Final Year Project

Performance Analysis of Design Patterns in Microservice Architecture

Rajit Banerjee

Student ID: 18202817

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Professor John Murphy

UCD School of Computer Science

University College Dublin

29th April 2022

Contents

1 **Introduction** 4

2 **Related Work and Ideas** 5

2.1 Microservices 5

2.2 Software Design Patterns 6

2.3 Common Design Patterns in Microservice Architecture 6

2.4 Issues and Challenges with Microservices 9

2.5 Performance Engineering 9

2.6 Summary 11

3 **System Design Overview** 12

3.1 Prototype 12

3.2 Case Studies 12

3.3 Deployment 14

3.4 Evaluation 14

3.5 Code Repository 14

4 **Case Study 1** 15

4.1 Overview 15

4.2 Design Patterns Implementation 16

4.3 Evaluation and Results 23

5 **Case Study 2** 30

5.1 Overview 30

5.2 Design Patterns Implementation 31

5.3 Evaluation and Results 37

6 **Conclusions** 44

6.1 Summary 44

6.2 Future Work 45

7 **Appendix** 50

7.1 Supplementary Figures 50
7.2 Initial Project Specification 56
7.3 Project Work Plan 58

Abstract

Performance engineering is an integral part of the software design lifecycle for microservice-based systems, especially with the growing importance of microservices in modern-day system design. When developing new microservices or refactoring monolithic applications into several services, design patterns and anti-patterns play an important role in providing guidance regarding documented good and bad practices in a given context. Performance analysis of such patterns is essential as they can greatly impact user experience and overall system performance.

In this project, case studies are conducted to implement and evaluate a number of common design patterns in microservice-based web applications. Two web applications developed for the case studies demonstrate contrasting inter-service communication styles - synchronous HTTP request/response versus asynchronous message queues - combined with secondary patterns. Performance modelling, API functional testing, and systematic performance testing to measure response times are used in the evaluation process. The key takeaways from this project's outcomes include:

- Design patterns have a massive impact on the performance of microservices. Systems should be designed to handle failures as far as possible.
- Possible performance issues and bottlenecks can be identified and rectified with rigorous modelling and testing. Observability and monitoring patterns are crucial to the process of detecting and troubleshooting regressions.
- Automated load testing is essential in the performance evaluation process, especially for distributed systems. Performance test results should be averaged over multiple iterations, since measurements can be unreliable.
- Asynchronous styles are preferred over blocking synchronous calls, especially for communication amongst internal microservices.
- Systems which show resource consumption linearity with respect to load are suitable for scaling. Stable systems should show response times increasing proportionally with the incident load.

Chapter 1: **Introduction**

Microservices have emerged in the last decade as an approach for architecting and organising software as a fleet of autonomous and specialised services, that are loosely coupled and easier to develop and maintain compared to traditional monolithic architectures. Automated and independent deployment, focus on business capabilities, communication via well-defined interfaces using lightweight APIs, decentralised governance and data management are commonly observed characteristics of this evolving architectural style.

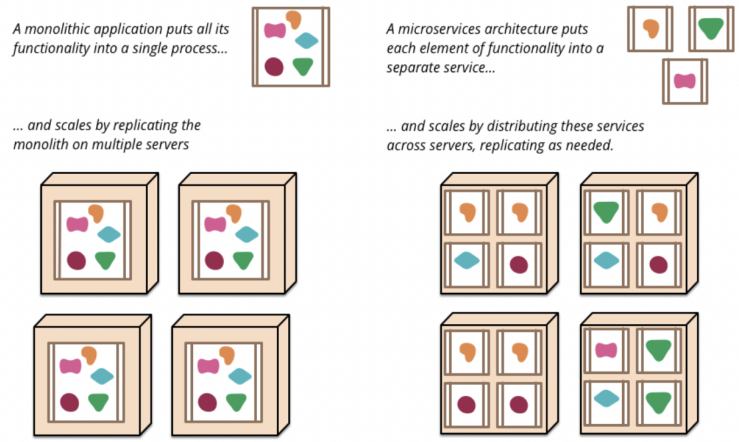


Figure 1.1: Monoliths versus microservices [1].

Design patterns in software engineering are well-documented templates describing common problems and recommended approaches in a given context. For nearly 30 years, design patterns and anti-patterns have guided developers to build software systems without having to re-invent the wheel every step of the way. As microservices haven't reached full maturity yet, architectural decisions guided by appropriate design pattern choices have a massive impact on the functionality and performance of applications. Performance engineering is an often ignored aspect of software design but is vital to its overall success, especially with the growth of business and changing demands. It encompasses performance analysis at design and deployment time, as well as performance monitoring and management at run-time. Although microservices have seen several stages of evolution over the years, innovation has been driven primarily by leading companies in the software industry, without significant contributions from academia.

The motivation for conducting this project is the lack of adequate studies on the effect of design pattern choices on microservice performance. It explores the performance impacts of a number of design patterns implemented by two microservice-based web applications. The systems have been developed to model a movie ticket reservation system, involving a client, three independent cinemas, and a client-facing intermediary service. The first case study explores patterns such as API gateway, circuit breaker and service discovery. The second study's application implements patterns including asynchronous messaging, monitoring of application metrics and health check API. In addition, both studies demonstrate indispensable patterns such as externalised configuration, separate database per service, and deployment of service instance per container. In the end, the performance of systems is evaluated using both quantitative and qualitative metrics, allowing us to draw some comparisons between the two case studies.

Chapter 2: Related Work and Ideas

The aim of this chapter is to provide readers with a holistic view of microservices, design patterns and performance evaluation, especially highlighting some of the important terminology and latest developments in the field. The discussion will consider the existing literature which illustrates how the topics have been previously explored, including the significance of performance engineering for microservices.

2.1 Microservices

Divide and conquer, the idea of breaking down complex systems into smaller manageable parts, is an ancient and proven paradigm which has been applied to computer science since the early 1960s. To tackle the complexities of software systems, concepts such as *modularity* and *information hiding* were introduced in 1972 by D. Parnas [2], as well as *separation of concerns* by E.W. Dijkstra in 1974 [3].

Regarding the term "microservices", the two most acknowledged definitions were given by Lewis and Fowler [1] and Newman [4], both around the same time in 2014. Newman considers microservices as a particular way of implementing SOA (Service-oriented Architecture) well, whereas Lewis and Fowler define it as a new architectural style contrasted against SOA:

"In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API."

The aforementioned definitions are first compared in 2016 by Zimmermann [5], who contrasted them with existing SOA principles and patterns. Since then, a number of systematic mapping studies have been published ([6], [7], [8]) to identify existing literature on microservices, in an effort to bridge the gap between academia (lack of many significant publications) and industry (racing past academia in terms of popularising and adopting microservices). In particular, Di Francesco et al. [8] cite both the prior studies conducted by Pahl and Jamshidi [6] and Alshuqayran et al. [7], and mention that due to the *bottom-up* approach (practical solutions first) that the industry has taken with microservices, many "fundamental principles and claimed benefits have still to be proven". By learning from practical applications, the industry has identified best practices, however aspects such as performance, functional suitability and maintainability of microservices at the industry-scale (instead of small-scale examples) are yet to proven in academia.

In the past decade or so, the adoption of microservices has been accelerated by the success of companies such as Amazon Web Services (AWS) [1], Netflix [2], Spotify [3] and Uber [4]. Amazon also has a whitepaper describing how microservices can be implemented using AWS cloud services, taking into consideration ways to reduce operational complexity and design distributed systems components (service discovery, data management, configuration management, asynchronous communication, and monitoring) [9].

[1] https://aws.amazon.com
[2] https://www.netflix.com
[3] https://www.spotify.com
[4] https://www.uber.com

2.2 Software Design Patterns

In software engineering and related fields, *design patterns* are generally defined as reusable solutions to commonly occurring problems in software design. Although design patterns cannot be directly converted to code (like an algorithm described in pseudo-code), they provide a blueprint on how a problem can be approached in various situations. Unlike *algorithms*, design patterns are not meant to define any clear set of instructions to reach a target, but instead provide a high level description of an approach. The characteristic features and final result are laid out, however the actual implementation of the pattern is left up to the requirements of the business problem and use case. Every "useful" design pattern should describe the following aspects: the intent and motivation, the proposed solution, the appropriate scenarios where the solution is applicable, known consequences and possible unknowns, as well as some examples and implementation suggestions.

Over half a century of software engineering experience has taught developers that it is indeed rare to come across a hurdle that hasn't been crossed before in some way, shape or form. Most obstacles and day-to-day decisions would have been tackled previously by another developer, thanks to which the idea of *best practices* has been formed over the years. Such solutions are accepted as superior, as they save time, are adequately efficient, and don't have many known side effects.

The most widely known literature on the topic is the 1994 textbook [10] by the Gamma, Helm, Johnson and Vlissides (Gang of Four), which is considered as the milestone work that initiated the concept of software design patterns. The authors, inspired by Christopher Alexander's definition of patterns in urban design [11], describe 23 classic patterns that fall under 3 main categories: *creational, structural*, and *behavioural* patterns.

- Creational patterns such as *Factory Method, Builder* and *Singleton* increase the flexibility and reusability of code by providing object creation mechanisms.
- Structural patterns such as *Adapter, Bridge* and *Facade* illustrate the process of building large, flexible and efficient code structures.
- Behavioural patterns such as *Chain of Responsibility, Iterator*, and *Observer* provide guidelines to distribute responsibilities between objects, and are specific to algorithms.

In recent times, design patterns have had a tendency of coming across as somewhat controversial, primarily due to a lack of understanding about their purpose. In this regard, it is important for developers to note that in the end, design patterns are merely guidelines and not hard-and-fast rules that must be conformed to.

2.3 Common Design Patterns in Microservice Architecture

Several attempts have been made to apply the concepts of software design patterns to microservices and categorise commonly seen patterns. In Fig. 2.1, C. Richardson provides a number of pattern groups, including *Decomposition, Data management, Transactional messaging, Testing, Deployment, Cross-cutting concerns, Communication style, External API, Service discovery, Reliability, Security, Observability* and *UI* [12].

Similarly, M. Udantha describes 5 different classes of design patterns applicable to microservices, namely *Decomposition, Integration, Database, Observability* and *Cross-cutting concerns* (see Fig. 2.2).

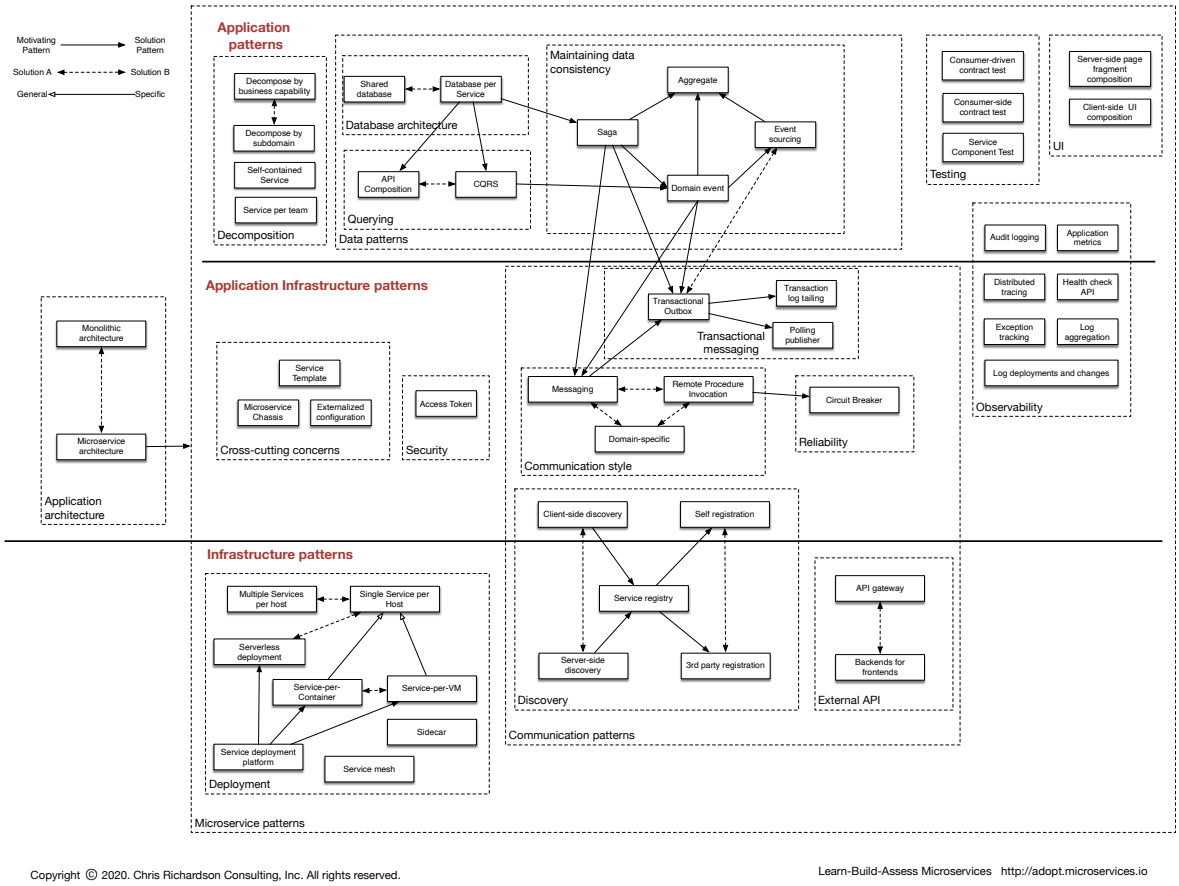


Figure 2.1: Groups of microservice design patterns [12].

It is important to note that despite minor differences in the categorisation of patterns, the groups suggested by the authors above are generally along the same lines, and aim to address the common principles of microservice design, such as scalability, availability, resiliency, flexibility, independence/autonomy, decentralised governance, failure isolation, auto-provisioning and CI/CD (continuous integration and delivery) [13].

- *Decomposition* patterns lie at the heart of microservice design, and illustrate how an application can be broken down by business capability, subdomain, transactions, developer teams, and so on. They also include refactoring patterns that guide the transition from monoliths to microservices.
- *Data management* patterns guide the design of database architecture (e.g. whether multiple services will share a database or each service will get a private database). They also lay out methods for maintaining data consistency, dealing with data updates and implementing queries.
- *Integration* patterns include API gateways, chain of responsibility, other communication mechanisms (e.g. asynchronous messaging, domain-specific protocols), as well as user interface (UI) patterns.
- *Cross-cutting concern* patterns describe ways of dealing with concerns that cannot be made completely independent, and result in a certain level of tangling (dependencies) and scattering (code duplication). Examples include externalising configuration, handling service discovery (client-side such as Netflix Eureka [5]; server-side like AWS ELB [6]), using circuit

[5] https://github.com/Netflix/eureka
[6] https://aws.amazon.com/elasticloadbalancing/

Figure 2.2: Udantha's 5 classes of microservice design patterns [13].

breakers, or practising Blue-Green deployment (keeping only one of two identical production environment live at any time). Service discovery and circuit breaker (for reliability) are also considered as communication patterns.

- *Deployment* patterns illustrate multiple ways of deploying microservices, including considerations about hosts, virtual machines, containerisation, number of service instances, as well as serverless options.
- *Observability* is a part of performance engineering, and such patterns are essential to any form of software design, since application behaviour must be continuously monitored and tested to ensure smooth working. Aggregating logs, keeping track of performance metrics, using distributed tracing, and maintaining a health check API are some invaluable practices which aid the troubleshooting process.

It is interesting to note that there are certain similarities between the *GoF*'s creational, structural and behavioural patterns [10] and the aforementioned microservice-specific patterns.

Apart from the sources mentioned above, there have been some studies conducted to recognise architectural patterns for microservice-based systems. In [14], Bogner et al. perform a qualitative analysis of SOA (Service-oriented Architecture) patterns in the context of microservices. Out of 118 SOA patterns (sources: [15], [16], [17]), the authors found that 63% were fully applicable, 25% were partially applicable and 12% were not at all applicable to microservices. Taibi et al. [18] tackle the issue of inadequate understanding regarding the adoption of microservice architectures. The authors explore a number of widely adopted design patterns, under the categories of *Orchestration and Coordination, Deployment* and *Data storage*, by elaborating the advantages, disadvantages and lessons learnt from multiple case studies. Thus, a catalogue of patterns is presented, all constituents of which demonstrate the common structural properties of microservices as discussed earlier.

2.4 Issues and Challenges with Microservices

Although microservice architectures have a number of benefits, it is important to note that they are not universally applicable to solve all problems at scale. Moreover, there are various trade-offs to consider, and *anti-patterns* ("bad practices") to avoid when building new microservice-based systems or transitioning from monolithic applications.

In [19], K. Cully investigates whether unforeseen performance issues can be introduced in a healthy system by independent communication and resiliency configuration of microservices. Using a custom-built rapid prototyping suite, Cully shows that multiple operational constraints in microservices can be conflicting, and optimising the performance within one part of microservice-based system can lead to major pitfalls in other parts.

M. Fowler argues in [20] that transitioning from a well-defined monolith to an ecosystem of microservices has various operational consequences and complexities, which demand certain competencies such as rapid resource provisioning (characteristic of cloud-native applications), observability and monitoring setup, CI/CD, as well as DevOps culture in the organisation. Fowler goes on to elaborate on some of the trade-offs that teams face when choosing microservices over monoliths. The costs associated with microservices include dealing with the distributed nature of the system: with issues such as slow remote calls, risk of failure, consistency, as well the increased operational effort required by teams [21].

Finally, it is important to avoid common anti-patterns that are known to be counterproductive when adopting the microservice architectural style. A few examples of such practices include [22], [23]:

- Distributed monolith - a monolith refactored into several smaller services, all of which are interdependent (tightly coupled).
- Dependency disorder - services must be deployed in a particular order to work.
- Shared database - resource contention due to all microservices sharing a single data store.
- API gateway - not using an API gateway, or using it incorrectly by building a gateway in every service.
- Entangled data - all services get complete access to all database objects (not following the principle of least privilege).
- Improper versioning - services such as APIs not designed for changes and upgrades, leading to reduced maintainability.

2.5 Performance Engineering

Having discussed the background research and ideas related to microservice-based systems and design patterns, it is now appropriate to narrow down the focus to performance engineering for distributed systems, especially microservices. Performance may be defined as the extent to which a system meets its timeliness objectives, and the efficiency with which this is achieved. Software performance engineering (SPE) is a vital part of the software design lifecycle, including, but not limited to performance simulation and modelling (e.g. using UML [7] diagrams), benchmarking, monitoring, and performance testing.

[7] https://www.uml.org

2.5.1 Distributed Systems

Innovation in performance engineering has often been driven by breakthroughs in industry, out of necessity. In fact, Netflix is well known for pioneering the concept of *chaos engineering* as early as 2011, when migrating to the cloud (AWS). To address the inadequacy of available resilience testing, the company invented a suite of tools (known as the "Simian Army" [24]), the most significant of which is Chaos Monkey [8]. Chaos engineering tests the performance of a large-scale system when subjected to experimental failure scenarios, especially in production environments.

Similarly, there exists the principle of *ownership* at Amazon, where it is the development team's responsibility to handle the entire software lifecycle including operations, performance testing, and monitoring ("you build it, you run it" [25]). Some contributions from academia include the introduction of a policy-based adaptive framework to automate the usage of diagnosis tools in the performance testing of clustered systems by Portillo-Dominguez et al. [26], [27]. The authors refer to the increased complexity of performance testing in distributed environments and provide a framework (PHOEBE) to increase time savings for testers.

2.5.2 Evaluation of Microservice-Based Systems

The 2015 study by Amaral et al. [28] stands out as the first work of its kind, as claimed by the authors: to analyse the performance of microservice architectures using containers (specifically Docker [9]). The paper provides a comparison of CPU performance and network running benchmarking for two models of designing microservices: master-slave and nested-container.

