating an exemplar micro-service with a well documented API that follows best practises and conventions. This will include implementation of things such as Rate Limiting and Pagination of results.
2. Publishing documentation detailing the specification of the API.
3. Establishing a template for an API client that solves the issue related to consumption of APIs that we face. The client will have to handle errors and rate limiting correctly, as well as ensuring it exports the the shape of request responses as part of its library.
4. Generating a library of API clients based on the template that the micro-services can import as dependencies. Essentially each micro-service will be able to produce an API client that a consumer can then use to communicate with it on demand.

If implemented properly, the solution will save time on development of new micro-services drastically. By being able to generate API clients for all of the services it needs to communicate with, less code will have to be manually written, and the code processing the outputs of the requests it makes can be written with confidence, as we will know exactly what data is returned. Additionally it will give us faith in the the handling of errors related to these requests, and if errors are thrown will help isolate exactly why and where they came from.

## **2.3 Types of Software Architectures**

I will be discussing two main forms of software architecture, the traditional Monolithic architecture, and the more contemporary, distributed micro-service architecture .

### **2.3.1 Monolithic Architecture**

Monolithic architecture is the classic way of shipping software. All of the code is stored under one umbrella. Back-end, database and front-end code are all encapsulated as a single unit.

The different software components can readily talk to one-another, no API usage between the internal components is needed.

### **2.3.2 Micro-service Architecture**

Conversely, micro-service architectures break down this monolithic structure into a number of self contained and loosely coupled services, responsible for a single aspect of the applications functionality. These services then communicate with each other, typically through an API, in order to pass information from one part of the system to another.

### **2.3.3 Monolith vs Micro-services**

The two architecture styles each offer advantages and disadvantages when compared with each other. Monolithic code-bases are simple to develop, test, and deploy. You only need to have one code-base stored locally, and you simply need to checkout onto a new branch and start coding to add new functionality. As everything is under one roof, you can have powerful testing suites that cover the entire application, both at a Unit level, but also as End-to-End tests. End-to-End tests are particularly potent as they allow the system to start tests at the UI simulating the end user, ensuring behaviour of the application is expected from start to finish.

However, monoliths can get messy. As functionality of the application grows, so does the code-base, and soon the system can consist of many hundreds and thousands of different classes and files. If the entire system isn't extensively Unit-Tested, small changes can easily introduce new bugs, and adding additional functionality can turn into an agile development nightmare. Monoliths can also make fully understanding the code-base as a whole extremely challenging, simply because there is too much code there. A new developer joining the team could be easily overwhelmed as they learn the system.

Micro-services alleviate a lot of these issues. As individual functionality is abstracted out as a new service, understanding how individual micro-services work is comparatively very easy. Micro-services also enable different teams to easily and effectively work on different components of the system in parallel. As long as correct versioning techniques are used, and the application has been logically broken down, then modification of one part of the system should have little to no effect on other parts. Comparatively, while monoliths are technically easy to deploy, any changes made to the monolith result in the entire stack being re-deployed, and in large

applications this can make throughput of small agile changes very low. Something as simple as changing some text in the front-end can take many hours to go through the deployment pipeline. Micro-services do have the disadvantage of being more difficult to deploy; the distributed nature of the system means that proper networking and fallback protocols need to be implemented to make sure that the system can happily work together as a cohesive unit.

Additionally, software following the micro-service architecture can consist of micro-services using completely different programming languages and technologies. Through their usage of APIs to communicate between services, each micro-service does not need to have any knowledge of the internal workings of any of the other services in order to function properly. This means that a company can use individual services as test beds for new technology and solutions, and look to incorporate those over time as new micro-services get added. This is something that the static nature of a monolithic architecture prevents, as everything is so tightly wound together.

Micro-services are heavily scalable, and if certain parts of the overall system need more resources or implementations of things such as load balancers, then those specific micro-services can be targeted. In monoliths, resources need to be added to the entire system as a whole. Failures within micro-service architectures are oftentimes handled more gracefully. If an error or a failure happens in one service, then it is isolated, especially in the cases of more technical issues such as memory leaks. However networking problems and errors can cause issues with micro-services, primarily by interrupting communication between them. If services' communication channels are knocked out, then that entire functionality is knocked out until the issue is resolved.

Overall, there are use cases for both architectural styles. Monoliths work very well for smaller code-bases, but as soon as the application grows and becomes cumbersome, more scalable micro-services are typically more appropriate.

## 2.4 RESTful API Architecture

REST is an acronym for Representation State Transfer, a term first coined by Roy Fielding in his doctoral thesis (Fielding 2000). REST follows the client-server architectural paradigm and is stateless, the client and server do not need to know about the others state. This paradigm allows different clients, all implemented differently, to interact with the same server, in the same way, and receive the same responses. Changes made to the clients have no bearing on the server



whatsoever.

Information in a RESTful API is abstracted out as a Resource. State is the information stored under the resource. When a client makes a request to the server, the format of the response, I.e. the Representation of the data, is specified and the data is returned in that format. Essentially RESTful APIs work by getting the client to describe exactly what they want, and how they want to receive it.

### 2.4.1 REST Requests

Requests made to a RESTful API contain a number of components

1. HTTP Verb: The HTTP verb defines the operation that will take place. There are four main HTTP verbs that see common use. Technically there are more but for general use cases they are not commonly used.
  - GET: This verb specifies that you wish to retrieve the information about the desired resource
  - POST: This verb informs the server that it is to create a new resource. Typically the information relating to the new resource is sent in the request body.
  - PUT: The PUT verb is used to modify a specified resource. As with POST the detail of the modification are typically sent in the request body.
  - DELETE: This verb is pretty self explanatory, and informs the server that the resource should be deleted
2. URL Endpoint: URL is an acronym for Uniform Resource Locator, and specifies the resource being requested, and where to request it from.

An example URL endpoint for the Bitbucket API is:

```
GET https://api.bitbucket.org/2.0/repositories?role=member
```

This URL requests a list of all public repositories in which the authorised user has read-access. It consists of a number of different components.

- Scheme: The scheme is used to identify how the URL is to be interpreted. In the example the scheme

`https`

This tells the client that HTTP over TLS is to be used.

- Host: This specifies the base address for where the request will be sent. In the example this will be

`api.bitbucket.org`

The request will be made under Bitbucket's API domain.

- Path: This specified the resources being requested. In the example it is

`/repositories`

In this case the resource being requested is all of the public repositories.

- Query: The query parameters are normally optional on requests, and are usually used to help further filter and define requests. Query parameters are added after a `?` and multiple query parameters can be added after each other through the `&` symbol. In the example URL the query parameter is

`role=member`

This requests results be filtered down to only those where the authenticated user has read-only rights on the repository.

3. Header: The header is used to pass additional information about the request and response. Both requests and responses have their own headers and different headers provide different information. Typical use cases for request headers are to allow the client provide credentials to authorise restricted requests. The client may provide a token which the server can then check and authenticate. If the client is authorised the response is sent, otherwise the server will send an Unauthorised error response. An example of a response header is the Location field, often used to specify the location of newly POSTed resources within the response.
4. Body: Request bodies are typically only needed for POST and PUT requests, and is used to pass in additional information used to create or update the resource. Body data can be sent in different forms, often identified within the header.

One important note about REST is that it is not a formal protocol, and is instead a set of more loose guidelines.

## 2.5 Alternatives to REST APIs

### 2.5.1 SOAP APIs

Simple Objects Access Protocol is an XML based messaging protocol. SOAP originated as the XML-RPC specification (Winer 1998) released by Microsoft in 1998, iterating through several versions before version 1.2 (*SOAP Version 1.2* 2003) reached W3C Recommendation status. This status indicates that the standard is ready for public consumption, having been tested and reviewed thoroughly.

Compared to REST it is far more verbose and rigid. SOAP is typically used by financial institutions, its rigid nature is ideal for usecases where security is a priority. For general usage SOAP requests and responses contain vast swathes of bloat, which in turn slows down communication. With SOAP you are also locked into using XML, which compared to more contemporary approaches like JSON, is less user-friendly and far less human-readable.

All of this reduces the viability of SOAP for use in micro-service APIs, the overhead is just too great compared to RESTful APIs.

### 2.5.2 GraphQL APIs

GraphQL is a contemporary data query language developed by Facebook. With GraphQL a single endpoint is provided, and clients request the specific information that they required. Whereas REST endpoints provide concrete response bodies for request, GraphQL returns only the information requested. This makes GraphQL a lot more lightweight than RESTful APIs, no redundant information is requested or sent.

The downside with GraphQL is it has yet to see widespread standardisation or adoptions. Far more tools and solutions support REST, however over the next few years we might see a shift away from REST and towards GraphQL.

## 2.6 RESTful API Documentation & Specification

### 2.6.1 Documentation

API documentation guides the end-user on how the API is to be used. It tells a developer how each endpoint interacts, from how the client should format a request, as well as describing the responses provided by the server.

Good API documentation should cover the following:

- Details for each API endpoint: Each API endpoint should be described in detail. Related HTTP verbs should be listed, path parameters should be described and the functionality that the endpoint fulfils should be described. Additionally all query parameters should be listed and explained, with details on the effect they have on the call. Any additional information about header parameters that the user should be aware of should also be listed.
- Detailed request information: For PUT and POST requests detailed information relating to the data the API server expects to be sent should be provided. Information about all fields in the body data should be explained, as well as data formats accepted by the endpoint. Good documentation will provide examples for each of the data formats.
- Detailed response information: The response format for the endpoint should be described in depth, each field explained and listed. Example responses help users of the API understand more clearly.
- Detailed lists of error information: If an API expects to throw errors in certain cases, but details of these errors are not provided, then the user is not able to handle them. All expected error messages should be listed, with details of the message, associated HTTP status code and an explanation of the cause.

Documentation should always be kept up to date, and updated as the API itself is modified. If an API is not documented, or the documentation is out of date, then the API will be unusable.

## 2.6.2 Specification

Where documentation is meant to be a human-readable explanation of an API, API specifications describe how an API behaves and interacts, and is typically written in specific specification languages. Oftentimes the specifications are transformed into machine-readable API definitions and used in creation of API servers, handling response and requests as specified, alongside adding layers of validation across the API.

## 2.7 RESTful API Best Practises

### 2.7.1 Correct Assignment of HTTP Status Codes

RESTful APIs involve communication between two parties, a server and a client. A client will typically make a request to a server, and HTTP status codes allow the server to provide a quick summary of whether the request was successful, and if not, what went wrong.

HTTP status codes are three digit numbers that are groups into five main categories.

- **Information Response:** These status codes occupy the 1xx band of numbers, such as 100 and 101. These status codes are typically used to communicate to the client that the request was successfully received, but more processing is to be done on it.
- **Success Response:** The success responses indicate to the client that the request is complete and successful, and cover the 2xx number range. The typical responses are the 200 and 201 status codes, with the former typically used in conjunction with a response body and the latter without, but there are a number of other response codes covering other use cases.
- **Redirection Response:** Covering the 3xx numbers, these status codes inform the client that some additional action must be taken in order to complete the request. Typically this involves redirection of the request to a different URL.
- **Client Error Response:** 4xx status codes are used to tell the client that some part of the request provided by them is incorrect or unexpected. Typical status codes for this category include the 400 Bad Request code informing the client that some part of the request body or URL is incorrect, the 401 Unauthorized code which tells the user that they have not

yet provided authentication to access the request and the 404 status code which informs the user that the request resource does not exist.

- **Server Error Response:** These status codes cover the 5xx series, and are used when some part of the request fails on the servers side. Where 4xx status codes are somewhat expected, 5xx codes typically involve unexpected crashes or exceptions. The 500 Internal Server Error code typically covers most use cases, but more specialised codes for things like Bad Gateways are also used.

### 2.7.2 Rate Limiting

Modern REST APIs can potentially be consumed by many thousands of potential clients. This can produce a great deal of strain on the the host server, especially if requests are concentrated in a tight time-frame. Rate limiting aims to help alleviate this issue by essentially throttling the throughput of requests made at any one time or time-period.

Typically requests will be associated with a specific user, either through tying in the IP address from which the request originated, or through any authenticated user information if the API requires it. A set volume of requests that the server allows over a period of time is specified, and that individual's request number is tracked and incremented as more requests are made. If the limit is exceeded, the user will be temporarily locked out of making any more requests to the API until after the time period has ended. A 429 status code is typically sent by the server to inform the client of this breach of the rate limits alongside information about when the lockout ends.

Rate limits also have the added benefit of helping to protect against more malicious Denial of Service attacks through the same principles.

### 2.7.3 Pagination of Large Results

RESTful APIs allow access to huge swathes of data. In order to illustrate the point we will look at the following Bitbucket API endpoint:

```
https://api.bitbucket.org/2.0/repositories
```

This endpoint provides a list of every public repository that exists in Bitbucket. There are

hundreds of thousands of repositories stored and if the response were to contain all of the information in single payload, it would be vastly too large.

Pagination allows us to break down large sets of data into more manageable chunks called pages. Pages have a length specified which acts as an upper bound for how many entries each page can contain. Rather than returning all of the data at once, the API will instead return a page, alongside some meta-data informing the client about current page number, as well as information relating to the total number of pages. Some implementations of pagination also contain links to the first, previous and next page but this differs across different APIs.

Pages as well as page size can typically be specified through use of query parameters on the endpoint, and when tied in with the page number, and total pages meta-data a client can move over the pages like they would a book.

#### **2.7.4 Well Formatted and Designed URL Endpoints**

In order for an API to be easy to use and navigate it must conform to certain best practises and standards, with the key ones being specified below.

##### **2.7.4.1 Using Nouns for all Resources**

RESTful APIs are resource-based, and each of these resources should be named using nouns rather than verbs. The key premise for this is that a resource is representing a thing, the very definition of a noun. The API is used to fetch the properties of these resources and using a verb to describe the resources does not make sense.

##### **2.7.4.2 Using Plurals for Resources**

APIs typically contain many different individuals of each resource type and so the resource name should be formatted in its plural form to properly convey this.

##### **2.7.4.3 Using Identifiers for Individual Resources**

While the resources will be pluralised, individuals of this chosen resource type should be able to be fetched through use of an identifier. Choosing the identifier carefully is important as the

request should be idempotent, that is every time the same identifier is used, the same resource is used.

### 2.7.5 Semantic Versioning of APIs

Over time REST APIs will change and evolve. APIs acts as a dependency for many different pieces of software and if the response bodies or URL endpoints change many of these integration can break, stopping functionality within this software from working.

Semantic versioning is the principle of breaking changes into three distinct categories.

