Degree Thesis, Åland University of Applied Sciences, Degree Programme in Information Technology

A Comparative Analysis of Spring MVC and Spring WebFlux in Modern Web Development

Alexandru-Valentin Catrina

2023:35

Date of approval: 24.11.2023
Academic Supervisor: Joakim Isaksson

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Alexandru-Valentin Catrina
Arbetets namn:	En jämförande analys av Spring MVC och Spring WebFlux i modern webbutveckling
Handledare:	Joakim Isaksson
Uppdragsgivare:	Crosskey Banking Solutions

Abstrakt

Detta examensarbete utför en jämförande analys av två populära Spring-baserade ramverk, Spring Boot MVC och Spring WebFlux, med fokus på deras prestanda och resursanvändning.

Den centrala forskningsfrågan som utforskas i denna avhandling är: “Hur förhåller sig Spring WebFlux till traditionell synkron webbutveckling när det gäller prestanda och skalbarhet?“

För att etablera en ram för utvärdering bygger studien på teorier relaterade till webbutveckling, asynkron programmering och prestandaanalys.

Undersökningen utgår från en systematisk jämförande ansats, som innefattar skapandet av representativa webbapplikationer både i Spring Boot MVC och Spring WebFlux.

Prestandamått, resursanvändning och svarstider mäts under varierande arbetsbelastningar för att bedöma ramverkens kapacitet.

Resultatet visar att Spring WebFlux har lägre resursanvändning jämfört med Spring Boot MVC, samtidigt som de uppvisar nästan samma svarstider.

Nyckelord (sökord)

WebFlux, Reactive, SpringBoot

Högskolans serienummer:	ISSN:	Språk:	Sidantal:
2023:35	1458-1531	Engelska	47 sidor

Inlämningsdatum:	Presentationsdatum:	Datum för godkännande:
15.09.2023	29.09.2023	24.11.2023

DEGREE THESIS
Åland University of Applied Sciences

Degree Programme:	Information Technology
Author:	Alexandru-Valentin Catrina
Title:	A Comparative Analysis of Spring MVC and Spring WebFlux in Modern Web Development
Academic Supervisor:	Joakim Isaksson
Commissioned by:	Crosskey Banking Solutions

Abstract

This thesis conducts a comparative analysis of two widely used Spring-based frameworks, Spring Boot MVC and Spring WebFlux, with a focus on their performance and resource utilization.

The central research question explored in this thesis is : “How does Spring WebFlux compare to traditional synchronous web development in terms of performance and scalability?”

The study leverages theories related to web application development, asynchronous programming, and performance analysis to establish a framework for the evaluation.

The research employs a systematic benchmarking approach, involving the creation of representative web applications in both Spring Boot MVC and Spring WebFlux. Performance metrics, resource usage, and response times are measured under varying workloads to assess the frameworks’ capabilities.

The results reveal that Spring WebFlux exhibits less resource utilization compared to Spring Boot MVC, while maintaining nearly the same response times.

In conclusion, the study highlights the strengths and weaknesses in Spring Boot MVC and Spring WebFlux.

Keywords

WebFlux, Reactive, SpringBoot

Serial number:	ISSN:	Language:	Number of pages:
2023:35	1458-1531	English	47 pages

Handed in:	Date of presentation:	Approved:
15.09.2023	29.09.2023	24.11.2023

TABLE OF CONTENTS

1 INTRODUCTION 5
1.1 Purpose 5
1.2 Methodology 5
1.3 Limitations 6
2 BACKGROUND 7
2.1 REST APIs 7
2.2 MVC (Model View Controller) 7
2.3 Traditional Synchronous Web Development 8
2.4 Reactive Programming and Asynchronous Web Development 9
2.5 Spring Framework 10
3 THEORETICAL FOUNDATIONS 12
3.1 Spring WebFlux and its components 12
3.2 Reactive Streams Specification 12
3.3 Project Reactor 13
3.4 Spring WebFlux Architecture 14
3.5 Java Persistence API (JPA) and Reactive Data Access 16
4 DESIGN AND IMPLEMENTATIONS 17
4.1 Application requirements 17
4.2 System Architecture 17
4.3 Implementation of Spring WebFlux REST API 19
4.4 Integration with R2DBC 21
4.5 Implementation of Spring MVC REST API 23
5 PERFORMANCE COMPARISON 26
5.1 Benchmarking methodology 27
5.2 Performance results and testing scenarios 28
5.3 Analysis of results 30
5.3.1 /add endpoint performance in the Spring WebFlux application 31
5.3.2 /add endpoint performance in the Spring MVC application 34
5.3.3 /hello endpoint performance in the Spring WebFlux application 37
5.3.4 /hello endpoint performance in the Spring MVC application 40
5.3.5 Summary of the result 43
5.4 Analysis of the result 43
6 CONCLUSION 45
REFERENCES 47

1 INTRODUCTION

1.1 Purpose

The rapid growth of internet users and connected devices has increased the demand for highly responsive and scalable web applications. Synchronous web development struggles to handle the high concurrency and throughput, leading to the exploration of new non-blocking and asynchronous techniques.

In the context of my developer role at Crosskey[1], where we utilize Spring WebFlux, this thesis will seek to answer the following research questions:

- How does Spring WebFlux compare to traditional synchronous web development in terms of performance and scalability?
- What are the key considerations for successfully implementing Spring WebFlux?

1.2 Methodology

To provide a good understanding of Spring WebFlux and its role in creating non-blocking REST APIs, the research will be carried out as following:

- A detailed review of the fundamentals of reactive programming, including its advantages and challenges, as well as a comparison with traditional synchronous programming.
- An in-depth exploration of the Spring WebFlux framework, including its architecture, core components, and the essential features that facilitate non-blocking REST API development.
- Design and implementation of non-blocking REST API using Spring WebFlux, showcasing its capabilities in creating highly concurrent and scalable applications.
- Performance evaluation of the implemented API, including benchmark against a comparable synchronous API, to demonstrate the differences in the adoption of a non-blocking and reactive approach.

[1] https://www.crosskey.fi/

1.3 Limitations

One of the limitations of this study is its primary focus on performance metrics related to time, such as response time and resource utilization. While this approach provides valuable insights into the efficiency and scalability of Spring Boot MVC and Spring WebFlux, it should be noted that the analysis does not encompass a comprehensive comparison between these frameworks in the context of real-world enterprise applications.

Another limitation worth mentioning is the relatively limited number of test scenarios employed in this study. The number of scenarios, while chosen to represent a range of common usage patterns, does not cover the full spectrum of real-world application behaviors and workloads. Enterprise applications often exhibit diverse usage patterns, and their performance characteristics can vary significantly under different conditions. As such, the findings presented in this study reflect the outcome observant within the scope of the selected test scenarios.

These limitations underscore the need for caution when generalizing the result to all possible enterprise application scenarios. Future research endeavors could expand the scope of analysis to include a more extensive set of test scenarios, thereby providing a more comprehensive understanding of the frameworks' performance under diverse conditions and usage patterns in the context of enterprise-scale software development.

2 BACKGROUND

2.1 REST APIs

A REST (Representational State Transfer) API is an architecture for designing network applications. REST APIs leverage the simplicity of the standard HTTP protocol to enable communication between various software components over the internet.

In a REST API, resources are the key components that the clients interact with. These resources can be any data object, and each resource is identified by a unique resource identifier (URI), which serves as an address for locating and accessing it.
Standard HTTP methods like GET, POST, PUT and DELETE are used to define the actions that can be performed on the resources.
The REST API offers several advantages, including:

1. Scalability: The stateless nature of REST APIs and their use of caching mechanisms allow them to handle a large number of requests and scale easily.
2. Flexibility: RESTful web services support total client-server separation where each part can evolve independently.
3. Independence: REST APIs are independent of the technology used.

(*What Is A RESTful API?*, n.d.)

2.2 MVC (Model View Controller)

The Model-View-Controller (MVC) is an architectural pattern that separates the application into three main components:

1. The Model is responsible for managing the application’s data and business logic. Data storage, retrieval and manipulation are done here as well.
2. The View is responsible for displaying the data to the user. It serves as the application’s user interface and is in charge of presenting the data in a user-friendly manner. The data that the View is presenting is received from the Model.
3. The Controller acts as an intermediary between the Model and the View. It processes user input, such as requests or actions, and communicates with the Model to retrieve

or update data. The Controller then updates the View to reflect any changes in the date. This component is responsible for managing the overall flow of the application and coordinating the Model and View. (*MVC Framework - Introduction*, n.d.)

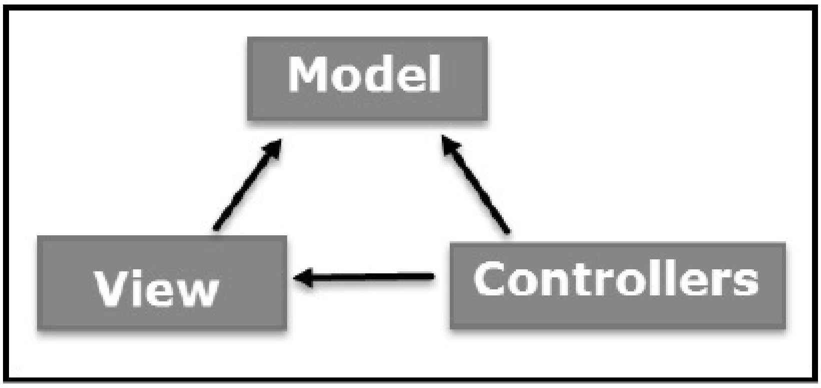


Figure 1. Model-View-Controller architecture (MVC Framework - Introduction, n.d.)

2.3 Traditional Synchronous Web Development

In synchronous web development, the client sends a request to the server and waits for the server to respond. During this time, the client remains idle and cannot process other requests or perform other tasks. This leads to a blocking behavior, where the user waits for the response before interacting with the website further.

Synchronous programming offers several advantages. It ensures predictable results as programs run in a linear fashion, simplifying debugging since there is typically only one code flow path to follow. Additionally, synchronous programs tend to consume less memory because they execute one task at a time, reducing the need to store the state of numerous concurrent tasks. This approach also fosters ease of comprehension, as tasks must be completed in a specific order, making the code more straightforward to follow. Moreover, when handling a limited number of tasks sequentially, synchronous programming can be more efficient, as it reduces the likelihood of encountering unexpected issues or errors. Lastly, synchronous programs have lower overhead as they do not involve the management of multiple tasks concurrently (Muscad, 2022).

Conversely, synchronous programming does come with its drawbacks. One notable limitation is slower execution, as tasks must wait for one another to finish before proceeding. This can lead to suboptimal performance in scenarios where parallelism would be more beneficial. Additionally, the inherent need to complete tasks in a successive order can make it challenging to scale the program, particularly when new tasks need to be added. This limits the program's ability to efficiently leverage multi-core processors and parallel processing. Hence, synchronous programming may not be the best choice when dealing with computationally intensive or highly concurrent tasks (Muscad, 2022).

2.4 Reactive Programming and Asynchronous Web Development

Reactive Programming and Asynchronous Web Development are modern approaches to building efficient, responsive, and scalable web applications. These techniques focus on the limitations of traditional synchronous web development by allowing a simultaneous processing of multiple tasks and delivering a seamless user experience.

Reactive programming can be employed in asynchronous web development to enhance the responsiveness and interactivity of web applications. By reacting to data changes and events, web applications can update their interface without the need for constant server communication or page refreshes. This leads to a more efficient user experience, particularly for complex web applications that handle real-time data, such as chat applications, data visualization tools or financial platforms.

Reactive programming and synchronous web development offer several significant advantages. Firstly, they contribute to improved performance by enabling simultaneous processing of tasks. This capability is valuable when handling complex, data-intensive operations. Additionally, these approaches enhance scalability, allowing applications to efficiently manage large volumes of data and user interactions. This scalability not only ensures smoother performance but also accommodates growing user demands. Furthermore, reactive programming and asynchronous development empower applications to be highly responsive, adapting fast to changes or user inputs in real-time. This real-time responsiveness contributes to a more interactive and engaging user experience. Lastly, these methodologies promote maintainability through their declarative nature, creating a separation of concerns within the codebase. This separation simplifies the process of maintaining and updating the code, reducing the likelihood of errors (Nolle, 2021).

However, alongside these benefits, there exist notable challenges. Integrating reactive systems into existing software can be a complex task, potentially requiring significant modifications or, in some cases, proving impossible. Additionally, embracing this approach often necessitates a paradigm shift for developers accustomed to traditional programming paradigms. Learning and adapting to the principles of reactive and asynchronous programming may demand time and effort. Finally, there is the risk of accumulating delays in reactive systems when an excessive number of processes are linked in the stream. Managing and optimizing this complexity is essential to ensure the continued efficiency and responsiveness of the application. Therefore, while these methodologies offer substantial advantages, they also require careful consideration and adaptation to address these challenges effectively (Nolle, 2021).