In 2017, Heinrich et al. explored the performance engineering challenges specific to microservices, namely testing, monitoring and modelling, with a focus on the need for efficient performance regression testing techniques, and the ability to monitor performance despite regular updates to software [29]. The paper identifies open issues and outlines possible research directions to tackle the lack of adequate performance engineering for microservices. In the same year, Gribaudo et al. took a simulation based approach to study the behaviour of microservice-based software architectures, and used a randomly generated (and realistic) overall workload to evaluate infrastructure setup, performance (defined indexes such as response time) and availability. The authors claim that their work is the first parametric simulation approach for performance modelling of microservices [30].

In more recent studies (2020), Eismann et al. discuss both the benefits and challenges introduced by microservice architectures from the perspective of performance testing [31]. Containerisation, granularity, access to metrics and DevOps integration are mentioned as characteristics that aid performance testing, whereas some pitfalls such as lack of environment stability, experiment reproducibility and detecting small changes for performance regression testing are also present. Gias et al. look at tailored performance engineering techniques for cloud-native microservices, which are increasingly becoming the industry norm due to the various benefits of cloud computing [32]. The authors use the RADON [10] project to address performance modelling challenges in microservices and FaaS (Function as a Service, such as AWS Lambda [11]), then go on to address aspects such as deployment optimisation, continuous testing and runtime management.

Finally, the primary inspiration for this project is the 2019 paper by A. Akbulut and H.G. Perros who take a closer look at 3 distinct microservice design patterns: API Gateway, Chain of Responsibility and Asynchronous Messaging, and compare performance analysis results (in terms of

[8] https://netflix.github.io/chaosmonkey/

[9] https://www.docker.com

[10] Rational Decomposition and Orchestration for Serverless Computing

[11] https://aws.amazon.com/lambda/

query response time, efficient hardware usage, hosting costs and packet loss rate) with different hardware configurations (e.g. virtual CPU and RAM) [33]. Some results from the first case study on the API Gateway pattern are shown in Fig. 2.3.

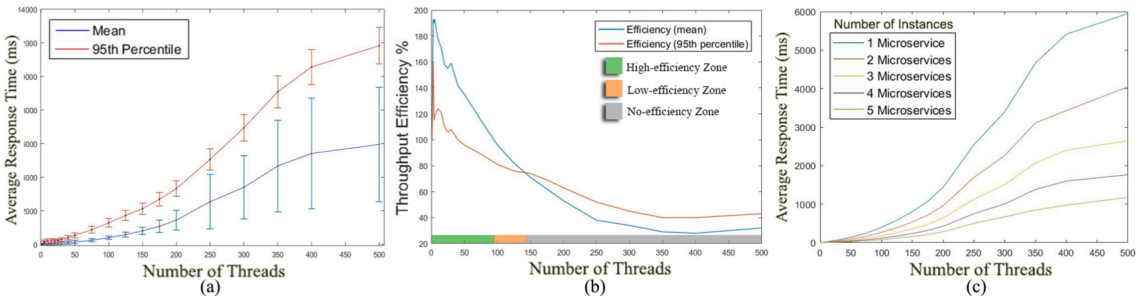


Fig. 1. Performance Results of Case Study I a)Mean and 95th Percentile of the Response Time vs Number of Threads, b)Efficiency Curves Using the Mean and the 95th Percentile of the Response Time, and c)Average Service Times for Different Number of Instances

Figure 2.3: Results from Akbulut and Perros' case study on the API Gateway design pattern [33], showing the variation in performance metrics with increasing number of threads used to generate asynchronous requests.

At the time of writing, Akbulut and Perros' work was found (to the best of our knowledge) to be the only paper to consider the performance analysis of design patterns for microservices, and this project aims to possibly replicate some of their results, then extend the study to other useful design patterns beyond the 3 that the authors have evaluated. This project will also take a similar approach by designing microservice-based applications using Docker containers, the performance engineering aspects of which will be studied.

2.6 Summary

The background research conducted reveals that in under a decade, microservices have gone from a nascent architectural style to becoming the de facto choice for building large-scale, flexible, loosely-coupled and maintainable software in the industry. Design patterns are templates for best practices that were initially applied to software design in general, but have been adapted for microservice architecture as well. They often guide the process of refactoring a system from a monolith to fine-grained services, considering aspects such as decomposition strategy, communication style, distributed data management, observability, and much more. Still, there are a number of hurdles to cross and anti-patterns to avoid when adopting microservices, and a proper needs assessment must be conducted before making the choice to disregard a well-designed monolith. Finally, performance engineering is a key contributor towards the success of any software, and is especially challenging and important for distributed systems such as microservice-based systems. The limitations of existing works include the lack of adequate performance analysis for various microservice design patterns, which has only been tackled by Akbulut and Perros [33] so far. This provides the motivation for proceeding with this project to extend the related works.

Chapter 3: System Design Overview

3.1 Prototype

In order to demonstrate various performance engineering practices in the context of microservice design patterns, we look at a simple movie ticket reservation system, consisting of 3 independent cinemas (Cineworld [1], Dundrum [2] and UCD [3]), as well as an intermediary (or broker) between clients and the cinema services. Each cinema has its own catalogue of movies (with an ID, name, and available showtimes). The client's primary objective would be to request the intermediary to fetch a list of movies from every active cinema and then display aggregated results. Next, the client selects a movie and showtime (along with other booking details) to make a reservation at a specific cinema. It is also possible to list the reservations made at a given cinema (useful for staff and administrators). The intermediary serves as a router for client requests to the cinema services, and in some cases, an aggregator of results.

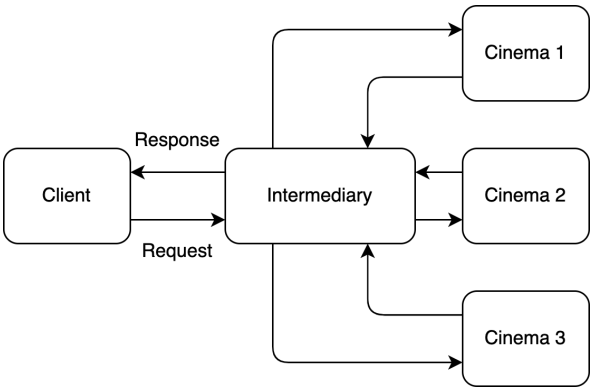


Figure 3.1: Prototype of movie ticket reservation system using microservices.

3.2 Case Studies

In this project, two separate web applications were designed to model the system described above, each employing a variety of common design patterns. The applications follow microservice architecture, with the intermediary and cinema services all being self-contained, loosely coupled and independently deployable. In a larger scale production system, it is likely that each independent cinema would in turn comprise a number of microservices to serve more specific purposes. Taking each application to be a *case study*, the next two chapters provide descriptions of the system design, implementation details, design pattern choices, and associated performance implications.

[1] https://www.cineworld.ie/
[2] https://www.movies-at.ie/movies/
[3] https://www.ucd.ie/studentcentre/cinema/

3.2.1 Communication Styles

When migrating from monolithic applications to microservices, decisions regarding communication mechanisms are vital, since a number of internal services must efficiently interact with each other as well as the clients, in order to perform well in a distributed environment. Managing data from responses, and aggregating results from various services also becomes a part of the equation. The two case studies in this project illustrate the two most common communication style patterns, namely request/response using APIs [4] (typically RESTful [5]) in case study 1, and asynchronous messaging in case study 2. Broadly speaking, the first style uses a synchronous protocol (HTTP [6]), where the client awaits a response from services after sending a request. Then, as the name suggests, the second style uses an asynchronous protocol (AMQP [7]) that uses message queues and doesn't wait for immediate responses. These have been separated for demonstration, but in practice, microservice-based applications tend to employ a combination of synchronous and asynchronous communication styles (including deferred synchronous and asynchronous callbacks) depending on the use case. Microservices often face the issue of "chattiness", due to excessive inter-service communication leading to performance degradation, which is a point to keep in mind when making architectural decisions. The aim is to enforce the autonomy of microservices during integration, and avoid anti-patterns such as a distributed monolith, where separate services end up being interdependent and tightly coupled.

3.2.2 Choice of Tools

One of the main benefits of microservices is the freedom of choice regarding programming languages used for development, since any suitable language may be used as long the service exposes an API adhering to known standards (typically HTTP and REST). The API is then used for inter-service communication. For the case studies in this project, Java and Spring [8] were chosen over other languages and frameworks primarily due to the dominance of Java in enterprise-level software, as well as the feature richness of Spring. Moreover, as shown in a 2017 study by Pereira et al. [34], Java ranks much higher in terms of time, memory and energy efficiency; which need to be considered when talking about system performance; compared to other popular languages such as Python or TypeScript/JavaScript. The Spring framework in Java offers a plethora of integrations and abstractions to implement microservice and cloud-related design patterns, through projects such as Spring Boot, Spring Data, Spring Cloud and Spring AMQP, to name a few that were used in the case study web applications in this project Apache Maven [9] was used for dependency management for Java.

Each web application exposes APIs for its microservices, whose endpoints can be invoked with HTTP requests like GET and POST. In commercial systems, a website (UI client) would invoke the backend API to perform actions, but for our case studies, it is sufficient to use a tool such as Postman [10] or VS Code's REST Client [11] to demonstrate the API functionalities locally.

[4] Application Programming Interface
[5] https://restfulapi.net/
[6] https://developer.mozilla.org/en-US/docs/Web/HTTP
[7] https://www.amqp.org/
[8] https://spring.io
[9] https://maven.apache.org
[10] https://www.postman.com
[11] https://marketplace.visualstudio.com/items?itemName=humao.rest-client

3.3 Deployment

In line with industry standards, Docker [12] is the infrastructure/containerisation tool used to deploy the fleet of microservices. According to Docker, Inc., a *container image* is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Using Docker provides a number of advantages such as ease of standardisation, compatibility, maintainability, CI/CD, resource and application isolation, and much more. Container deployment and application testing was performed on an Ubuntu 18.04 LTS compute server (*dunnion*, maintained by the UCD School of Computer Science [13]), with a 10 core Intel(R) Xeon(R) Silver 4114 CPU (base frequency: 2.20 GHz), and 125GB system memory.

3.4 Evaluation

In order to evaluate each case study, a three-pronged approach can be taken: performance modelling, manual API testing, and performance testing using Apache JMeter [14]. Since standardisation is essential for performance modelling, the Open Management Group's (OMG) *Unified Modelling Language (UML)* is used as the common design language for sequence diagrams. *Use cases* in UML and the *scenarios* that describe them are the key focus of the software performance engineering modelling process. Such scenarios show the interactions between objects and the flow of messages between them. After modelling, VS Code's REST Client is used to manually test the APIs, to demonstrate all functional endpoints and expected results. The tool also provides the time taken by the system to process a request. Finally, more rigorous performance testing of APIs is carried out with the help of JMeter, an industry-leading open source load testing tool. Performance testing is an umbrella term for various categories of tests such as load, stress, soak, spike or unit testing, each serving its own purpose. For evaluation here, the focus is on load testing - by simulating a number of virtual users using threads. The objectives of load testing include verifying system stability at light and heavy loads, exposing performance flaws, confirming linearity of resource usage with respect to system load, bottleneck identification and analysis, determining system load handling capacity, and so on. A comparison can then be drawn between the two case studies since they both implement the same system prototype.

3.5 Code Repository

The code for this project can be found on UCD School of Computer Science's internal GitLab server: https://csgitlab.ucd.ie/rajitbanerjee/microservice-design-patterns.

[12] https://www.docker.com
[13] https://www.ucd.ie/cs/
[14] https://jmeter.apache.org/

Chapter 4: Case Study 1

4.1 Overview

The pilot case study implements the movie ticket reservation system prototype described in the previous chapter, primarily using microservices communicating with REST APIs. Representational State Transfer implies that a client requesting a resource from any service cues the server to transfer back the current state of the resource in a standardised representation. The methodology and implementation details are elaborated on below.

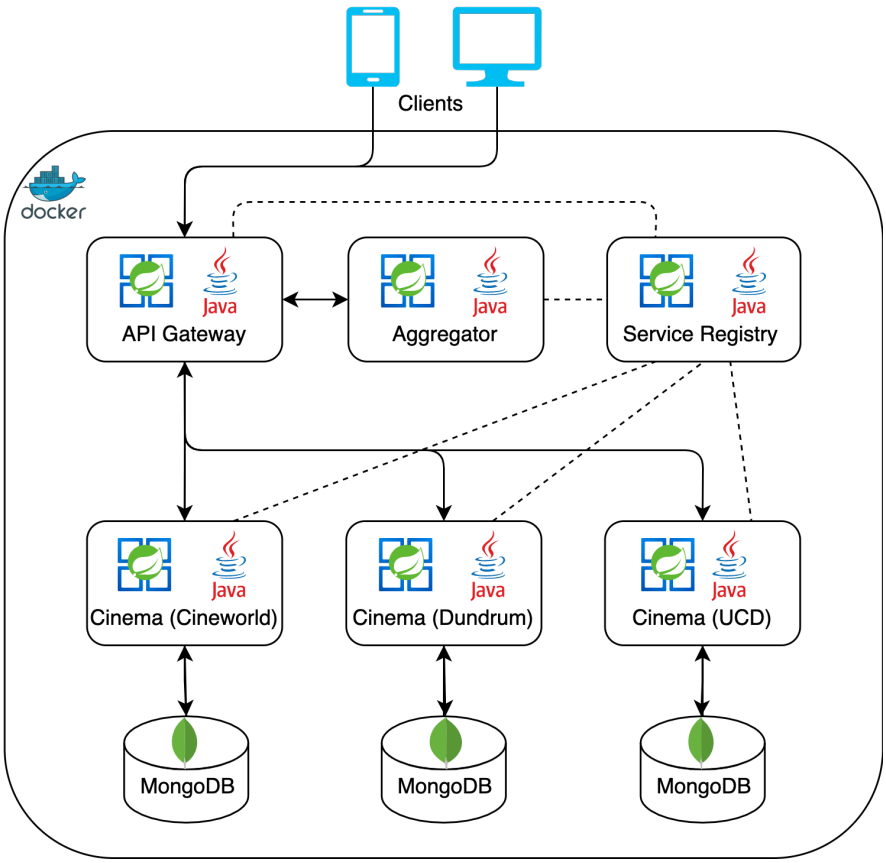


Figure 4.1: System design for the first case study.

The architecture diagram in Fig. 4.1 depicts the organisation of the microservices involved: an API gateway service to act as an interface between clients and other services, an aggregator service to gather results, a central service registry, and three independent cinema services - Cineworld, Dundrum and UCD. The following microservice design patterns were employed while developing this web application:

4.2 Design Patterns Implementation

4.2.1 API Gateway

An *API gateway* is one of the most fundamental and commonly used design patterns in microservice architecture. The `api-gateway-service` microservice offers an entry point for all clients so that multiple services (e.g. cinemas) can be exposed, and the gateway can route requests to other services as appropriate. In such situations, an API gateway relieves the clients from setting up and managing a separate endpoint for every service. It also aids the inter-service communication between microservices when required, for instance, the `aggregator-service` contacts the cinema services via the gateway again.

Without a gateway, any change (e.g. refactoring) in the service APIs would necessitate corresponding changes in the client code as well, which creates several development and management troubles. However, when a gateway is placed between clients and services, further consolidation or decomposition of a service would only require a simple routing logic change in the gateway, instead of updating the client.

An added benefit of this pattern would be the ease of deployment of new versions of microservices, which can be done in parallel with existing versions, since API gateway routing would provide control over endpoints presented to clients, and flexibility over feature release strategies. A minor variation of the API gateway pattern is called *backends for frontends (BFF)*, where a separate gateway handles request routing for different kinds of clients, such as web applications, mobile applications and third party applications (e.g. IoT devices). API endpoint management in appropriate gateway services is a more focussed approach to handling routing logic for different kinds of user interfaces, with their own set of pre-requisites.

Despite its popularity and usefulness, the API gateway pattern brings with it a few issues that should be taken into consideration. For instance, it could become a bottleneck or single point of failure, if resiliency and fault tolerance measures are not put in place. Performance testing methods such as load testing should be mandatory to prevent failure scenarios.

In this case study, the `api-gateway-service` uses the Spring Cloud [1] project to introduce a gateway [2] in the application. The gateway is configured using the module's `application.yml` resource file, which Spring Boot [3] can access to set up all the routing logic supported by the application. From a performance perspective, an API gateway provides several benefits, such as options to set up request authentication, rate-limiting to prevent service over-use and DDoS (distributed denial-of-service) attacks, monitoring and analytics endpoints (using Spring Boot Actuator), reducing latency by serving cached responses, as well as load balancing. A very simple gateway configuration example is shown below. The `lb://` prefix in `uri:` makes use of Spring Cloud Gateway's built-in load balancer, to provide round-robin client-side load balancing features during calls to other microservices. A 503 Service Unavailable Error is returned when a service cannot be found, which is more appropriate than a naive Java exception stack trace shown when using `http://` or `https://` prefixes instead.

```
spring:
  cloud:
    gateway:
      routes:
      - id: myRoute
        uri: lb://service
```

[1] https://spring.io/projects/spring-cloud
[2] https://spring.io/projects/spring-cloud-gateway
[3] https://spring.io/projects/spring-boot

```
          predicates:
          - Path=/service/**
```

Listing 4.1: Sample Spring Cloud Gateway configuration.

Spring Cloud Gateway paves the way for discussion on the next two design patterns: *circuit breaker* and *service discovery*, thanks to the Spring integrations available to simplify configuration and maintenance.

4.2.2 Circuit Breaker

To prevent cascading network or service failures resulting in system performance disasters, the *circuit breaker* pattern is best applied in the API gateway service to adopt a "fail-fast" approach. This was primarily achieved by adding a `CircuitBreaker` filter to the Spring Cloud API gateway routing logic, a snippet of which is shown below.

```
spring:
  application:
    name: api-gateway-service
  cloud:
    gateway:
      routes:
        ...
        - id: cinema-cineworld-service-route
          uri: lb://CINEMA-CINEWORLD-SERVICE
          predicates:
            - Path=/cinema/cineworld/**
          filters:
            - name: CircuitBreaker
              args:
                name: cinema-cineworld-service-cb
                fallbackUri: forward:/fallback/cinema-cineworld-service
```

Listing 4.2: Snippet from the API gateway service's application properties.

Spring Cloud supports several different circuit breaker implementations such as Netflix Hystrix, Resilience4J, Sentinel and Spring Retry, of which reactive Resilience4J [4] is preferred when used together with Spring Cloud Gateway as shown in the listing below. Reactivity is necessary in the circuit breaker since Spring Cloud Gateway is itself built using Project Reactor [5].

```
private Resilience4JConfigBuilder builder(String id) {
  return new Resilience4JConfigBuilder(id)
          .timeLimiterConfig(
                  TimeLimiterConfig.custom().timeoutDuration(Duration.
  ofSeconds(3)).build())
          .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults());
}
```

Listing 4.3: Circuit breaker configuration in API gateway application.

In the API gateway application, Resilience4J was configured with default settings, plus a timeout of *3 seconds* for demonstration purposes. This implies that any API endpoint accessed via the gateway that takes over 3 seconds to respond will cause the circuit breaker to trip, and forward

[4] https://github.com/resilience4j/resilience4j
[5] https://projectreactor.io/docs

the requester to the fallback URI - which needs to be specified for every route in the gateway configuration application properties. The timeout ensures that no transient faults in a microservice will degrade the system performance (primarily response time).

```
@GetMapping("/{name}")
public String message(@PathVariable("name") String serviceName) {
    return String.format("Fallback: Circuit broken in %s!", serviceName);
}
```

Listing 4.4: Code snippet from `FallbackController.java`.

The fallback controller is shown above, and this application simply logs the name of the service where the circuit was broken.

4.2.3 Service Discovery

Service discovery is an essential design pattern for modern microservices, where the network location (host name/IP address and port number) may be dynamic due to the industry dominance of virtualisation and containerisation solutions to run microservice-based applications. Hard-coding the locations of services is then considered an anti-pattern, since it necessitates changes in multiple places when services are scaled up or down. Hence, a natural solution is to set up a service registry, which maintains the locations of all active services and provides access to registered services through logical addresses (such as `lb://AGGREGATOR-SERVICE` or `lb://CINEMA-CINEWORLD-SERVICE` instead of actual network locations).