- Major: These are the big sweeping changes, any changes that will break or change the fundamental workings of the API will fall under this category. An example of this would be the removal of existing fields within the responses from the API.
- Minor: These changes can changes functionality, but in a backwards compatible way. These could be the addition of new optional fields such that existing data can still be used.
- Patch: These changes are typically bug fixes, again completely backwards compatible.

These API versions are typically represented in the format X.Y.Z where X represents the Major version, Y the Minor and Z the Patch version such that the version 1.2.1 will be the first Major, with two Minor version increments and one Patch version increment on the current Minor version.

For the most part as the Minor and Patch versions are backwards compatible, we do not have to deal with them outside of documentation tracking the changes. Major versions with their breaking changes are the key issue.

The standard way of handling Major versions is by including the version number within the path of the URL endpoint that the API exposes. If we look back at the Bitbucket repositories endpoint we can see this:

```
https://api.bitbucket.org/2.0/repositories
```

After the base URL the number 2.0 is specified. This identifies the seconds major version of the API is to be used. As long as each major version is documented adequately, users should be able to consume any of the versions or all of versions without ever dealing with breaking changes. Any



big changes will simply involve creating a new endpoint with an incremented version number within the path and all of the existing calls to the old endpoints will be unchanged.

### **2.7.6 Query Parameters for Filtering and Sorting**

Falling under the same vein as pagination, the large data-sets that RESTful APIs often provide typically need to provide support for filtering and sorting the results into subsets. The most common way of injecting this support into APIs is through optional query parameters.

These query parameters are appended to the end of the URL endpoint and allow different fields filters and sorting parameters to be passed into the server affecting the results returned to be tweaked on demand by the client.

## **2.8 Conclusion**

In this chapter I have looked at the different aspects of the problem area, primarily looking at the difference between micro-services and monoliths, different types of APIs before focusing more in-depth on RESTful APIs.

## Chapter 3

# Requirements and Specifications

### 3.1 Introduction

In this section I will explore the actual requirements that the project aims to meet. The functionality of the end products will be specified creating tangible goals that can be tested against.

As previously stated, a lot of this project will be based on my experience at Hindsight migrating from a Java based monolithic architecture, to a collection of Node.js micro-services. I will look at the problems I have hit during this migration and potential ways of resolving them.

### 3.2 Requirement Gathering

No formal requirements gathering will take place, instead requirements will be pulled out of my last three years creating and working with the micro-services and their APIs at Hindsight Software. Essentially I will draw on my employment as a case study, and look to solve the issues we have encountered during our work.

#### 3.2.1 Speeding up the Development of RESTful Node.js Micro-services with the OpenAPI Specification

During development of an application, the bulk of development time should be spent building and testing the logic for the end behaviour of the program, not on the configuration and frameworks used. Choosing the right tools and libraries to allow development of new micro-services to be

as fast and fluid as possible is crucial to the longevity and extensibility of the application.

The first part of the project will involve building a RESTful micro-service using a collection of tools aiming to fulfil these criteria that we can use as a template for future micro-services at Hindsight. Additionally, facilities for features such as API parameter and payload validation, rate limiting of endpoints and support for documentation must be included. There must also be functionality supporting exporting an OpenAPI Specification representing the micro-services API. This specification will be used to generate API clients using the OpenAPI Generator tool later.

The micro-service designed will be simple, fundamentally acting simply as a test-bed and exemplar project for the libraries used, but will include all standard functionality expected for a modern application, including use of a database system.

### **3.2.2 Creating a Reliable Template TypeScript API Client**

Once the API for the micro-service has been created, I will look at creating a custom API client consuming it. The aim of this client here is to produce a client library for the micro-service API that any number of other services can install and use to communicate with the target micro-service.

The client library will need to facilitate automatic retry mechanisms on certain failed requests, support the use of query parameters for filtering and sorting of results as well as exposing the details of request and response bodies as TypeScript types to the library user.

The library must also be easily imported into other Node.js micro-services so that the other services can readily consume the API exposed by the client.

### **3.2.3 Injecting the new API Client Functionality into Auto-generated API Clients from the OpenAPI Generator**

The final aim goal of the project is to automatically generate client libraries fulfilling the above criteria using the OpenAPI Generator. The tool provides options for a number of different styles of clients, each utilising different technology stacks and frameworks, and so the first job will be determining which of the offerings is the best fit. From there I will compare the functionality against the selected option and my custom client, and then look at adding any missing pieces

into the template files used during generation.

### 3.2.4 Summary

The overarching goal of the project as a whole is to create a suite of tools that speeds up and make development of a collection of RESTful micro-services easier. Generating clients automatically from API specifications allows all of the inter-micro-service communication to be modularised, allowing micro-services to import the client library for each service that they wish to communicate with. As the APIs of a micro-service is updated and a new endpoint is added, a new API specification will be created, in turn creating a new version of the client library through the code generation tool. The library will then be published allowing all services using it as a dependency to have access to the new endpoint as soon as it goes live.

Testing tools and frameworks without examples is next to impossible and as such the project will involve the creation of an exemplar micro-service with an API made available for testing. The requirements for the micro-service framework will be written for and tested against this particular micro-service. This will allow me to properly analyse the requirements against a tangible service and also allow me to write unit tests to create further evidence of the fulfilment of the requirements.

## 3.3 System Requirements

### 3.3.1 Node.js Micro-service & its API

1. **Generate an OpenAPI Specification file:** The micro-service should be able to generate an OpenAPI Specification file representing it's API.
2. **Use well-formed URLs:** The URLs used for the API should follow best practises. Nouns should be used within the URL paths and plurals should be used for all resources.
3. **Provide endpoints for each of the major HTTP verbs:** At least one endpoint using GET, POST, PUT and DELETE should be created. GET endpoints for fetching data, POST for creating new resources, PUT for updating a resource, and DELETE for deleting a resource.

4. **Assign the correct HTTP status codes to both Success and Error Responses:**  
2xx series codes should be used for successful operations, 4xx operations should be used for any client related errors and 5xx status codes should be used for any errors relating to the server itself.
5. **Validate both request and response payloads:** The data being passed in and out of the API should be properly defined and validated. All request bodies, query parameters and path parameters should be validated and checked against the specification and any failures in the validation should throw a HTTP error with status code 400.
6. **Provide useful documentation for the API:** There must be a mechanism in place for viewing documentation related to the API that is easy to update as the API is updated. It should provide details about endpoints, parameters and request and response bodies.
7. **Utilise Rate Limiting:** The API should make use of Rate Limiting to protect itself against Denial of Service attacks.
8. **Make use of Pagination on large responses:** Large array based responses should be broken down and paginated to reduce the size of response payloads when related requests are made. The response should detail content of the current page, number of the current page and information of total number of pages related to the request. The API should facilitate accessing different pages.
9. **Provide access to a database to support CRUD usage of different HTTP Verbs:**  
The micro-service should connect to a database allowing actual data to be created, fetched, modified and deleted.

### 3.3.2 The ideal TypeScript API Client Library

1. **Support all endpoints across the API:** All of the endpoints available within the API should be accessible from the client library.
2. **Provide details of the responses types returned by endpoints:** The response bodies of each endpoints should be exposed as TypeScript types to allow consumer code to easily interact with the data provided.

3. **Provide details of all input parameters for the endpoints:** All path parameters, query parameters and body fields should be available to input for each endpoint.
4. **Ensure all important error information is correctly is passed on:** Any errors produced by the API should be properly passed onto the client consumer, including HTTP status code and any other important error details.
5. **Automatically retry requests that fail with certain HTTP status codes:** Errors relating to rate-limiting or server errors should be retried a finite number of times, after a non-determinate amount of time has passed to try to avoid the end-user having to make another request in the future.

### 3.3.3 Analysis of OpenAPI Generator and Modifications Required

1. **Generate an API client from scratch through use of an OpenAPI specification:**  
Given an OpenAPI Specification it must be possible to create a fully functional API client library using the OpenAPI Generator tool with my custom changes.
2. **Support all endpoints across the API:** All of the endpoints available within the API should be accessible from the client library.
3. **Provide details of the responses types returned by endpoints:** The response bodies of each endpoints should be exposed as TypeScript types to allow consumer code to easily interact with the data provided.
4. **Provide details of all input parameters for the endpoints:** All path parameters, query parameters and body fields should be available to input for each endpoint.
5. **Ensure all important error information is correctly is passed on:** Any errors produced by the API should be properly passed onto the client consumer, including HTTP status code and any other important error details.
6. **Automatically retry requests that fail with certain HTTP status codes:** Errors relating to rate-limiting or server errors should be retried a finite number of times, after a non-determinate amount of time has passed to try to avoid the end-user having to make another request in the future.

## 3.4 Software Development Life-cycle Methodology

### 3.4.1 Agile and Kanban

I will be employing an Agile way of working through the software development process, in particular loosely basing my methodology on the Kanban framework.

Essentially the development process will be split into a number of different chunks, which can be transitioned through a number of different stages of development.

The stages of development I will use will be **Designing**, **In Production**, **Testing** and **Ready**.

**Designing** will cover the initial design stages, from overall system architecture to RESTful API design.

**In Production** will encapsulate all the actual writing of the code, getting base functionality working.

**Testing** will involve both User-based Testing and Unit Testing the code, from using the systems end to end to actually writing the Unit Tests and ensuring test coverage is adequate.

**Ready** will simply show that the component is in its final, complete form.

In order to track the stages each piece of work is in, I will use a Jira project with a Kanban board, breaking the components into Jira Issues which can be transitioned individually through my pipeline.

## 3.5 Conclusion

In this chapter I have looked at the overall goals that my project aims to fulfil, as well as the individual requirements for each of the software components and I have detailed my plans to use the Kanban Agile Framework during development of the various solutions.

## Chapter 4

# Legal, Social, Ethical and Professional Issues

### 4.1 Introduction

In this chapter the various factors for consideration throughout this project will be reviewed. These factors ranged from legal and ethical concerns, to those regarding the social and professional impact this project would have on both Hindsight, and the community in general.

### 4.2 Legal Considerations

My main legal concern was ensuring that Intellectual Property is not breached, both concerning Hindsight and any libraries that I use in the project.

All of the libraries and tools used across the project are fully open source, allowing fair usage of the technology without any legal concern.

Information regarding specific business logic of the micro-services used at Hindsight was deliberately omitted, with a generic exemplar micro-service being developed in their stead. This allowed me to work through the project without any concerns around the breach of any confidential intellectual property that Hindsight's services contain.



### **4.2.1 Computer Misuse Act**

This project is fully compliant with the Computer Misuse Act of 1990, with a number of considerations being taken into account across the development of the project.

None of the software produced would fall under malware, and care was taken when choosing the libraries used by the project to ensure that they are all widely used and frequently updated. Choosing widely known and trusted libraries reduces the chance of third party malware being injected into the code-base, furthermore NPM provides tools to audit libraries for any known vulnerabilities adding another degree of protection.

No user data needed to be collected for this project, so any concerns around illegal or even unethical collection were avoided.

### **4.2.2 Data Protection Act**

As stated previously this project contained no data collection. That being said had collection been necessary, any data stored would have been stored on an encrypted hard-drive on my business MacBook. The MacBook is secured with both a pass-code and a YubiKey, providing adequate security for any stored data.

## **4.3 Social Considerations**

The overarching aim of this project is to produce a framework for developing RESTful micro-service architectures without spending excessive amounts of time on service configuration and communication between services. This would allow developers to spend more time on the actual business logic driving new functionality. This results in improvements to both individual developer job satisfaction as they can spend more of their time working on the interesting parts of the code-base and the profitability from the business side of things by reducing development time.

## **4.4 Ethical Considerations**

### **4.4.1 Privacy and Consent**

No data is being collected during development or by the software developed and so privacy and consent is not really an issue.

### **4.4.2 Agency and Identity**

The framework produced over the course of the project aims to offer some guidelines when developing micro-services, rather than imposing a stringent way of working.

## **4.5 Professional Considerations**

Over the course of the project I followed the same professional standards as I would when working on any Hindsight project.

All development took place on properly secured and audited machines to safeguard the developed code and any other information stored on the computers.

All code developed was treated the same way as any production ready code would be, with sensible naming conventions and code style being followed throughout.

## Chapter 5

# System Design

### 5.1 Introduction

The project aims to bring together a set of libraries and tools together in such a way that any number of micro-services can be generated and integrated together with ease, whilst still fulfilling the best practices and requirements that I have set out for them. In order to showcase how these systems work together I am going to create an exemplar micro-service and API client using them. This section of the report will explain and justify the design decisions I have made when curating the list of libraries and tools as well as detailing the actual architecture and system design of the services.

The project is going to produce three differing systems. The first will be an exemplar Node.js TypeScript RESTful micro-service, developed to showcase techniques to develop the services in general. It will be a pretty basic service, fulfilling the basic functionality expected of a modern micro-service. The second will be a TypeScript NPM module, acting as a client library that consumes the API of the micro-service and allows the Node.js service that imports the module to access the API. The final part of the project is a number of modifications made to the OpenAPI Generator typescript-axios templates. These changes will add any missing functionality to the generated client, using my NPM module as a baseline.

### 5.1.1 Client-Server API Interactions

A foundational goal of the project is to manage effectively the interactions between micro-services. Each individual micro-service will utilise a RESTful API to expose its functionality to other services within the application through the use of HTTP requests. This follows the typical Client-Server model, where the service making the request acts as the client, and the owner of the API being requested acts as the server. Any number of micro-services can make requests to any of the APIs within the overarching application and so the system design needs to facilitate this in a scalable way.

The end goal of the project is to provide the ability for all micro-services in a distributed application to produce an API client for their APIs that the other services can use without any client related code needing to be written. This will allow more development time to be put into the actual application logic of the system, reduce duplicated code across multiple services and provide a communication platform that is robust and reliable.

## 5.2 Premise of the micro-service

The aim of the project is to provide a framework of tools for creating services in general and not any particular service. The micro-service created as part of the project will be a simple example showcasing the interactions between the tools and how they are used, the actual business logic of the system having no real bearing on the project itself. The service will simulate a simple Stock Control component that could be used by any sort of supermarket or shopping application. This should allow me to demonstrate all I need to without getting bogged down in business logic complexities. It will be connected to a simple back-end database, and allow different products to be added, updated and deleted from the system.

## 5.3 RESTful API Design

The first part of the system to be designed is the REST API for the exemplar micro-service. One of the key features of a micro-service architecture is that it allows the different services to be written in different frameworks and languages, and so the API must be language-agnostic. As such all request and response bodies will use JSON. This means that although the micro-service,

based on the technology stack at Hindsight will be a Node.js service written in TypeScript, theoretically any programming language that can handle JSON payloads will be supported.