2.5 Spring Framework

The Spring Framework is a widely-used open source Java-based framework for developing enterprise applications and simplifying Java development. It was introduced in 2003 by Rod Johnson and it aims to address the complexity of traditional Java development by providing a lightweight, modular and extensible solution (*Introduction to Spring Framework*, n.d.).

The Spring Framework is built on several core principles including Dependency Injection (DI), Aspect-Oriented Programming (AOP), and Convention over Configuration. These principles promote the creation of flexible, maintainable and testable applications by decoupling components, simplifying configurations and reducing boilerplate code.

The framework is organized in about 20 modules. These modules are then grouped into Test, Core Container, AOP, Aspects, Instrumentation, Web and Data Access/Integration as shown in Figure 2.

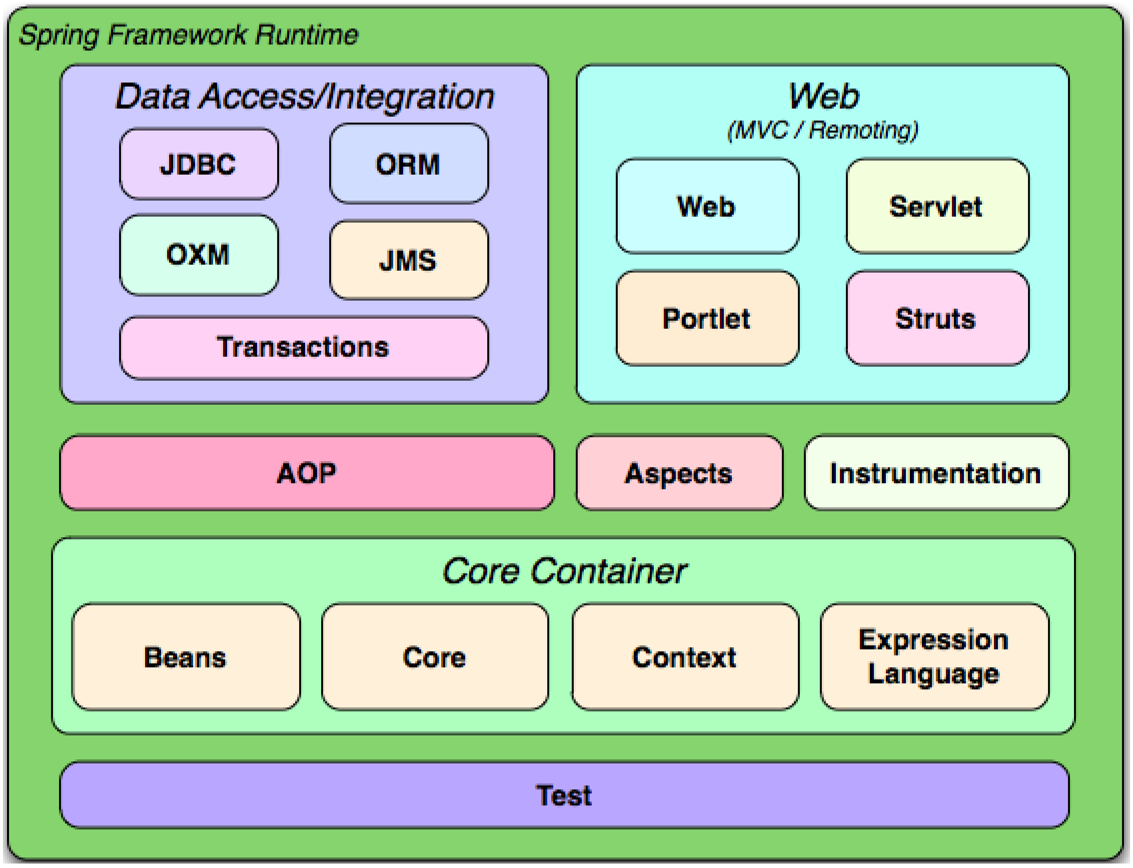


Figure 2. Overview of Spring Framework (Introduction to Spring Framework, n.d.)

The Core Container provides the fundamental parts of the framework. Here we have the “Bean Factory” that removes the need for programmatic singletons and allows for decoupling the configuration and dependencies from the actual program logic.

Data Access/Integration provides the support to interact with the database.

The Web module provides basic web-oriented integration features.

AOP and Instrumentation work together to provide a way to modularize cross-cutting concerns and manage application resources.

The Test module supports testing of Spring components, provides consistent loading of Spring ApplicationContext and provides the ability to mock objects.

(*Introduction to Spring Framework*, n.d.)

3 THEORETICAL FOUNDATIONS

3.1 Spring WebFlux and its components

Spring Web Flux is a part of the Spring Framework designed for building reactive and non-blocking web applications. It was introduced as an alternative to the traditional Spring MVC where Spring WebFlux leverages reactive programming principles and supports a fully asynchronous, event-driven programming model in order to create highly scalable and responsive applications.

Spring WebFlux is built on the Reactive Streams Specifications and utilizes the Project Reactor library. It is presented as a solution for creating web applications that efficiently handle a large number of concurrent connections, making it suitable for high-performance and real-time applications (*Spring Framework Documentation*, n.d.).

3.2 Reactive Streams Specification

The Reactive Stream Specification (*Reactive Streams,* n.d.) was created to provide a standardized approach for building asynchronous, non-blocking systems that can efficiently handle large data streams while maintaining backpressure (flowcontrol) to prevent resource exhaustion.

> Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols. (*Reactive Streams*, n.d.)

In the Reactive Stream Specification, four main components are defined that work together to enable asynchronous stream processing:

- Publisher: The publisher is responsible for producing data and emitting it to one or more subscribers.
- Subscriber: The subscriber consumes the data produced by the publisher.
- Subscription: The subscription represents the link between the publisher and the subscriber.
- Processor: The processor acts as both a publisher and a subscriber. It can be used to implement various data processing patterns, such as filtering and mapping.

The subscriber has the ability to request a specific amount of data from the publisher and the publisher must respect the backpressure signals from its subscribers. This enables dynamic backpressure management.

This initiative is closely related to the Reactive Stream Manifesto, that is a set of guiding principles and characteristics for building reactive systems (Bonér et al., 2014). It presents the four main traits of reactive systems:

- Responsive: Reactive systems should respond in a timely manner and maintain consistent performance under various loads.
- Resilient: systems should be designed to be able to handle failures and continue to function despite errors or other failures.
- Elastic: Reactive systems should be able to scale horizontally, efficiently managing resources.
- Message Driven: Reactive systems should rely on asynchronous message-passing to establish boundaries between components, enabling coupling, isolation, and location transparency.

The Reactive Streams Specification promotes interoperability between different libraries and frameworks, enabling seamless integration of various reactive libraries, such as Project Reactor.

3.3 Project Reactor

Developed by Pivotal Software, Project Reactor is built on top of the Reactive Streams Specification and is a pivotal reactive programming library for building non-blocking and asynchronous applications. As described by Maldini and Baslé (2023), it is a foundational component of the Spring WebFlux framework and provides a powerful and flexible toolset for building responsive, resilient and scalable systems.

Project Reactor introduces two primary reactive types: Flux and Mono. These types represent asynchronous sequences of data and serve as building blocks for creating reactive data streams.

- Flux is a reactive type that can emit zero or many items. It represents a potentially infinite stream of data and supports various operations such as filtering, mapping and merging.
- Mono is a reactive type that can emit zero or one item. It represents a single value or the absence of it and can be used for asynchronous computation or signaling the completion of a task.

The set of operators offered in Project Reactor enables us to perform various operations on reactive date streams, such as data transformation, filtering, error handling, and combining multiple streams. These operators empower developers to build complex data processing pipelines with ease.

Schedulers in Project Reactor are responsible for managing the execution of reactive operations. They give the capability to control the concurrency of their applications by offloading work to different threads or specifying the parallelism level. By efficiently managing concurrency, it ensures responsive and performant applications, even under heavy loads.

Backpressure is a crucial feature in reactive systems, allowing subscribers to control the rate at which they consume data from publishers. Project Reactor supports backpressure, ensuring that slower subscribers do not get overwhelmed by faster publishers. This contributes to the stability and scalability of reactive applications.

3.4 Spring WebFlux Architecture

Spring WebFlux was introduced in Spring 5 as an alternative to the traditional Spring MVC. Its architecture is built around the Reactive Streams Specification. Figure 3 shows a comparison of Spring MVC and WebFlux.

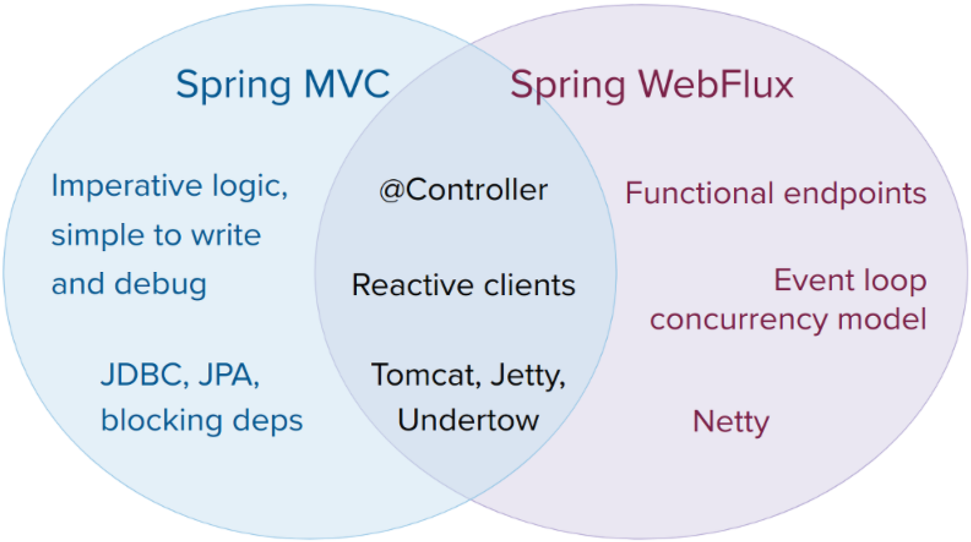


Figure 3. Spring MVC and WebFlux Spring WebFlux Overview (2023)

Spring WebFlux offers support for two programming models. Annotated Controllers are similar to Spring MVC but use reactive types for data processing like Flux and Mono. Functional Endpoints is a lightweight functional programming model in which functions are used to route and handle requests and contracts are designed for immutability (*Spring WebFlux Overview,* 2023).

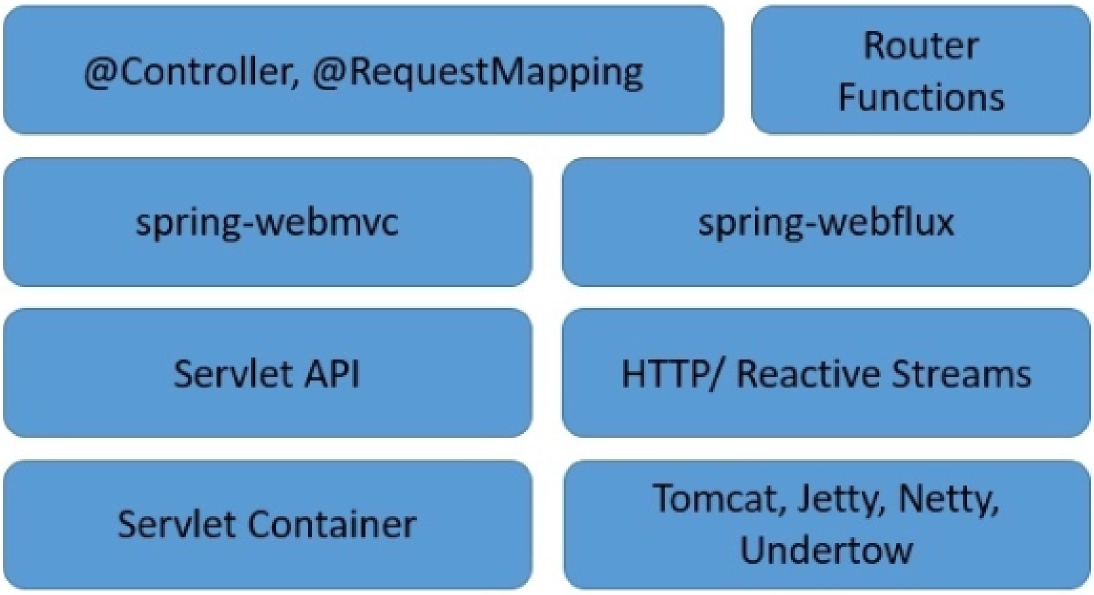


Figure 4. Spring WebFlux architecture (Chandrakant, 2020)

Figure 4 shows the Spring WebFlux architecture. We can see that Spring WebFlux sits parallel to the traditional Spring MVC framework and does not necessarily replace it.