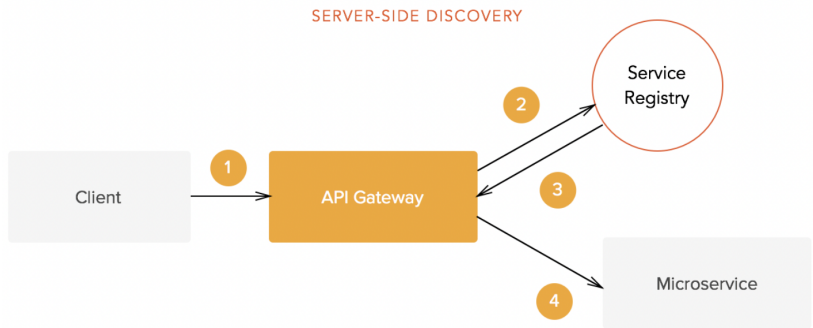


Figure 4.2: A simple service registry setup [35].

Fig. 4.2 depicts the gist of the service discovery pattern, and Fig. 4.1 shows how a service registry was implemented in this case study using the Spring Cloud Netflix (Eureka) project [6]. The server and client Maven dependencies are `spring-cloud-starter-netflix-eureka-server` and `spring-cloud-starter-netflix-eureka-client` respectively. The `eureka-server` service uses an `@EnableEurekaServer` annotation in its main application class to denote its server status (maintaining the service registry), and every other microservice uses a corresponding `@EnableEurekaClient` annotation (to self-register with the Eureka server on application startup). The server and sample client configurations are shown in the listings below:

```
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

Listing 4.5: Snippet from Eureka server's application properties.

[6] https://spring.io/projects/spring-cloud-netflix

The server is prevented from trying to register itself. For the sample client configuration, the Spring Boot application attempts to contact the Eureka server on `code.client.serviceUrl.defaultZone`, which is specified in `docker-compose.yml`. Eureka clients also send regular heartbeat messages to the server (set using `lease-renewal-interval-in-seconds`) to maintain their active status. Once the server stops receiving heartbeats below an expected threshold, it starts evicting the instances from the registry; this is known as self-preservation.

```
eureka:
  instance:
    lease-renewal-interval-in-seconds: 5
  client:
    register-with-eureka: true
    fetch-registry: true
    serviceUrl:
      defaultZone: http://${EUREKA_SERVER:eureka-server}:${EUREKA_PORT
  :8761}/eureka/
```

Listing 4.6: Snippet from a Eureka client's application properties.

Netflix Eureka also makes a dashboard available (Fig. 4.3) at the configured port (8761), which shows the active services, system status, and other useful information about memory usage, number of CPUs, server uptime, and so on.

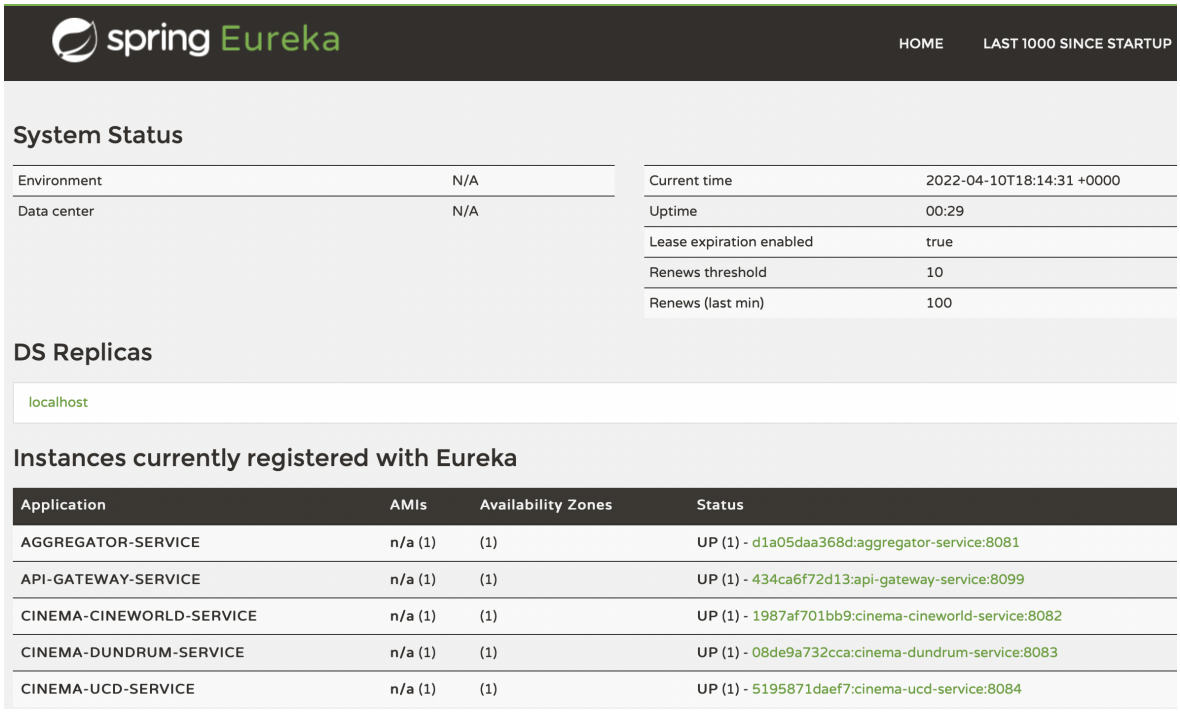


Figure 4.3: Eureka dashboard.

4.2.4 Aggregator

The service *aggregator* design pattern for microservices is useful when similar calls need to be made to multiple microservices. On receiving a request from the client via an API gateway, the aggregator service dispatches the request to appropriate internal backend microservices and then combines the responses into a single structure, which is returned to the requester. The aggregator itself doesn't need to know the routing details of internal services, since it can communicate with them through the API gateway again. The performance benefit of this pattern is the reduction

in communication overhead and chattiness between the client and the various internal cinema microservices.

Moreover, in this case study, the aggregator service relies on the service discovery pattern to find the registered cinema services at any given time. The code snippet from `aggregator-service` below illustrates how a custom discovery service function is used to fetch the addresses of all active cinemas, which the aggregator can use to send identical GET or POST requests to (received from the client via the gateway). The primary advantage of this design choice is the fact that the aggregator does not need to be made aware when the number of cinema services changes, as long as the necessary registration or eviction process is completed with the Eureka server.

```
for (String serviceUrl : discoveryService.getCinemaUrlPrefixes()) {
    String url = apiGatewayHost + serviceUrl + endpoint;
    if (httpMethod.matches("GET")) {
        results.add(restClient.getForObject(url, type));
    } else if (httpMethod.matches("POST")) {
        results.add(restClient.postForObject(url, body, type));
    }
}
```

Listing 4.7: Snippet from `AggregatorService.java`.

In the given movie ticket reservation system, the function of the aggregator is to gather the list of movie showtimes from every registered cinema and return a compiled list to the client. It should also be noted that an alternative to an aggregator microservice is to perform response aggregation in an API gateway itself, which results in the *gateway aggregation* pattern.

4.2.5 Externalised Configuration

Externalised configuration is a cross-cutting concern design pattern, that is especially useful in separating changeable application properties from the business logic. Spring Boot has built-in support for processing external configuration from various sources, such as Java properties files, YAML files, environment variables, and command-line arguments. The values set by the developer for such properties are injected into Spring beans [7] using the `@Value` annotation (among other alternatives).

Shown below are the environment variable settings for Cineworld's cinema service in the first case study's Docker Compose file.

```
cinema-cineworld-service:
  ...
  environment:
    - SERVER_PORT=8082
    - EUREKA_SERVER=eureka-server
    - DATABASE_HOST=mongodb
    - DATABASE_PORT=27017
  ...
```

Listing 4.8: Snippet from `docker-compose.yml`.

Using the above environment variables, Spring Boot then sets the application properties. Externalising configuration also allows the development team to maintain all properties in one place (say the Docker Compose file) such that any required changes will only need to be made in that file,

[7] https://www.baeldung.com/spring-bean

instead of searching for every occurrence of the property in the application source code. Alternatively, a dedicated microservice can be maintained for storing configurations required by multiple other services.

```
server:
  servlet:
    context-path: /cinema/cineworld
  port: ${SERVER_PORT}
spring:
  application:
    name: cinema-cineworld-service
  data:
    mongodb:
      database: cinema-cineworld
      host: ${DATABASE_HOST:mongodb}:${DATABASE_PORT:27017}
eureka:
  ...
  client:
    ...
    serviceUrl:
      defaultZone: http://${EUREKA_SERVER:eureka-server}:${EUREKA_PORT
  :8761}/eureka/
```

Listing 4.9: Snippet from `cinema-cineworld-service application.yml` file.

4.2.6 Database per Service

In monolithic applications, databases are shared across services, typically in a tiered approach (involving a web tier, services tier, cache tier and data tier). However, this leads to a large central database that becomes a single performance bottleneck for all data operations and is considered an anti-pattern when used with microservice architecture. For the microservices in our system to be truly independent, their persistent data stores must also be loosely coupled. Each service must have ownership over its domain data and logic under an autonomous lifecycle. This is achieved using the *database per service* design pattern.

Database independence in microservices can introduce additional concerns, as illustrated by the CAP theorem [8], according to which a distributed data store can only provide two of the following: consistency, availability and partition tolerance. In this project, MongoDB [9], a popular NoSQL database, is set up using Docker.

```
services:
  mongodb:
    image: mongo:latest
    container_name: mongodb
```

Listing 4.10: Docker Compose file snippet for MongoDB server.

Although only a single server was used in this project (serving at the default port - 27017), every microservice was configured to use a separate database. For instance, in the code listing below, `spring.data.mongodb.database` was set to `cinema-cineworld`, which creates and uses a separate database for Cineworld service, independent from other cinemas. When dealing with services with high throughput, multiple database servers may need to be provisioned, which can be achieved with

[8] https://www.ibm.com/cloud/learn/cap-theorem
[9] https://www.mongodb.com

Docker service scaling or using an orchestration tool like Kubernetes. The complications of a web application's integration with a persistent data store is made trivial with the help of the Spring Data MongoDB [10] project.

```
spring:
  ...
  data:
    mongodb:
      database: cinema-cineworld
      host: ${DATABASE_HOST:mongodb}:${DATABASE_PORT:27017}
```

Listing 4.11: Snippet from Cineworld cinema's application properties.

Although all services use MongoDB in this project for simplicity, the database per service design pattern allows every service to use a different kind of data store if the need arises, for instance, a service storing a lot of relational data could use MySQL, whereas another service storing network data might use a suitable graph database.

4.2.7 Service Instance per Container

As discussed in the project outline, Docker was used to containerise and deploy the microservices. The services were built as a multi-module Maven project, each having an independent Dockerfile for deployment. Docker Compose [11] simplifies the process of defining and running multi-container Docker applications: running `docker-compose up` starts and runs the services defined in the Compose file, provided that each service has a valid Dockerfile.

```
docker-compose down --remove-orphans
docker-compose build --no-cache
docker-compose up
```

Listing 4.12: Sample Docker Compose commands to start the microservices.

The wide range of benefits of containerising microservices makes Docker the industry-leading tool for service deployment. Developers can choose any suitable programming language and framework (including different versions) since Docker provides isolated environments with all required dependencies. The *service instance per container* design pattern allows each microservice to be packaged as a Docker container image, permitting the deployment of multiple instances of the microservice as separate containers if required.

There are far-reaching performance benefits of this pattern, including service reproducibility, independent deployment and scaling, hardware resource (CPU/memory) constraining capabilities, service instance monitoring, reliability and fault tolerance. Most importantly, Docker containers are much faster than traditional virtual machines (VMs) thanks to lightweight architecture (sharing the host OS kernel instead of requiring a complete OS or hypervisor). In large scale projects, container orchestration tools such as Kubernetes [12], Marathon (Mesos) [13], Amazon ECS [14] and EKS [15] are widely used to automate the operational effort required to manage containerised microservices, which is key for well-run DevOps.

The three cinema microservices have identical Dockerfiles, as shown below. Each of them uses

[10] https://spring.io/projects/spring-data-mongodb
[11] https://docs.docker.com/compose/
[12] https://kubernetes.io
[13] https://mesosphere.github.io/marathon/
[14] https://aws.amazon.com/ecs/
[15] https://aws.amazon.com/eks/

a JDK 11 base image, copies the compiled JAR file into the container and then runs it after a 5-second wait. Using Docker Compose, each service's Dockerfile is used to start the application.