In line with REST API best practises, all endpoint resources will be named with nouns, and the plural form will be used. The noun used for the base API will be shoppingItems. A shoppingItem representing one item within the API and shoppingItems representing the collection as a whole. As such, the base endpoint will be:

`/REST/1.0/shoppingItems`

Support will need to be provided for all CRUD operations for an API, supporting the HTTP verbs GET, POST, PUT and DELETE. Both a singular shoppingItem and the contents of the collection need to be fetched, and a subset of the properties of individual shoppingItems will need to be updated. New shoppingItems will need support for creation, and existing shoppingItems need to be able to be deleted.

### 5.3.1 The API

Descriptions of the planned API endpoints can be found below:

*All request and response bodies will be formatted as JSON.*

#### 5.3.1.1 The shoppingItem Response Object

*Each shoppingItem will have the following properties:*

name: A string representing the unique name of the shoppingItem.  
category: An enumerated type representing the type of shoppingItem.  
numberOfStock: A representation of the number of the shoppingItem in stock  
inStock: A boolean representing the state of numberOfStock > 0

#### 5.3.1.2 API Endpoints

GET /REST/1.0/shoppingItems

Description:

Retrieve all shoppingItems

#### Query Parameters:

page: page number of results returned by the query  
pageSize: the number of results per page  
inStock: filter items matching the inStock state provided  
category: filter items matching the category provided

#### Additional note:

Endpoint will be paginated, pages accessed using the page query  
parameter + the page and totalPages fields within the response body

Success Status Code: 200

#### Response:

```
{  
  shoppingItems: shoppingItem[]  
  pageSize:  
  page: current page number  
  totalPages: number of total pages  
}
```

#### POST /REST/1.0/shoppingItems

##### Description:

Create a new shoppingItem

##### Request Body:

```
{  
  name: Name of the new shoppingItem,  
  category: Category of the new shoppingItem,  
  numberOfStock: Number of stock for the new shoppingItem,  
}
```

Success Status Code: 201

Response: No Response returned

#### GET /REST/1.0/shoppingItems/{name}

##### Description:

Get an individual shoppingItem by name

Path parameters:

name: The name of the item being queried

Success Status Code: 200

Response:

shoppingItem

DELETE /REST/1.0/shoppingItems/{name}

Description:

Delete an individual shoppingItem by name

Path parameters:

name: The name of the item being queried

Success Status Code: 204

Response: No Response returned

PUT /REST/1.0/shoppingItems/{name}/increaseStock

Description:

Increase the named shoppingItem stock by value provided in body

Path parameters:

name: The name of the item being queried

Request Body:

```
{  
  value: number to increment by  
}
```

Success Status Code: 200

Response:

shoppingItem

PUT /REST/1.0/shoppingItems/{name}/decreaseStock

Description:

Decrease the named shoppingItem stock by value provided in body

Path parameters:

name: The name of the item being queried

Request Body:

```
{  
    value: number to decrement by  
}
```

Success Status Code: 200

Response:

shoppingItem

PUT /REST/1.0/shoppingItems/{name}/category

Description:

Update the named shoppingItem category

Path parameters:

name: The name of the item being queried

Request Body:

```
{  
    category: new category value  
}
```

Success Status Code: 200

Response:

shoppingItem

## 5.4 Technologies and Libraries

Having introduced the exemplar micro-service, I will now walk through the libraries and frameworks that I have collected together to support the generation of composable micro-services. The specific choices have been made in order to put together a way of working that meets the requirements for micro-service development that were identified in Chapter 3. The results of this project are intended to be used for the generation of production level code and so it is



important to use well-established libraries in order to provide robust and well-tested solutions to the problems in micro-service architectures that have previously been found when developing large scale projects.

#### **5.4.1 Node.js**

The micro-service will be implemented using the Node.js run-time environment. This allows me to model the micro-service after the micro-services used at Hindsight and has the added benefit of providing me with familiarity, meaning I do not have to learn anything new. Node.js is used industry wide for back-end APIs due to its asynchronous and non-blocking event handling and is also relatively lightweight whilst maintaining good performance. Through the use of NPM, a package manager for Node.js, hundred of thousands of public libraries and modules can be imported and utilised, providing access to well-tested and reliable code. This allows Node.js developers to avoid writing common code, as well as providing them to modularise their code into both public and private packages that can be shared across code bases. During the development of both the micro-service and the client I will be importing a number of public NPM packages into my code.

#### **5.4.2 TypeScript**

Node.js services are traditionally written in vanilla JavaScript, but through the use of NPM support can also be added for TypeScript.

TypeScript at its core is just a superset of JavaScript, aiming to alleviate some of the foundational issues with language by adding features such strong type and generics support to JavaScript. Upon compilation, TypeScript is compiled back into vanilla JavaScript and so can typically be used anywhere JavaScript is used.

Having used both JavaScript and TypeScript during development of micro-services, I could never go back to JavaScript. Personally I find the support added by TypeScript allows me to develop software far, far faster. The addition of types allows methods to define exactly what they return, and when tied in with a good IDE, the code prediction facilities provided by TypeScript allow a far more agile way of working. TypeScript code is also naturally documented by its types, parameters types are explicitly specified allowing some issues to show up as errors in the IDE before any code is even run through type checking.

Because TypeScript is a superset of JavaScript, if for some reason some component does need to be written in raw JavaScript, that is also accepted by the compiler.

### **5.4.3 OpenAPI Specification**

The OpenAPI specification is a language-agnostic RESTful API specification that is widely used across the industry. In my opinion it is the most user-friendly specification out there and provides a huge ecosystem of different tools and integrations. It will be the API specification I will be using for this project.

The specification document can be written in JSON or YAML, and provides details about all aspects of the REST API. When tied into language-specific tools the API specification can be used to generate both API clients and servers.

### **5.4.4 Micro-service specific libraries**

In this section I will describe and justify the libraries and frameworks I have decided to use in development of the exemplar micro-service.

#### **5.4.4.1 Express.js**

Express.js is a framework I will be using as the base of my micro-service. The Node.js ecosystem contains a number of widely used frameworks, but Express.js is arguably the most popular. It is lightweight and extensible, and handles a lot of the more tedious aspects of setting up a Node.js application. Express allows middleware to be easily plugged in, allowing a developer to easily import and use libraries for handling a huge volume of different use cases. Common middleware includes cookie and session support as well as things like custom error handling.

The Express framework is also the framework that most of the micro-services at Hindsight use, and so I am also very familiar with it.

#### **5.4.4.2 TSOA**

TSOA is an NPM library that aims to treat the controller and data models of an application as the single source of an truth for the API. It defines routes for the endpoints and validates

request payloads based on information provided the controllers and data models. The beauty of this approach is that as changes are made to either of these two parts of the system, the API is automatically updated to support the changes. TSOA allows tightly-woven typings across the application, all the way from the database layers up to the responses returned by the controllers.

The library uses a number of decorators in conjunction with TypeScript in order to define how the generated API will function. The developer decorates the methods in the controllers with information about the endpoint URL, HTTP verbs to support and default response code that the endpoint will return. Parameters within the controller methods are then decorated to define URL path parameters, query parameters as well as the request body payload. TSOA automatically picks up the response for the endpoint from the return type of the method.

It generates routes for a number of different frameworks, including Express, and automatically validates incoming data against the API definitions. If the format of the data sent is not as expected, or the fields do not match, then it will reject the request and throw an error.

TSOA can also generate an OpenAPI Specification for the API. Through usage of the JSDoc markup language, tonnes of additional information can be added to allow the generated specification to act more like living documentation for the API. Example requests and responses payloads can be included, as well as endpoint and parameter descriptions.

The library does a lot of the heavy lifting for a micro-service, especially the tasks concerning routing and data validation, alongside handling the OpenAPI Specification requirement and so TSOA will be an integral part of the technology stack I use.

#### **5.4.4.3 Swagger-UI-Express**

Swagger-UI-Express is another NPM package that generates Swagger-UI based API Documentation. It essentially takes an OpenAPI Specification in JSON format, and generate a HTML page representing the specification in an easily-readable format.

It also allows requests to be made to the API directly from the web page, allowing the API to be utilised without any client related code. This allows easy testing of the API during development without having to manually type out convoluted CURL commands in order to make requests.

When injected as middleware, the package can be used to create a documentation endpoint that automatically serves the Swagger-UI HTML page. By tying in both this package, and TSOAs

specification generation, we can automatically publish API documentation. This means that as soon as code changes are published, a new API specification will be generated and documentation for the modified API will follow instantly.

#### **5.4.4.4 Express-Rate-Limit**

This NPM package is a very simple rate limiting middleware. It is lightweight, but comes with the caveat that it recognises requests as coming from the same entity, and so all users will share the same rate limit timer. As this project is focused on the interactions between an API client and server, this limitation is not currently an issue, and this library provides the rate limiting proof of concept needed for the clients functionality to be testing against.

#### **5.4.4.5 Mongoose**

In order to properly support the CRUD requirements of the API, a simple and lightweight MongoDB database will be integrated into the micro-service.

Mongoose is an NPM module that handles the vast majority of the logic involved in creating and interacting with a MongoDB database, cutting down on a huge volume of otherwise boilerplate code. It is schema-based, and so can also validate requests made to and from the database. It also provides support for injecting TypeScript types into its model definitions, allowing strongly typed response definitions to be returned from database queries. This allows the same type definitions to be used across all layers of the micro-service, and when used with TSOA decorators can provide API definitions using these typings. This means if the underlying database format and specifications are changed, then the API definition itself will be changed too.

#### **5.4.4.6 Mongoose-Pagination-V2**

The Mongoose-Pagination-V2 NPM package is a library that provides access to a Mongoose plugin. The plugin facilitates the use of a wrapper method on database queries that splits up the result of the query into paginated chunks.

Pagination can be fairly easily obtained using vanilla Mongoose via the Skip and Limit operators. Skip simply tells the query to Skip x number of documents, whereas Limit gives an upper bound on the number of documents returned. Pagination can be simulated by setting the value of Skip

to the Page Number of the request multiplied by the value of Limit. This will set the starting position of the query to the "Page Number" requested. However this approach does not easily expose the number of available pages to the consumer of the request and so makes iterating through the pagination less friendly. It is also not scalable for huge collections, and the use of Skip and Limit can become CPU and I/O bound for large sets of documents.

The Mongoose-Pagination-V2 library instead is cursor-based and so does not have the same scalability concerns. It also provides all of the extra meta-data related to pagination and makes exposing exactly what data is being displayed to the user painless.

### **5.4.5 Client specific libraries**

In this section I will look at the tools and libraries I will within the API Client, as well as how I plan on publishing the client as a module.

#### **5.4.5.1 Creating the Client as an NPM Library**

Seeing as the project is based on Node.js micro-services, configuring the client library as an NPM library makes the most sense. The client can easily be packaged and published to the NPM registry, either publicly or privately, and the micro-services can then readily install and utilise it. This also allows me to import other NPM packages as dependencies for my client, saving on the amount of otherwise unnecessary home-brewed code. Physically publishing the package is out of scope of this project, but the fundamental setup for the process will be done.

#### **5.4.5.2 Axios**

The key functionality of an API client is making requests to an API, and so choosing the right library for making HTTP requests is key.

The library Request is a very popular and easy to use solution for making HTTP requests, and is what a lot of the Hindsight micro-services use to communicate between services. The library was however deprecated in early 2020, and so following developer best practises, an alternative should be used.

Axios is a widely used alternative to Request. It provides a lot of powerful functionality like the ability to cancel requests, and automatically parses JSON data and encodes URL parameters.

Requests can be heavily configured, providing options for things like transformation of request data into different formats, serialisation of parameters and the use of proxy servers.

Axios is also promise based. Node.js is asynchronous, and promises are a mechanism for dealing with asynchronicity without having to resort to a large number of callbacks. This usage of promises allows Axios to provide the ability to intercept both requests and responses before they are passed back to the calling function. A key use case for this in my project will be facilitating automatic retries on certain failed requests.

#### **5.4.5.3 Axios-Retry**

Axios-Retry is a simple NPM library that provides access to an Axios interceptor that allows failed requests to be caught and sent again. The interceptor can be configured to retry the request any amount of times, and can take in arguments for delays between retries as well as conditions for which status codes get retried. Although fundamentally a very small library, it saves me from writing code unnecessarily.

#### **5.4.6 OpenAPI Code Generator**

The OpenAPI Generator provides a huge number of template configurations supporting different programming languages and frameworks. In this section I will explain why I chose the typescript-axios option.

##### **5.4.6.1 The typescript-axios templates**

The base of my handmade REST Client is the Axios HTTP library, and so utilising the typescript-axios configuration for the OpenAPI Generator should make pulling changes across the software fairly simple.

The generated client is also setup as an NPM package, and so fulfils my concerns around easily importing and using the client in Node.js services. As a baseline, the generated code only has TypeScript and Axios as dependencies and therefore has no deprecated libraries in use. This should make it a fairly stable option longterm as Axios and TypeScript are both likely to stick around.

## 5.5 Microservice

The micro-service is a Express based Node.js service broken down into a three layer architecture: the *data-layer*, *middleware* and *service-layer*.

It is worth noting again that the complexities of the actual behaviour of the service are unimportant, and all of the design decisions made for this example can be used across any real micro-service. The focus is on the general architectural decisions and choice of libraries and tooling.

### 5.5.1 data-layer

This layer will handle all of the business logic of the service, namely interacting with the database in order to fetch the information sent and requested through the API.

#### 5.5.1.1 models/ShoppingItemModel.ts

The models directory will be where the Mongoose models and schemas for the service are defined. The schemas define the fields required for the documents within a collection, allowing validation by Mongoose automatically when the create and update wrappers are called. The model acts as interface for interacting with the schema, wrapping the CRUD operations boilerplate and packaging it into a simple object.

This file will define the Mongoose schema for the `ShoppingItem` collection. It will define the shape of the data, and create the actual collection in MongoDB. It will then create a Mongoose model that exposes all of the CRUD operations related to the collection. This will allow the data-agents to interact with the database without writing unnecessary boilerplate code. The `Mongoose-Paginate-V2` plugin will also be wired into the schema providing isolated access to the paginate functionality within only this collection. If any further collections are to be added, the pagination function will not be needed as default and so the dependency is unnecessary.