WebFlux supports a wide variety of runtimes, Netty being the standard one. It also includes WebClient, which is a reactive non-blocking client for HTTP requests. (Chandrakant, 2020)

3.5 Java Persistence API (JPA) and Reactive Data Access

When it comes to data persistence in Java applications, JPA and Reactive Data Access represent two different approaches. JPA focuses on traditional data access for relational databases, while Reactive Data Access focuses on asynchronous data access in reactive applications.

Many JPA implementations often use JDBC under the hood, which is a lower-level API used to interact with relational databases. A widely used Reactive Data Access is R2DB, which stands for Reactive Relational Database Connectivity. Figure 5 shows the Spring Data JDBC and the potential reactive replacement (Dokuka & Lozynskyi, 2018).

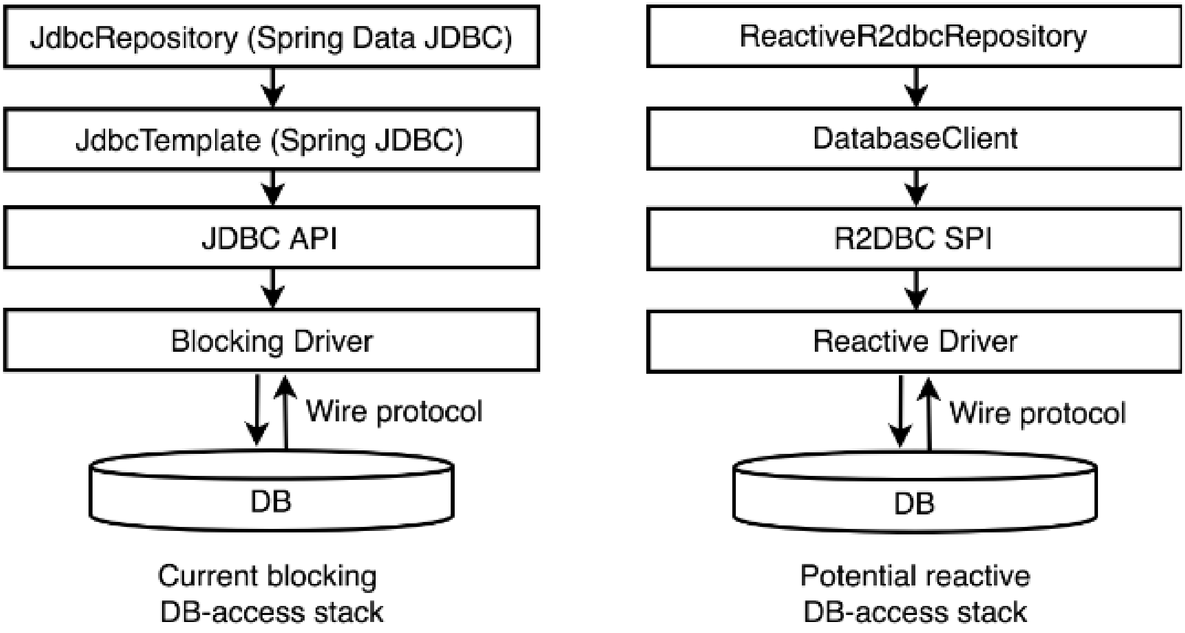

JdbcRepository (Spring Data JDBC)
JdbcTemplate (Spring JDBC)
JDBC API
Blocking Driver
Wire protocol
DB
Current blocking DB-access stack
ReactiveR2dbcRepository
DatabaseClient
R2DBC SPI
Reactive Driver
Wire protocol
DB
Potential reactive DB-access stack

Figure 5. JDBC and R2DBC (Dokuka & Lozynskyi, 2018)

4 DESIGN AND IMPLEMENTATIONS

4.1 Application requirements

The focus of the application for this thesis is to develop a reactive REST API for items stored in a database. The application is expected to support the operation of adding records. To accomplish this, the following requirements have been established:

- A fully reactive REST API constructed using Spring WebFlux framework. This will ensure efficient, non-blocking communication and allow the system to handle a high volume of concurrent requests effectively.
- Utilization of Netty as the underlying web server to enhance the performance of the application. Netty will be configured as the default web server for the Spring WebFlux application.
- Adoption of reactive data access techniques utilizing the R2DBC driver, facilitating non-blocking database interactions. The application will be connected to an in-memory H2 database, streamlining development and testing processes.

In addition to the reactive REST API, this thesis will also examine the performance difference between the traditional Spring Boot application and the reactive web application. The traditional Spring Boot application will implement the corresponding functionality using a blocking and synchronous approach.

The comparison will involve measuring various performance metrics, such as response times, throughput, and resource utilization, under varying workloads and concurrency levels.

4.2 System Architecture

The application's architecture, incorporating Netty as the underlying web server is shown in figure 6.

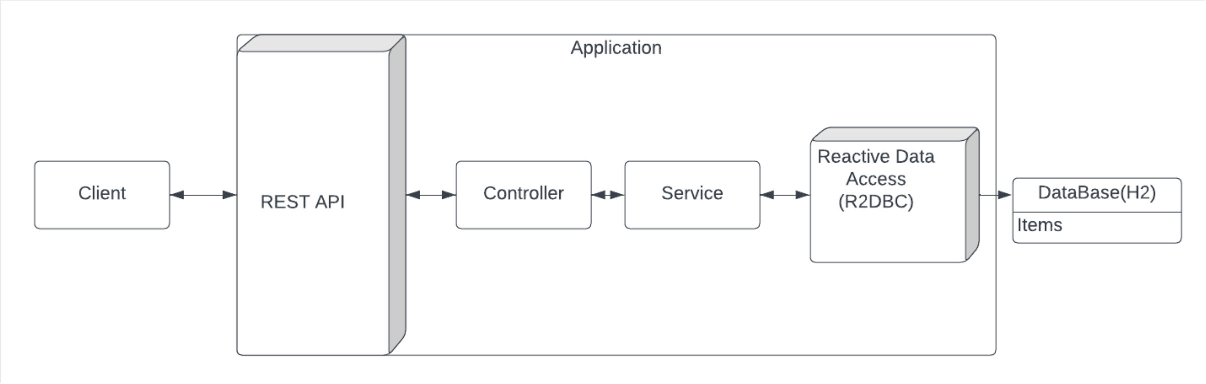


Figure 6. Application's architecture

1. Client Layer:

   The client will send HTTP requests to the API and receive responses in a standardized format. It can be seen as other services that interact with the REST API.

2. REST API Layer:

   The REST API layer will expose endpoints for creating item records.

3. Controller Layer:

   The controller layer consists of the class responsible for mapping incoming HTTP requests to appropriate methods in the service layer. It will define the REST API endpoints, validate the incoming request data and handle any API-specific exceptions.

4. Service Layer (Business Logic):

   This layer will consist of services responsible for processing and validating incoming requests from the controller layer. These services will perform necessary business logic operations and coordinate the data access layer to interact with the database.

5. Data Access Layer (R2DBC):

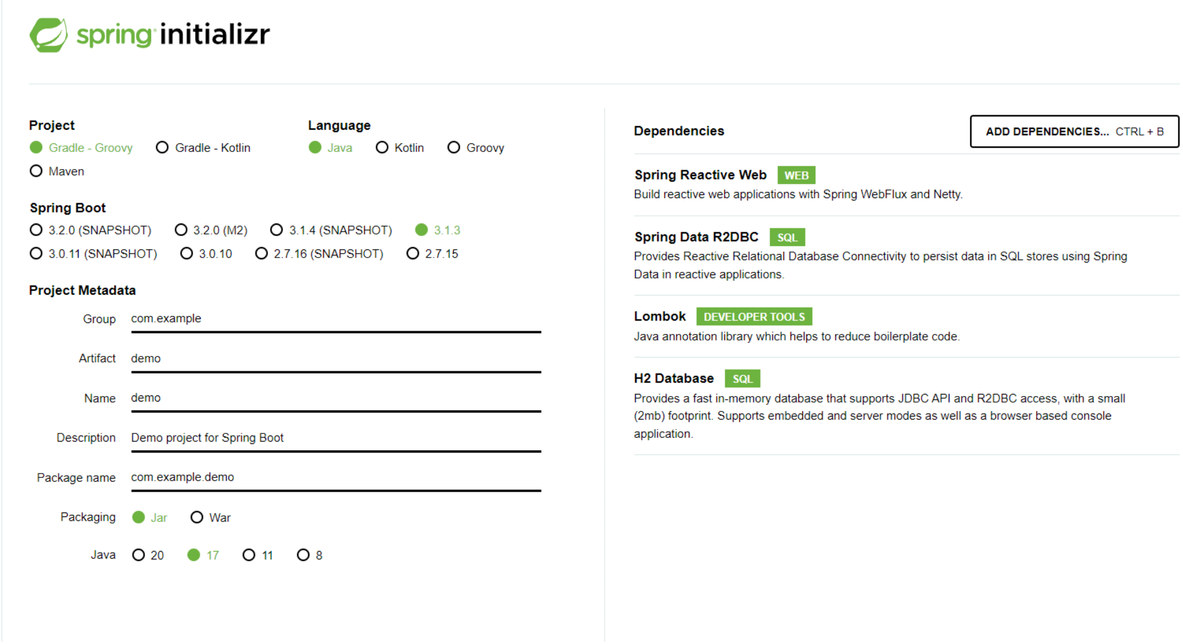
   The data access layer will provide an abstraction for interacting with the in-memory H2 database in a non-blocking manner. It will be responsible for executing CRUD operations on persons records and their information, as well as handling any database-related transactions.

6. Database (H2):

   The application will utilize an H2 database for storing persons records and their information. The database will offer a lightweight and flexible solution for development and testing purposes.

4.3 Implementation of Spring WebFlux REST API

To set up the project, the Spring Initializr[2] has been used. Using the default setting, as shown in Figure 7, with Spring framework 3.0.6 and Java 17, we need to add the following dependencies for R2DBC, WebFlux, Lombok, and H2 Database:



spring initializr

Project
● Gradle - Groovy ○ Gradle - Kotlin
○ Maven

Language
● Java ○ Kotlin ○ Groovy

Spring Boot
○ 3.2.0 (SNAPSHOT) ○ 3.2.0 (M2) ○ 3.1.4 (SNAPSHOT) ● 3.1.3
○ 3.0.11 (SNAPSHOT) ○ 3.0.10 ○ 2.7.16 (SNAPSHOT) ○ 2.7.15

Project Metadata

Group: com.example
Artifact: demo
Name: demo
Description: Demo project for Spring Boot
Package name: com.example.demo
Packaging: ● Jar ○ War
Java: ○ 20 ● 17 ○ 11 ○ 8

Dependencies ADD DEPENDENCIES... CTRL + B

Spring Reactive Web WEB
Build reactive web applications with Spring WebFlux and Netty.

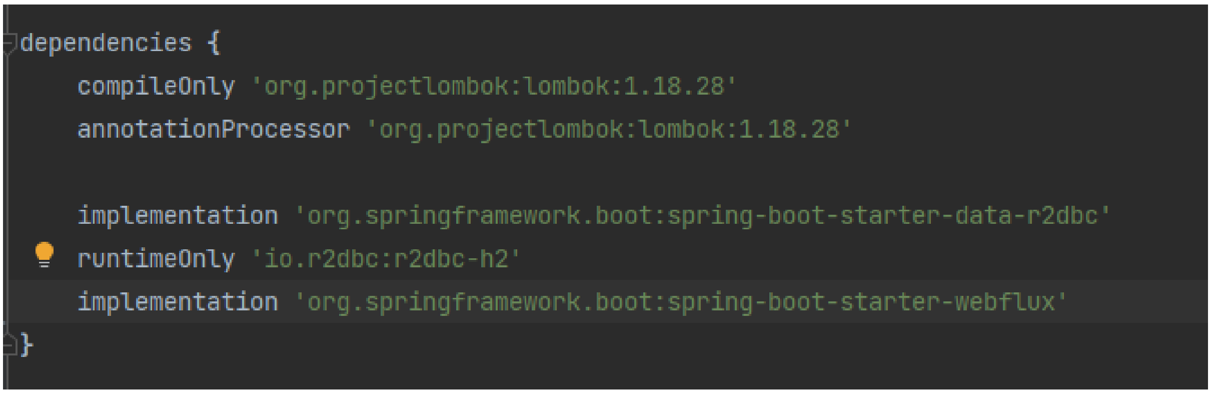
Spring Data R2DBC SQL
Provides Reactive Relational Database Connectivity to persist data in SQL stores using Spring Data in reactive applications.

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

H2 Database SQL
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Figure 7. Spring Initializr (Spring Initializr, n.d.)