```
FROM openjdk:11-jre-slim
COPY target/cinema*.jar /cinema.jar
CMD sleep 5 && java -jar /cinema.jar
```

Listing 4.13: Dockerfile for cinema services.

4.3 Evaluation and Results

To evaluate the first case study, the aforementioned plan involving performance modelling, manual API testing and JMeter load testing is followed.

4.3.1 Performance Modelling

Fig. 4.4 shows a sequence diagram for a scenario where the client wishes to fetch the list of movie showtimes from all active cinema microservices and view the aggregated results. The diagram components follow the specifications from OMG (2011) [36], showing a series of *objects* (e.g. Client, Aggregator, Cinemas) with vertical *lifelines* (downward flow of time). Opaque rectangles on top of lifelines are *activation boxes* (i.e. method-invocation boxes) which indicate an object handling/processing a message.

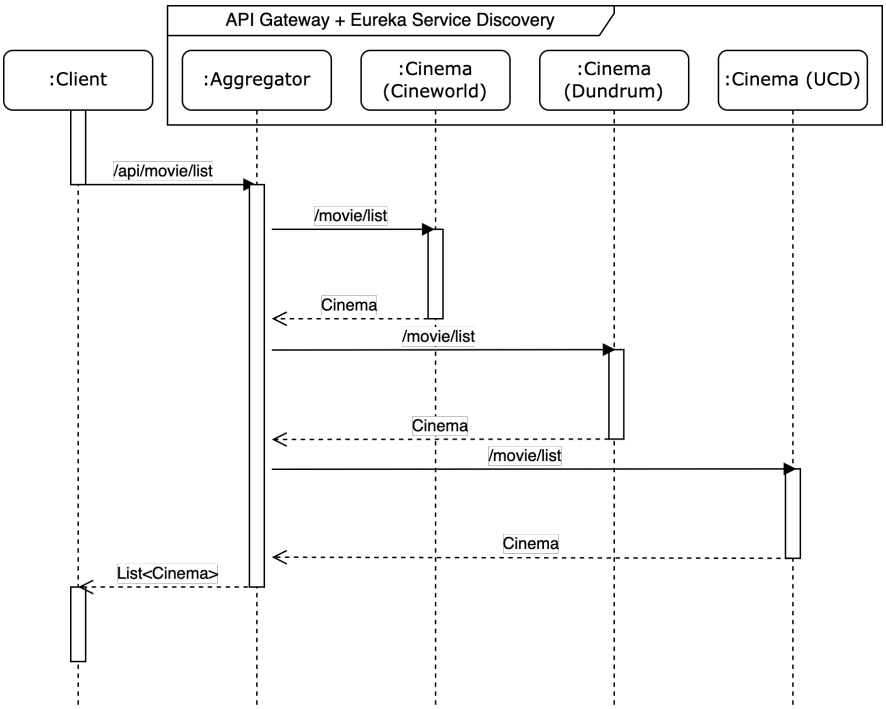


Figure 4.4: UML sequence diagram for listing movies from all cinemas (case study 1).

Sequence of steps:

- Client calls the API endpoint `/api/movie/list` exposed by the Aggregator service, via the

API gateway and Eureka service discovery. All synchronous calls are represented by closed arrowheads with solid lines.

- When received (dashed lines) by the Aggregator, it requests each active Cinema service (again via the gateway, discovering the ports using Eureka), for their respective list of movie showtimes.
- Each service responds with a `Cinema` object, containing a unique identifier, cinema name, and list of movies (including showtimes).
- The Aggregator then combines all `Cinema` objects into a list, to return to the Client.
- Since this entire flow is synchronous, the connection between the Client and Aggregator remains open while the latter contacts all cinemas.

From the sequence diagram, one can infer that the API gateway (and possibly the aggregator) service can turn into a potential performance bottleneck if appropriate preventive measures are not taken under heavy load, especially since all the inter-service communication is synchronous and blocking. Similar modelling can also be done for other use cases, such as making a reservation or listing reservations at a given cinema.

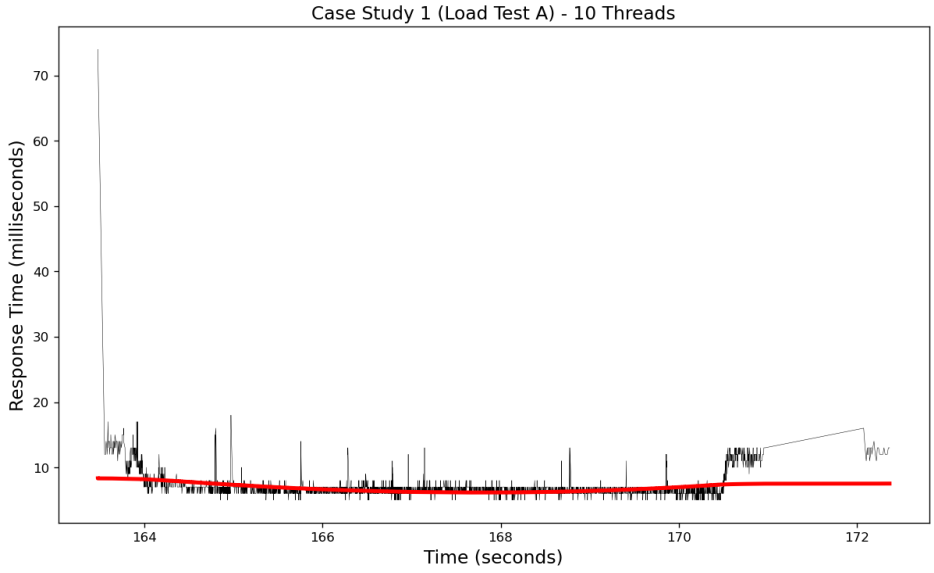
4.3.2 Manual API Testing

Using VS Code, the time taken by the application to process a given HTTP request (stored under `src/main/resources/http/` in `api-gateway-service`) was measured. All calls were made to the client-facing API gateway on port 8099, which internal microservices also use to communicate with each other. The following endpoints were tested to demonstrate the functionality of the web application:

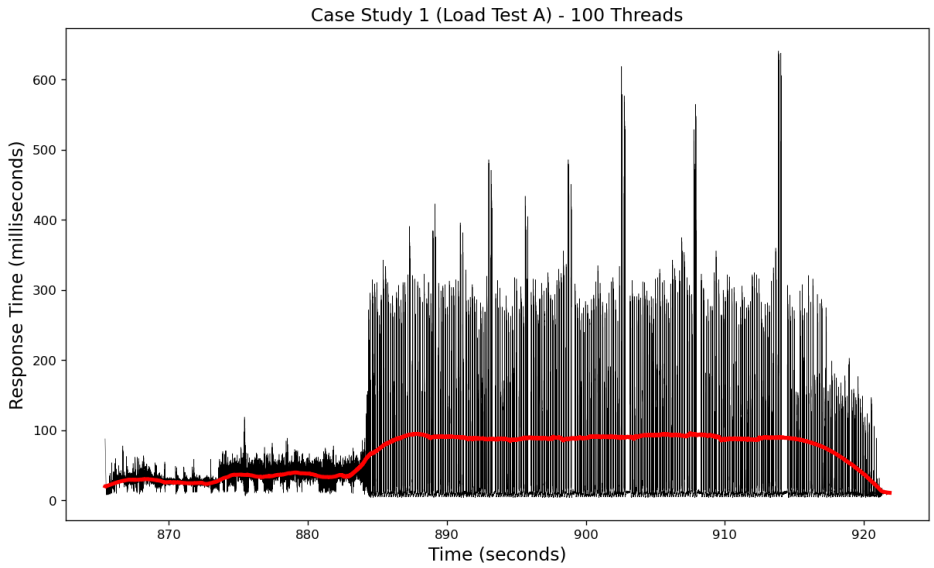
- `/api/movie/list` (Fig. 7.1): As specified in the API gateway routing configurations, all requests to `/api` are forwarded to the Aggregator service using a logical address. Actual network locations are maintained only by the Eureka service registry, where all services must register on startup. A GET request to the `/api/movie/list` endpoint returns the list of movie showtimes (HTTP status 200 OK) from all active cinema services in 153 milliseconds (ms).
- `/cinema/cineworld/reservation/make` (Fig. 7.2): A sample ticket reservation can be made at Cineworld by directing the request only to the appropriate service, using a POST request to `/cinema/cineworld/reservation/make`. The body of the request includes booking details such as the client name and email, movie ID, date, showtime, number of tickets, ticket type/category and total amount paid. In a production system, these details would be provided by the frontend, after the client enters information on a website. An HTTP status 201 Created response implies that the reservation was successfully stored in the cinema database, with the server returning the new reservation in 152 ms.
- `/cinema/cineworld/reservation/list` (Fig. 7.3): To ensure that the previous reservation at Cineworld was successful, one can list all the reservations at the cinema. In a production system, such an endpoint would be privileged and restricted to admin users, so that any external client doesn't get access to other clients' booking details. A 200 OK response in 45 ms returns a list of reservations, containing a single entry for Jane Doe's booking made earlier.
- `/cinema/cineworld/reservation/delay/4` (Fig. 7.4): To illustrate a working circuit breaker implementation, a simple `/delay/4` endpoint for Cineworld can be invoked, which simply sleeps for 4 seconds. To prevent unresponsive services, the circuit breaker in the API gateway was configured to time out after 3 seconds, and display a fallback message (HTTP code 500 Internal Server Error) containing the service name where the timeout occurred.

4.3.3 Performance Testing with JMeter

Finally, using CLI mode on Apache JMeter, the microservices were load tested. Two test plans (A and B) were designed: to fetch the list of movie showtimes from all cinemas and then to make a ticket reservation at Cineworld. For both tests, the number of threads (users) was varied from 10 to 100 in intervals of 10, with a ramp-up period of 1 second, i.e. the time taken by JMeter to get all threads up and running. Importantly, all load testing was performed over 1000 iterations to improve the reliability of measurements.



(a)



(b)

Figure 4.5: Load test A response times using (a) 10 threads and (b) 100 threads.

Load test A is focussed on the `/api/movie/list` endpoint, with the call to the aggregator service (via API gateway) returning the full list of movie showtimes from Cineworld, Dundrum and UCD cinemas. Fig. 4.5 shows the response times measured at the two load extremities - 10 threads and 100 threads. Since response time measurements are known to be noisy, a Savitzky-Golay filter was used for data smoothing (plotted in red). The filter uses convolution, by applying the linear least squares method to fit successive subsets (chosen size = 10,000) of adjacent data points with

a low degree polynomial (chosen = 2). The results show that at a low workload of 10 threads, the response time is more or less constant at under 10 milliseconds (ms). There is a noticeable initial spike (70 ms +), likely due to the time taken to establish the database connection and populate the cache (cold start). The load was increased in intervals of 10 threads, up to the heavy load scenario of 100 threads per second. In that situation, the recorded response times jumped from 30-50 ms to about 100 ms (noticeable in the plot between timestamp 880 and 890 seconds). The variation in response time appeared to increase significantly when using 100 threads, suggesting a need for system scaling to handle heavier workloads without performance degradation.

Threads	responseCode	
10	200	100%
20	200	100%
30	200	76%
	500	24%
40	200	82%
	500	18%
50	200	84%
	500	16%
60	200	92%
	500	8%
70	200	93%
	500	7%
80	200	89%
	500	11%
90	200	85%
	500	15%
100	200	79%
	500	21%

Figure 4.6: Proportion of 200 and 500 HTTP response codes during load test A at different load levels.

Fig. 4.6 shows the response codes observed during load test A, 200 implying success and 500 signifying a server error. The circuit breaker timeout pattern employed in this case study prevents unresponsive server behaviour at heavy workloads, with fast failures reflected in the increase in the proportion of errors at heavy load points such as 90 (15%) and 100 (21%) threads respectively. Yet, this is preferable to the server taking 3 plus seconds to respond to client queries. However, steps must be taken to minimise the error probability - with the recommended approach being an increase in service capacity (horizontal or vertical scaling as appropriate) to handle a heavier load.

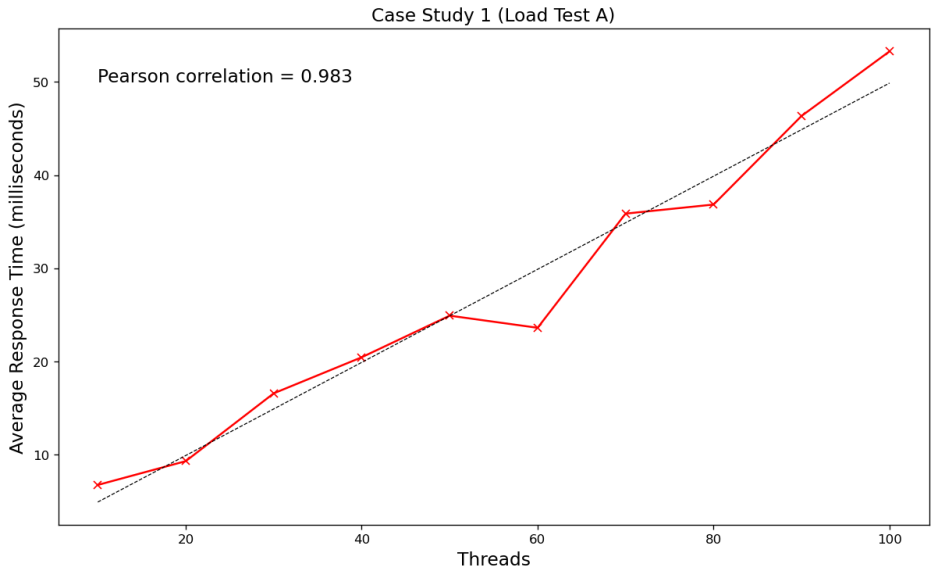


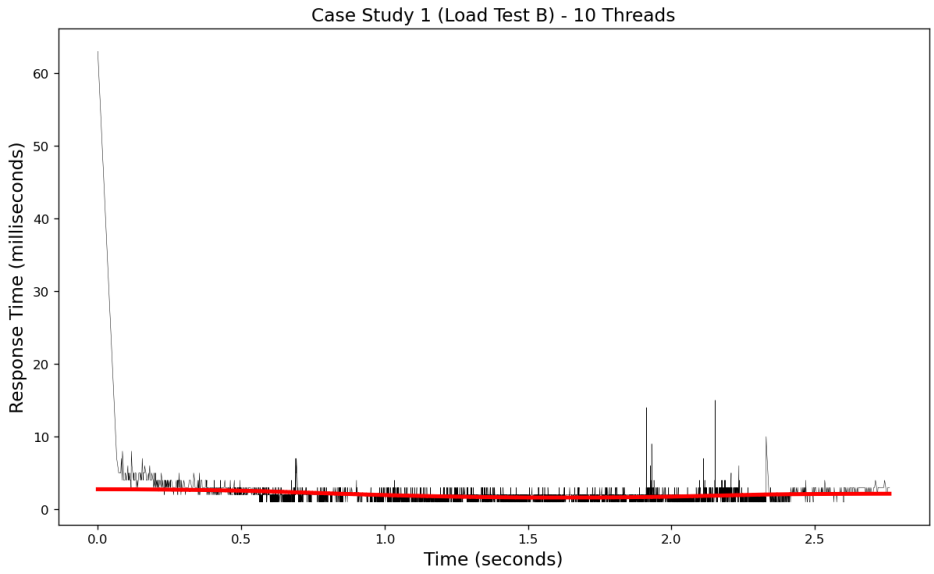
Figure 4.7: Average response time vs number of threads in load test A.

One of the primary aims of load testing is to demonstrate the system resource usage being proportional to the applied load, which is a precondition for scalability. This is well illustrated by Fig. 4.7 (also showing a dashed trend-line), with a strong positive Pearson correlation of **0.983** between average response time and the number of threads (load).

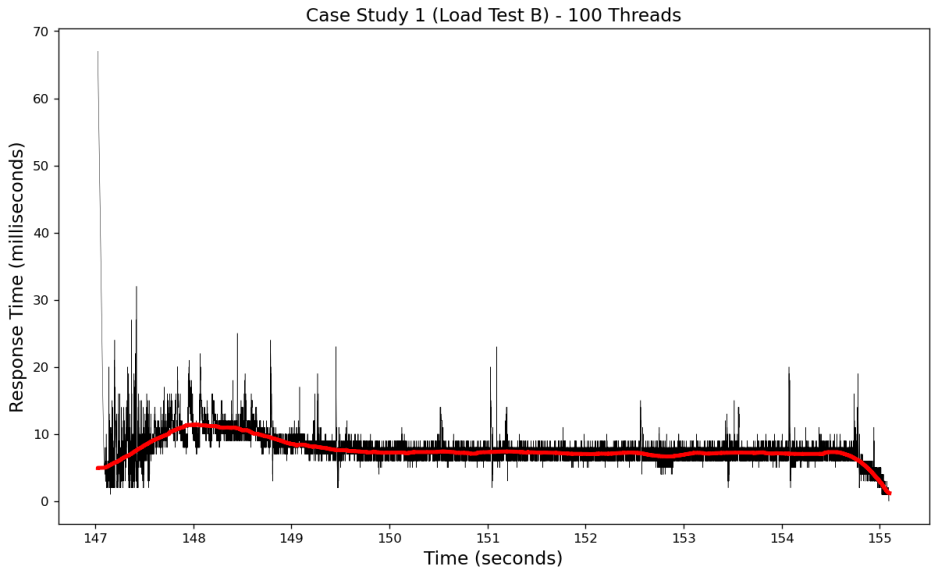
For load test B, a similar strategy as above was applied to check the ticket reservation functionality of the application. The API endpoint `/cinema/cineworld/reservation/make` directly contacts the Cineworld service via the API gateway. Due to the reduced inter-service communication, the response times measured at both load extremities - 10 threads and 100 threads were more or less constant at about 3 ms and 10 ms respectively. As expected, making a reservation at a single cinema was considerably quicker than aggregating movie showtimes from all services. The listing below shows the ticket booking details used as the POST request body - identical to the payload used for manual API testing.

```
{
  "clientName": "Jane Doe",
  "clientEmail": "jane.doe@ucd.ie",
  "movieId": "CNW001",
  "date": "2022-04-15",
  "showTime": "18:00",
  "tickets": 2,
  "ticketType": "Gold",
  "amount": 30.00
}
```

Listing 4.14: Dummy payload for load test B POST request.



(a)



(b)

Figure 4.8: Load test B response times using (a) 10 threads and (b) 100 threads.

Fig. 4.9 showing a 100% success rate (all 201 Created HTTP response codes) at all load levels reaffirms the fact that relatively low response times for reservation requests allows the system to handle a load even greater than 100 threads per second before collapsing and requiring a scale-up.

Threads	responseCode	
10	201	100%
20	201	100%
30	201	100%
40	201	100%
50	201	100%
60	201	100%
70	201	100%
80	201	100%
90	201	100%
100	201	100%

Figure 4.9: Proportion of HTTP response codes during load test B at different load levels.

As seen before in load test A, the linearity of resource consumption with respect to system load in test B is confirmed by a Pearson correlation score of **0.985**. The straight-line plot in Fig. 4.10 shows a proportional increase in mean response time with the load increasing gradually from 10 to 100 threads.

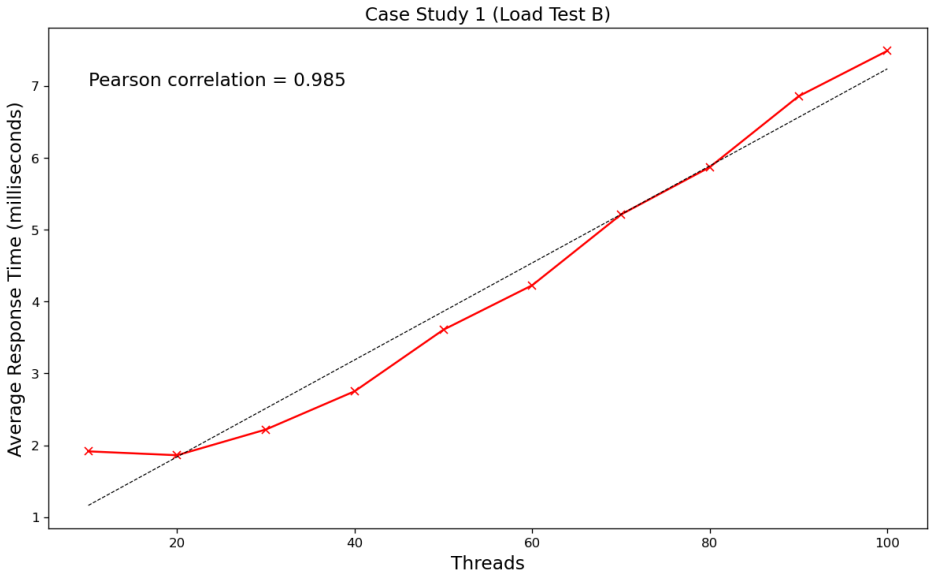


Figure 4.10: Average response time vs number of threads in load test B.

Chapter 5: Case Study 2

5.1 Overview

The second case study implements the same system prototype for a movie ticket reservation system as seen in the pilot study. However, a different set of microservice design patterns is explored this time, including a significant change in inter-service communication style from synchronous REST request/response to asynchronous messaging.

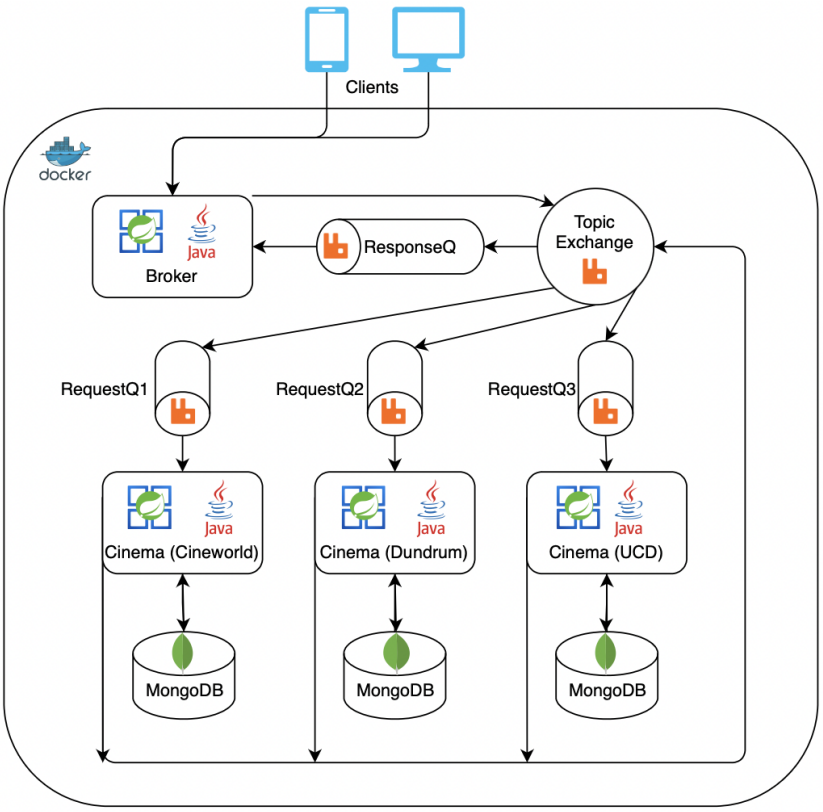


Figure 5.1: System design for the second case study.

The system architecture shown in Fig. 5.1 is similar to the previous study's architecture in the sense that the three cinemas are still independent microservices with separate MongoDB database connections. However, the intermediary that intercepts client requests has been changed from an API gateway plus aggregator setup to a message broker that uses message queues to communicate with other services. For external clients, the broker still exposes a modified REST API, but with separate endpoints to send a request and fetch a response. Requests are hence asynchronous from the client's perspective too since a single endpoint is no longer expected to deliver results while maintaining an open connection. The design patterns implemented by this web application are described in detail below:

5.2 Design Patterns Implementation

5.2.1 Asynchronous Messaging

For inter-service communication, the *asynchronous messaging* is the recommended design pattern over synchronous request/response. Although external clients should still avail of a synchronous REST API to contact the intermediary (broker), the internal cinema services should communicate with the broker asynchronously to prevent blocking. There exist multiple realisations of this pattern, including the use of notifications, HTTP polling, publish/subscribe, or messaging queues request/response.

In this case study, RabbitMQ [1], a widely popular open-source asynchronous messaging technology, was used as the tool of choice. RabbitMQ implements AMQP (Advanced Message Queueing Protocol), a wire-level message-oriented middleware protocol with defining features such as message orientation, queueing, routing (point-to-point and publish/subscribe), reliability and security. The Spring AMQP [2] project was used to greatly reduce the amount of boilerplate code required for infrastructure setup, by providing high-level abstractions: a Listener container to asynchronously process inbound messages, RabbitTemplate to send/receive messages, RabbitAdmin to auto-declare queues, exchanges and bindings.

The main components of the messaging infrastructure are:

- *Request queues* - Every cinema microservice is configured to receive requests from the broker via a corresponding request queue.
- *Response queue* - All cinema services send responses to a common response queue, that the broker consumes messages from before returning responses to the client.
- *Topic exchange* - An exchange in RabbitMQ is a routing agent, for directing messages to/from different queues using header attributes, bindings and routing keys. A single topic exchange (given name: `cinema`) is used in this application, that is appropriate for messaging based on pre-defined "patterns" of routing keys as described below.
- *Bindings and routing keys* - Bindings are used to link the four queues (3 for requests, 1 for response) to the cinema topic exchange, based on different routing keys. For instance, a message with routing key `request.to.cinema` is broadcast by the exchange to each request queue. Each cinema service is then able to process an identical request (e.g. listing movie showtimes). A response message from each cinema with key `response.from.cinema` is sent to the common response queue. The broker can then combine the message responses from each cinema and present a full list to the client. For special cases, such as client requests related to a single cinema service (e.g. making a reservation, listing reservations), a separate routing key with the pattern `request.to.cinema.cinemaName` tells the topic exchange to send the request to only a single queue corresponding to the appropriate `cinemaName`.
- *Message converter* - Message converters are used to handle the interconversion/serialisation of Java objects (initial stage - from cinema services - JPA [3] repositories), Spring AMQP messages (during transmission via queues) and finally the JSON response (shown to the client).

Fig. 5.2 shows the 4 queues set up for requests and response messages, using the RabbitMQ Management plugin interface.

[1] https://www.rabbitmq.com
[2] https://spring.io/projects/spring-amqp#overview
[3] Java Persistence API

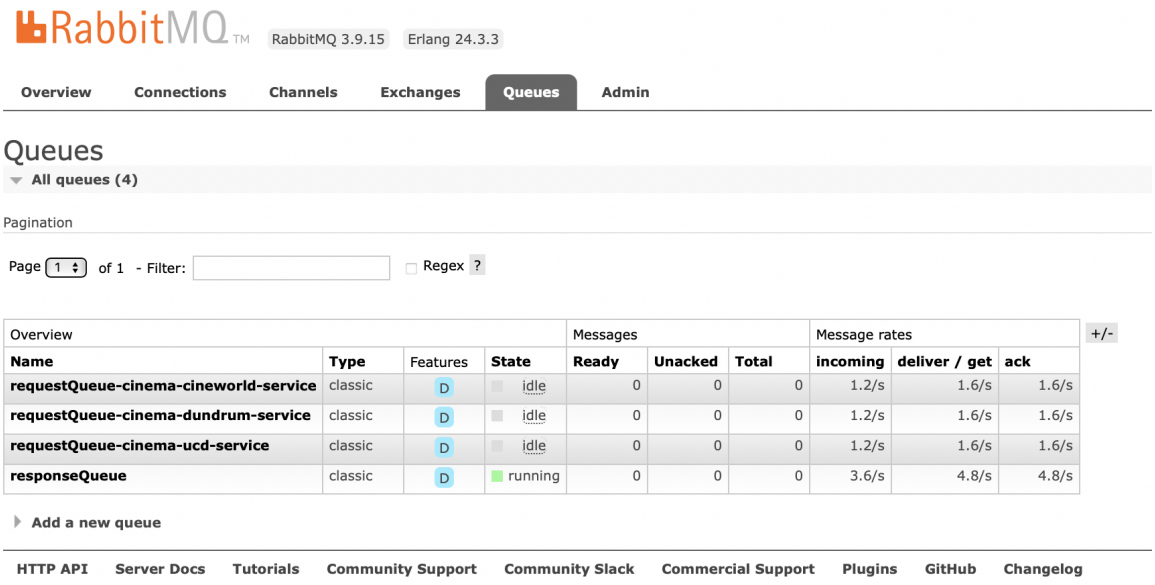


Figure 5.2: The Queues section from the RabbitMQ Management UI (http://localhost:15672). Default login credentials: `guest:guest`.

In each microservice, a `config.MessagingConfig` `@Configuration` class defines the above infrastructure as Spring beans using Spring AMQP abstractions. As previously mentioned, the broker service exposes a synchronous REST API for the client. This is a way to implement the asynchronous request/response pattern as seen in Fig. 5.3). The first API call enqueues a request to be processed and returns a temporary result like a new endpoint or the request ID that was forwarded. Then a second API call actually returns the data requested, after ensuring that the result is mapped to the correct request (usually using an ID).

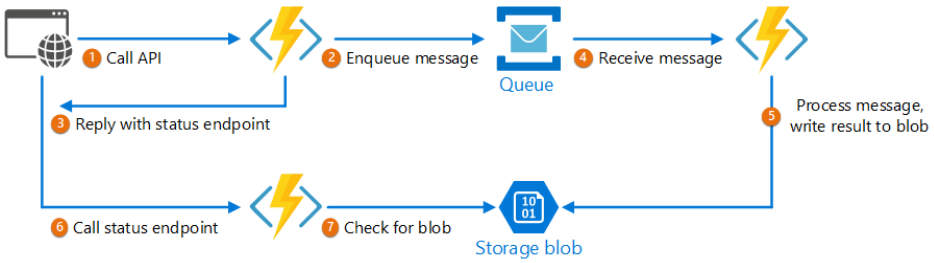


Figure 5.3: Sample asynchronous request/response pattern.
Source: Microsoft, https://github.com/mspnp/cloud-design-patterns

For example, one function of the system is to list the movie the showtimes from all cinemas. Instead of immediately returning the movies, the `/api/movie/list` endpoint simply broadcasts the request to all cinemas and returns a broker-generated *correlation ID*. This ID is used to map the asynchronous request to the responses. The broker uses a `@RabbitListener` to monitor the response queue, and any received messages are cached - ready to return to the client when the proper correlation ID is provided via the corresponding API endpoint: `/api/movie/list/{correlationId}`. Unlike the two-step synchronous broker-client communication, the broker-cinema communication channels are completely asynchronous. A `RabbitTemplate` and `MessagePostProcessor` are used to format messages from the broker, and published to the appropriate queues for the cinemas. All plain Java objects are serialised by the message converter. The individual cinema services also use a `@RabbitListener` to monitor the request queue for broker requests, then perform database read/write operations, and finally route the response to the common response queue.

The code listing below shows a snippet from the broker service, with an HTTP GET mapping to send a request (e.g. listing movie showtimes from cinemas) and return a correlation ID `String`, then a separate GET mapping using the ID to fetch the response list asynchronously.

