|  |  |
| --- | --- |
| **Module Title** | Advanced Programming |
| **Module Code** | KF7014 |
| **Academic Year / Semester** | 2024-25 / Semester 1 |
| **Module Leader** | Dr Maria Salama |

|  |  |
| --- | --- |
| **ASSESSMENT – Development of microservices-based application** | |
| **Individual Component** | |
| **Student Name** | Mubashir Zamir |
| **Student ID** | W24029487 |
| **Programme** | MSc Advanced Computer Science |

**Words count: 962**

*Table of Contents*

[1. Introduction 3](#_Toc183550026)

[2. Architecture diagram 4](#_Toc183550027)

[3. Database Schema 5](#_Toc183550028)

[A. Citizen Microservice 5](#_Toc183550029)

[B. Provider Microservice 5](#_Toc183550030)

[C. City Microservice 5](#_Toc183550031)

[4. Class diagram 6](#_Toc183550032)

[A. API Gateway 6](#_Toc183550033)

[B. Citizen Microservice 7](#_Toc183550034)

[C. Provider Microservice 8](#_Toc183550035)

[D. City Microservice 9](#_Toc183550036)

[5. APIs documentation 10](#_Toc183550037)

[6. Critical evaluation 12](#_Toc183550038)

[References 14](#_Toc183550039)

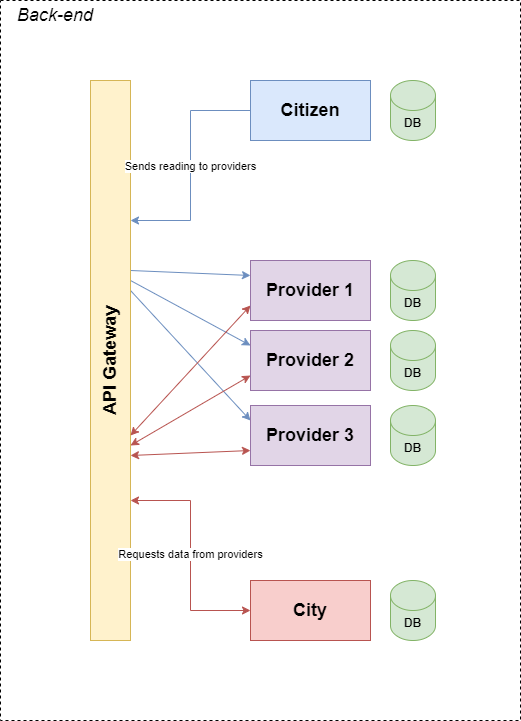
[Acknowledgements 14](#_Toc183550040)

# Introduction

This document is a report, overviewing and illustrating via diagrams, the solution implemented for the individual part for this module. The solution consists of the following:

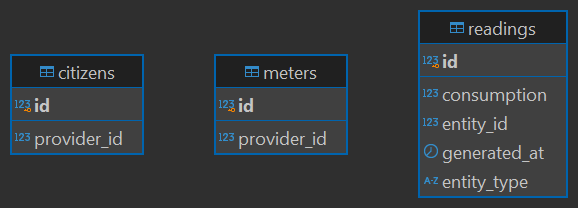
1. Citizen Microservice
   * Simulates 100 citizens and 50 smart meters generating readings in kWh and communicating them to the 3 electricity providers
2. Provider Microservice
   * Stores the readings sent by the citizens/smart meters
3. City Microservice
   * Polls the 3 different electricity providers for new data and consolidates and stores it in its database
4. API Gateway
   * Used for inter-service communication via REST
5. docker-compose.yml
   * Serves as the ‘Main Program’ i.e. launching the following components with their required dependencies e.g. databases
     1. 1 instance of Citizen
     2. 3 instances of Provider
     3. 1 instance of City
     4. 1 instance of the API Gateway