As illustrated in Figure 8, the Gradle dependencies displayed were generated via the Spring Initializr platform.



```
dependencies {
    compileOnly 'org.projectlombok:lombok:1.18.28'
    annotationProcessor 'org.projectlombok:lombok:1.18.28'

    implementation 'org.springframework.boot:spring-boot-starter-data-r2dbc'
    runtimeOnly 'io.r2dbc:r2dbc-h2'
    implementation 'org.springframework.boot:spring-boot-starter-webflux'
}
```

Figure 8. Gradle dependencies

Lombok is a Java library (*lombok:1.18.28*) that provides annotations to reduce boilerplate code in Java applications, such as setter and constructor generation.

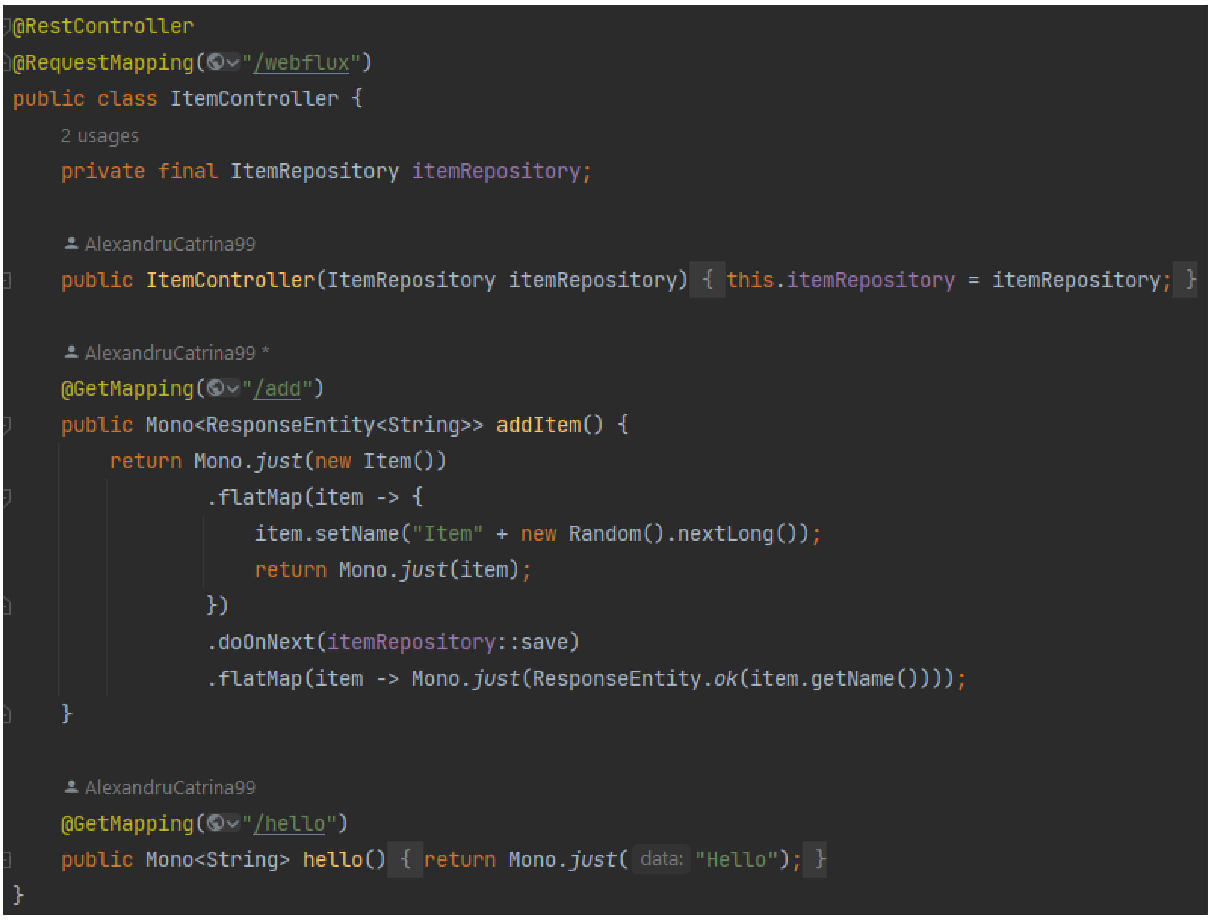
spring-boot-starter-data-r2dbc is a starter pack for using Spring Data with R2DBC (Reactive

[2] https://start.spring.io/

Relational Database Connectivity). R2DBC is an initiative to provide a reactive programming API for relational databases.

r2dbc-h2 is a reactive driver for the H2 (in memory) database, which allows for the use of H2 in a non-blocking, reactive manner.

spring-boot-starter-webflux provides the tools to build asynchronous and non-blocking web applications.



```java
@RestController
@RequestMapping("/webflux")
public class ItemController {
    2 usages
    private final ItemRepository itemRepository;

    AlexandruCatrina99
    public ItemController(ItemRepository itemRepository) { this.itemRepository = itemRepository; }

    AlexandruCatrina99 *
    @GetMapping("/add")
    public Mono<ResponseEntity<String>> addItem() {
        return Mono.just(new Item())
                .flatMap(item -> {
                    item.setName("Item" + new Random().nextLong());
                    return Mono.just(item);
                })
                .doOnNext(itemRepository::save)
                .flatMap(item -> Mono.just(ResponseEntity.ok(item.getName())));
    }

    AlexandruCatrina99
    @GetMapping("/hello")
    public Mono<String> hello() { return Mono.just( data: "Hello"); }
}
```

Figure 9. ItemController java class

In Figure 9, we observe the definition of a class named *ItemController*. This class is designed as Spring's RESTful web service controller, as indicated by the *@RestController* annotation. Further, a base request mapping of */webflux* is provided by the *@RequestMapping* annotation, indicating that all encompassed endpoints will be prefixed with this path. The controller is primarily responsible for managing operations related to an *item* and is dependent on *ItemRepository* for data access. The constructor is utilized for dependency injection, assigning the injected *ItemRepository* bean to this field.

Both endpoints, */add* and */hello*, are defined as HTTP GET methods by the *@GetMapping* annotation.

/add - endpoint:

The method *addItem* invokes the *save* method of the *ItemRepository*, resulting in saving a new item in the database.

The return value is a reactive stream *Mono<ResponseEntity<String>>* that returns a *Mono* that wraps a response entity with an *ok*(200) status containing the name of the newly saved item.

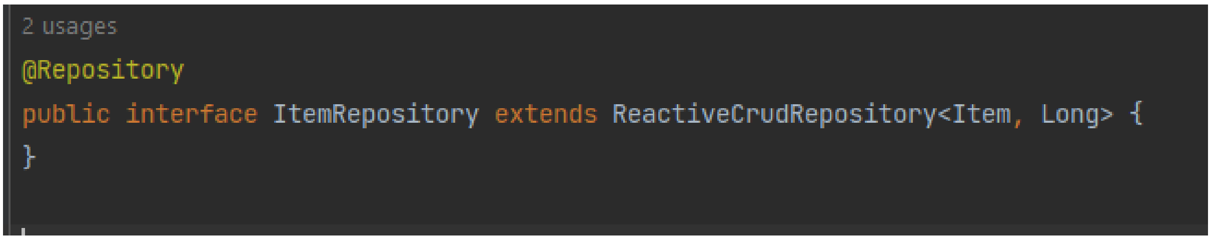
/hello - endpoint:

This endpoint produces a single asynchronous value of type *Mono<String>* bearing the string *"Hello"*.

4.4 Integration with R2DBC

The decision to use R2DBC often goes hand in hand with the adoption of the Spring WebFlux to create a fully reactive application.

The code in Figure 10 shows the integration with R2DBC by extending Spring Data's *ReactiveCrudRepository*. This interface is specially designed to work seamlessly with R2DBC, thereby giving the ability to handle database operations reactively. By extending it any CRUD (Create, Read, Update, Delete) operations performed through this repository will be non-blocking, providing better scalability, especially under high loads.



```
2 usages
@Repository
public interface ItemRepository extends ReactiveCrudRepository<Item, Long> {
}
```

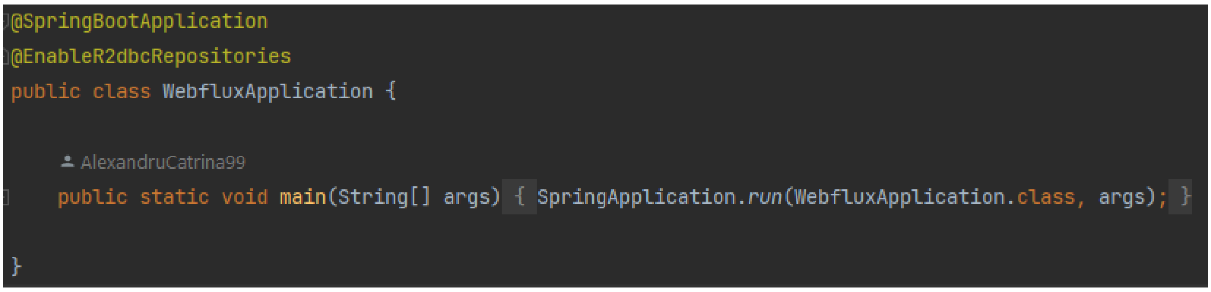
Figure 10 Repository interface

The generic types *<Item, Long>* specify the type of object the repository will manage (*Item*) and the type of the object's unique identifier (*Long*).

By virtue of extending *ReactiveCrudRepository*, *itemRepository* inherits several methods for data access without requiring explicit implementations such as *save()*.

The *@Repository* annotation is a Spring-specific stereotype annotation that indicates that the class defined is intended to be a Repository.

Figure 11 shows the *WebfluxApplication* class with the *@EnableR2dbcrepositories* annotation that is pivotal in configuring the Spring Boot application to use R2DBC repositories. By including this annotation, the application enables the scanning for interfaces extending reactive repository interfaces like *ReactiveCrudRepository*, such as *ItemRepository* detailed in Figure 10. Consequently, this setting facilitates the integration of the reactive database operations defined in their repository interfaces.



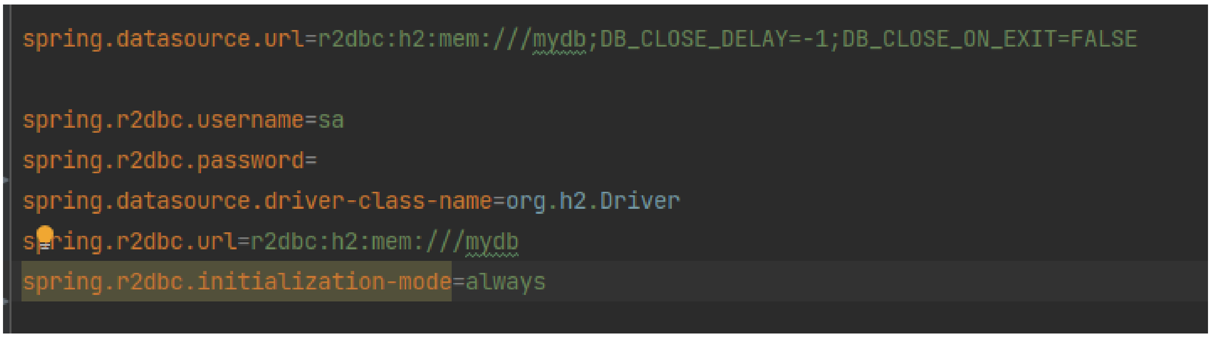
```java
@SpringBootApplication
@EnableR2dbcRepositories
public class WebfluxApplication {

    AlexandruCatrina99
    public static void main(String[] args) { SpringApplication.run(WebfluxApplication.class, args); }

}
```

Figure 11. The WebFluxApplication and its main method.

The *@SpringBootApplication* annotation is a convenience annotation that streamlines the setup of Spring-based applications. When applied to the main application class, it automatically performs tasks like component scanning, auto-configuration, and acts as the entry point for the application. This simplifies development by reducing the need for explicit configuration and boilerplate code, making it easier to build Spring applications.

And the *main()* method is the entry point for the Spring Boot application. It calls the *SpringApplication.run()* method to bootstrap the application. The application context is initialized, and all the enabled features like R2DBC repositories are set up.

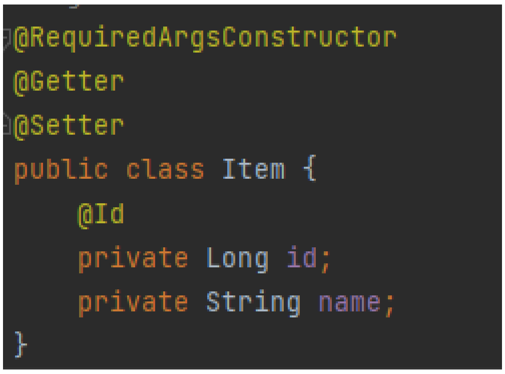
As for the R2DBC configurations, they are set in a separate *application.properties* file.



```properties
spring.datasource.url=r2dbc:h2:mem:///mydb;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE

spring.r2dbc.username=sa
spring.r2dbc.password=
spring.datasource.driver-class-name=org.h2.Driver
spring.r2dbc.url=r2dbc:h2:mem:///mydb
spring.r2dbc.initialization-mode=always
```

Figure 12. Application property file

In Figure 12 above, the standard configurations for the R2DBC H2 - database such as url, login credentials, and the driver (H2 in our case) are included.