```
@GetMapping("/list")
public String listMovies() {
    String endpoint = "/movie/list";
    String routingKey = requestRoutingKey;
    String correlationId =
            brokerService.sendRequest(new RequestMessage(endpoint, null),
  routingKey);
    return correlationId;
}

@GetMapping("/list/{correlationId}")
public List<Cinema> listMoviesResponse(@PathVariable String correlationId)
  {
    List<Message> responses = brokerService.fetchResponseFromCache(
  correlationId);
    return responses.stream()
            .map(r -> (Cinema) SerializationUtils.deserialize(r.getBody()))
            .collect(Collectors.toList());
}
```

Listing 5.1: Code snippet from `MovieController.java` in `broker-service`.

As mentioned earlier, any asynchronous responses from cinemas are first cached in the broker using the correlation ID, then the API allows reading from the cache with the correct ID via a separate endpoint. In terms of system performance compared to case study 1, there is no longer a need for the client to be blocked after sending a request (in anticipation of an immediate response). Requests can be sent to the broker, and then fetched later when needed using the correlation ID. As a result, the client can continue with other tasks after sending a series of requests to be processed, and then collect the responses when they need to be displayed by some frontend service.

5.2.2 Application Metrics

For successful performance engineering in a distributed system, observability and monitoring patterns are indispensable. Monitoring is often defined as capturing the behaviour of a system by performing health checks and collecting metrics over time, to aid the process of anomaly detection and root cause analysis. While maintaining text logs is also important, aggregated metrics over time provide insights into the historical and current state of a system. To understand the behaviour of an application and be able to troubleshoot performance issues or service degradations, adequate monitoring infrastructure must be in place, along with visualisation tools.

In applications using RabbitMQ, the most popular monitoring stack includes Prometheus [4] (monitoring toolkit) and Grafana [5] (metrics visualisation). Although a default management plugin [6] exists (exposing RabbitMQ metrics for nodes, connections, message rates, and so on), long term metric storage and visualisation services like Prometheus and Grafana are known to be more suitable for production systems.

[4] https://prometheus.io
[5] https://grafana.com/grafana/
[6] https://www.rabbitmq.com/management.html

The `rabbitmq_prometheus` plugin exposes all RabbitMQ metrics on port 15692, in Prometheus text format. A dedicated Dockerised Prometheus server is also set up (configuration file: `prometheus.yml`) to monitor *itself* (port 9090), the RabbitMQ Prometheus metrics (port 15692) as well as a Dockerised `node-exporter` service (Prometheus exporter for host machine hardware and OS metrics) (port 9100). A screenshot of the Prometheus target dashboard is shown in Fig. 5.4, with the 3 targets defined in `prometheus.yml` for scraping metrics.

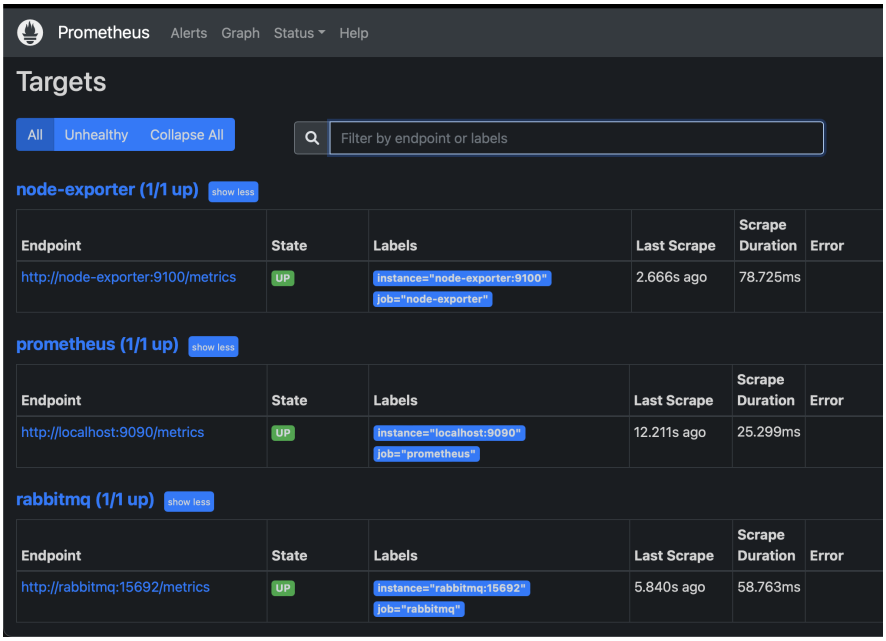


Figure 5.4: Prometheus targets dashboard on `http://localhost:9090`.

Due to the limited features of the RabbitMQ management plugin and the Prometheus dashboard alone, Grafana (port 3000, default credentials `admin:admin`) is the visualisation tool of choice, which can use the Prometheus server as a data source. The RabbitMQ team provides a number of predesigned Grafana dashboards for RabbitMQ and runtime metrics (e.g. `RabbitMQ-Overview.json` shown in Fig. 5.5; runtime memory allocators, inter-node communication), which are specified in the local `./grafana/` directory.

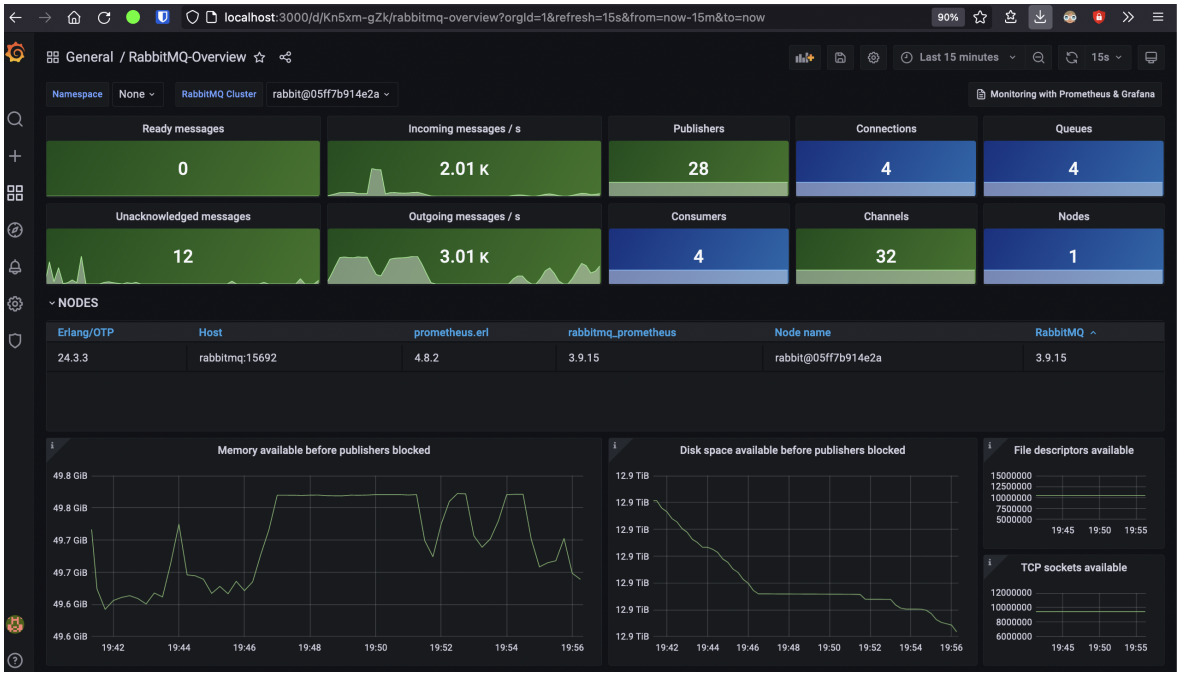


Figure 5.5: Grafana RabbitMQ overview dashboard on http://localhost:3000 during a load test.

5.2.3 Health Check API

As discussed earlier, monitoring is crucial for any application, the basic aspect of which happens to be *health checking*. This is especially true for modern-day microservices, which are usually hosted on the cloud: affected by several factors such as network latency, bandwidth, and performance of "black-box" host machines. Hence, the health of services should be verified regularly as a part of the monitoring infrastructure, to maintain SLAs (service level agreements) and promised availability levels.

The RabbitMQ HTTP API reference, including details on health check endpoints, is available here [7]. The functioning of the following health check endpoints (with status code 200 if healthy) is demonstrated (using VS Code's REST Client) in supplementary figures:

- `/api/health/checks/alarms` (Fig. 7.5): Responds a 200 OK if there are no alarms in effect in the cluster, otherwise responds with a 503 Service Unavailable.
- `/api/health/checks/certificate-expiration/{within}/{unit}` (Fig. 7.6): Checks the expiration date on the certificates for every listener configured to use TLS. Responds a 200 OK if all certificates are valid (have not expired), otherwise responds with a 503 Service Unavailable. Valid units: days, weeks, months, years. The value of the `within` argument is the number of units. So, when `within` is 2 and `unit` is "months", the expiration period used by the check will be the next two months.
- `/api/health/checks/protocol-listener/{protocol}` (Fig. 7.7): Responds a 200 OK if there is an active listener for the given protocol, otherwise responds with a 503 Service Unavailable. Valid protocol names are: amqp091, amqp10, mqtt, stomp, web-mqtt, web-stomp.

A few other useful endpoints are listed here, with details available in the API reference:

[7] https://pulse.mozilla.org/api/index.html

- `/api/health/checks/local-alarms`
- `/api/health/checks/port-listener/{port}`
- `/api/health/checks/virtual-hosts`
- `/api/health/checks/node-is-mirror-sync-critical`
- `/api/health/checks/node-is-quorum-critical`

The next three design patterns are invaluable to a microservice-based application like the ones implemented in case studies 1 and 2, and are hence common to both:

5.2.4 Externalised Configuration

Docker Compose environment variables are used to configure all the microservices in this application. The environment variables set by Docker are translated into Spring Boot application properties, e.g. `SERVER_PORT` becomes `server.port`, `SPRING_RABBITMQ_HOST` becomes `spring.rabbitmq.host`, and so on, without even having to specify the properties in the standard `application.yml`.

```
spring:
  application:
    name: cinema-ucd-service

data_file: data/movies.json

amqp:
  exchange: cinema
  request:
    routingKey: request.to.cinema
    serviceOnlyRoutingKey: request.to.cinema.ucd
  response:
    routingKey: response.from.cinema
```

Listing 5.2: UCD cinema's `application.yml`.

The listing above shows how the data file used to populate MongoDB, and other RabbitMQ configurations such as the name of exchange and routing keys are set in the application properties. When access is required in the source code, Spring uses the `@Value` annotation to read said properties. As mentioned in case study 1, externalising configuration in this manner helps with organisation and reduces avoidable development (human) errors.

5.2.5 Database per Service

Maintaining a separate MongoDB database for each cinema microservice is still the most practical approach in a distributed system to avoid a data operation bottleneck. A server is set up with Docker, and multiple databases are created again with the help of Spring Boot environment variables such as `SPRING_DATA_MONGODB_DATABASE` and `SPRING_DATA_MONGODB_PORT`.

```
cinema-ucd-service:
  ...
  environment:
    ...
    - SPRING_DATA_MONGODB_HOST=mongodb
```