### 5.5.1.2 data-agents/ShoppingItemAgent.ts

The data-agents acts as a middleman between the controllers and the models. They take the data passed in from the controllers, do any processing or formatting required for the actual database queries. They then pass in the formatted data into the wrapper methods from the model with any configuration options relevant to the request. They then do any post-processing on the results that the controller might need before bubbling the result back up. The data-agents also handle and wrap any errors that might be thrown by the models and transform them into more helpful HTTP errors.

The `ShoppingItemAgent` will be the data-agent responsible for calling and handling the `ShoppingItemModel`. It will contain a method related to each of the REST API endpoints, and using the data passed into it from the `ShoppingItemController` will interact with the `ShoppingItem` collection. It will handle any logic involved in both the pre and post-processing of data and will map an errors thrown into one of the errors from the `ErrorLibrary`.

#### `createShoppingItem()`

Method related to the creation of a `ShoppingItem` within the database. Will call the `findOneAndUpdate()` model method. The use of the `ShoppingItem` name as the unique identifier for the collection means that namespace collisions need to be avoided and so using the `findOneAndUpdate` function with the `upsert` flag set will overwrite the previous document with the name if the same name is used again.

#### `getShoppingItem()`

Fetch an individual `ShoppingItem` from the collection based on a name field passed in. Simple usage of the `findOne()` model method using a query built around the name field. If no results are returned then a custom `ShoppingItemNotFoundError` is thrown with



an assigned HTTP status code of 404.

#### `getShoppingItems()`

Method for requesting multiple `ShoppingItems`, either the entire collection, or a subset based on parameters for `ShoppingItemCategory` or the `inStock` boolean.

Results will be paginated, and information related to current page and total number of pages will be returned in the response.

Different pages can be accessed through usage of the `Page` parameter and the number of results per page can be changed through the `PageSize` parameter. If no results are found an empty array will be returned.

#### `updateShoppingItemCategory()`

Simply update the `ShoppingItemCategory` of the document with the queried name. Uses the `findOneAndUpdate()` model method and the new, updated version of the document will be returned. If no results are found for the name passed in then a custom `ShoppingItemNotFoundError` is thrown with an assigned HTTP status code of 404.

#### `increaseShoppingItemStock()`

Increment the `NumberOfStock` field of the document with the name passed in by the number in the `value` field. If the stock level is raised from 0 then the boolean `InStock` will be flipped. Again the method uses `findOneAndUpdate()` and returns the updated document.

If no `ShoppingItem` is found with the queried name then a custom `ShoppingItemNotFoundError` is thrown with an assigned HTTP status code of 404.

#### `decreaseShoppingItemStock()`

Decrement the `NumberOfStock` of the targeted document. Validation catches negatives number and sets the `NumberOfStock` to 0 in these cases. Model method used is `findOneAndUpdate()` with the modified form of the document being returned. If no `ShoppingItem` is found with the queried name then a custom `ShoppingItemNotFoundError` is thrown with an assigned HTTP status code of 404.

`deleteShoppingItem()`

Delete the targeted `ShoppingItem` using the name field. Uses the model function `findOneAndRemove()` to handle the deletion. If no `ShoppingItem` is found with the name passed in then a custom `ShoppingItemNotFoundError` is thrown with an assigned HTTP status code of 404.

## 5.5.2 middleware

This will be where the logic used across the micro-service will be abstracted and stored, mainly TypeScript types and error definitions. This will also be where the literal middleware is configured such as the rate limiting configuration and error handler.

### 5.5.2.1 `enums/ShoppingItemCategories.ts`

Simple enumeration types library for the micro-service. Helpful for ensuring only certain field values can get through the TSOA validation and across the controllers and data-agents.

A simple enumeration type detailing the valid `ShoppingItem` categories.

### 5.5.2.2 `types/ErrorLibrary.ts`

The `ErrorLibrary` takes the generic `ErrorWrappers` and contextualises them by assigning a value to the unique `ErrorIdentifier` field.

This allows the error to be easily identified at a glance. We can simply look at where the specific error in the library is thrown and

investigate the cause from there. Not so important in this small service, but in a large distributed system, identifying the root cause of errors is extremely important and so this added context is a necessity as errors will be potentially bubbling up across multiple services.

#### 5.5.2.3 ShoppingItem.ts

Simple TypeScript interface for defining the ShoppingItem type for use across the service. Additionally usage of JSDoc annotation around the fields allows TSOA to add descriptions of each property to the OpenAPI Specification on generation.

#### 5.5.2.4 ErrorHandler.ts

Custom error handling middleware. Has explicit handling of the ErrorWrapper errors thrown by the data-agents as well as handling the validation thrown by TSOA. This ensures the additional meta properties of the error are returned to the user and that the status codes are set as expected. If the error is neither of these formats it gets piped onto a handler for generic errors which uses a default error message and status code if they are missing.

All of the error bodies are returned as JSON to maintain the language-agnostic nature of the API.

#### 5.5.2.5 ErrorWrapper.ts

Custom extensions of the default Node.js Error object. Adds custom fields for additional details, status code and a unique error identifier. This generic ErrorWrapper is further extended by implementations for each expected HTTP status code, where the extended class simply defines the status code for its parent ErrorWrapper object.

### 5.5.2.6 RateLimiter.ts

Super simple implementation of the Express-Rate-Limit package, essentially assigning configuration values.

### 5.5.3 service-layer

The *service-layer* will handle all of the logic associated with serving and processing the HTTP requests. This will be where all the controllers are defined and where the TSOA generated route files are created.

#### 5.5.3.1 controllers/ShoppingItemController.ts

The ShoppingItemController is where all of the TSOA related definitions and decorating is configured.

The base ShoppingItemController class is decorated using the Route() decorator. This defines the base URL that all of the method specific URLs will append. In this case it is set to "/REST/1.0/shoppingItems".

JSDoc notation will be used across the controller class, primarily before method definitions to provide TSOA with information about method descriptions and examples for fields and parameters.

Additional decorators are injected in to help document any expected errors that the API will throw. This is done through the use of the @Res decorator and will add the error information to the OpenAPI Specification.

Other Decorators used:

@Get() @Post() @Put() @Delete():

Map URL and HTTP Verb to the

function.

Under the hood URL is set to {baseUrl}/{URL}

Get("/{name}") => GET /REST/1.0/shoppingItems/{name}

@Path():

Define a path parameter, maps the field within the URL to a variable.

@Query():

Define a query parameter, maps the field within the URL to a variable.

@Body():

Define the request body, maps the request body to a variable.

@Response():

Define errors, subtly different from @Res and only works with more static errors like the TSOA ValidationError.

@Example():

Define an example response for the generated OpenAPI Specification.

See the API documentation for detailed information on the API.

createShoppingItem()

Mapped to:

POST /REST/1.0/shoppingItems

Passes in the request body information to the createShoppingItem() function in the ShoppingItemAgent and decorates the response status code as 201 to represent the resource successfully being created.

getShoppingItems()

Mapped to:

GET /REST/1.0/shoppingItems

Calls getShoppingItems() method from the agent and passes in the optional query parameters for filtering and pagination if they are present. Simply returns the response from the agent alongside a 200 status code.

getShoppingItem()

Mapped to:

GET /REST/1.0/shoppingItems/{name}

Passes in the path parameter name to the getShoppingItem() agent method. Returns the response from the function with a HTTP status code 200.

updateShoppingItemCategory()

Mapped to:

PUT /REST/1.0/shoppingItems/{name}/category

Call the updateShoppingItemCategory() agent function with both the name path parameter and the request body fields. As we are returning the updated document set the status code to 200.

increaseShoppingItemStock()

Mapped to:

PUT /REST/1.0/shoppingItems/{name}/increaseStock

Call the increaseShoppingItemStock() agent function with both the name path parameter and the request body fields. As we are returning the updated document set the status code to 200.

decreaseShoppingItemStock()

Mapped to:

PUT /REST/1.0/shoppingItems/{name}/decreaseStock

Call the decreaseShoppingItemStock() agent function with both

the name path parameter and the request body fields. As we are returning the updated document set the status code to 200.

`deleteShoppingItem()`

Mapped to:

`DELETE /REST/1.0/shoppingItems/{name}`

Pass in the name path parameter to the `deleteShoppingItem()` agent method. Decorate the response status code as a 204 to inform the API consumer the resource was deleted.

#### 5.5.3.2 routes/Routes.ts

TSOA generated routes for the REST API based on information provided in the controller files.

#### 5.5.4 Root Files

Some files relating package and actual server configuration do not really belong in any of the layers and so are stored at the root level.

##### 5.5.4.1 App.ts

The `App.ts` file contains all the logic for injecting all of the middleware into the application such as the custom error handlers, the rate limiting functionality from `RateLimiter.ts` as well as the mongoose database connection configuration.

The file also defines a static route `"/REST/1.0/documentation"` that serves the SwaggerUI HTML page from the `swagger-ui-express` package. The `generateHTML()` method is called on the TSOA generated OpenAPI Specification `"swagger.json"` and is returned as the routes response.

#### 5.5.4.2 Server.ts

Simple configuration file for starting up the Node.js server.

#### 5.5.4.3 tsoa.json

Configuration files for TSOA specifying the framework that the routes should be generated for as well as the directories for the different files to be pulled from and where others will be generated.

### 5.5.5 Commands Used

#### 5.5.5.1 Generating routes through TSOA

```
tsoa routes
```

#### 5.5.5.2 Generating an OpenAPI Specification from the Controllers

```
tsoa spec
```

#### 5.5.5.3 Compiling & building the Service

```
tsc --outDir dist --experimentalDecorators
```

#### 5.5.5.4 Starting the Service

```
node dist/Server.js
```

## 5.6 API Client

The next part of the project will be determining the functionality that a generic API Client should have. This will allow me to work out what feature sets need to be added to the generated client templates later on.



In order to do this I will create a custom handcrafted API client for my micro-service. The API Client will be built as an NPM Package essentially acting as a wrapper for other imported libraries. The architecture will be simple, with the package essentially broken down into two directories, *utils* and *microservice-client*.

### 5.6.1 utils

The *utils* directory will store the more abstract functionality, the methods for generically making HTTP requests and handling errors without the context for the specific micro-service API.

#### 5.6.1.1 Http.ts

The `Http` file contains the `Http` class, built on top of the `Axios` library.

##### `constructor()`

The constructor generates an `AxiosInstance` using the `baseUrl` and optional `customHeaders` parameters that will be used for making the HTTP requests. Upon creation the constructor will also inject the interceptor for handling automatic retries of certain requests into the `AxiosInstance` using the `axios-retry` library.

##### `request()`

Generic wrapper for the `AxiosInstance.request()` method. Injects a custom error handler into the the response for the `request()` method to insure all of the relevant error information from the API is pulled out before being parsed as an `ErrorWrapper` object.

The method signature allows inputting of the HTTP verb to be used, the URL endpoint to be requested as well as optional fields for request body and query parameters.

A switch statement is used based on the HTTP verb to determine how the request is going to be sent, and what parts of the request

will be provided.

Subsequent wrappers will be provided for this generic function each providing specific HTTP verb support.

`get()`

Wrapper for the `request()` method which injects the HTTP verb GET into the method. Method signature provides a required parameter URL which will also be passed into the `request()` method when called. Optional parameters for query parameters also provided.

`post()`

This wrapper injects in the HTTP verb POST and contains required parameters for both URL and the request body, both of which are passed in the `request()` method. Optional parameters for query parameters also provided.

`put()`

This wrapper injects in the HTTP verb PUT and contains required parameters for both URL and the request body, both of which are passed in the `request()` method. Optional parameters for query parameters also provided.

`delete()`

Wrapper which injects the HTTP verb delete into the `request()` method alongside a URL, request body and optional query parameters.

#### 5.6.1.2 `HttpVerbs.ts`

Simple enumeration type for defining the HTTP Verbs used in the library.

### 5.6.1.3 ErrorWrapper.ts

Custom Error format shared with the same file in the micro-service. Used to ensure that no important information or context is lost when the errors are handled and passed through the library.

### 5.6.2 microservice-client

This directory will be where the actual export functionality of the library will be located. It will contain a single file, handling the logic for directing the HTTP requests using the utility functions from the files in the *utils* directory.

#### 5.6.2.1 MicroserviceClient.ts

The file defines the `MicroserviceClient` class. The class contains a member variable importing the `Http` class and wraps its methods with contextual information about the API endpoints and responses.

`constructor()`

Simple constructor that creates a new `Http` instance using the microservices base URL.

`createShoppingItem()`

Calls the `Http.post()` method passing in the a body passed from the method parameters and URL for the associated endpoint

`getShoppingItems()`

Calls the `Http.get()` method with the URL for the API endpoint and and passes in optional query parameter object from the optional method parameters

`getShoppingItem()`

Calls the `Http.get()` method with a custom URL with path parameters

from the method parameters

`increaseShoppingItemStock()`

Calls the `Http.put()` method with the endpoint URL and a body  
built from the method parameters

`decreaseShoppingItemStock()`

Calls the `Http.put()` method with the endpoint URL and a body  
built from the method parameters

`updateShoppingItemCategory()`

Calls the `Http.put()` method with the endpoint URL and a body  
built from the method parameters

`deleteShoppingItem()`

Calls the `Http.delete()` method with the endpoint URL and a body  
built from the method parameters

### 5.6.3 Root Files

#### 5.6.3.1 `Index.ts`

Basic file defining the functions that the library will expose as exports.

### 5.6.4 Building and Publishing the Library

Physically publishing the package is out of scope of the project, but I will explain how it would be done.

Creating and publishing NPM modules is simple. The `package.json` file specifies a number of fields including name of the package, current version of the module as well as information about its git repository. Once the index file has been defined and the export members specified, the module can be published using the NPM or Yarn package managers.

The module will then be stored on the module registry and can then be imported using a package manager.

## 5.7 Modifying the OpenAPI Code Generator

### 5.7.1 Generating the Client using the Base Templates

In order to generate the Client I will use the `openapi-generator-cli` NPM package. This means that I do not have to fork and run the whole software stack locally until I have to.

Once the OpenAPI Specification has been created, the Client can be generated through simply running the command:

```
openapi-generator-cli generate
-g typescript-axios
-i PATH_TO_SPECIFICATION
-o OUTPUT_DIR
--additional-properties=npmName=CLIENTNAME,
```

The command will generate a new API client based on the base `typescript-axios` templates. By passing in the `npmName` field in as an additional property will cause the `package.json` files and other related NPM files to be automatically generated. This is necessary for both publication and for installing the libraries dependencies.