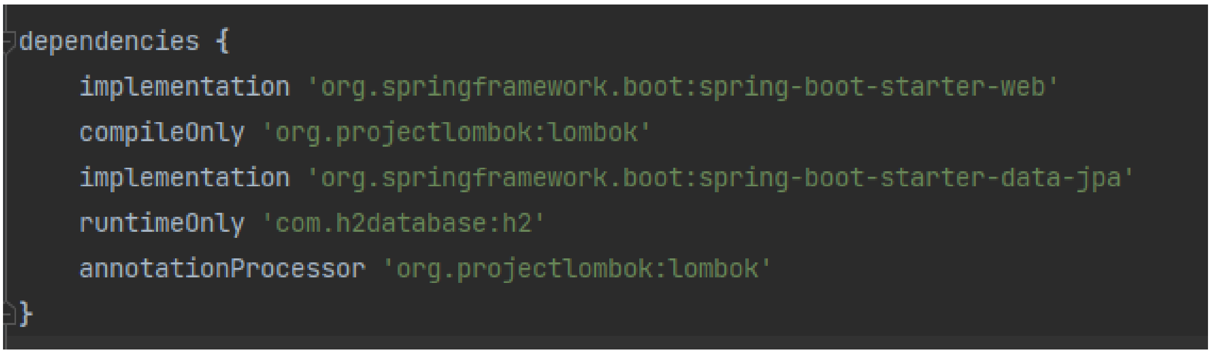
```
@RequiredArgsConstructor
@Getter
@Setter
public class Item {
    @Id
    private Long id;
    private String name;
}
```

Figure 13. The Item entity

Figure 13 presents the Item entity class, structured to model the underlying *Item* table within a relational database. The Lombok annotations here generate a constructor, getters for both fields and respective setters for both fields. The *@Id* annotation signifies that the field *id* serves as the primary key for the *Item* table.

4.5 Implementation of Spring MVC REST API

The Spring MVC application has the same functionality as the WebFlux application but is written in a non-reactive blocking manner.

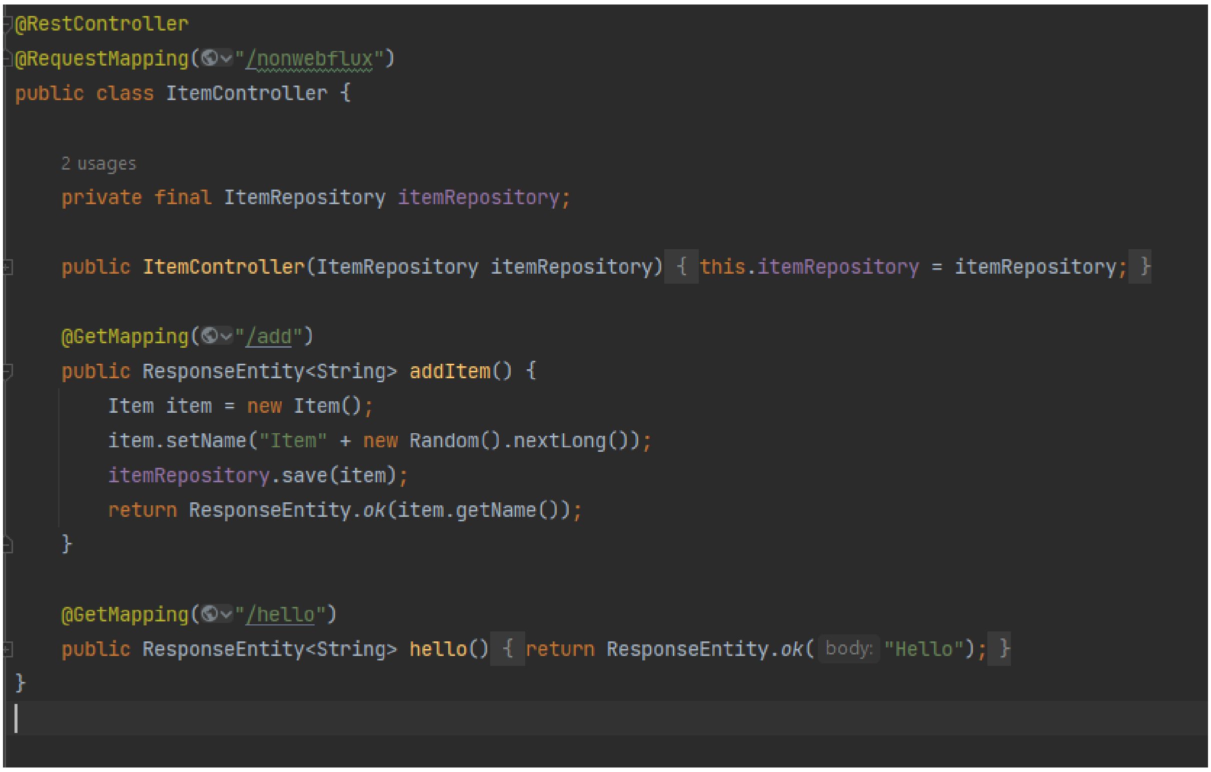


```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    runtimeOnly 'com.h2database:h2'
    annotationProcessor 'org.projectlombok:lombok'
}
```

Figure 14. SpringBoot Web MVC dependencies

In Figure 14 we see all the dependencies that are used for the Spring MVC application. The main difference is that *spring-boot-starter-web* is used instead of *spring-boot-starter-webflux*, together with *spring-boot-starter-data-jpa*. *spring-boot-starter-web* serves as the foundation for building web-based applications using the Spring Framework. Specifically, it facilitates the creation of RESTful web services by leveraging Spring MVC. Unlike reactive frameworks such as Spring WebFlux, this package is based on a blocking I/O model and utilizes a separate thread for each request-response cycle.

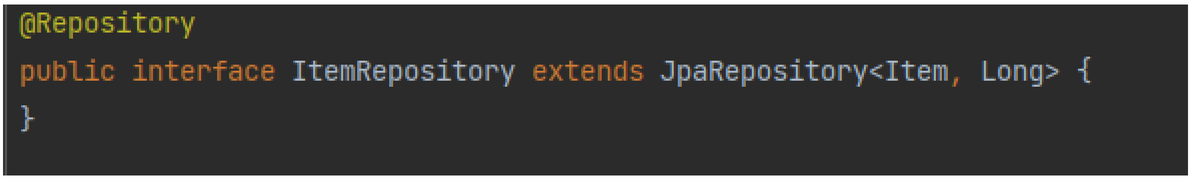
spring-boot-starter-data-jpa provides an abstraction layer for object-relational mapping through Java Persistence API (JPA) and Hibernate. It streamlines database interactions by offering simplified CRUD operations and query execution and operates on a blocking model.



```
@RestController
@RequestMapping("/nonwebflux")
public class ItemController {

    2 usages
    private final ItemRepository itemRepository;

    public ItemController(ItemRepository itemRepository) { this.itemRepository = itemRepository; }

    @GetMapping("/add")
    public ResponseEntity<String> addItem() {
        Item item = new Item();
        item.setName("Item" + new Random().nextLong());
        itemRepository.save(item);
        return ResponseEntity.ok(item.getName());
    }

    @GetMapping("/hello")
    public ResponseEntity<String> hello() { return ResponseEntity.ok( body: "Hello"); }
}
```

Figure 15. Spring MVC ItemController java class

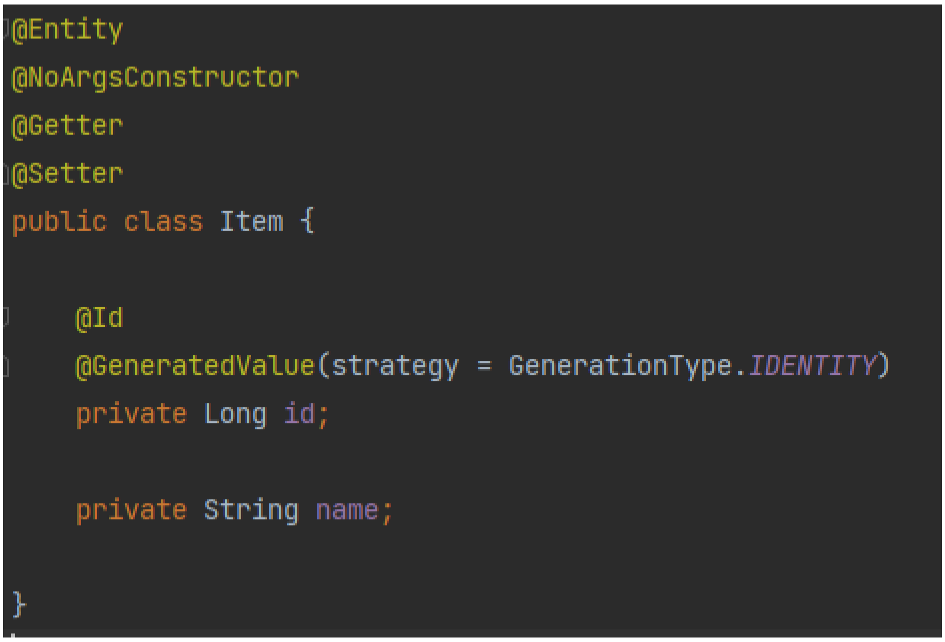
Figure 15 shows the *ItemController* implementation for the Spring MVC controller which includes the same functionality as the WebFlux controller. The main difference is that it directly returns a *ResponseEntity*-type. The injected repository is now a JPA repository that has a similar method *save()* but operates in a blocking manner.



```
@Repository
public interface ItemRepository extends JpaRepository<Item, Long> {
}
```

Figure 16. JPA repository

Figure 16 encapsulates the data layer abstraction for *Item* entities in a Spring Boot MVC application. It leverages the features of Spring Data JPA to minimize boilerplate code while offering a robust and flexible API for database interactions.



```
@Entity
@NoArgsConstructor
@Getter
@Setter
public class Item {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

}
```

Figure 17. Spring MVC Item

As can be seen in Figure 17, the only difference in the *Item* class is the *@Entity* annotations which indicates that the class is a JPA entity.

5 PERFORMANCE COMPARISON

Evaluating the performance of web applications requires planning and appropriate tooling. While there are various options available for performance testing, the suitability of each tool can differ based on various factors like ease of use, ability to be configured, and specific needs of the application being tested.

Initially, the performance evaluation was planned to be conducted using Bombardier[3] (release-1.2)—a fast, cross-platform HTTP benchmarking tool. Bombardier's minimal setup and straightforward interface made it an appealing choice for benchmarking our Spring WebFlux application. However, the tool encountered several issues that inhibited a comprehensive performance analysis. These issues ranged from compatibility challenges to limitations in generating diverse traffic patterns necessary for a thorough evaluation.
Due to the shortcomings faced by Bombardier, the focus was shifted to Gatling[4], a high-performance open-source load testing framework. This framework facilitates the simulation of intricate user behaviors, thereby aiming to provide an accurate reflection of application performance. Additionally, Gatling presents a set of comprehensive reporting tools, capturing metrics like mean response time, and response time distribution, providing valuable insights into the application's behavior during high load periods. For this study Gatling 3.9.5 was used.

To monitor the resource utilization of the web applications VisualVM[5] was used. VisualVM is a monitoring, troubleshooting, and profiling tool for Java applications. It can attach to locally and remotely running Java processes, providing a broad perspective on resource utilization such as CPU, memory, threads, and classes. This makes it a suitable candidate for understanding the intricate details of the JVM during the load test.
During the monitoring VisualVM 2.1.6 was used.

[3] https://github.com/codesenberg/bombardier
[4] https://gatling.io/
[5] https://visualvm.github.io/

5.1 Benchmarking methodology

To quantify the performance and scalability of the reactive web application developed using Spring WebFlux and R2DBC (as detailed in Figures 7 to 13) the comparison will be made against a similar Spring Boot MVC application that has the same functionality implemented in a non-reactive manner.

Machine Specifications:

- Operating System: Windows 11 Pro 64-bit (10.0, Build 22621) (22621.ni_release.220506-1250)
- Processor: Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz (20 CPUs), ~3.6GHz
- Memory: 32 GB RAM DDR4
- Network: localhost:8080 (0 latency)
- Benchmarking Tool: Gatling 3.9.5 and VisualVM 2.1.6
- Storage: 1TB SSD

Test Scenarios:

- Concurrent User Access towards */hello*-endpoint : Multiple users accessing the application simultaneously with varied think times.
- Concurrent User Access towards */add*-endpoint : A call to the database is made each time the endpoint is called.