```
6       - SPRING_DATA_MONGODB_DATABASE=cinema-ucd
7       - SPRING_DATA_MONGODB_PORT=27017
8     ...
```

Listing 5.3: Snippet from `docker-compose.yml` for UCD cinema.

5.2.6 Service Instance per Container

The ease of spinning up Docker containers makes this deployment pattern the industry standard for microservices. This is especially true for this second case study, where much additional infrastructure is required, such as RabbitMQ messaging system, Prometheus and Grafana servers for monitoring, which are all deployed as independent Dockerised services. Exposing ports such as 9090 (Prometheus), 3000 (Grafana), 15672 (RabbitMQ management) provides access to various user interfaces to host machine browsers. Using Docker Swarm or Kubernetes with Docker allows container scaling and orchestration, which are important when the load creates a need for multiple containers across multiple host machines.

5.3 Evaluation and Results

Similar to the previous case study, the same approach of modelling, manual API testing and JMeter load testing is adopted here.

5.3.1 Performance Modelling

The significant change in communication mechanism from synchronous REST alone (case study 1), to asynchronous request/response (external) and messaging (internal) is expected to yield performance benefits. Fig. 5.6 shows quite a different sequence diagram for the identical movie listing use case explored in case study 1's evaluation.

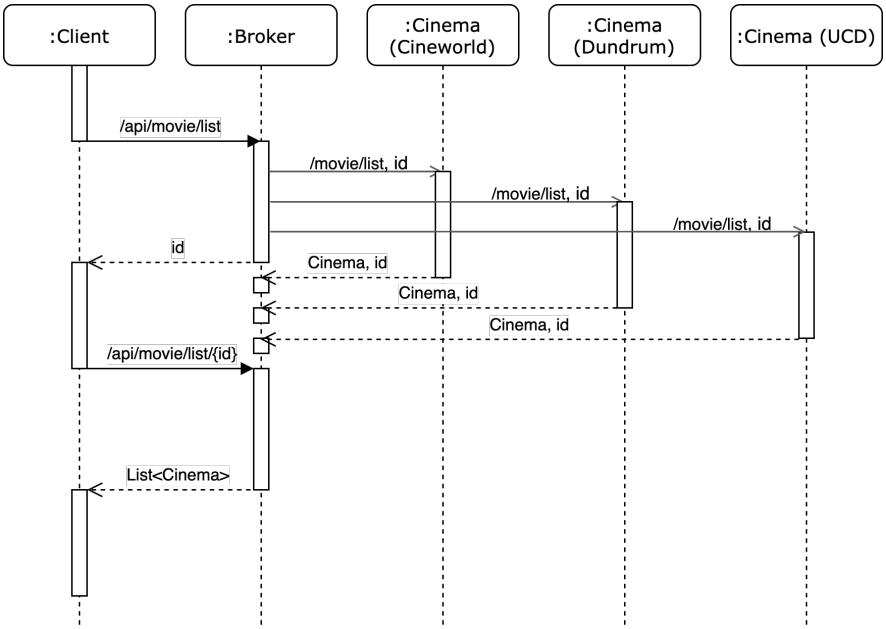


Figure 5.6: UML sequence diagram for listing movies from all cinemas (case study 2).

Sequence of steps:

- Firstly, contacting the Broker via synchronous REST only returns a correlation ID of the forwarded request, instead of a list of movies.
- Next, the Broker sends asynchronous messages (open arrowheads, solid lines) to each Cinema via the RabbitMQ infrastructure (topic exchange, queues, routing keys), without expecting an immediate response.
- When a Cinema is done processing the request, it returns the list of movie showtimes to the Broker, along with the request ID. This is known as an asynchronous callback.
- The Broker caches the responses from Cinemas using the same correlation ID, ready to be requested by the Client.
- When the Client makes another synchronous REST API call to the Broker with the appropriate ID, the Broker returns any responses that were cached in the time since the initial request.

The above model suggests the possibility of higher overall system performance compared to the first case study, since all inter-service communication is asynchronous and non-blocking. This is a well-accepted approach when dealing with microservice architecture - to have the external client or frontend facing service (like the broker) expose a synchronous REST API, while the internal microservices (like the cinemas with the broker) communicate asynchronously (here, via message queues).

Requests are linked to their corresponding responses by some form of correlation ID. However, care must still be taken to scale the Broker service as needed to handle increasing load, since it is the single point of entry for external clients.

5.3.2 Manual API Testing

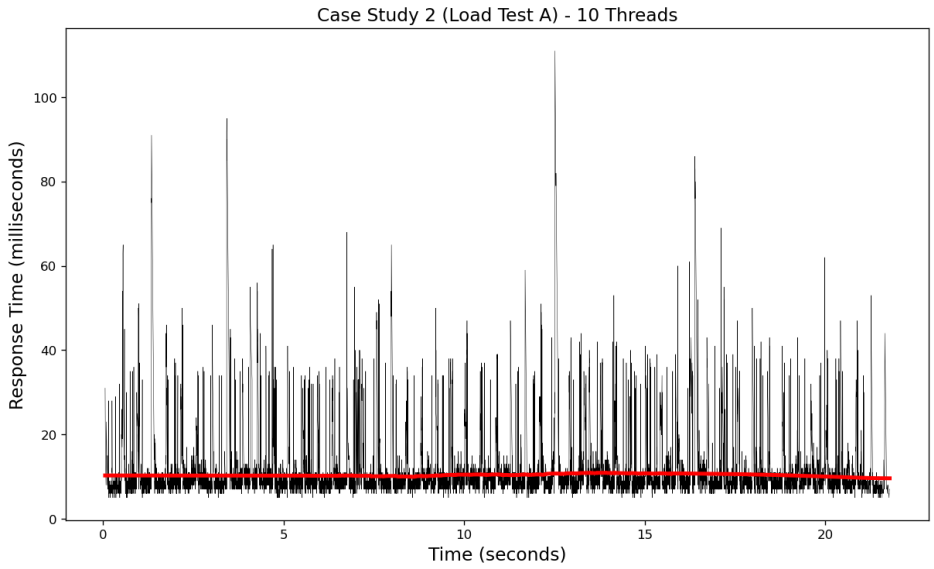
Again, using VS Code's REST Client, various HTTP requests (stored under `src/main/resources/http/`) to the client-facing Broker service (port 8099) were timed. Internal cinema services use RabbitMQ queues, a topic exchange and routing keys to pass messages to/from the central Broker. The following endpoints were tested to demonstrate the functionality of the web application:

- `/api/movie/list` (Fig. 7.8): The `/list` endpoint sends a request to the Broker to list the movie showtimes from all cinema services. Since the Broker communicates with the cinemas via asynchronous messaging, only an alphanumeric series of characters (separated by hyphens) is returned by the GET request (in 160 ms), which is the request's correlation ID.
- `/api/movie/list/{correlationId}` (Fig. 7.9): On providing the same correlation ID received earlier as a path variable in a new GET request, the full list of movie showtimes from all cinemas is returned (in 78 ms). The Broker uses the time between the two requests to cache responses from any cinemas that are can process the request in the given time frame.
- `/api/cinema/cineworld/reservation/make` (Fig. 7.10): A targeted request to the Cineworld cinema must also go via the message Broker (`/api` prefix), unlike the first case study where such a request is directly sent to the appropriate cinema via an API gateway. Here, the Broker creates a custom routing key to forward the request only to the appropriate cinema's request queue, instead of broadcasting it to all queues. A POST request to Cineworld contains booking details identical to those seen in the first case study, and a correlation ID is returned in 65 ms.
- `/api/cinema/cineworld/reservation/make/{correlationId}` (Fig. 7.11): A GET request to the `/reservation/make` endpoint with the appropriate correlation ID yields a response in 31 ms with the successful reservation details.
- `/api/cinema/cineworld/reservation/list` (Fig. 7.12): It only takes 10 ms to fetch the correlation ID for a reservation list GET request to Cineworld.
- `/api/cinema/cineworld/reservation/list/{correlationId}` (Fig. 7.13): In 13 ms, the server can list all the reservations at Cineworld, thus confirming the success of the earlier `/reservation/make` request.

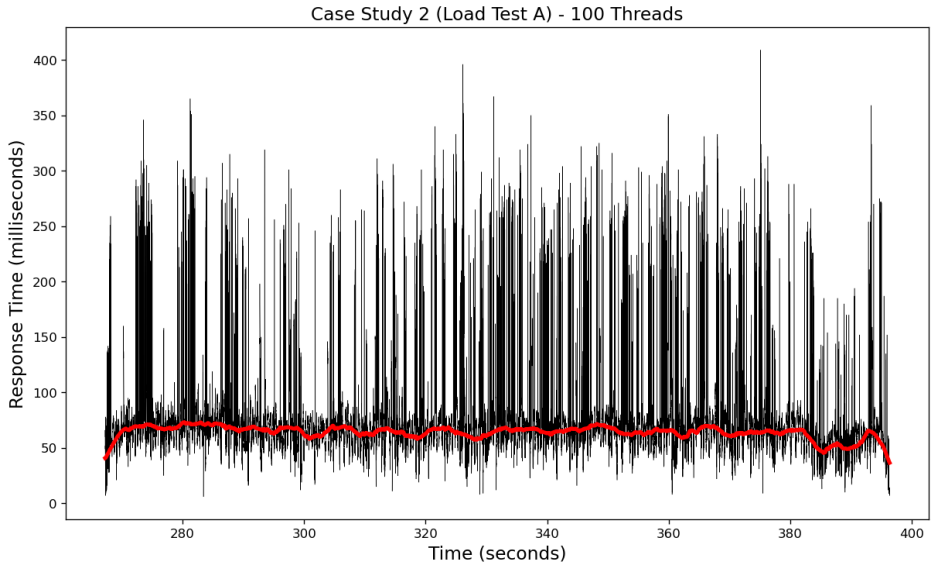
This asynchronous request/response pattern is especially useful in preventing client blocking (as seen in the first case study), since the client can now perform other tasks after sending a series of requests to the Broker, without having to wait for every single request to receive a full response before moving on.

5.3.3 Performance Testing with JMeter

Using JMeter again, two load tests (A and B) were designed in a similar fashion to case study 1. However, due to the asynchronous nature of communication in case study 2, each test plan includes two HTTP requests: one to send the initial request to the Broker and get a correlation ID, and another to use the correlation ID to fetch the final response. A regular expression extractor (post-processor) was used in JMeter to extract the correlation ID from the response body after the first request, then use it as a path variable for the second request. Other test parameters are identical to case study 1, for instance: constant load increments from 10 to 100 threads, 1-second ramp-up, and 1000 test iterations at each load level for reliability.



(a)



(b)

Figure 5.7: Load test A response times using (a) 10 threads and (b) 100 threads.

Load test A involves the Broker API endpoints `/api/movie/list` and `/api/movie/list/{correlationId}`. As seen in Fig. 5.7, the response time at both load levels - 10 threads and 100 threads - were far more variable, but lower on average compared to the case study 1 observations. However, applying a Savitzky-Golay filter (red line) as before helps smoothen the data points for more readable plots. Both load scenarios showed constant response times approximately under 10 ms and 75 ms - both lower than the case study 1 observations for identical loads. Tuning the RabbitMQ queue configurations (e.g. resource allocations) to better suit the system under test would reduce the variability in response times.

Fig. 5.8 proves that error probabilities for load test A in case study 2 (less than 3%) are negligible compared to those observed in case study 1 (Fig. 4.6: 15-21% under heavy load). A naive assumption incorporated into the test plan is that the Broker is able to finish caching the response before the second request to get the full response is received - and this is shown to be true for the vast majority of cases (the few failures - under 3% - are depicted by the 500 server error codes in the table). However, a better approach would be to separate the consecutive requests with a fixed

waiting period, and disregard the wait time when plotting average response times and calculating other metrics. The idea behind asynchronous requests as shown here is that the client does not require an immediate response to a query, but emulating such a scenario is difficult when response timing is involved while keeping track of correct correlation IDs.

Threads	responseCode	
10	200	98%
	500	2%
20	200	98%
	500	2%
30	200	98%
	500	2%
40	200	97%
	500	3%
50	200	97%
	500	3%
60	200	99%
	500	1%
70	200	99%
	500	1%
80	200	98%
	500	2%
90	200	97%
	500	3%
100	200	99%
	500	1%

Figure 5.8: Proportion of HTTP response codes during load test A at different load levels.

Finally, a familiar linear plot with high Pearson correlation of **0.998** in Fig. 5.9 confirms the linearity of resource utilisation with respect to load. An exponential curve instead would be indicative of a struggling system under constant load increments.

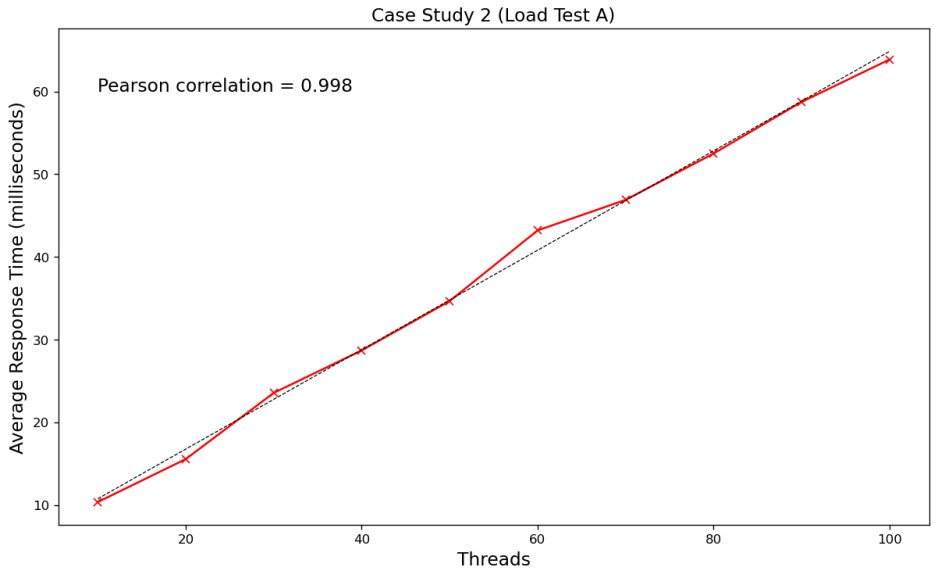
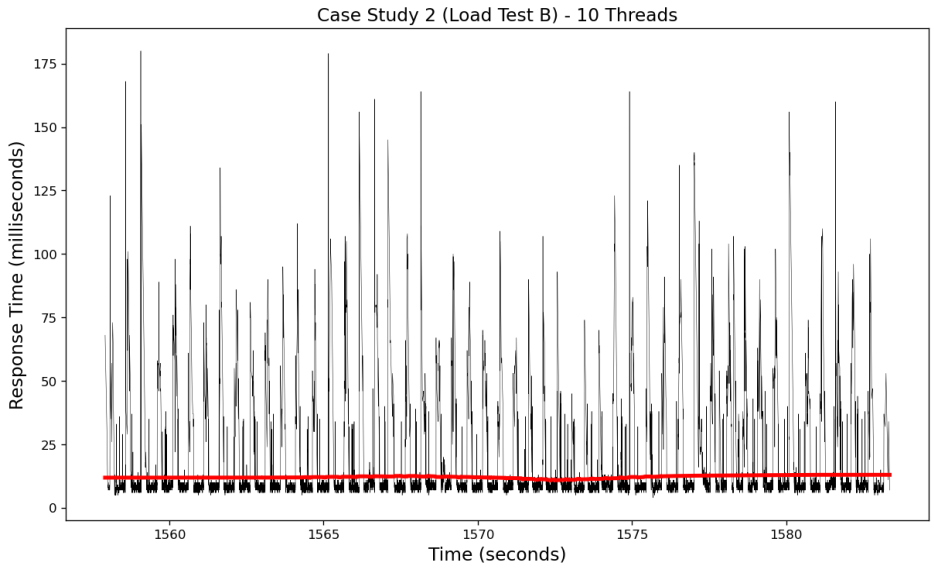
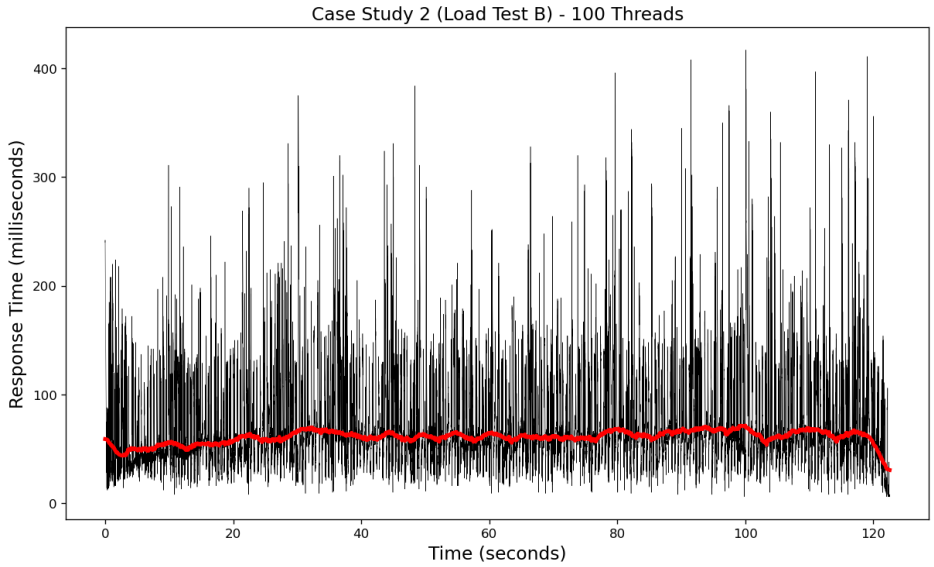


Figure 5.9: Average response time vs number of threads in load test A.

Load test B is designed to check the ticket reservation functionality, corresponding to endpoints `/api/cinema/cineworld/reservation/make` and `/api/cinema/cineworld/reservation/make/{correlationId}`, with an identical client booking payload (Jane Doe, Cineworld) for POST requests. The second request in each step is a simple GET request with the extracted correlation ID as a path variable.



(a)



(b)

Figure 5.10: Load test B response times using (a) 10 threads and (b) 100 threads.

Fig. 5.10 shows highly variable response times, but mostly constant Savitzky-Golay smooth red curves, measured at approximately 10 ms and 50 ms for load levels of 10 threads and 100 threads respectively. In this case, the average response times were higher than those in the previous case study, possibly due to the effect of two communication round trips, and request/response travel via a congested Broker service. This suggests that the Broker and RabbitMQ queues need to be configured with greater hardware resources to avoid a performance bottleneck.

Threads	responseCode	
10	200	98%
	500	2%
20	200	98%
	500	2%
30	200	97%
	500	3%
40	200	97%
	500	3%
50	200	97%
	500	3%
60	200	97%
	500	3%
70	200	97%
	500	3%
80	200	96%
	500	4%
90	200	97%
	500	3%
100	200	96%
	500	4%

Figure 5.11: Proportion of HTTP response codes during load test B at different load levels.

Since the POST and GET requests were performed without a time gap in between, there is a small percentage of errors observed here (Fig. 5.11), while there were none in case study 1 (Fig. 4.9). As mentioned earlier, these errors are observed in cases where the Broker doesn't finish storing microservice responses for some correlation IDs, consequently the client cannot access the endpoint by providing an ID. Scaling the services is a plausible solution since the prerequisite is satisfied - resource utilisation levels are directly proportional to the system load (Pearson correlation of **0.992** between average response time and the number of threads in the load test).

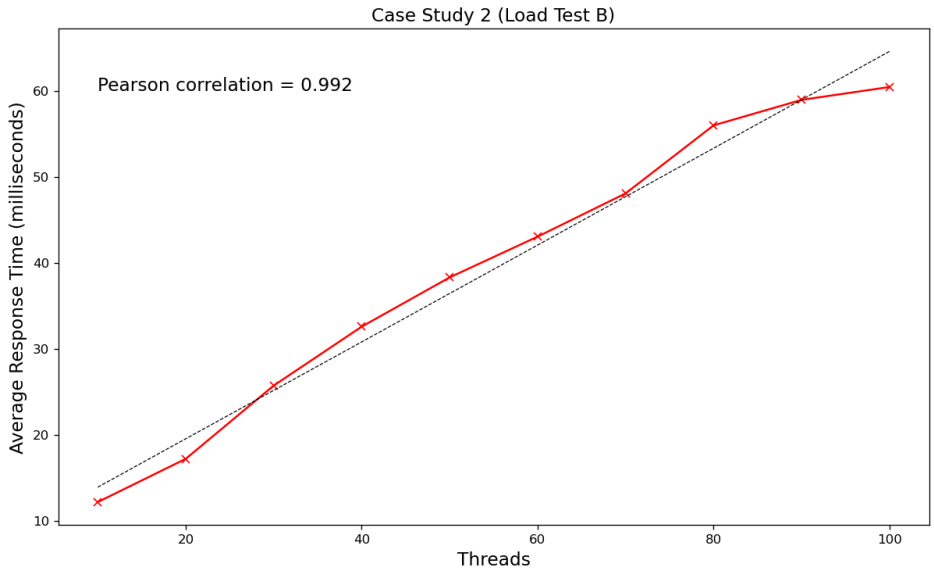


Figure 5.12: Average response time vs number of threads in load test B.

Chapter 6: **Conclusions**

In conclusion, it is appropriate to reflect on the performance impact of microservices, design patterns, as well as the interpretation of case study evaluation results.

6.1 Summary

With the rising popularity of microservices, the number of systems migrating from monolithic to fine-grained microservice architecture is increasing by the day. Design patterns are essential to developers working with microservices, with every effort made to avoid anti-patterns which can hinder system performance.

Applying the most suitable design patterns for a given business use case is critical to achieving a functional system, and importantly, a *performant* system able to deal with changing demand. Performance engineering is a vital part of the software development lifecycle, and clear objectives must be set to achieve the best performance at a given budget.

By building two web applications using a wide range of microservice design patterns, the performance benefits and shortcomings of each pattern were clearly demonstrated. While synchronous REST APIs are required for communication with some clients (e.g. frontend/UI), adopting an asynchronous strategy such as messaging for internal communication is highly recommended in order to avoid client blocking.

For microservices exposing an API for clients, using an API gateway along with automatic service discovery greatly reduces the necessary effort in client-side code. When using message queues, correctly configuring the infrastructure to use adequate hardware resources is an important task. Monitoring and observability patterns are crucial to tracking system performance over time, especially with useful visualisations instead of simple log dumps. Some design patterns such as externalising configuration, having a separate database per service, or using Docker to deploy a single service instance per container, are extremely useful and applicable to the vast majority of microservice-based applications.

Evaluation of case studies in this project suggests the following key takeaways:

- Choosing an appropriate evaluation strategy is vital and is dependent on clear, pre-defined performance goals.
- Performance modelling and manual API testing are useful in the early stages of development. Any performance regressions and bottlenecks can be identified by modelling and then verified by large-scale testing. In the above case studies, the API gateway and Broker services are shown to be potential performance bottlenecks (single point of entry and failure e.g. due to congestion) and are hence prime candidates for service scaling to avoid issues.
- Automated performance testing, such as load testing, is essential for reliable performance analysis. Simple load testing reveals that the first case study (API gateway + REST) was observed to be approximately **25%** slower (response time) and **7** times more error-prone on average to fetch an aggregated list of movie showtimes from all services, under

heavy load (100 threads) compared to the much faster and more reliable second case study (asynchronous messaging). However, when contacting a single cinema service to make a reservation, the API gateway triumphs with about **5** times faster responses compared to the queueing system. There were several factors (both hardware and software configurations) affecting the observed results, and it is quite possible to further optimise the services to obtain more distinctive results in each case study. Importantly, as long as common anti-patterns are avoided, no single design pattern is superior to another: the choice between similar patterns should depend on the business requirement.

- Since performance measurements can vary significantly, ensuring reliability through multiple test iterations is required. Data smoothing techniques should also be applied to obtain more usable plots.

Overall, this work has been a success in implementing various common microservice design patterns and analysing their performance impact. As is the case with all projects, there is a scope for improvement, to be built upon and continued in future works.

6.2 Future Work

This project has a number of limitations that could be tackled by similar projects in the future. First and foremost, countless microservice design patterns and anti-patterns are constantly evolving and could be demonstrated through further case studies. Moreover, showing the migration process from monoliths to microservices using refactoring patterns should be an insightful study.

Today, the vast majority of microservices are deployed in the cloud (AWS, Google Cloud Platform, Microsoft Azure, and the like), thanks to various benefits such as ease of scalability and auto-provisioning resources, faster release, reduced cost, ease of management, and much more. Consequently, several cloud-specific design patterns have emerged. With cloud deployment, the use of container orchestration tools like Kubernetes are indispensable in the industry for managing a large number of containerised services, and to enable ease of scaling, reliability and fault tolerance.

To evaluate microservice-based applications, only simple and limited load testing was carried out in this project, entrusting more rigorous stress, spike and integration tests to future studies.

In the end, performance must not be an afterthought in the lifecycle, since delivering well-performing systems is arguably equally important as software functionality. Practices involving microservice architectures are still growing and developing constantly, and the shape they take in the future would be positively impacted by engineering and management teams taking measures to integrate performance considerations into the development process.

Acknowledgements

I would like to express my sincere gratitude towards:

- Professor John Murphy, my project supervisor, for his guidance, support and understanding throughout the project.
- Associate Professor Rem Collier, for sharing his knowledge and experience with distributed systems and technologies.
- My team (Ulysses - Enterprise Engineering) at Amazon Web Services (Dublin), for introducing me to the world of microservice architecture.
- UCD School of Computer Science, for providing access to a Linux compute server.
- My family and friends - especially Sharon Farrell, Ciara Murphy and Alexander Bourke, for their constant encouragement.

Bibliography

[1] J. Lewis and M. Fowler. 'Microservices: a definition of this new architectural term.' (25th Mar. 2014), [Online]. Available: https://martinfowler.com/articles/microservices.html (visited on 26th Oct. 2021).

[2] D. Parnas, 'On the Criteria To Be Used in Decomposing Systems into Modules,' *Communications of the ACM*, vol. 15, pp. 1053–1058, Dec. 1972. DOI: 10.1145/361598.361623.

[3] E. W. Dijkstra, 'On the role of scientific thought,' 1974. [Online]. Available: https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html.

[4] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Dec. 2014, Book.

[5] O. Zimmermann, 'Microservices tenets: Agile approach to service development and deployment,' *Computer Science - Research and Development*, vol. 32, Nov. 2016. DOI: 10.1007/s00450-016-0337-0.

[6] C. Pahl and P. Jamshidi, 'Microservices: A Systematic Mapping Study,' Jan. 2016, pp. 137–146. DOI: 10.5220/0005785501370146.

[7] N. Alshuqayran, N. Ali and R. Evans, 'A Systematic Mapping Study in Microservice Architecture,' Nov. 2016, pp. 44–51. DOI: 10.1109/SOCA.2016.15.

[8] P. Di Francesco, P. Lago and I. Malavolta, 'Architecting with Microservices: a Systematic Mapping Study,' *Journal of Systems and Software*, vol. 150, Apr. 2019. DOI: 10.1016/j.jss.2019.01.001.

[9] Amazon Web Services, 'Implementing Microservices on AWS,' Tech. Rep., 9th Nov. 2021. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices-on-aws.html (visited on 28th Nov. 2021).

[10] E. Gamma, R. Helm, R. Johnson and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994, ISBN: 0201633612.

[11] C. Alexander, S. Ishikawa and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977, ISBN: 0195019199.

[12] C. Richardson. 'A pattern language for microservices,' [Online]. Available: https://microservices.io/patterns/index.html (visited on 28th Nov. 2021).

[13] M. Udantha. 'Design Patterns for Microservices.' (30th Jul. 2019), [Online]. Available: https://dzone.com/articles/design-patterns-for-microservices-1 (visited on 26th Oct. 2021).

[14] J. Bogner, A. Zimmermann and S. Wagner, 'Analyzing the Relevance of SOA Patterns for Microservice-Based Systems,' Mar. 2018.

[15] T. Erl, *SOA Design Patterns*. Boston, MA, USA: Pearson Education, 2009.

[16] T. Erl, B. Carlyle, C. Pautasso and R. Balasubramanian, *SOA with REST - Principles, Patterns and Constraints for Building Enterprise Solutions with REST*, ser. The Prentice Hall service technology series. Prentice Hall, 2012.

[17] A. Rotem-Gal-Oz, *SOA Patterns*. Shelter Island, NY: Manning Pubications Co., 2012.

[18] D. Taibi, V. Lenarduzzi and C. Pahl, 'Architectural Patterns for Microservices: A Systematic Mapping Study,' in *CLOSER*, SciTePress, 2018, pp. 221–232.

[19] K. Cully, 'Performance Problems Inherent to Microservices with Independent Communication and Resiliency Configuration,' M.S. thesis, University College Dublin, Ireland, Mar. 2020.

[20] M. Fowler. 'Microservice Prerequisites.' (28th Aug. 2014), [Online]. Available: https://martinfowler.com/bliki/MicroservicePrerequisites.html (visited on 29th Nov. 2021).

[21] M. Fowler. 'Microservice Trade-Offs.' (1st Jul. 2015), [Online]. Available: https://martinfowler.com/articles/microservice-trade-offs.html (visited on 29th Nov. 2021).

[22] V. Alagarasan. 'Seven Microservices Anti-patterns.' (24th Aug. 2015), [Online]. Available: https://www.infoq.com/articles/seven-uservices-antipatterns/ (visited on 29th Nov. 2021).

[23] J. Kanjilal. 'Overcoming the Common Microservices Anti-Patterns.' (2nd Nov. 2021), [Online]. Available: https://www.developer.com/design/solving-microservices-anti-patterns/ (visited on 29th Nov. 2021).

[24] Netflix Technology Blog. 'The Netflix Simian Army.' (19th Jul. 2011), [Online]. Available: https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116 (visited on 29th Nov. 2021).

[25] C. O'Hanlon, 'A Conversation with Werner Vogels: Learning from the Amazon Technology Platform,' pp. 14–22, 2006. DOI: 10.1145/1142055.1142065.

[26] A. O. Portillo-Dominguez, M. Wang, J. Murphy, D. Magoni, N. Mitchell, P. F. Sweeney and E. Altman, 'Towards an Automated Approach to Use Expert Systems in the Performance Testing of Distributed Systems,' New York, NY, USA: Association for Computing Machinery, 2014. DOI: 10.1145/2631890.2631895.

[27] O. Portillo, P. Perry, D. Magoni and J. Murphy, 'PHOEBE: an automation framework for the effective usage of diagnosis tools in the performance testing of clustered systems,' *Software: Practice and Experience*, Apr. 2017. DOI: 10.1002/spe.2500.

[28] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar and M. Steinder, 'Performance Evaluation of Microservices Architectures Using Containers,' in *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015, pp. 27–34. DOI: 10.1109/NCA.2015.49.

[29] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte and J. Wettinger, 'Performance Engineering for Microservices: Research Challenges and Directions,' in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ser. ICPE '17 Companion, L'Aquila, Italy: Association for Computing Machinery, 2017, pp. 223–226, ISBN: 9781450348997. DOI: 10.1145/3053600.3053653.

[30] M. Gribaudo, M. Iacono and D. Manini, 'Performance Evaluation Of Massively Distributed Microservices Based Applications,' May 2017, pp. 598–604. DOI: 10.7148/2017-0598.

[31] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović and A. van Hoorn, 'Microservices: A Performance Tester's Dream or Nightmare?' In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20, Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 138–149, ISBN: 9781450369916. DOI: 10.1145/3358960.3379124.

[32] A. U. Gias, A. van Hoorn, L. Zhu, G. Casale, T. F. Düllmann and M. Wurster, 'Performance Engineering for Microservices and Serverless Applications: The RADON Approach,' in *Companion of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20, Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 46–49, ISBN: 9781450371094. DOI: 10.1145/3375555.3383120.

[33] A. Akbulut and H. G. Perros, 'Performance Analysis of Microservice Design Patterns,' *IEEE Internet Computing*, vol. 23, no. 6, pp. 19–27, 2019. DOI: 10.1109/MIC.2019.2951094.

[34] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes and J. Saraiva, 'Energy efficiency across programming languages: How do energy, time, and memory relate?' In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017, Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267, ISBN: 9781450355254. DOI: 10.1145/3136014.3136031.

[35] S. Peyrott. 'An Introduction to Microservices, Part 3: The Service Registry.' (2nd Oct. 2015), [Online]. Available: https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/ (visited on 10th Apr. 2022).

[36] Object Management Group. 'OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1.' (2011), [Online]. Available: https://www.omg.org/spec/UML/2.4.1/Superstructure/PDF (visited on 17th Apr. 2022).

[37] Amazon Web Services. 'What is DevOps?' [Online]. Available: https://aws.amazon.com/devops/what-is-devops/ (visited on 26th Oct. 2021).

[38] C. Richardson, *Microservices Patterns: With Examples in Java*. Manning Publications, Oct. 2018, Book.

[39] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly Media, Mar. 2017, Book.

[40] M. Kamaruzzaman. 'Effective Microservices: 10 Best Practices.' (23rd Nov. 2019), [Online]. Available: https://t.co/ZM78yg19OR?amp=1 (visited on 26th Oct. 2021).

[41] M. Kamaruzzaman. 'Microservice Architecture and its 10 Most Important Design Patterns.' (15th Dec. 2020), [Online]. Available: https://t.co/9Du1U3X4Q6?amp=1 (visited on 26th Oct. 2021).

[42] S. Kappagantula. 'Everything You Need To Know About Microservices Design Patterns.' (25th Nov. 2020), [Online]. Available: https://www.edureka.co/blog/microservices-design-patterns (visited on 26th Oct. 2021).

Chapter 7: Appendix

7.1 Supplementary Figures

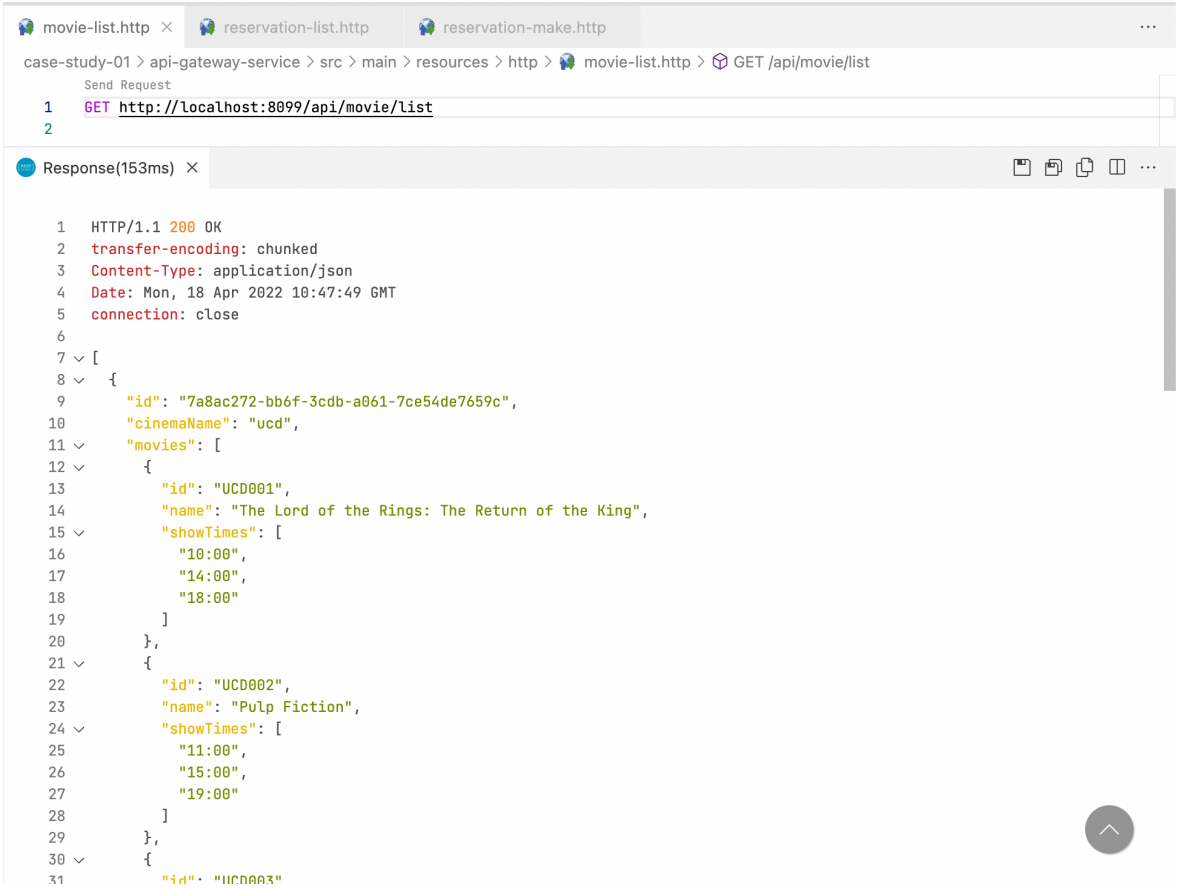


Figure 7.1: Case Study 1 - manual API testing (listing all movie showtimes from various cinemas).

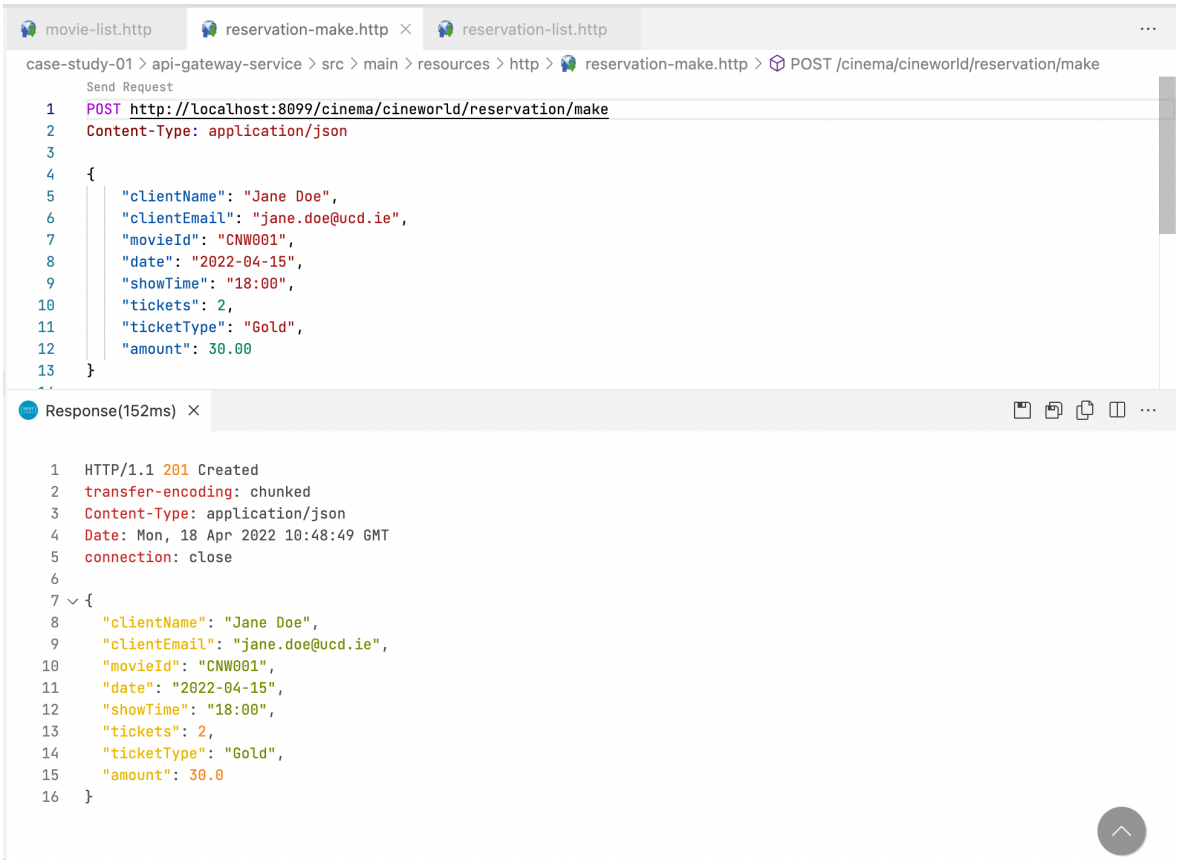


Figure 7.2: Case Study 1 - manual API testing (making a reservation at Cineworld cinema).

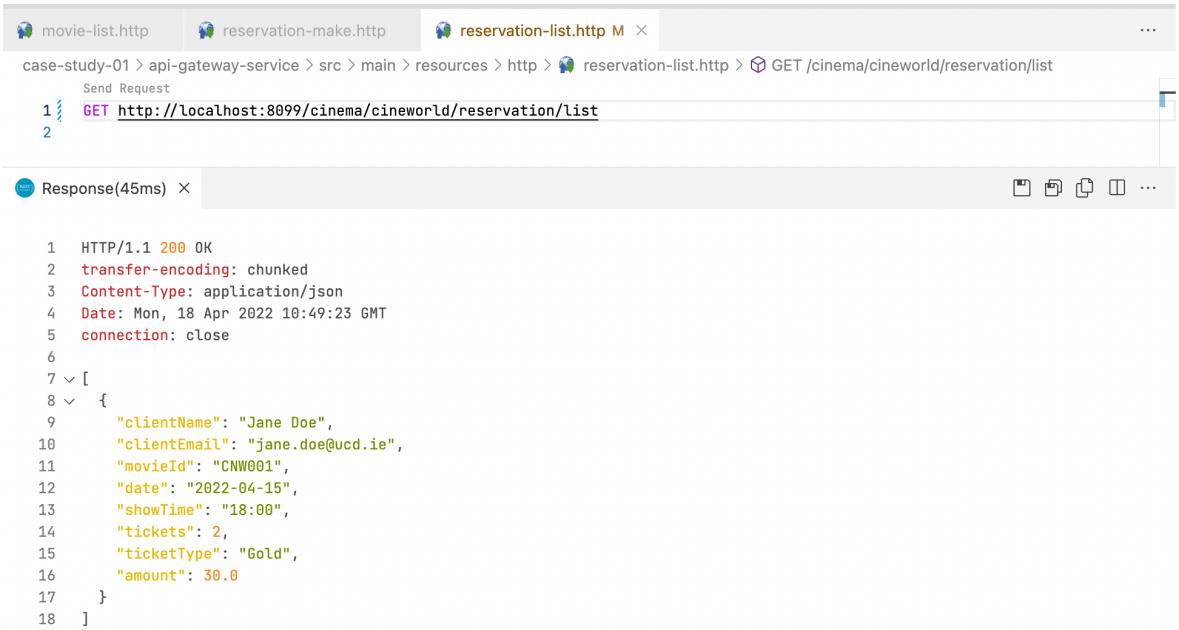


Figure 7.3: Case Study 1 - manual API testing (listing all reservations at Cineworld to ensure the previous reservation was successful).

reservation-delay.http U

case-study-01 > api-gateway-service > src > main > resources > http > reservation-delay.http > ...