# Architecture diagram



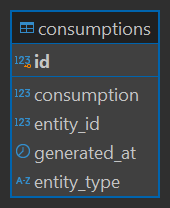
# Database Schema

## Citizen Microservice

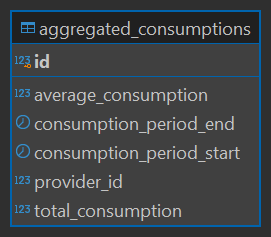


* + The provider\_id columns in the citizens and meters table hold the IDs of the providers that the citizens/meters will ping
  + The consumption column in the readings table holds the current reading of a citizen/meter in order to always generate a greater reading in the next cycle

## Provider Microservice

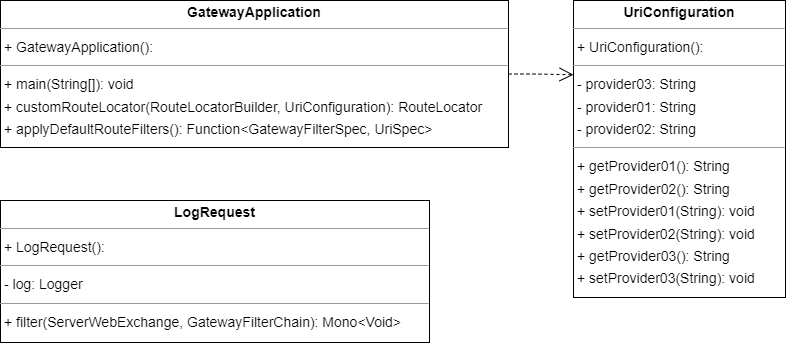


## City Microservice



# Class diagram

## API Gateway



## Citizen Microservice

## Provider Microservice

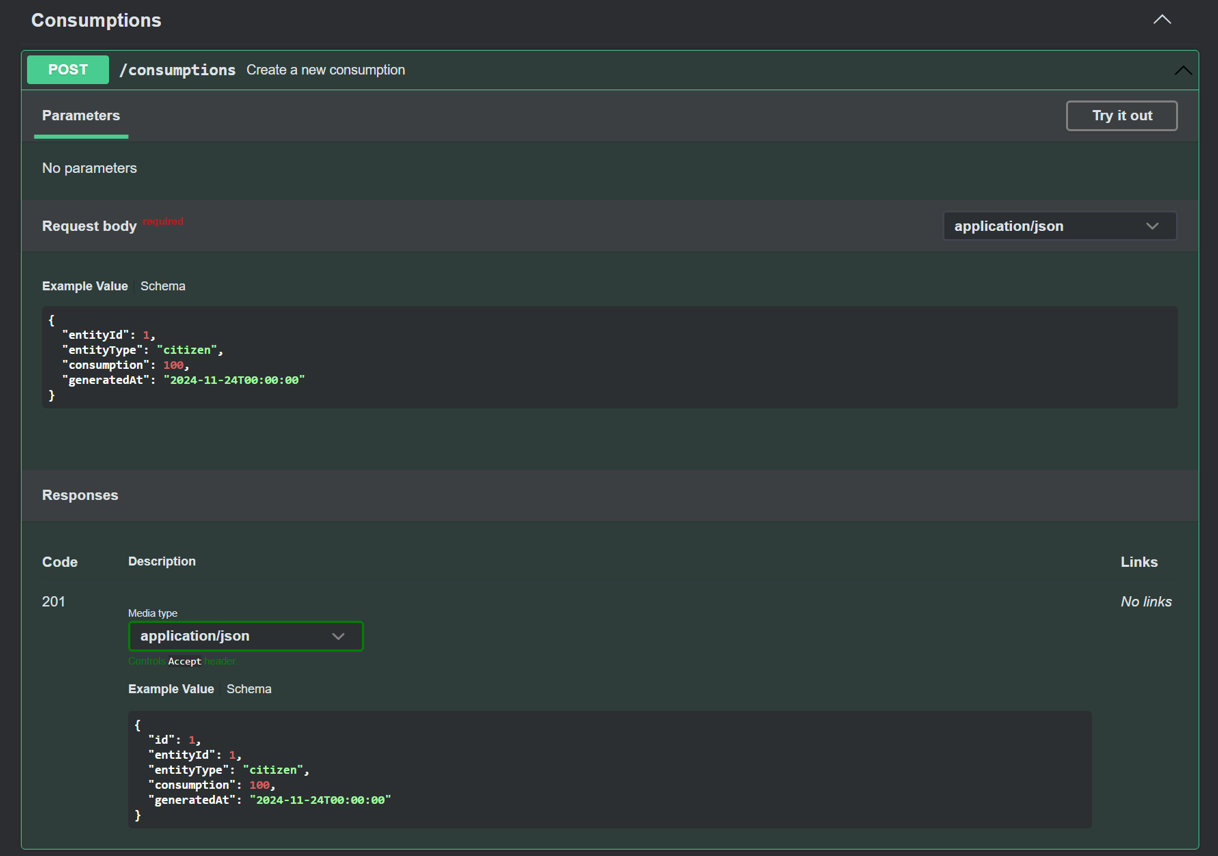
## City Microservice

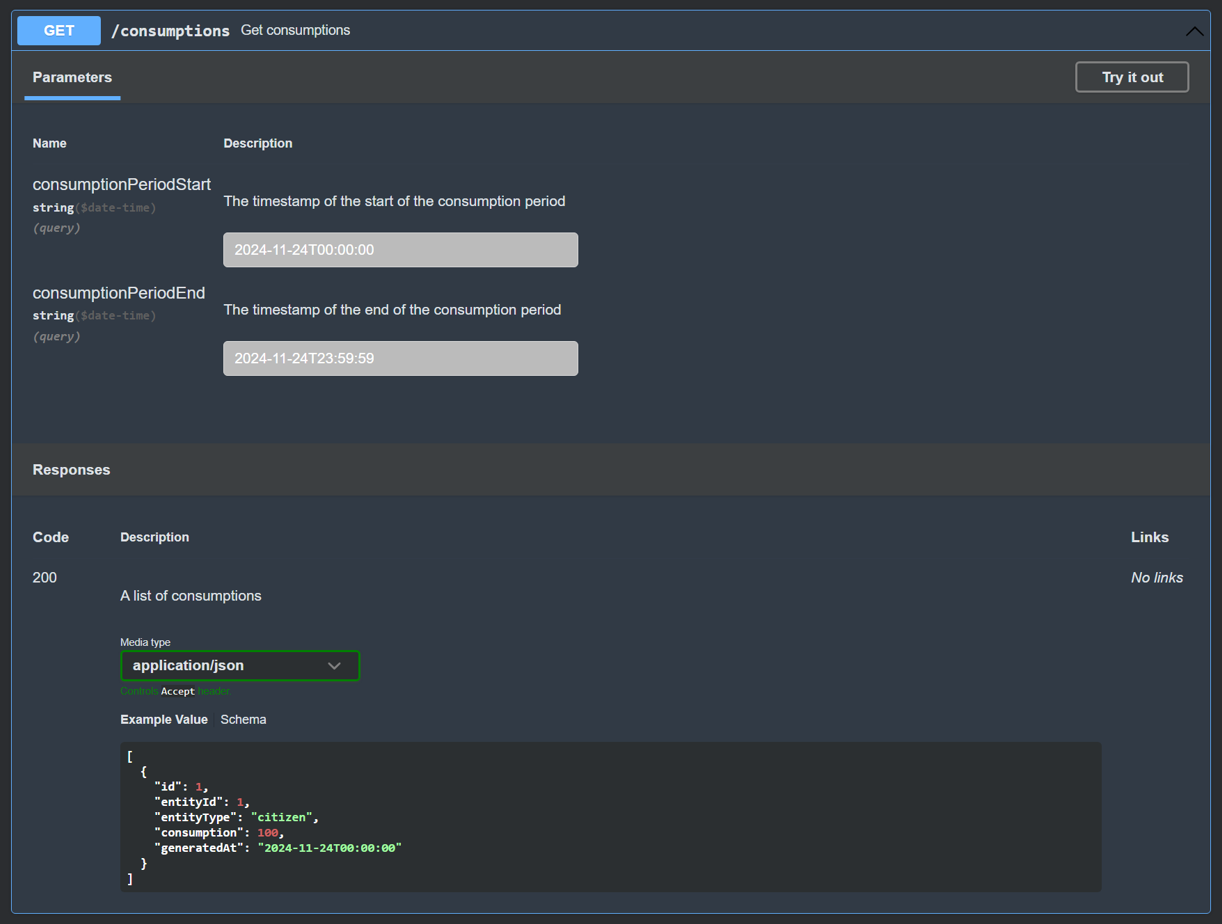
# APIs documentation

The application in its current form exposes only the following two APIs from each Provider (1, 2, 3).

A screenshot of a computer

Description automatically generated





# Critical evaluation

As an individual with previous experience in exclusively developing monolithic applications, primarily using Laravel (PHP) and a few Node.js applications, the assessment was a challenging yet valuable learning experience. With exposure to different technologies e.g. Spring Boot, Docker etc. and a complete paradigm shift to microservice architecture there is a lot to take away.

On its surface, the assessment brief seemed quite simple however, there was a lot of complexity hidden inside it. To summarize, the focus of the assessment was microservices development. Specifically, a smart city scenario where the Citizen microservice was responsible for mocking hundreds of citizens and smart meters that generated electricity consumption readings at regular intervals and reported them to the Provider microservice which stored these readings. The City microservice regularly polled the 3 different instances of the Provider and consolidated and stored the data in its own database. Meanwhile, the API gateway acted as the medium for communication between the services.