Metrics Captured:

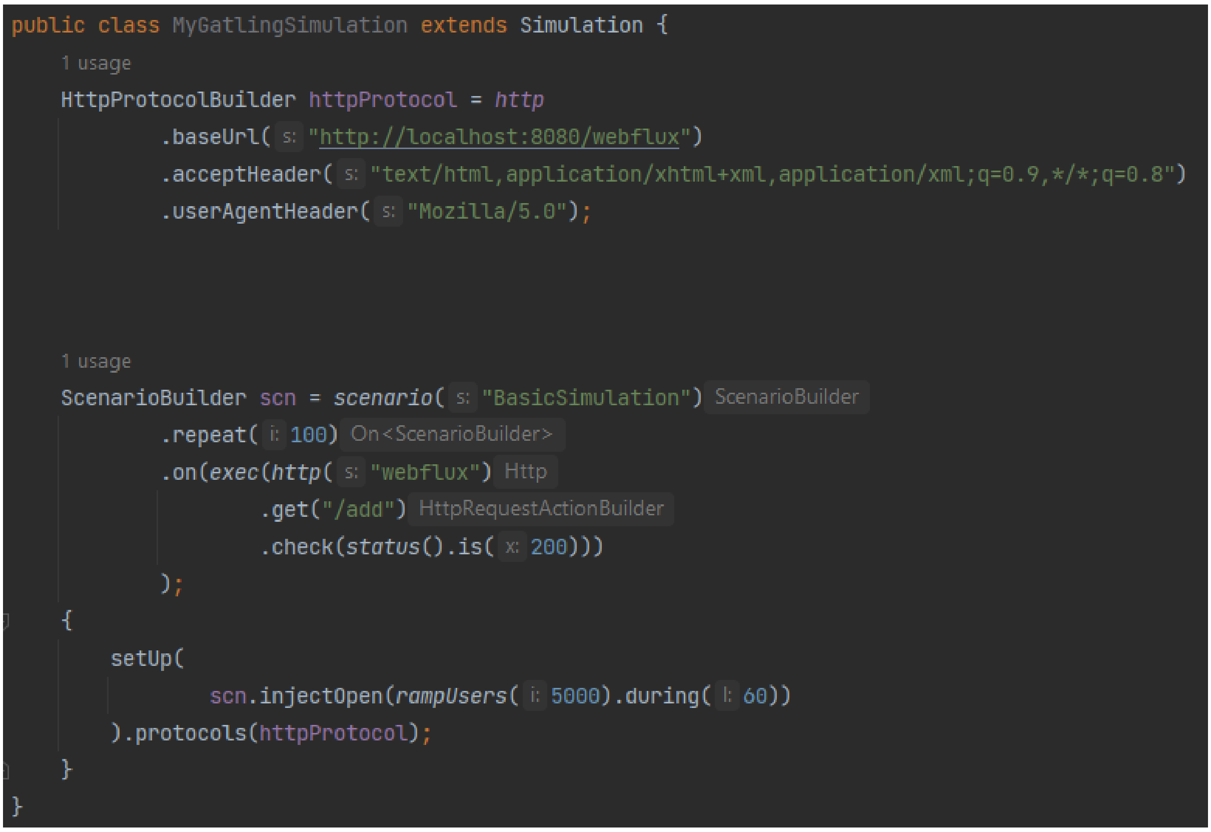
- Response Time: Average, median.
- Throughput: Number of requests processed per second.
- Error Rate: Percentage of failed requests over the total number of requests.
- Resource Utilization: CPU and Memory usage statistics gathered with VisualVM

Data Analysis

The data collected was statistically analyzed to draw conclusions on the application's performance characteristics. Gatling's in-built reporting tools were instrumental in this phase, generating graphical reports that elucidated both the strengths and bottlenecks of the application architecture.

5.2 Performance results and testing scenarios

For the testing scenarios with a focus on HTTP-based web applications, this chapter provides an analytical perspective on how different components and configurations impact performance metrics like response time and resource utilization. In Figure 18, a Gatling Simulation Script is used to simulate and measure the performance of our web applications running on a local server. This will serve as a cornerstone for understanding how performance tests are designed, executed and analyzed.



```
public class MyGatlingSimulation extends Simulation {
    1 usage
    HttpProtocolBuilder httpProtocol = http
            .baseUrl( s: "http://localhost:8080/webflux")
            .acceptHeader( s: "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8")
            .userAgentHeader( s: "Mozilla/5.0");



    1 usage
    ScenarioBuilder scn = scenario( s: "BasicSimulation") ScenarioBuilder
            .repeat( i: 100) On<ScenarioBuilder>
            .on(exec(http( s: "webflux") Http
                    .get("/add") HttpRequestActionBuilder
                    .check(status().is( x: 200)))
            );
    {
        setUp(
                scn.injectOpen(rampUsers( i: 5000).during( l: 60))
        ).protocols(httpProtocol);
    }
}
```

Figure 18. MyGatlingSimulation class

Figure 18 shows a class named *MyGatlingSimulation* which extends the *Simulation* class from Gatling.

HTTP Protocol Configuration:

- *HttpProtocolBuilder httpProtocol* defines the base URL of the server as *http://localhost:8080/webflux*
- A basic and common configuration is used to accept various types of headers with varying quality parameters and sets the user-agent as *Mozilla/5.0*.

Scenario Definition:

- *ScenarioBuilder scn* defines a scenario named *BasicSimulation*.
- Repeats 100 times.
- Makes an HTTP GET request to the specified, */add*, endpoint.
- Checks if the status code is 200 (OK).

Test Setup:

- *setUp* specifies the test setup configuration.
- Utilizes the scenario *scn*.
- Injects 5000 virtual users into the system, with the number of users increasing gradually over a period of 60 seconds.
- Associates the defined HTTP protocol settings with the scenario via *.protocols(httpProtocol)*.

The overarching objective of testing methodology is to provide an in-depth analysis of web application performance. Specifically, we aim to draw a comparative performance analysis between Spring WebFlux and Spring MVC architectures. This comparison will help us understand how these two different frameworks respond under load, latency, and other stress conditions.

Our baseline for these tests will be the setup described in Figure 18. This setup, initially targeted at a WebFlux application running on *http://localhost:8080/webflux*, simulates 5000 virtual users ramping up over a period of 60 seconds, making 100 repeated GET requests to the */add* endpoint. We will observe key performance indicators such as response time and throughput during this test.

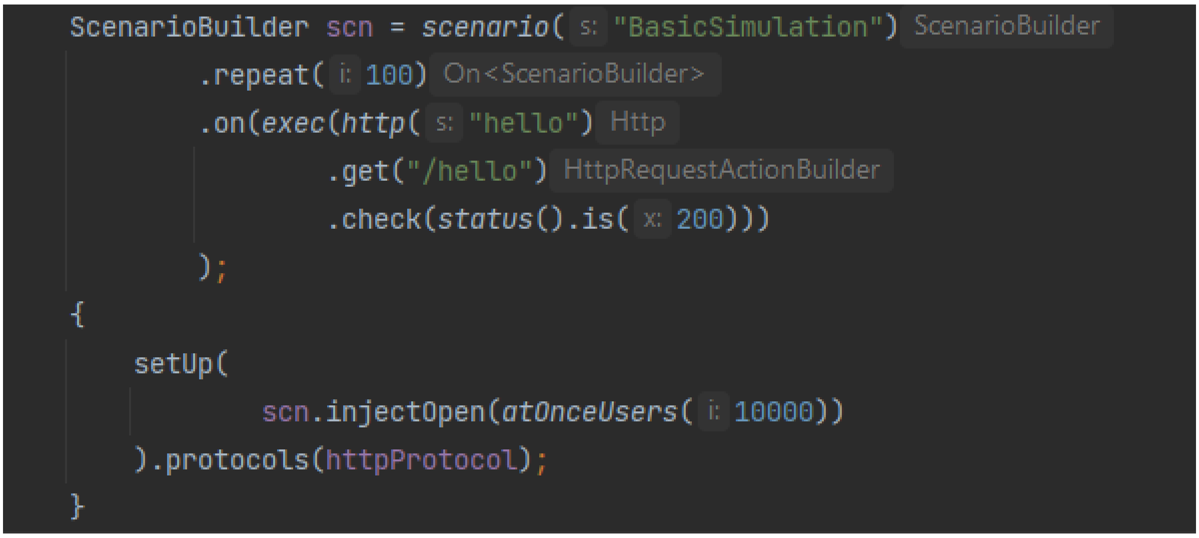
Variations in the Tests:

1. WebFlux Baseline Test with */add* endpoint: Our first test will follow the configuration and settings from Figure 18, targeting a WebFlux application.
2. Spring MVC Baseline Test with */add* endpoint: Using the same configurations as in Figure 18, we will run a similar test on a Spring MVC application to compare the baseline performances.
3. WebFlux High Load Test with */hello* endpoint: In this test, we will significantly ramp up the scale by simulating 1,000,000 requests to the */hello* endpoint on the WebFlux

application. This will help us assess how the framework behaves under extreme stress conditions.

4. Spring MVC High Load Test with */hello* endpoint: Following a similar approach, we will test the Spring MVC application with 1,000,000 requests to the */hello* endpoint to compare its performance under high load conditions.

The following scenario will be used in order to test the */hello* endpoint of both our applications:



```
ScenarioBuilder scn = scenario("BasicSimulation")
        .repeat(100)
        .on(exec(http("hello")
                .get("/hello")
                .check(status().is(200)))
        );
{
    setUp(
            scn.injectOpen(atOnceUsers(10000))
    ).protocols(httpProtocol);
}
```

Figure 19. The scenario used for testing the /hello endpoint

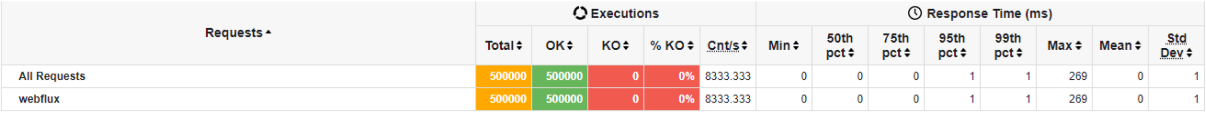
By running these tests, we aim to achieve a multi-dimensional view of our application's performance. This will not only allow us to identify potential bottlenecks in different parts of the system but also to evaluate the impact of different API endpoints on the overall performance of WebFlux and Spring MVC applications.

5.3 Analysis of results

The primary focus of this chapter is to analyze the data collected from the comprehensive performance tests detailed in the preceding chapters. This will involve evaluation of key performance indicators such as response time, throughput and resource utilization under different test conditions. The ultimate aim is to derive meaningful patterns and conclusions that can help in making informed decisions about choosing between Spring WebFlux and Spring MVC for specific use cases.

5.3.1 */add* endpoint performance in the Spring WebFlux application

Response Time:



Requests ▴	Executions					Response Time (ms)							
	Total ⇕	OK ⇕	KO ⇕	% KO ⇕	Cnt/s ⇕	Min ⇕	50th pct ⇕	75th pct ⇕	95th pct ⇕	99th pct ⇕	Max ⇕	Mean ⇕	Std Dev ⇕
All Requests	500000	500000	0	0%	8333.333	0	0	0	1	1	269	0	1
webflux	500000	500000	0	0%	8333.333	0	0	0	1	1	269	0	1

Figure 20. WebFlux /add - Response time

Figure 20 shows that the mean response time for the request made to the */add* endpoint was 0 milliseconds, with a standard deviation of 1 millisecond. The max response time was 269 milliseconds and 95% of the requests were fulfilled within 1 millisecond.