### 5.7.2 Determining the differences between the generated client and my client

The API clients generated contain a lot of overlapping features with my home-brewed client, as well as some functionality I omitted from it. Support for query parameters is provided and types are generated from the response and request body definitions allowing full type support for the method signatures generated. It also supports a number of different authentication and authorisation mechanisms including OAuth and use of JWT tokens, although my API does not utilise any of these technologies. All of the functionality is implemented using the same `AxiosInstance` integration within my implementation of the client allowing a lot of the feature sets of each to be readily injected into the other.

The key pieces of missing functionality are the custom error handling for the failed API requests, with the bloat-heavy Axios errors being thrown by default, and no support provided for the automatic retries on failed requests.

### 5.7.3 Modifying the generated client

Due to the use of the same Axios library, and its AxiosInstance methods, most of the code from my custom client, and the clients generated by the OpenAPI Generator, is generally interchangeable.

Rather than jumping straight into modifying the Mustache template files, I can instead modify the raw TypeScript files that is output by the generator to test the interactions between the new code and the existing. This will make debugging any issues clearer than trying to get the generation to work on top of the actual code that is produced.

Adding support for the automatic retries on failed requests will be relatively simple, all I need to is add the axios-retry as a dependency and inject the same code I use within my code, into the AxiosInstance that the generated API used.

Parsing the errors thrown by the client will be handled in a similar way. The file defining the custom ErrorWrapper class can simply be cut out of my code and pasted within the generated library, and the code for handling the error thrown can be cut out of my AxiosInstance definitions and used to catch the errors thrown by the new API client.

I will then test that the behaviour of the client is as expected by creating a test script that calls each of the API endpoints. The inputs will be a mixture of valid and invalid parameters to check to see that successful requests are still handled correctly, and that any errors thrown are sent in format that is expected. If all of the changes work as expected, I can move onto abstracting all of these changes into the template files.

### 5.7.4 Modifying the template files

The modifications that need to be made to the template files used in the generator will be relatively simple. All of the functionality related to actually populating fields with values from the specification is already in the generated client, and so all of the additions made will all be static blocks of code.

The custom `ErrorWrapper` class file can simply be cut and pasted within the template file directory, and the code defining up the error handler can be pulled from my API and copied into the templates. This will all be appended onto the template code relating to setting up the `AxiosInstance` that forms the basic of the API requests.

In the same vein getting the automatic retries working simply involves adding the `axios-retry` library into the template `NPM package.json` file before cut-pasting the module setup into the `AxiosInstance` definitions.

### 5.7.5 Generating a client with the new templates

Once the templates have been modified I can generate a new client using them. The client generated should be identical to the that I modified. The command used to generate the client using the new templates is almost identical to the previously used command, with the addition of another field specifying template directory.

The command is:

```
openapi-generator-cli generate
-g typescript-axios
-t PATH_TO_TEMPLATE_DIRECTORY
-i PATH_TO_SPECIFICATION
-o OUTPUT_DIR
--additional-properties=npmName=CLIENTNAME,
```

### 5.7.6 Conclusion

In this chapter I have looked at the libraries I will be using in development of the software as well as API design and software architecture of the solutions. I have also described and made justifications all of the design decisions made, specifically around the interactions between the different layers of the micro-service and the file breakdown of my custom API client. A breakdown and explanation of how I plan on adding functionality into the OpenAPI Generator has also been described.

## Chapter 6

# Testing and Evaluation

### 6.1 Introduction

This section will detail my plans on how I will test the framework that I have put together and how I will evaluate how well it meets the objectives and specifications that I have defined for it.

Creating a test suite for micro-services typically involves a lot of different moving parts, and I will begin by explaining how I collected a number of libraries and tools to create a comprehensive set of unit tests. I will showcase these tools by creating an exemplar test suite for the shopping micro-service, with all of the design and architectural decisions made during the development of the test suite being explained. Whilst the test suite will be written for the exemplar micro-service developed for this project, the methodology for the design and implementation of the unit tests will be applicable for any micro-service built using the template that I have created.

Each of the software components need to be evaluated against the original objectives and specifications lined out for them. The second half of the chapter will contain this evaluation with justification for whether I feel that the individual objectives have been met.

### 6.2 Unit Testing with SuperTest, Jest and Nock

My Unit Tests for both the micro-service and the client will use a number of key libraries, SuperTest, Jest and Nock. Whilst they will test the sample services I have created, the main aim will be to produce a suite of tools for general micro-service testing.



### 6.2.1 Supertest

Supertest is a Node.js library used for testing HTTP. By passing in the server export from the App.ts we can perform Unit Tests on the actual HTTP requests made through the server.

All aspects of the response can be tested including headers and status code, with the body of the response being the focal point of my testing. This will allow me to ensure that everything returned by the API is returned in both the format and shape that is expected, and the appropriate errors are thrown on respective requests.

### 6.2.2 Jest

Jest is a lightweight and very user-friendly JavaScript testing framework developed by Facebook. The tests themselves are split into separate processes and run in parallel, resulting in extremely fast and performant test suites.

Jest requires very little boilerplate configuration, and pretty much works out of the box making it very easy to get straight to writing Unit Tests on the code-base.

The framework also has the Istanbul code coverage tool built in. Istanbul tracks how much of our code is actually covered by our Unit Tests such as:

- Functions: Are all the functions and methods tested
- Branching: Are all branches in the decision-making logic tested
- Statements: Are all variable assignments, return values etc tested

The reports generated by Istanbul detail what code isn't covered too, allowing the test suite to be expanded to cover these areas. The higher the code coverage, the more of the code-base that is tested, which in turn should reduce the numbers of bugs introduced to the system.

It should be noted that Code Coverage and Test Coverage are different. Where Code Coverage tests how much of the code is tested in some way, Test Coverage measures those tests against actual Specifications and Requirements, testing the actual coverage of the expected functionality. As such where Code Coverage shines for Unit Tests, Test Coverage is a very good metric for Acceptance Testing.

For this project I will only be testing Code Coverage as the bulk of my physical tests will be Unit Tests.

### **6.2.3 Nock**

Nock is a Node.js library used for mocking HTTP requests. Nock intercepts the HTTP requests sent by the server or application, and rather than actually forwarding the error to the destination it returns a static response body or error defined by the developer.

It is particularly useful for interacting with external APIs as it mimics the request sent by the consuming service without actually interacting within the API. This allows the unit tests to be run without connecting to the internet or running any of the dependencies of the service being tested locally.

Nock allows creation of highly configurable mocks and every component of the requests and responses can be refined. This includes everything from headers to status codes.

Once a request is intercepted the mock is consumed. This allows different instances of the exact same request to return different responses. This is particularly useful for testing things like rate limit handling, as the first few requests which are limited will fail with a 429 status code, but subsequent requests that are not limited will return successfully.

All of the unit tests for my API Client will make use of Nock so that it can be tested in isolation from the micro-service.

### **6.2.4 Code Coverage Target**

Oftentimes it can be extremely challenging to hit one hundred percent code coverage. While I will aim to hit as close to complete coverage as possible, I will be aiming for a minimum of 85% of overall code coverage for the important files within both the client and micro-service. There are likely to be certain parts of the code that aren't feasible to reach during unit-testing, particularly around handling of unexpected failures around the database connections.

It is important in software projects to identify a suitable balance between effort involved in increasing coverage targets and the benefits gained. Whilst the process of identifying test objectives is out of scope for this project, we have informally used a risk-based testing approach (Amland 2000) to focus tests on those areas of code with highest exposure, and reduced testing

effort on low exposure areas of code where the potential impact of faults is low. The important result is that the test tools that have been identified fully support risk-based testing.

### **6.2.5 Unit Testing the Micro-service**

The testing of the micro-service will be broken down into two key sections, testing that the controllers work and respond as expected, and that the data-agents process and store the data correctly.

#### **6.2.5.1 Testing the Controllers**

In order to test the controllers and the REST API exposed through them I will make use of the SuperTest library. SuperTest allows programmatic HTTP requests to be made to the server within the unit tests, allowing testing of the same URL endpoints that the live API will expose.

Upon calling the HTTP requests I will use Jest assertions to verify that the fields returned within the response are correct. To do this I will have to create static response bodies for each of the requests that I wish to test and compare them against those actually returned by the request. If the fields match those expected, then the API is working as expected.

Each of the URL endpoints within the micro-service will have a small suite of unit tests written for them. Requests that should trigger successful responses will be validated as will as requests that should trigger expected errors, such as doing GET requests on resources that do not exist. The HTTP status codes will be tested, ensuring that they are being assigned correctly and any response or error bodies will also be checked to ensure that they contain all of the content that they should. Essentially any documented request and response for the API should be tested.

#### **6.2.5.2 Testing the Data Agents & Models**

Testing the data-agents and models involves interacting with the actual database operations. We do not want our unit tests ever interacting with live data and so standard practise is to connect to a completely separate test-focused database. This is easy to do with Mongoose, and we can simple create a new database connection as part of the unit tests. Once the database connection is defined we can simple inject a number of test documents into the collections to act as our test data.

A small sample of test documents will be created upon the start up of the unit tests and saved into the test database. Different values for each of the model fields will be used across the suite of documents. This will allow tests to be written that verify that the correct subset of the collection is returned for the different filtering and sorting options on the data-agent methods.

Each of the operation involved in fetching existing documents will utilise the test documents created upon database startup. This will allow assertions to be run against the actual document definitions reducing boilerplate and keeping the unit tests lightweight. However, for the operations involving modifying or creating documents the tests will consist of two stages. The first will be defining the parameters used for the creation or modification of the resource, as well as creating a JSON representation of the resource after the operation has been defined. The method can then be called assertions can be tested to ensure that the operation did not fail. In order to test that the actual data saved is what is expected, a second request fetching the new data needs to be made. This can be done directly through the mongoose model using the primary key used in the create or modify operation, and the response can be tested against the previously defined JSON object. This two-stage process ensures that not only does the operation not fail, the actual information stored in database is exactly what is expected.

### **6.2.6 Unit Testing the Handcrafted Client**

Testing the client will involve using Nock to intercept and mock the responses of all the outgoing requests that the client sends out. These will simulate both successful and failed requests and allow all expected cases to be tested against.

#### **6.2.6.1 Intercepting HTTP Requests to the micro-service with Nock**

There are a number of different behavioural aspects that need to be tested within the client. The successful responses need to be properly handled and parsed into objects that are expected, any errors thrown by the API need to be caught and handled without losing any crucial information and certain failed requests need to be retried automatically.

Testing handling of successful responses is easy with Nock. The endpoint simply need to be passed into the Nock constructor before calling the `.reply()` method with both the status code and response body that the mocked request should return. An example can be seen below.

```

const request = Nock('http://localhost:3000/REST/1.0')
  .get('/shoppingItems')
  .reply(200,
    {
      page: 1,
      totalPages: 1,
      shoppingItems: [{
        "name": "apple",
        "category": "Fruit",
        "numberOfStock": 110,
        "inStock": true
      }]
    })

const res = await client.getShoppingItems();

```

The above code simply initialises a new variable *request* and assigns it to the request made to the */shoppingItems* endpoints. The spoofed response is assigned the status code 200, imitating a successful operation, and an example of the correct response payload is also injected in. The *getShoppingItems()* can then be called. As far as the methods calling the endpoints are concerned they will have interacted with what they believe to be the live API, but behind the scenes the request will have been intercepted and replaced with the Nock. The consuming code will then treat the faked response the same as a legitimate one and the outputs of the methods can be asserted against to check that they are working correctly.

Failed requests can be built and used in the exact same way. A status code and an error payload just need to be assigned to the Nock request to endpoint and it's good to go.

```

const failedRequest = Nock('http://localhost:3000/REST/1.0')
  .get('/shoppingItems/mango')
  .reply(404,
    {
      "errorIdentifier": "ShoppingItemNotFound",
      "message": "Shopping Item not found with params:
        {

```

```

        \"name\": \"mango\"
    }"

    })

    const res = await client.getShoppingItem('mango');

```

The code above simply creates a Nock imitating a request made for a resource that does not exist. The method that consumes the request is expected to throw an instance of the *ErrorWrapper* object using the data from the error payload and status code. The unit tests for the failed requests will test that the expected object is thrown and that it contains the fields and values that are expected. This is easily done through simple assertions on the object properties within the unit tests.

Once a request has been intercepted and assigned a matching Nock, the Nock is consumed. Any subsequent requests will need to have separate Nock objects assigned to them. This allows the automatic retry facilities of the client to be tested quite easily. The rate limit retry system works by making the first request, checking the status code returned in the case of an error and if supported, a number of retries of the same request will be made with incremental delays between them. Essentially a successful retry system will contain a number of failed requests, followed by a final successful request. With Nock we can define all of these requests and the system should handle them as if they were interacting with an actual rate-limited API. An example for testing a request made successfully after three retries is given below.

```

const failThrice = Nock('http://localhost:3000/REST/1.0')
    .get('/shoppingItems')
    .times(3)
    .reply(429)

const retriedSuccessRequest = Nock('http://localhost:3000/REST/1.0')
    .get('/shoppingItems')
    .reply(200,
        {
            page: 1,
            totalPages: 1,
            shoppingItems: [{
                "name": "apple",

```

```

        "category": "Fruit",
        "numberOfStock": 110,
        "inStock": true
    }]
  })
  const res = await client.getShoppingItems();

```

There are two Nocks defined above. The first behaves the same as the previous spoofed errors with one exception, it has the additional *.times()* function call attached. All this does it tell Nock that the request should exist three times. This means that when the axios-retry library makes retries the request after the initial failure, it will receive two subsequent rate-limited errors.

The second Nock is a typical successful mocked response for the same endpoint. When the retry is triggered a third and final time this response will be returned. When the *getShoppingItems()* method is called the output should contain the information provided by the final success. As with the previous tests this can be tested by simple asserting the fields provided in the response body outputted by the method.

## 6.3 End-to-End Testing the Auto-generated Client

### 6.3.1 Migrating the Unit Tests from my Client

In order to test the client generated from the OpenAPI Generator I will pull out the unit tests from my handcrafted client and copy them across. The tests will require some very light modification but all of the Nock descriptions and assertions on the responses should be transferable. The only modifications should be changes to the names of methods called and any variance in the method signatures.

The clients package.json file will also need some additional libraries to be imported in order for the unit tests to run, however this will have no bearing on the actual functionality of the client. These will include the Jest framework and Nock library.