```
Send Request
1  GET http://localhost:8099/cinema/cineworld/reservation/delay/4
2
```

Response(1048ms)

```
1  HTTP/1.1 500 Internal Server Error
2  Content-Type: text/plain;charset=UTF-8
3  Content-Length: 53
4  connection: close
5
6  Fallback: Circuit broken in cinema-cineworld-service!
```

Figure 7.4: Case Study 1 - manual API testing (reservation delay circuit breaker).

alarms.http

case-study-02 > broker-service > src > main > resources > http > monitoring > health > alarms.http > GET /api/he

```
Send Request
1  GET http://localhost:15672/api/health/checks/alarms
2  Authorization: Basic guest:guest
```

Response(4ms)

```
1  HTTP/1.1 200 OK
2  cache-control: no-cache
3  connection: close
4  content-length: 15
5  content-security-policy: script-src 'self' 'unsafe-eval' 'unsafe-inline'; object-src 'self'
6  content-type: application/json
7  date: Sun, 17 Apr 2022 15:02:55 GMT
8  server: Cowboy
9  vary: accept, accept-encoding, origin
10
11 {
12   "status": "ok"
13 }
```

Figure 7.5: RabbitMQ health check - alarms.

certificate-expiration.http

case-study-02 > broker-service > src > main > resources > http > monitoring > health > certificate-expiration.http >

```
Send Request
GET http://localhost:15672/api/health/checks/certificate-expiration/2/months
Authorization: Basic guest:guest
```

Response(6ms)