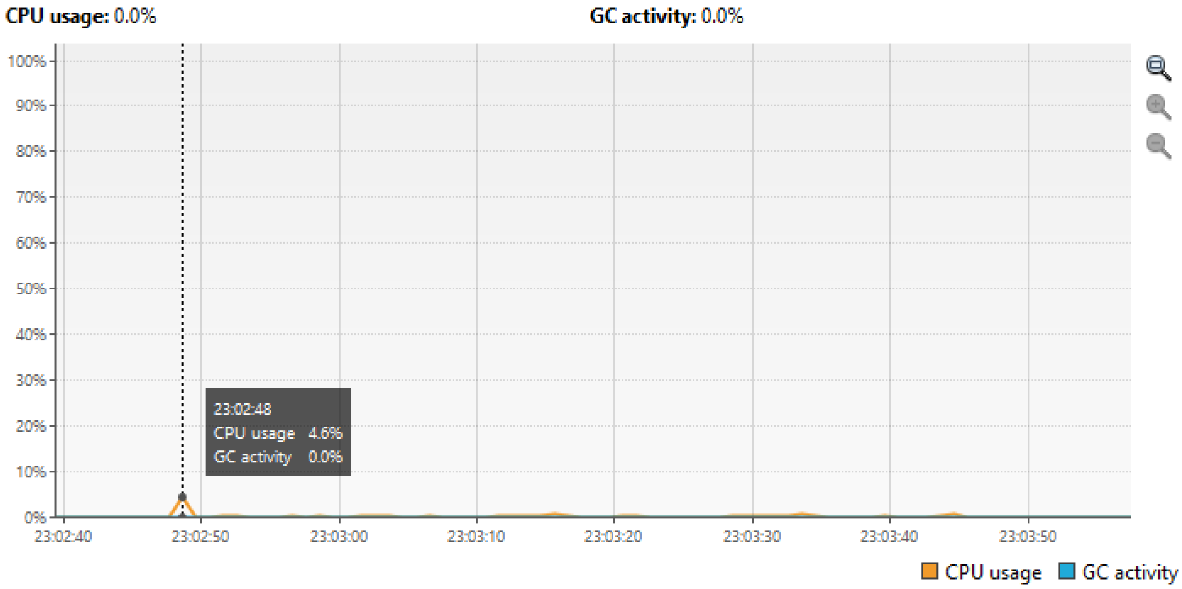


Figure 21. WebFlux /add - CPU usage

From Figure 21 we notice that the max CPU usage when calling the */add* endpoint was 4.6% at the beginning of the test.

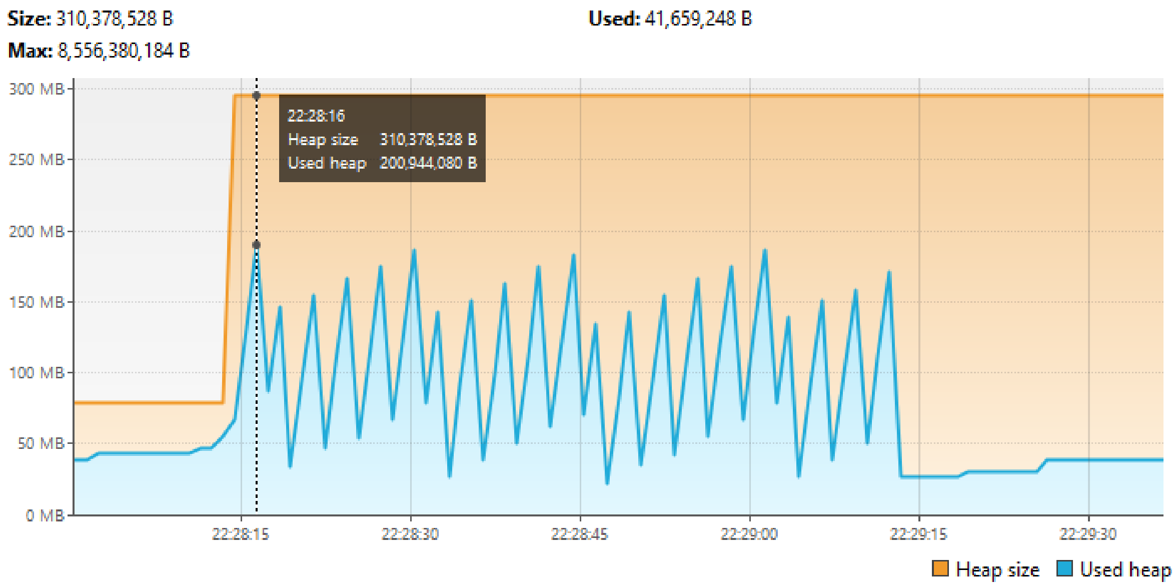


Figure 22. WebFlux /add - Heap memory

In Figure 22 we see that the heap memory size and utilized converted to megabytes (MB) was:

Max heap size: 310 MB

Max used heap: 200 MB

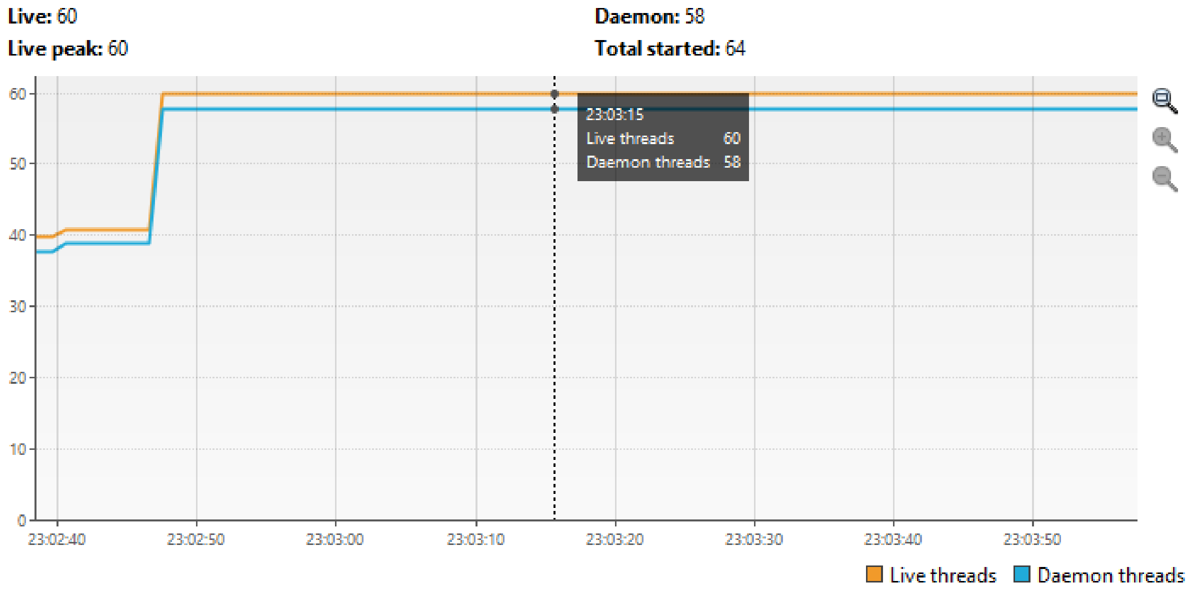


Figure 23. WebFlux /add - Number of threads

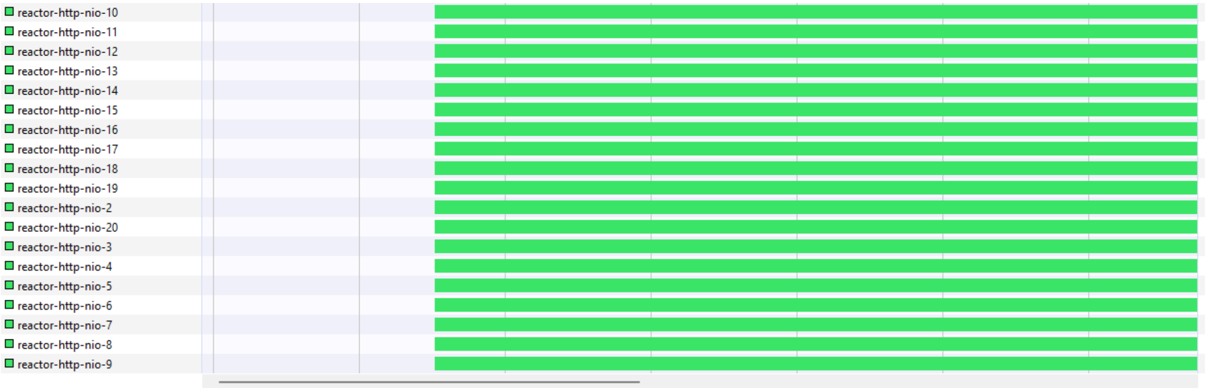
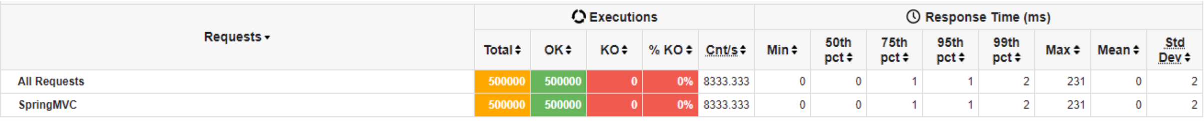


Figure 24. WebFlux /add - The reactor threads created

A total of 60 threads were running during the test as shown in Figure 23, and in Figure 24 we can see the *reactor-http-nio* threads that were used during the test. Of the total 60 threads, 22 of them were worker threads mostly used for database calls. The remaining 18 threads are distributed across JVM, Spring, and Netty.

5.3.2 */add* endpoint performance in the Spring MVC application

Response Time:



Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	500000	500000	0	0%	8333.333	0	0	1	1	2	231	0	2
SpringMVC	500000	500000	0	0%	8333.333	0	0	1	1	2	231	0	2

Figure 25. SpringMVC /add - Response time

Figure 25 shows that the mean response time for the request made to the */add* endpoint was 0 milliseconds, with a standard deviation of 2 milliseconds. The max response time was 231 milliseconds and 95% of the requests were fulfilled within 1 millisecond.

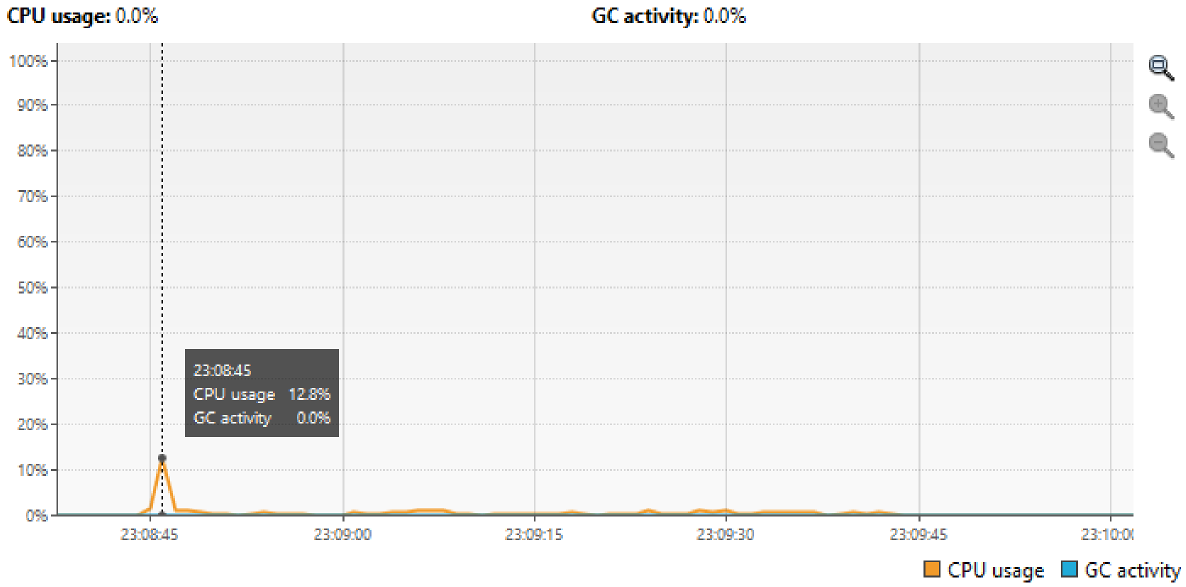
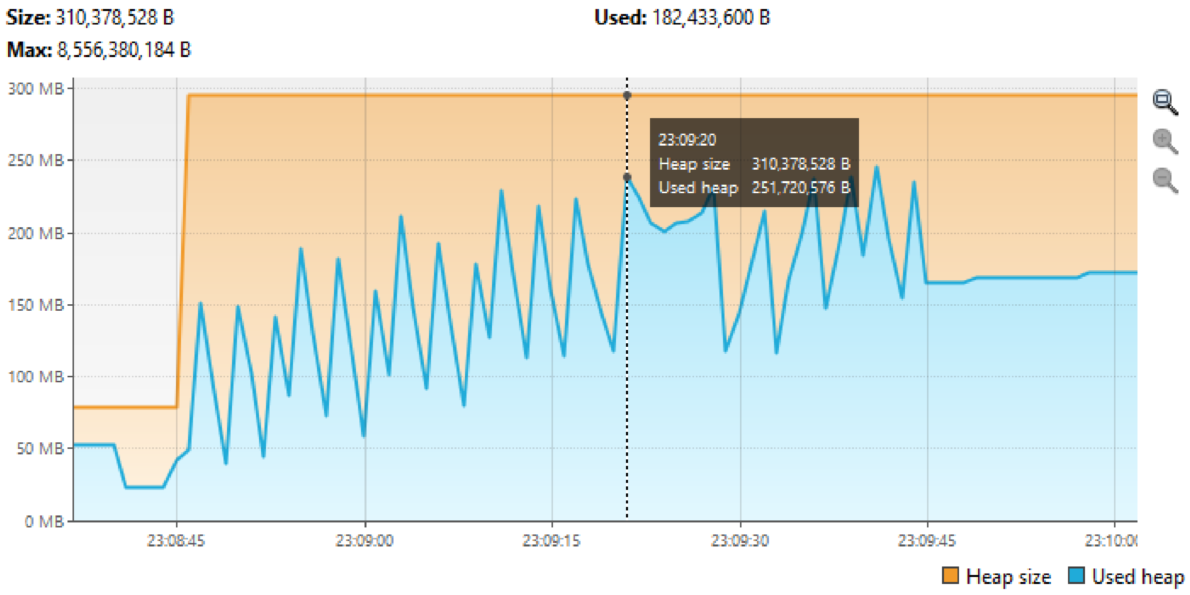


Figure 26. SpringMVC /add - CPU usage

From Figure 26 we notice that the max CPU usage when calling the */add* endpoint was 12.8% at the beginning of the test.