## 6.4 Results of the Tests

In this section I will analyse the Code Coverage reports generated for each of the services to get a general idea of how comprehensively the test suites cover the code-base. The test suite has been designed to focus on the actual business logic of the micro-service and the clients. This is where the core functionality of the systems are located, and some areas of the code do not have specific tests covering them, specifically around failures on database connection or any sort of network connection error. This will lead to some expected gaps in the coverage, which should be shown via line details in the report. If these gaps in coverage are subsequently identified as high risk areas in a quality review, the reports provide sufficient detail to identify the areas for which additional tests should be defined

### 6.4.1 Reading the Code Coverage Reports

The generated reports contain a detailed breakdown of which files and components of the code are covered, as well as details of the degree to which they are covered. It breaks the coverage down into four key sections, Statements, Branches, Functions and Lines. It also lists the lines in the code which are not covered at all in tests. For the purpose of this report I am going to omit analysis of the Line coverage, as I think the other 3 aspects are more important.

#### 6.4.1.1 Statements

This covers the Statements within the software. Variable assignment, function calls and return values all fall under under this umbrella and the reports details how many of these statements have been executed when the tests have been run. Statement coverage should typically be relatively high as you want the core of your code base to be heavily tested, covering most of the expected use case for the software. However if there are some niche branching paths or function calls that can be difficult to reach with unit tests, it might be unfeasible, or simply not worth the time investment, to strive for full coverage.

#### 6.4.1.2 Branches

The branch coverage shows how many of the branches in the applications control flow have been executed. Typical examples for this are your standard if or switch statements. Maintaining



100% coverage on control flow can be very difficult, and there are likely to be branches within the code that are not really expected to execute unless in the case of things like server failures. Writing unit tests covering these cases can be time consuming, and as they are not a typical use case of the software, oftentimes are not worth the time investment.

### 6.4.1.3 Functions

Function coverage should be as close to 100% as possible and unit tests should be in place for every important function in the application. The function report details how many of the defined functions have actually been executed within the unit tests.

### 6.4.1.4 Lines

The line coverage tracks how many of the lines within the applications codes have been executed and in the same vein as branch coverage, can be extremely hard to reach perfect coverage. That being said aiming for as high a value as possible within sensible time constraints is best.

## 6.4.2 The Micro-service

The report below shows the Code Coverage report for the ShoppingItems micro-service.

Figure 6.1: Code Coverage Report for the Exemplar Micro-service

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	91.54	72.73	90.38	91.27	
src	87.1	0	80	86.67	
App.ts	100	100	100	100	
Server.ts	0	0	0	0	1-5
src/data-layer/data-agents	98.21	94.44	100	100	
ShoppingItemAgent.ts	98.21	94.44	100	100	40
src/data-layer/models	100	100	100	100	
ShoppingItemModel.ts	100	100	100	100	
src/middleware	84.38	37.5	60	83.33	
ErrorHandler.ts	75	38.46	50	71.43	41,46-50
ErrorWrapper.ts	90	33.33	66.67	90	36
RateLimiter.ts	100	100	100	100	
src/middleware/enums	100	100	100	100	
ShoppingItemCategories.ts	100	100	100	100	
src/middleware/types	90.91	100	75	90.91	
ErrorLibrary.ts	90.91	100	75	90.91	22
src/service-layer/routes	88.89	75.86	94.44	88.89	
routes.ts	88.89	75.86	94.44	88.89	115,138,161,186,209,231,272,275,299,305,309
src/service-layer/controllers	96.77	87.5	100	96.3	
ShoppingItemController.ts	96.77	87.5	100	96.3	106
Test Suites: 2 passed, 2 total					
Tests: 28 passed, 28 total					
Snapshots: 0 total					
Time: 5.365 s					
Ran all test suites.					
Done in 7.46s.					

The files we are particularly interested in tracking are *ShoppingItemAgent*, *ShoppingItemModel*, *RateLimiter* and *ShoppingItemController* as this is where the vast majority of the business and application logic is stored.

#### 6.4.2.1 Statements

Starting with the statement coverage on *ShoppingItemAgent* we can see that the file has 98% coverage. The report shows the line 40 of the file is uncovered, and upon inspection we can see that it is handling any unexpected errors thrown by the database. This code is relatively unimportant, more acting as a safety net should an unexpected issue occur. Testing this kind of code is relatively unimportant as it has no real bearing on the day to day usage of the application and so the test suite doesn't cover it. In a perfect world test cases would be written for it, but with time constraints both for this project, and that you would also typically see in a consumer facing application the time cost doesn't really justify it. The 98% coverage exceeds my previously stated desired coverage of 85%.

*ShoppingItemModel* has 100% statement coverage as does *RateLimiter*.

*ShoppingItemController* reaches 97% coverage, with a single uncovered line at line 106. Looking at line 106 in the codebase we can see that it is handling any unexpected errors thrown. As stated before, writing test cases for these events is time consuming and so I opted to move on. Once again, coverage exceeds that of the target.

#### 6.4.2.2 Branches

The branches follow the exact same pattern as the statement report, with the files without full coverage being impacted by the same lines once again and so I will not go into any more detail on the matter here.

*ShoppingItemModel* *RateLimiter* both had 100% branch coverage, *ShoppingItemAgent* reached 94% coverage and *ShoppingItemController* reached 86% coverage. All files exceeded target.

#### 6.4.2.3 Functions

All of the files being watched maintain 100% function coverage in the report.

### 6.4.3 The Handcrafted Client

Figure 6.2 shows the report for the Code Coverage of my custom API client.

Figure 6.2: Code Coverage Report for the Handcrafted Client

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	95.29	83.33	100	95.18	
src	0	100	100	0	
index.ts	0	100	100	0	1-3
src/microservice-client	100	100	100	100	
MicroserviceClient.ts	100	100	100	100	
src/util	96.67	83.33	100	96.55	
ErrorWrapper.ts	100	100	100	100	
Http.ts	95.45	81.25	100	95.24	43-44
HttpVerbs.ts	100	100	100	100	
ShoppingItemCategories.ts	100	100	100	100	
Test Suites: 1 passed, 1 total					
Tests: 15 passed, 15 total					
Snapshots: 0 total					
Time: 6.483 s					
Ran all test suites.					
Done in 8.92s.					

The key files we will be looking at are *MicroserviceClient* and *Http*. The other files are mainly utility files providing Enums and Type declarations.

#### 6.4.3.1 Statements

Statements coverage of *MicroserviceClient* is 100% and coverage of *Http* is 95%. Both reaching the targeted coverage.

If we dive into the uncovered lines within the *Http* file we can see lines 43 and 44 handle any unexpected errors when physically sending the request. Errors under this category would typically be any sort of internet connection issues and as the focus of my testing is on the actual interactions with the API, I have not created any test suites for these cases.

#### 6.4.3.2 Branches

The coverage for branches within *Micro-serviceClient* is 100% and for *Http* is 81%.

The *Http* coverage fails to reach the 85% coverage, but upon inspection we can see that it is only one branch that is not covered, the same code mentioned in the statement report analysis. Covering this case has no real bearing on the success of the project, and so I will opt to leave it

be for now.

### 6.4.3.3 Functions

Both targeted files reached 100% function coverage within the test suites.

### 6.4.4 The Auto-generated Client with the modifications

The report for code coverage of the API client generated from the OpenAPIGenerator templates I modified are shown in Figure 6.3. The unit tests were pulled out of my handcrafted client and so with some very minor method signature changes, the test-suite itself is mostly identical across the two clients.

Figure 6.3: Code Coverage Report for the Generated Client with my Modifications

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	85.44	59.45	77.46	85	
api.ts	86.48	57.8	73.21	86.93	239,272,276,283,315,322,350,357,390,401,405,409,435,439,446,479,483,490,627-695
base.ts	91.53	69.05	100	89.8	55-56,104,110-111
configuration.ts	30	0	50	22.22	67-73
index.ts	100	100	100	100	
jest.config.ts	0	100	100	0	2
Test Suites: 1 passed, 1 total					
Tests: 16 passed, 16 total					
Snapshots: 0 total					
Time: 6.118 s					
Ran all test suites.					
Done in 8.13s.					

I am not going to break down the report for the generated client. There is a lot of functionality within the client I do not care about in the context of this project, and my focus is that the actual test cases I have transferred from my own client pass, as these represent the objectives I will be measuring the success of the generator changes against. As we can see in the figure above, all of the tests passed.

## 6.5 Evaluation against Project Requirements and Specifications

In this section I will look back at the objectives for the project and determine whether they have been met for each piece of software.

### 6.5.1 The Exemplar Micro-service

In order to assess how well the framework meets the project objectives, we will now evaluate them against the exemplar micro-service. This is a simple service that could be used by any shopping application and is designed to showcase the combinations of libraries that would be used in any real micro-service.

#### 6.5.1.1 The Objectives for the Micro-service

1. *Generate an OpenAPI Specification file*
2. *Use well-formed URLs*
3. *Provide endpoints for each of the major HTTP verbs*
4. *Assign the correct HTTP status codes to both Success and Error Responses*
5. *Validate both request and response payloads*
6. *Provide useful documentation for the API*
7. *Utilise Rate Limiting*
8. *Make use of Pagination on large responses*
9. *Provide access to a database to support CRUD usage of different HTTP Verbs*

#### 6.5.1.2 Generate an OpenAPI Specification file

The TSOA library provides the facilities for taking the API definitions defined into the controller files and generates an OpenAPI Specification file from them. It does this through the command:

```
yarn run tsoa spec
```

The output from this command is either a YAML or JSON OpenAPI Specification.

#### 6.5.1.3 Use well-formed URLs

When designing the ShoppingItem API for my micro-service, the API design best practices stated in Chapter 2 were used as a basis.

Those best practises are as follows:

1. Resources should be named with nouns
2. Resource names should be pluralised
3. Individual resources should be fetched through use of identifiers

The base URL for the REST API is the following:

```
/REST/1.0/shoppingItems
```

By looking at the URL we can see that the ShoppingItem resource is named after a noun, and the noun is in it's plural form.

The API endpoint for fetching an individual Shopping item is:

```
/REST/1.0/shoppingItems/{name}
```

The endpoint is an extension of the base URL, where name is the unique name of the targeted individual resource. This name acts as the unique identifier for the resource which in turn checks off the third best practice.

#### **6.5.1.4 Provide endpoints for each of the major HTTP verbs**

There are four major HTTP verbs that this objective is referencing: GET, POST, PUT and DELETE. Each of these verbs have associated endpoints within the ShoppingItems API.

1. GET

```
/REST/1.0/shoppingItems  
/REST/1.0/shoppingItems/{name}
```

2. POST

```
/REST/1.0/shoppingItems
```

### 3. PUT

```
/REST/1.0/shoppingItems/{name}/category  
/REST/1.0/shoppingItems/{name}/increaseStock  
/REST/1.0/shoppingItems/{name}/decreaseStock
```

### 4. DELETE

```
/REST/1.0/shoppingItems/{name}
```

#### 6.5.1.5 Assign the correct HTTP status codes to both Success and Error Responses

Correctly assigning HTTP status codes are crucial if consumers are going to successfully use the REST API. The micro-service API follows the standardised assignment of status codes explored in Chapter 2.

Success responses for any request that returns a resource are assigned a 200 status code alongside the response body. For the creation of a new resource the status code 201 is assigned and a 204 status code is used when a resource is deleted.

When a resource is not found a status code of 404 is used, and in the case of a bad request 400 is used. Any unexpected error on the server side is assigned a 500 status code. A request that breaks the rate limits of the API is assigned a 429 status code.

#### 6.5.1.6 Validate both request and response payloads

The TSOA library is used for creating all of the routing and API generation for the micro-service. Out of the box TSOA handles all of the validation for the API endpoints, using TypeScript types to validate both incoming and outgoing payloads and blocking any that do not match the expected format.

When an endpoint is defined that takes in a request body, such as the endpoint associated with the creation of a new `ShoppingItem`, the body is cast as a TypeScript type. When a request hits the API endpoint TSOA takes the body from the HTTP message and compares it to the

assigned type. If there are missing required fields, has unexpected fields or the body is of an unexpected format then it will throw a `ValidationError` with a 400 status code to the consumer.

Similarly the response body is defined by a TypeScript type and assigned to the return type of the controller method associated with the endpoint. As TypeScript is a strongly-typed language this tightly weaves the response into the lower level code handling the business logic of the request. If the method that the controller calls returns a different type then TypeScript itself will throw an error on compile time.

#### **6.5.1.7 Provide useful documentation for the API**

The micro-service has the ability to generate an OpenAPI Specification representing a full definition of its API. The `swagger-ui-express` library takes this document and parses it into a HTML page fully detailing the endpoints using the information provided in the specification. An API endpoint that serves this page can be assigned to the server, allowing the creation of a full documentation endpoint that any user can hit to get the details of the API.

Whilst this library generates the actual page, it works in tandem with TSOA. When TSOA generates the OpenAPI specification it scours the types and methods used for both additional decorators and JSDoc notation. These optional additions to the files allows developers to add a large volume of extra information and descriptions to various fields, methods and payloads. This extra data is then injected into specification file. By documenting the code itself, developers can automatically generate fully descriptive API documentation.



Figure 6.4: Documentation snippet from the micro-service documentation URL

GET /REST/1.0/shoppingItems

Get array of ShoppingItems

Parameters [Try it out](#)

Name	Description
inStock boolean (query)	Filter results based on whether stock value is greater than 0 or not <input type="text" value="--"/>
category string (query)	The Category to filter results by Available values : Frozen, Fruit, Vegetable, Dairy, Bakery <input type="text" value="--"/>
page number(\$double) (query)	Page of results to return <input type="text" value="page - Page of results to return"/>
pageSize number(\$double) (query)	Size of page to return <input type="text" value="pageSize - Size of page to return"/>

Responses

Code	Description	Links
200	Ok	No links

Media type:  Examples:

Controls Accept header

Example Value | Schema

```
{
  "name": "apple",
  "category": "Fruit",
  "inStock": true,
  "numberOfStock": 14
},
{
  "name": "cabbage",
  "category": "Vegetable",
  "inStock": true,
  "numberOfStock": 14
}
```

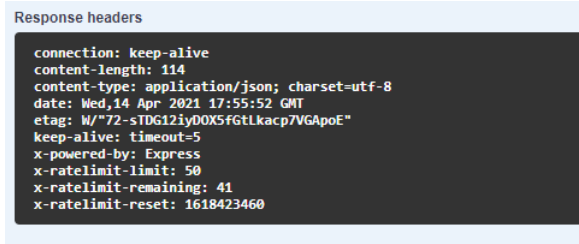
Figure 6.4 above shows part of this HTML page. Information relating to each of the query parameters, an overall endpoint description and example response payload are provided.

Outside of acting as documentation, the swagger-ui-express library also allows requests to be made directly through the web-page, saving time writing complex and tedious CURL commands when testing the API.