```
HTTP/1.1 200 OK
cache-control: no-cache
connection: close
content-length: 15
content-security-policy: script-src 'self' 'unsafe-eval' 'unsafe-inline'; object-src 'self'
content-type: application/json
date: Sun, 17 Apr 2022 15:03:10 GMT
server: Cowboy
vary: accept, accept-encoding, origin

{
  "status": "ok"
}
```

Figure 7.6: RabbitMQ health check - certificate expiration within a time unit (2 months).

protocol-listener.http

case-study-02 > broker-service > src > main > resources > http > monitoring > health > protocol-listener.http > G

```
Send Request
GET http://localhost:15672/api/health/checks/protocol-listener/amqp1
Authorization: Basic guest:guest
```

Response(6ms)

```
HTTP/1.1 503 Service Unavailable
cache-control: no-cache
connection: close
content-length: 126
content-security-policy: script-src 'self' 'unsafe-eval' 'unsafe-inline'; object-src 'self'
content-type: application/json
date: Sun, 17 Apr 2022 15:04:03 GMT
server: Cowboy
vary: accept, accept-encoding, origin

{
  "status": "failed",
  "reason": "No active listener",
  "missing": "amqp1",
  "protocols": [
    "http",
    "http/prometheus",
    "clustering",
    "amqp"
  ]
}
```

Figure 7.7: RabbitMQ health check - protocol listener.

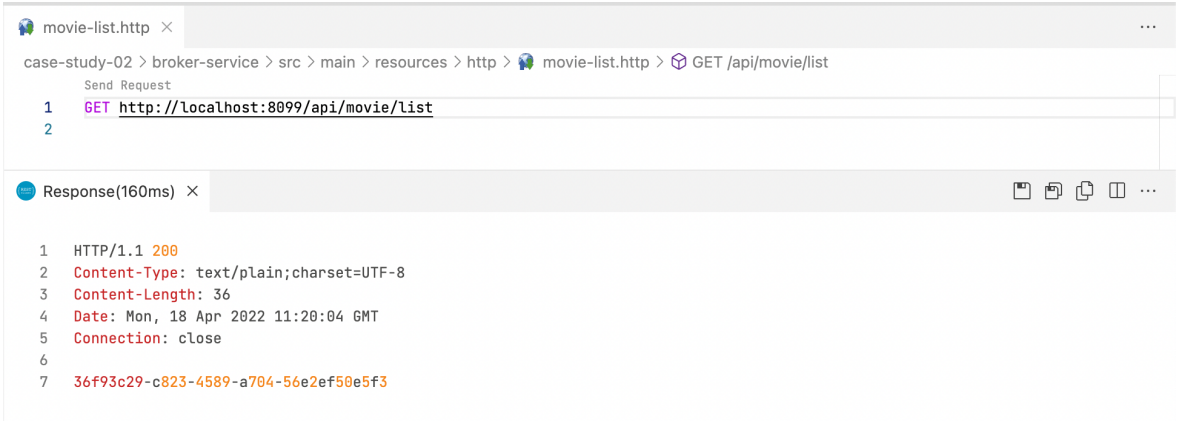


Figure 7.8: Case Study 2 - manual API testing (send request to list all movies from various cinemas).

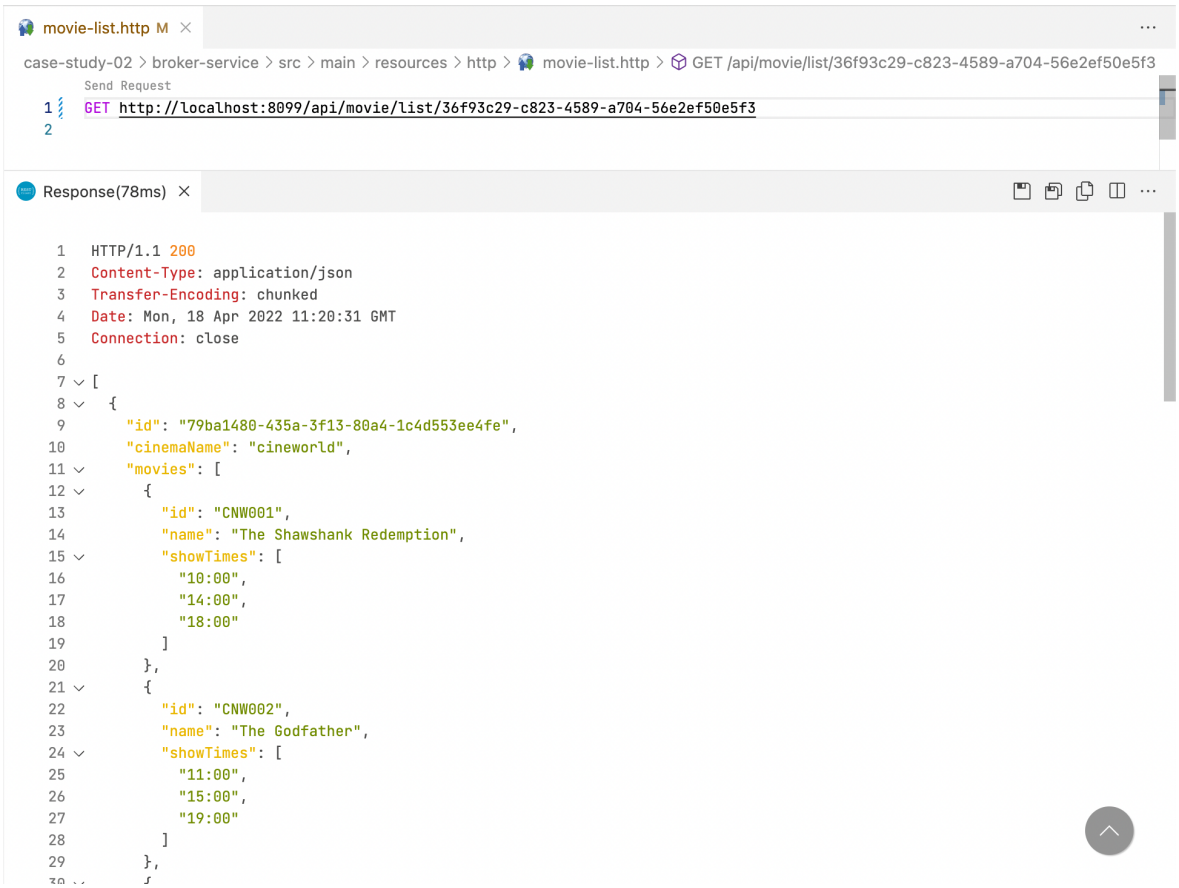


Figure 7.9: Case Study 2 - manual API testing (receive response listing all movies).

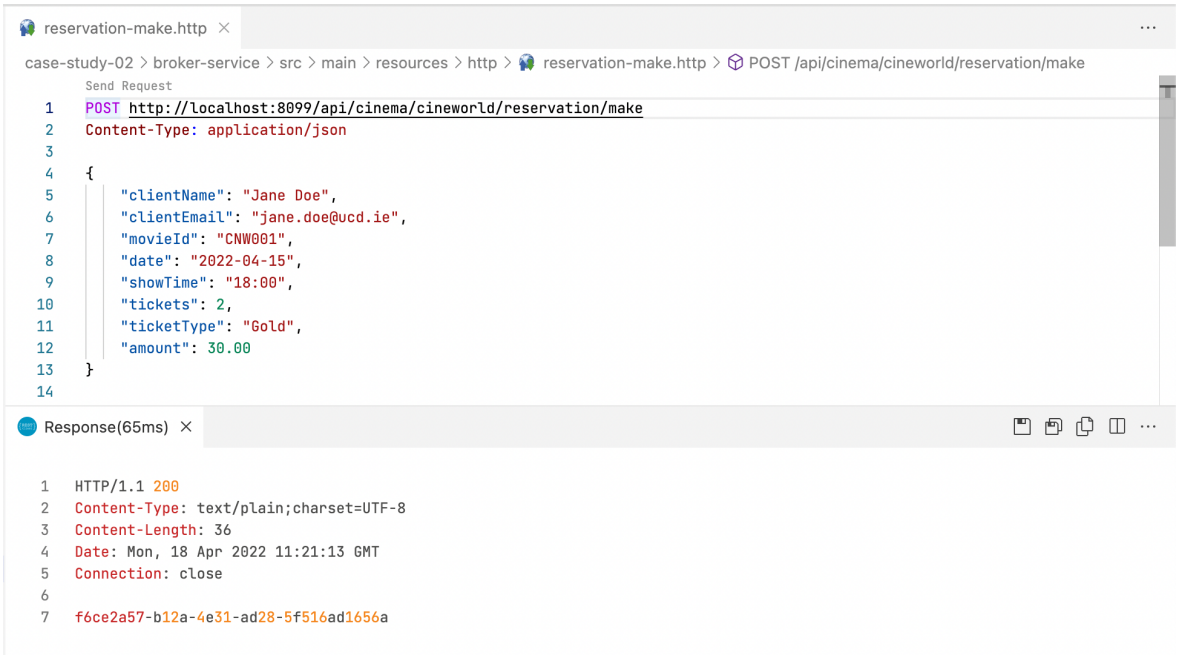


Figure 7.10: Case Study 2 - manual API testing (send request to make a reservation at Cineworld cinema).

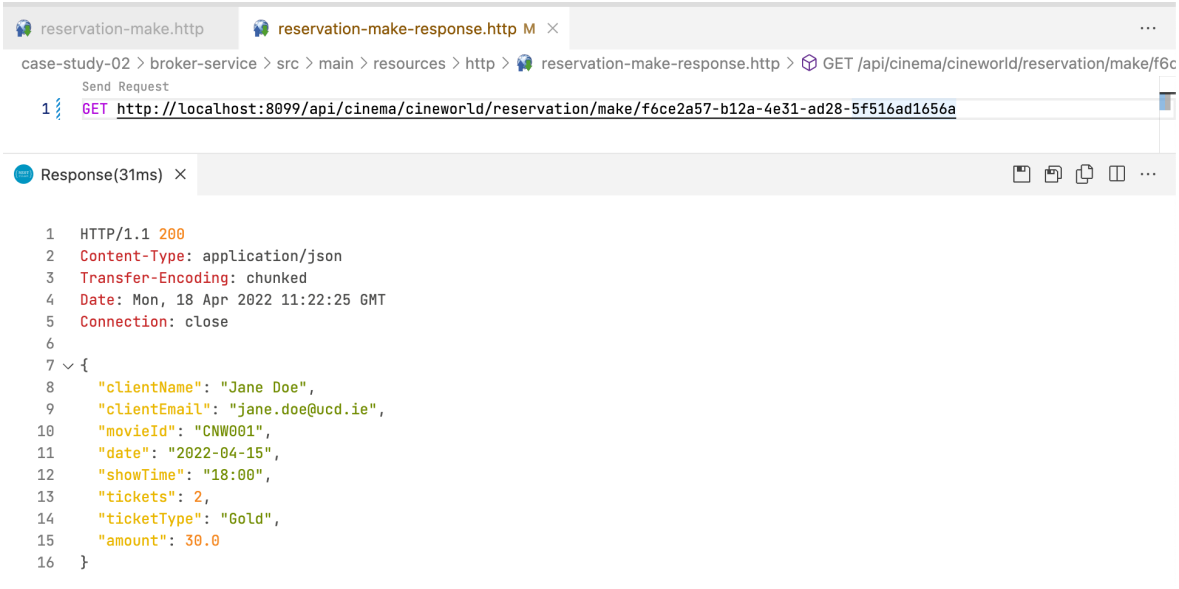


Figure 7.11: Case Study 2 - manual API testing (receive response from Cineworld showing reservation).

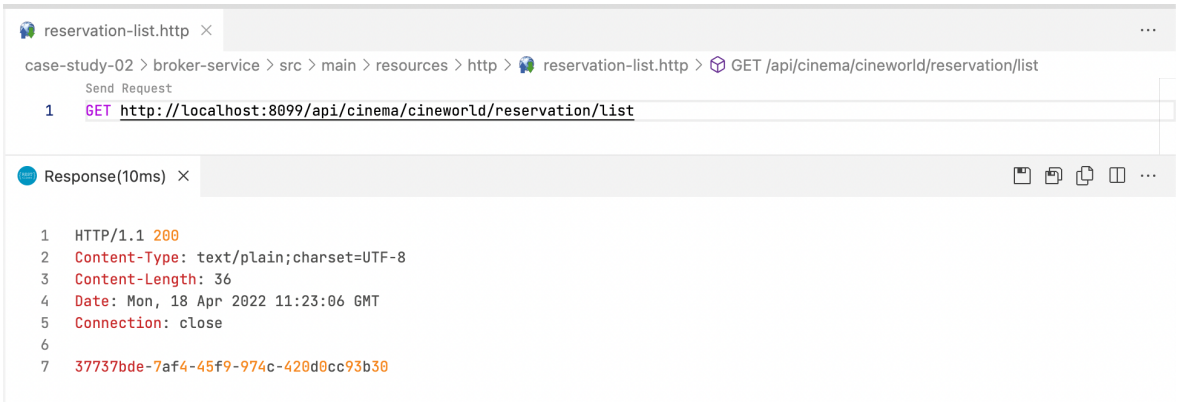


Figure 7.12: Case Study 2 - manual API testing (send request to list all reservations at Cineworld).

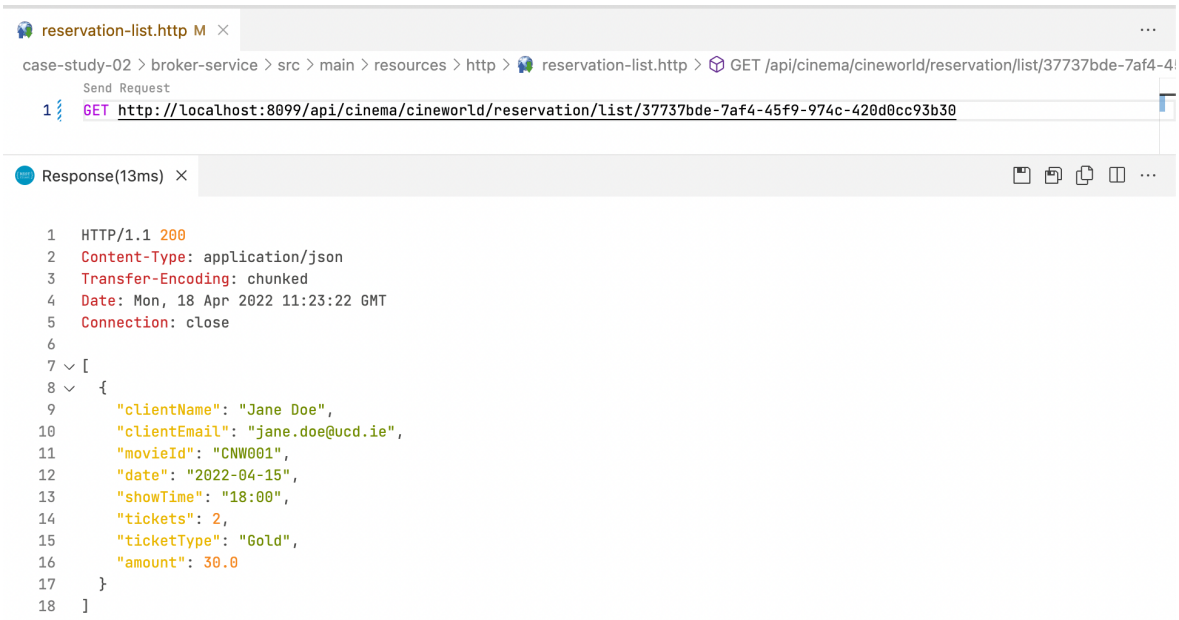


Figure 7.13: Case Study 2 - manual API testing (receive response showing all of Cineworld's reservations).

7.2 Initial Project Specification

7.2.1 Problem Statement

Microservice architecture is a style of designing software systems to be highly maintainable, scalable, loosely coupled and independently deployable. Moreover, each service is built to be self-contained and implement a single business capability. Design patterns in software engineering refer to any general, repeatable or reusable solution to recurring problems faced during the software design process. This project aims to study the performance engineering practices associated with a number of *microservice design patterns*, considering both qualitative and quantitative metrics to evaluate their benefits and shortcomings depending on the business requirement and use case. A non-exhaustive list of design patterns that could be explored is as follows:

- API Gateway

- Asynchronous Messaging
- Chain of Responsibility
- Database or Shared Data
- Circuit Breaker
- Externalise Configuration
- Aggregator
- Branch

Employing some of the aforementioned design patterns, sufficiently complex simulations will be designed for the performance engineering experiments. The project will also look at some common issues in microservices, and how they compare with traditional monolithic architectures.

7.2.2 Background

Microservices have gained traction in recent years with the rise of Agile software development and a DevOps [37] approach. As software engineers migrate from monoliths to microservices, it is important to make appropriate choices for system design and avoid "anti-patterns". Although no one design pattern can be called the "best", the performance of systems can be optimised by following design patterns suited to the use case, with the right configuration of hardware resources.

7.2.3 Related Work

Due to their popularity, microservices have been written about extensively in books like [38], [39], [4]. Articles such as [40], [41], [42], [13], [1] discuss the intricacies of microservice architecture as well as the trade-offs between various common design patterns. In [19], the performance problems inherent to microservices are explored, with evaluations performed using a custom-built prototyping suite. Akbulut and Perros [33] dive into the performance analysis aspect of microservices that is being proposed in this project, where they consider 3 different design patterns.

7.2.4 Datasets

Any data that is to be used or analysed in this project will be generated during the course of experiments. There are no dependencies on additional datasets.

7.2.5 Resources Required

A non-exhaustive list of resources is specified below, following a preliminary needs assessment.

- Languages/Frameworks: Java, Spring Boot, Python
- Tools: Git, Docker, Apache JMeter
- Database: MongoDB
- Compute: Linux server maintained by the UCD School of Computer Science

7.3 Project Work Plan

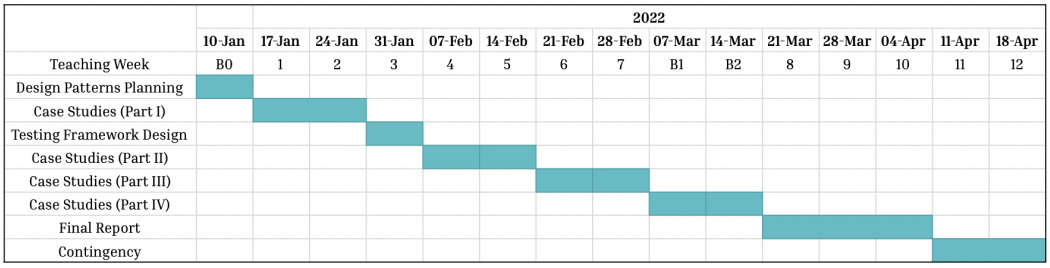


Figure 7.14: Gantt chart for project timeline.

- **Design Patterns Planning (1 week)**: The initial planning here is of particular significance and will decide the direction and pace of the project's implementation phase. Various microservice design patterns will be considered to select a few important patterns which can be easily demonstrated using simulations.
- **Testing Framework Design (1 week)**: Designing a simple testing framework (e.g. load testing plans) for the first case study will facilitate the adoption of similar strategies for subsequent case studies.
- **Case Studies (Parts I, II, III, IV) (8 weeks)**: These case studies will form the bulk of the project, and will be split into four parts for convenience, each comprising a set of related design patterns (each group of case studies will possibly address 2-3 patterns). The majority of effort required here will be concerned with the back-end development of dummy microservice-based systems using containers (Docker). Evaluation, both qualitative and quantitative, is well integrated with the implementation phase of the project, since performance analysis/testing will be carried out in tandem with the case study experiments.
- **Final Report (3 weeks)**: Although the core parts of the report should be written as progress is made with tasks, a dedicated period is set aside for refinement and completion.
- **Contingency (2 weeks)**: Time set aside to be used only in the event of unforeseen issues or challenges.