The mindset shift required to develop an application in microservices architecture was fascinating. The services had to be completely independent with each other and yet sensible communication between them had to be supported. While the independent part of it made development easier (on a service level), reducing the cognitive load, complexity and scope a service had to cover, implementing *sensible* inter-service communication was quite challenging. Hence, the need for timestamp columns in the database schemas illustrated above. The timestamps were especially helpful for communication between the City and Provider microservice, enabling interval-based retrieval of electricity consumption records.

These inter-service communication challenges reminded me of the lessons learned in the module about ‘chatty’ microservices and why they are discouraged. Take for example the seemingly simple GET /consumptions endpoint exposed by the Provider service. Considerable time and thought went behind designing it so it was useful to the City microservice without introducing any kind of loose coupling. If I had to design several more such APIs for communication between two such services, I would seriously consider consolidating them as recommended.

Some of the things I am particularly proud of are the design patterns I utilized while designing the application. I tried to use Dependency Injection wherever possible (apparent through my class diagrams) as encouraged by the Spring Boot framework. From what I have learned and researched, it is especially helpful when mocking dependencies for tests and swapping out implementations e.g. changing payment method (Card or PayPal) at runtime. Moreover, I kept things quite modular and encapsulated, making separate classes for scheduled activities, request classes and notifiers.

However, that is not to say that my solution is perfect. Especially, for the cases where communication between my microservices fails, there is no retry mechanism or something in that vein. Instead, I have simply caught such exceptions and logged them. Imagine a request from my Citizen to Provider fails. That data won’t ever reach the Provider and subsequently will not be included in the City’s database. Similarly, if a request between the City and the Provider fails, then for that requested time interval the City’s database would have no data.

All in all, this exercise gave great insight into thought and preparation that goes into microservices development plus the pros and cons of using microservices and distributed systems. Most of the cons were due to my inexperience with microservices and the fact that in real life, different teams would be working on different microservices spreading the workload and planning accordingly.

# References

* + <https://docs.spring.io/spring-boot/index.html>
  + <https://docs.docker.com/>
  + <https://learn.openapis.org/>
  + <https://app.diagrams.net/>

# Acknowledgements

* Took help from chatgpt.com for the main docker-composer.yml which orchestrates the whole application with the required dependencies and instances of the microservices
* Took help from stackoverflow.com for spring-boot specific concepts e.g. the implementation of ConsumptionFilter in the provider microservice
* Took help from chatgpt.com + stackoverflow to learn how to make asynchronous requests in Spring Boot.