#### 6.5.1.8 Utilise Rate Limiting

The express-rate-limit library is used to add rate-limiting functionality to the micro-service. Whilst it is a relatively simple integrations, applying a global rate limit to the service rather than a per user limit, it works fine as a proof of concept. Outside of intercepting requests that would break the limit and throwing a 429 error, it also inject a number of rate limit meta-data headers into the responses of the API.

Figure 6.5: Example of Rate Limit response headers provided by the API



The above figure shows the response headers include information about the rate limit itself, number of remaining requests before the limit is exceeded and a timestamp for when the current limit expires. This information can be used by the API consumer in order to request any failed requests, or plan how to make a series of requests to the API.

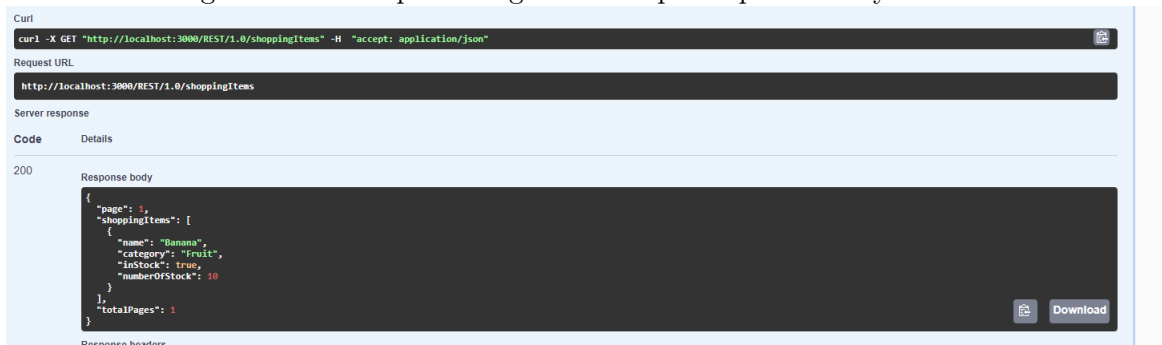
#### 6.5.1.9 Make use of Pagination on large responses

The micro-service breaks down requests returning multiple resources into chunks called pages. Through usage of the `mongoose-paginate-v2` library this pagination is done directly at the database request level. When the data-agent queries the database, the library wraps the request and returns a subset of the result based on optional parameters passed into the method signature. It also provides a number of meta-data fields providing additional information about the pagination including total number of pages and the current page number.

By handling the pagination at the data-agent layer, we do not have to do any additional processing on the request in order to pull out the pages resulting in simpler and more accessible code. It also allows a number of query parameters to be provided at the controller level which can simply be plugged into the optional paging parameters in the data-agents methods signature. This allows the API consumer to cycle through the pages by passing in the desired information into the query parameters, with default options selected if none of these parameters are provided.

A paginated response example is shown in Figure 6.6 below.

Figure 6.6: Example of Paginated Response provided by the API



#### 6.5.1.10 Provide access to a database to support CRUD usage of different HTTP Verbs

The micro-service connects to a MongoDB database which stores all of the services resource information. When the `createShoppingItem()` method is called a new document is created in the `ShoppingItems` collection and operations fetching, modifying and deleting the resource(s) can be triggered through the API and data-agent.

### 6.5.2 The Handcrafted Client

By evaluating my handcrafted client against its objectives, I can determine whether it successfully fulfilled its function.

#### 6.5.2.1 The Objectives for the Handcrafted Client

1. *Support all endpoints across the API*
2. *Provide details of the responses types returned by endpoints*
3. *Provide details of all input parameters for the endpoints*
4. *Ensure all important error information is correctly is passed on*
5. *Automatically retry requests that fail with certain HTTP status codes*

#### 6.5.2.2 Support all endpoints across the API

The client contains methods for calling each of the endpoints exposed by the API, with the exception of the endpoint:

GET /REST/1.0/documentation

An API consumer would never have to call this endpoint during an integration, and it would instead be used by the API server to create a public facing documentation page.

The methods and their corresponding endpoints are detailed in the table below.

Table 6.1: Method name to URL mapping for the Handcrafted Client

Method Name	URL Endpoint
getShoppingItems()	GET /REST/1.0/shoppingItems
getShoppingItem()	GET /REST/1.0/shoppingItems/{name}
createShoppingItem()	POST /REST/1.0/shoppingItems
deleteShoppingItem()	DELETE /REST/1.0/shoppingItems/{name}
updateShoppingItemCategory()	PUT /REST/1.0/shoppingItems/{name}/category
increaseShoppingItemStock()	PUT /REST/1.0/shoppingItems/{name}/increaseStock
decreaseShoppingItemStock()	PUT /REST/1.0/shoppingItems/{name}/decreaseStock

### 6.5.2.3 Provide details of the responses types returned by endpoints

The handcrafted client is built using TypeScript which allows us to build up Types for each of the response bodies returned by the API. The Axios library uses generics to provide details of the expected response type of the request. By casting the return types of each method to the custom response types I defined, Axios will inject this typing into the data object that it returns.

Figure 6.7 shows an example of this casting.

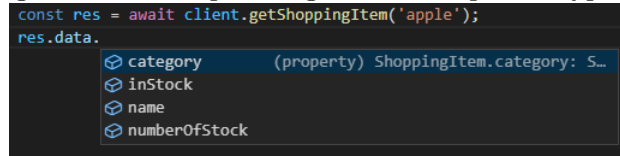
Figure 6.7: Example Response Type Casting

```
public getShoppingItem = async (itemQuery: string): Promise<AxiosResponse<ShoppingItem>> => {
```

Consumers of the method can then access all of the fields from the response when the data field is pulled out of the methods output.

Figure 6.8 shows the usage of this data field.

Figure 6.8: Example Usage of the Response Typings

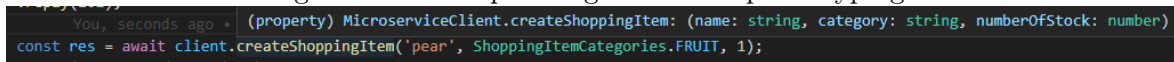


#### 6.5.2.4 Provide details of all input parameters for the endpoints

As with detailing the shapes of the responses, TypeScript allows type casting for each of the input parameters for methods. These input parameters each represent the various path, query and request body parameters for the API endpoints, and proper type definitions within the method signatures allow full details of these parameters to be exposed to the user.

Figure 6.9 below shows how the client exposes the typings of these fields through the method signature.

Figure 6.9: Example Usage of the Request Typings



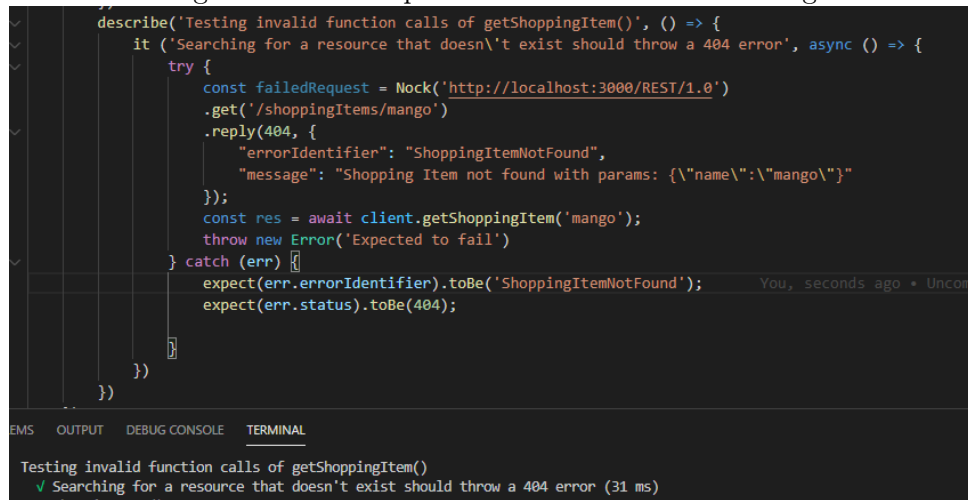
#### 6.5.2.5 Ensure all important error information is correctly is passed on

The handcrafted client makes use of a custom Axios interceptor to catch and handle any errors thrown during the requests. The fields are pulled out of the error payload and are used to create a local instance of the same `ErrorWrapper` that the the micro-service itself throws. This ensures that both the server and client in the request cycle are using the same error format when a request fails.

The unit test suite contains a number of test case that explicitly check that these errors are handled as expected.

Figure 6.10 shows a passing unit test that checks the error thrown on a request that is expected to fail.

Figure 6.10: Example Unit Test for Error Handling



```
describe('Testing invalid function calls of getShoppingItem()', () => {
  it('Searching for a resource that doesn\'t exist should throw a 404 error', async () => {
    try {
      const failedRequest = Nock('http://localhost:3000/REST/1.0')
        .get('/shoppingItems/mango')
        .reply(404, {
          "errorIdentifier": "ShoppingItemNotFound",
          "message": "Shopping Item not found with params: {\\"name\\":\\"mango\\"}"
        });
      const res = await client.getShoppingItem('mango');
      throw new Error('Expected to fail')
    } catch (err) {
      expect(err.errorIdentifier).toBe('ShoppingItemNotFound');
      expect(err.status).toBe(404);
    }
  })
})
```

Testing invalid function calls of getShoppingItem()  
✓ Searching for a resource that doesn't exist should throw a 404 error (31 ms)

### 6.5.2.6 Automatically retry requests that fail with certain HTTP status codes

The client makes use of the *axios-retry* library to retry failed requests. The library injects an Axios interceptor that checks the HTTP status code of the failed response, and if it falls under the subset of codes that should be retried, it will re-submit the request. Each request can be retried a total of 3 times before the request is aborted and an error message is bubbled back up to the user.

The test suite contains a set of tests that validate this behaviour. These tests cover both successful and failed retry attempts and give enough coverage to give confidence in the feature. The key focus for these tests is retrying rate limited requests.

The unit tests covering retrying rate limited requests are shown below in Figure 6.11.

Figure 6.11: Example Unit Test for Retrying Rate Limited Requests

```
describe('Testing a rate limited getShoppingItems request', () => {
  it('Should return the list of shoppingItems after succeeding on the 4th attempt', async () => {
    const failThree = Nock('http://localhost:3000/REST/1.0')
      .get('/shoppingItems')
      .times(3)
      .reply(429)
    const retriedSuccessRequest = Nock('http://localhost:3000/REST/1.0')
      .get('/shoppingItems')
      .reply(200, {page: 1, totalPages: 1, shoppingItems: [{name: "apple", category: "Fruit", numberOfStock: 110, inStock: true}]});

    const res = await client.getShoppingItems();
    expect(res.data).toEqual({page: 1, totalPages: 1, shoppingItems: [{name: "apple", category: "Fruit", numberOfStock: 110, inStock: true}]});
  })
  it('Should throw a 429 when the final retry fails', async () => {
    const failFourTimes = Nock('http://localhost:3000/REST/1.0')
      .get('/shoppingItems')
      .times(4)
      .reply(429);
    try {
      const res = await client.getShoppingItems();
      throw new Error('Expected to fail');
    } catch (err) {
      expect(err.status === 429)
    }
  })
})
})
```

IS OUTPUT DEBUG CONSOLE TERMINAL

Testing a rate limited getShoppingItems request  
✓ Should return the list of shoppingItems after succeeding on the 4th attempt (1684 ms)  
✓ Should throw a 429 when the final retry fails (1693 ms)

### 6.5.3 The Modified Auto-generated Client

The final component of the project was taking the default Typescript-Axios template for the OpenAPIGenerator and modifying it to produce a generated client that will satisfy the same objectives as my home-brewed client. Evaluation of the objectives against a client generated for the micro-service API are detailed in the following section, however a client generated for any API would workt.

#### 6.5.3.1 The Objectives for the Auto-generated Client

1. *Support all endpoints across the API*
2. *Provide details of the responses types returned by endpoints*
3. *Provide details of all input parameters for the endpoints*
4. *Ensure all important error information is correctly is passed on*
5. *Automatically retry requests that fail with certain HTTP status codes*

#### 6.5.3.2 Support all endpoints across the API

The OpenAPIGenerator uses a number of Mustache template files to define the shape of the generated code. Information is pulled out of the OpenAPI Specification for the API and used to populate parameters within these template files. These populated templates are then used

by a templating engine to create the files containing the code. As the specification generated by the micro-service contains definitions for each of the API endpoints, the generated client will in turn contain methods handling all of them too.

The only endpoint missing from the specification, and subsequently the client, is the endpoint serving the API documentation. This however is by design as this endpoint is not needed for operation of the API itself.

The table below shows a mapping of the various API endpoints to the associated client methods.

Table 6.2: Method name to URL mapping for the Generated Client

Method Name	URL Endpoint
getShoppingItems()	GET /REST/1.0/shoppingItems
getShoppingItem()	GET /REST/1.0/shoppingItems/{name}
createShoppingItem()	POST /REST/1.0/shoppingItems
deleteShoppingItem()	DELETE /REST/1.0/shoppingItems/{name}
updateShoppingItemCategory()	PUT /REST/1.0/shoppingItems/{name}/category
increaseShoppingItemStock()	PUT /REST/1.0/shoppingItems/{name}/increaseStock
decreaseShoppingItemStock()	PUT /REST/1.0/shoppingItems/{name}/decreaseStock

### 6.5.3.3 Provide details of the responses types returned by endpoints

The templates used for client generation are also based on the Axios library. As with my client, the generated client injects types into the generic Axios method calls to define the shape of the response bodies returned when the requests are made.

There are however some slight differences between the two clients. Where my custom types were statically defined, the types within the generated are built using information provided from the specification file themselves. Functionally this produces the same result, with the library providing details of the response bodies for each request through the data object returned by Axios.

### 6.5.3.4 Provide details of all input parameters for the endpoints

As with the response details, the Typescript code generated uses information from the specification to create strongly typed methods for each of the API endpoints.



Each method contains typed method parameters representing each of the URL parameters, query parameters or request body. Individual URL or query parameters are assigned a TypeScript type best matching the type in the specification. In the case of fields that can only match specific values, enumeration types are generated. Any JSON payload from the specification generates a TypeScript interface that the field is then assigned.

#### **6.5.3.5 Ensure all important error information is correctly is passed on**

Both the custom error wrapper and the error handler middleware are pulled out of my handcrafted client, and injected in an identical way into the templates for the OpenAPI Generator. This results in the behaviour being identical across the two clients, with the test-suite covering cases validating this.

#### **6.5.3.6 Automatically retry requests that fail with certain HTTP status codes**

As with the error handling, the mechanism for retrying failed requests is shared across both the handcrafted client, and the client template for the generator. Functionally the two clients handle the retries identically, and both services contain unit-tests covering this functionality.

## **6.6 Conclusion**

In this chapter I detailed the strategies used when testing the software as well as recording the results of this testing. The systems were then evaluated against the specifications originally set in Chapter 3.

## Chapter 7

# Evaluation of the New Framework

### 7.1 What is the Purpose of this Project

Over the course of the project I have built up a collection of libraries and tools that can be used to rapidly speed up the development of sets RESTful micro-services. The overarching aim was to streamline the process of creating the API for a micro-service and then consuming the API from another service.

Typically there can be a number of headaches during these processes and a number of the issues that I have come across whilst working at Hindsight served as inspiration for this project.

Some of the core issues the project aimed to address are summarised below:

- Ensuring the request and response payloads are being handled correctly between services
- Having to manually look up and define the TypeScript types representing the API payloads for every API a service consumes
- Losing details about thrown errors when requests travel through multiple micro-services
- Lots of duplicated client code, with each micro-service consuming the same API containing different variations of the same code
- Lack of support for handling soft failure cases, such as rate limiting. Particularly a problem when interacting with large scale commercial APIs such as the GitHub and Bitbucket APIs

## 7.2 The Framework

Whilst not explicitly a formal framework, the suite of libraries and tools used in the creation of the example micro-service and API client will be utilised in the same way a framework would. Overall both of the exemplar applications have fulfilled their requirements fully which bodes extremely well for live usage of the framework.

### 7.2.1 Creating the API for a Micro-service

One of my biggest takeaways from the development of the Shopping micro-services was how little time was actually spent on it. Creation of the API itself took a couple of hours at most, with the library TSOA handling the vast majority of the heavy lifting involved. The only real work involved from me was ensuring that the actual design of the API was solid and met the requirements I had set for it. This is done through defining both the URL endpoints for the API and the TypeScript Type definitions for each of the request and response payloads that the API would receive and send respectively. All of the incoming and outgoing data is fully validated by TSOA using these definitions, ensuring that only the explicitly defined data formats and fields are processed without rejection.

TSOA also generates an OpenAPI Specification representing the API which can be pulled out and used in client generation without any input from me. The specification file has the added benefit of generating living documentation for the API when used in tandem with the Swagger-UI-Express library. As modifications are made to the API a new specification can be generated, which in turn leads to new documentation being served to the user. Substantially less time needs to be spent documenting the API with all of the information used being pulled directly out of the code, whether it be out of the TypeScript Type definitions, or through JSDoc style comments. This intertwines the code-base and the API documentation very tightly which in turn helps to ensure that as changes are made to one, the other accurately represents them.

Error handling and pagination of results were both also specified within the Objectives for the micro-service. During development of the micro-service I designed and integrated a custom error message format that can be used to pass far more detailed error information through sets of micro-services. By designing a format that can be used across all future micro-services we can ensure that the errors are handled correctly across the board, with minimal loss of any meta-data

associated with them. This error format was tied into custom middleware for handling errors thrown by the service and ensuring it was correctly packaged and passed forward as the new format.

Pagination was injected directly into the MongoDB database. By handling it at the lowest level far less boilerplate wrapping responses from database queries needs to be written, and parameters specifying page size and number can be passed directly from the controller level down to the data level. It also helps to standardise usage of pagination and the same query parameters specifying the size and iteration of the pages can be used across all of the APIs for a set of micro-services.

The framework was pain free to use, and allowed me to get the service up and running extremely quickly. Although the micro-service created was simple and just an example for how it would be used in the future, the framework allowed me to quickly get stuck into the actual business logic and functionalities of the service. This is key to successfully shipping software, as wasting time on what is essentially configuration prevents you from delivering features to end users.

### **7.2.2 Modular Client Generation using API Specification Files**

The modifications made to the template library used by the OpenAPIGenerator produced an API client that satisfied all of the requirements defined. It allows an OpenAPI Specification generated by TSOA to create a fully functional API client that covers every endpoint defined in the API.

The clients generated take the form of NPM modules which allow a single instance of the client to be generated for each micro-service and installed as a dependency to all the other services that need to consume that API. This completely removes a great deal of duplicated code that would otherwise be shared across multiple services. Although not implemented in the example micro-service due to a lack of a CI pipeline, one use case would be to automatically generate and update a client for the micro-service every time changes are made to the micro-service. This would ensure that all consumers of the API always tracks the latest version and is risk free as long as the API follows semantic versioning to avoid any breaking changes. Setting up a pipeline to produce a workflow was out of scope for this project, but implementing the solution would be straightforward.

The generation of the client is done through running a single command, and requires no further

input from the developer outside of importing the module saving a great deal of time that would otherwise be used creating handcrafted requests.

The client library is built using TypeScript, and through the response and request definitions within the API specification, types are generated for all inputs and outputs of the API. These types are then assigned to the return types and method parameters of the functions that call the API and provide details of all of these fields to anybody importing the library. This allows any developers to easily integrate the client into other services as they can pull all of the fields out of the response objects safely and ensure that they are sending the correct data fields and format to the API when making requests.

Overall the client works seamlessly and as it is generated from the specification without any need for modifications will save a lot of time when integrated into micro-services at Hindsight.

### **7.3 Using the Framework at Hindsight**

Moving forward the aim is to integrate the framework at Hindsight. It will be used to create any new micro-services at Hindsight and any existing services that consume the new ones will make use of the client generation to communicate with the APIs.

As of the time of writing, a new micro-service is being built from the ground up using all of the tools covered in this project and will be the first service fully integrating the framework. Currently the service is functional but not yet release ready, utilising the same set of libraries and tools that this project has put together.

For the most part the client generation has been untouched, with only a minor modification being made to how the base URL of the API endpoints is generated. The production micro-services used at Hindsight sit inside Docker containers, and so getting the correct address for the service required a little refining within the template.

Once the business logic of the service is fully tested and signed off on, it will be deployed and an instance of a service using the outputs of this project will be live.

## 7.4 Conclusion

In this chapter we have shown that the framework developed in this project meets the core objectives by evaluating it through the development and testing of an exemplar micro-service. The framework is now being used for the development of production level code at Hindsight and initial feedback indicates that only a minor refinement of the framework was needed.

## Chapter 8

# Conclusion

### 8.1 Overview of the Project

We started the project by looking at the lessons learned during my time at Hindsight, and identified a number of common problems that have been observed when working with a RESTful Micro-service architecture. These issues were used as the inspiration for the project and were used to produce both the Objectives for the project as a whole, and the Requirements for the actual framework produced. We looked at best practices and standards for RESTful APIs in general alongside two contemporary alternative API methodologies. We also looked at what a micro-service is at its core, and how the approach differs from a more traditional monolithic software architecture.

Ethical concerns for the project were then detailed and the actual Requirements for the software components were defined. Once I had the list of Requirements, work began on the design of the framework itself, both on the development on the code, and the process for ensuring it was tested properly.

Overall the project has been a success. As shown in the latter half of Chapter 6 and in Chapter 7 the Requirements for the different software components were all met, and the actual framework produced works well.

The framework provides the facilities to quickly develop RESTful micro-services and allows generation of an API client from the services API specification. The micro-services serve API documentation from a documentation endpoint providing details for entire API. This includes

all the information needed to effectively make use of the API, whether it be information about how requests are to be made to the API or the shape and format of the responses returned by it.

A unit test-suite was developed for each of the examples showcasing the framework, covering all of the cases laid out by the Software Requirements. In Chapter 6 we showed that this test-suite not only passed fully, but also provided ample code coverage of the software. This level of code coverage allows more confidence to be had in the tests, as it shows exactly what is and is not covered.

The biggest indicator of success though is the fact that the framework is already starting to see usage at Hindsight, showing that it is in a good enough state to produce production level code used in live services.

## **8.2 Enhancing the Framework in the Future**

As we increase the number of micro-services at Hindsight, and as we look to improve things like scalability of the individual services, additional features and refinements to existing features are going to be necessary. The framework and client generation is dynamic, with changes easily injected into the existing code, allowing the software to be molded as we need it to. While nothing is essential at the time of writing, below I have detailed some potential additions and changes that could be made that fell out of the scope of the project.

### **8.2.1 More advanced Rate-limit support for the micro-services**

The express-rate-limit library used in the exemplar micro-service is handy, but relatively simple. As a proof of concept for handling rate limiting it works fine, and for general usage of a small scale web application it's functional. However, it pools all of the individual users of the service under one roof, and applies a global rate-limit to everyone at the same time.

The primary application we work on at Hindsight embeds itself into Jira, and so deals with both a large number of individual users, but also a large set of tenants, the tenant being the organisation that the individual user falls under. In our use-case, it makes more sense to have a tenant based rate-limit.



Typically this would be done through use of an in-memory database such as Redis to create a data entry tracking the requests each tenant makes, with each request incrementing a counter value. Once the counter reaches the limit, a flag will be tripped and will block subsequent requests with a rate-limit network message. A timestamp enforcing the end of the limit will be created, and once that time has been reached, the limit will be reset and we will start again.

### **8.2.2 More intelligent mechanism for retrying rate-limited requests**

Like with the rate-limiting of the micro-service, the automatic retry mechanism is functional, but fairly simple. The axios-retry library uses a set of user-defined status code to check whether the failed request should be retried. When a request fails, a HTTP status code is returned to the client, and the library compares this against the user defined list and if it matches retries it.

This works for most failed requests, but specific handling in the case of rate-limited requests would make it a bit more robust. By pulling out the expiration timestamp for the current rate-limit we could potentially retry a limited request at a point in the future where we know it will be accepted. However, the user doesn't want to be waiting a long time for the request to be retried, so logic for determining an acceptable time-frame would have to be added too, which turns an otherwise simple addition into a more difficult task.

## Appendix A

### Ethics Self-Check Form

# SAGE-HDR

---

Response ID	Completion date
640816-640807-77493571	6 May 2021, 10:52 (BST)

1	Applicant Name	Matthew Cooper
1.a	University of Surrey email address	mc00905@surrey.ac.uk
1.b	Level of research	Undergraduate
1.b.i	Please enter your University of Surrey supervisor's name. If you have more than one supervisor, enter the details of the individual who will check this submission.	Paul Krause
1.b.ii	Please enter your supervisor's University of Surrey email address. If you have more than one supervisor, enter the details of the supervisor who will check this submission.	p.krause@surrey.ac.uk
1.c	School or Department	Computer Science
1.d	Faculty	FEPS - Faculty of Engineering and Physical Sciences Sciences

2	<b>Project title</b>	A Framework for the Effective Development of Microservice Architectures
3	<b>Please enter a brief summary of your project and its methodology in 250 words. Please include information such as your research method/s, sample, where your research will be conducted and an overview of the aims and objectives of your research.</b>	Project aims to simplify the development of applications following the RESTful Micro-service architecture by creating a framework for creating, integrating and testing new services. Project is done with my employer - Hindsight Software Ltd - and aims to solve a number of issues that I have come across whilst migrating from a monolithic architecture to a set of micro-services. Not a whole lot research wise as most of the information based on the report/background has been pulled out of my time working but will mostly be focussed about ensuring that the output of the framework matches best practices so that it meets the required quality for production code.
4	<b>Are you making an amendment to a project with a current University of Surrey favourable ethical opinion in place?</b>	NO
5	<b>Does your research involve any animals, animal data or animal derived tissue, including cell lines?</b>	NO

<b>7</b>	<b>Does your project involve any of the following: human participants (including human data and/or any human tissue*); or is your project linked to engineering and/or the physical sciences?</b>	NO
----------	---	----

<b>8</b>	<b>Will you be accessing any organisations, facilities or areas that may require prior permission? This includes organisations such as schools (Headteacher authorisation), care homes (manager permission), military facilities etc. If you are unsure, please contact RIGO.</b>	NO
----------	---	----

<b>9</b>	<b>Will you be working with any collaborators or third parties to deliver any aspect of the research project?</b>	YES
<b>9.a</b>	<b>Is an agreement to work with collaborators already in place?</b>	YES

10	Is your project a service evaluation or an audit?	NO
----	---	----

11	Does your funder, collaborator or other stakeholder require a mandatory ethics review to take place at the University of Surrey?	NO
----	--	----

12	Are you undertaking security-sensitive research, as defined in the text above?	NO
----	--	----

30	Declarations	<ul style="list-style-type: none"> <li>• I confirm that I have read the University's Code on Good Research Practice and ethics policy and all relevant professional and regulatory guidelines applicable to my research and that I will conduct my research in accordance with these.</li> <li>• I confirm that I have provided accurate and complete information regarding my research project</li> <li>• I understand that a false declaration or providing misleading information will be considered potential research misconduct resulting in a formal investigation and subsequent disciplinary proceedings liable for reporting to external bodies</li> <li>• I understand that if my answers to this form have indicated that I must submit an ethics and governance application, that I will NOT commence my research</li> </ul>
----	--------------	---

4 / 5

		<p>until a Favourable Ethical Opinion is issued and governance checks are cleared. If I do so, this will be considered research misconduct and result in a formal investigation and subsequent disciplinary proceedings liable for reporting to external bodies.</p> <ul style="list-style-type: none"> <li>• I understand that if I have selected any options on the higher, medium or lower risk criteria then I MUST submit an ethics and governance application (EGA) for review before conducting any research. If I have NOT selected any of the higher, medium or lower risk criteria, I understand I can proceed with my research without review and acknowledge that my SAGE answers and research project will be subject to audit and inspection by the RIGO team at a later date to check compliance</li> </ul>
--	--	--

31	<b>If I am conducting research as a student:</b>	<ul style="list-style-type: none"> <li>• I confirm that I have discussed my responses to the questions on this form with my supervisor to ensure they are correct.</li> <li>• I confirm that if I am handling any information that can identify people, such as names, email addresses or audio/video recordings and images, I will adhere to the security requirements set out in the relevant Data protection Policy</li> </ul>
----	--	---

# Bibliography

- Amland, S. (2000), ‘Risk-based testing:: Risk analysis fundamentals and metrics for software testing including a financial application case study’, *Journal of Systems and Software* **53**(3).
- Fielding, R. (2000), Architectural Styles and the Design of Network-based Software Architectures, PhD thesis, University of California, Irvine.
- SOAP Version 1.2* (2003), Technical report, W3C.
- Winer, D. (1998), Xml-rpc specification, Technical report, Microsoft.