Size: 310,378,528 B

Max: 8,556,380,184 B

Used: 182,433,600 B

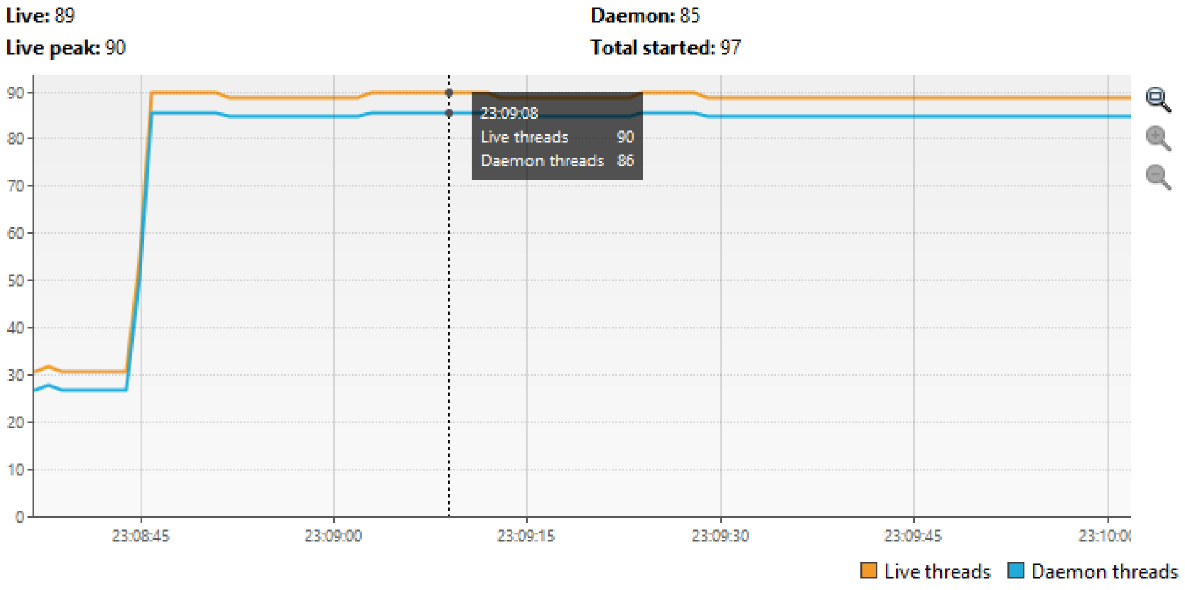
23:09:20
Heap size 310,378,528 B
Used heap 251,720,576 B

Figure 27. SpringMVC /add - Heap memory

In Figure 27 we see the heap memory size and utilized converted to megabytes (MB) was:

Max heap size: 310 MB

Max used heap: 250 MB



Live: 89

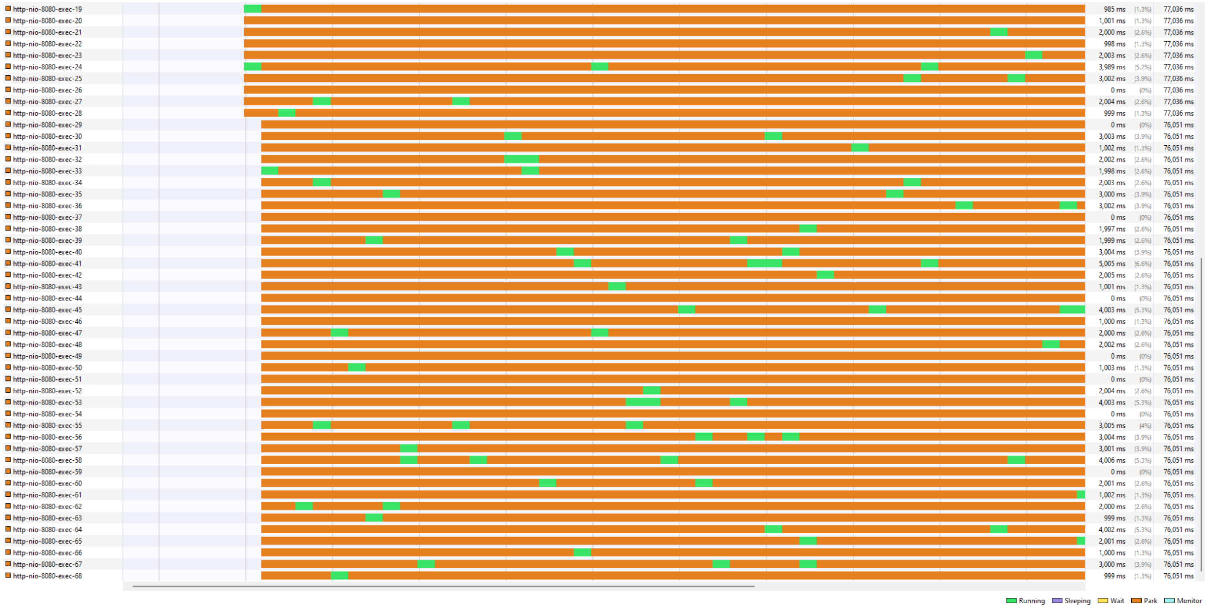
Live peak: 90

Daemon: 85

Total started: 97

23:09:08
Live threads 90
Daemon threads 86

Figure 28. SpringMVC /add - Number of threads



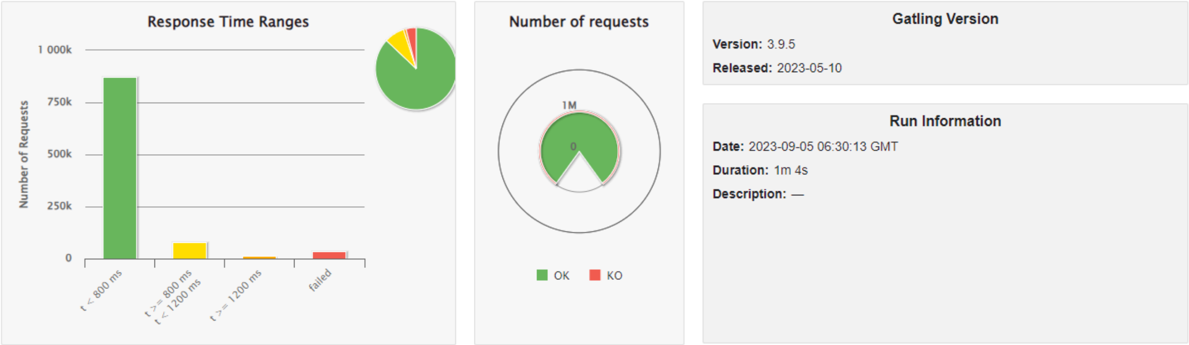
http-nio-8080-exec-19	985 ms	(1.3%)	77,036 ms
http-nio-8080-exec-20	1,001 ms	(1.3%)	77,036 ms
http-nio-8080-exec-21	2,000 ms	(2.6%)	77,036 ms
http-nio-8080-exec-22	998 ms	(1.3%)	77,036 ms
http-nio-8080-exec-23	2,003 ms	(2.6%)	77,036 ms
http-nio-8080-exec-24	3,989 ms	(5.2%)	77,036 ms
http-nio-8080-exec-25	3,002 ms	(3.9%)	77,036 ms
http-nio-8080-exec-26	0 ms	(0%)	77,036 ms
http-nio-8080-exec-27	2,004 ms	(2.6%)	77,036 ms
http-nio-8080-exec-28	999 ms	(1.3%)	77,036 ms
http-nio-8080-exec-29	0 ms	(0%)	76,051 ms
http-nio-8080-exec-30	3,003 ms	(3.9%)	76,051 ms
http-nio-8080-exec-31	1,002 ms	(1.3%)	76,051 ms
http-nio-8080-exec-32	2,002 ms	(2.6%)	76,051 ms
http-nio-8080-exec-33	1,998 ms	(2.6%)	76,051 ms
http-nio-8080-exec-34	2,003 ms	(2.6%)	76,051 ms
http-nio-8080-exec-35	3,000 ms	(3.9%)	76,051 ms
http-nio-8080-exec-36	3,002 ms	(3.9%)	76,051 ms
http-nio-8080-exec-37	0 ms	(0%)	76,051 ms
http-nio-8080-exec-38	1,997 ms	(2.6%)	76,051 ms
http-nio-8080-exec-39	1,999 ms	(2.6%)	76,051 ms
http-nio-8080-exec-40	3,004 ms	(3.9%)	76,051 ms
http-nio-8080-exec-41	5,005 ms	(6.6%)	76,051 ms
http-nio-8080-exec-42	2,005 ms	(2.6%)	76,051 ms
http-nio-8080-exec-43	1,001 ms	(1.3%)	76,051 ms
http-nio-8080-exec-44	0 ms	(0%)	76,051 ms
http-nio-8080-exec-45	4,003 ms	(5.3%)	76,051 ms
http-nio-8080-exec-46	1,000 ms	(1.3%)	76,051 ms
http-nio-8080-exec-47	2,000 ms	(2.6%)	76,051 ms
http-nio-8080-exec-48	2,002 ms	(2.6%)	76,051 ms
http-nio-8080-exec-49	0 ms	(0%)	76,051 ms
http-nio-8080-exec-50	1,003 ms	(1.3%)	76,051 ms
http-nio-8080-exec-51	0 ms	(0%)	76,051 ms
http-nio-8080-exec-52	2,004 ms	(2.6%)	76,051 ms
http-nio-8080-exec-53	4,003 ms	(5.3%)	76,051 ms
http-nio-8080-exec-54	0 ms	(0%)	76,051 ms
http-nio-8080-exec-55	3,005 ms	(4%)	76,051 ms
http-nio-8080-exec-56	3,004 ms	(3.9%)	76,051 ms
http-nio-8080-exec-57	3,001 ms	(3.9%)	76,051 ms
http-nio-8080-exec-58	4,006 ms	(5.3%)	76,051 ms
http-nio-8080-exec-59	0 ms	(0%)	76,051 ms
http-nio-8080-exec-60	2,001 ms	(2.6%)	76,051 ms
http-nio-8080-exec-61	1,002 ms	(1.3%)	76,051 ms
http-nio-8080-exec-62	2,000 ms	(2.6%)	76,051 ms
http-nio-8080-exec-63	999 ms	(1.3%)	76,051 ms
http-nio-8080-exec-64	4,002 ms	(5.3%)	76,051 ms
http-nio-8080-exec-65	2,001 ms	(2.6%)	76,051 ms
http-nio-8080-exec-66	1,000 ms	(1.3%)	76,051 ms
http-nio-8080-exec-67	3,000 ms	(3.9%)	76,051 ms
http-nio-8080-exec-68	999 ms	(1.3%)	76,051 ms

Figure 29. SpringMVC /add - The http threads created

A total of 90 threads were running during the test as shown in Figure 28, and in Figure 29 we can see a part of the 68 different *http-nio-8080-exec* threads that were used during the test. The red part generally means that the thread was in a *parked* state. This means that these threads are not executing any task at that moment.

The green color marks when the threads are in *runnable* state, meaning they are actively executing tasks or are ready to execute tasks as soon as they get CPU time.

5.3.3 */hello* endpoint performance in the Spring WebFlux application



Gatling Version

Version: 3.9.5

Released: 2023-05-10

Run Information

Date: 2023-09-05 06:30:13 GMT

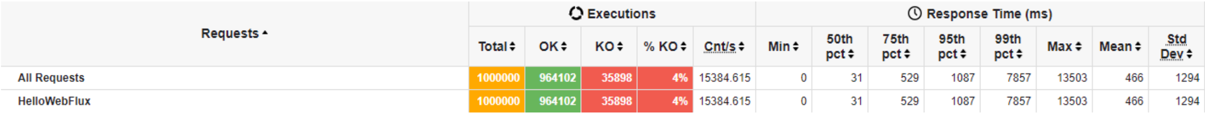
Duration: 1m 4s

Description: —

Figure 30. WebFlux /hello - Test overview

In Figure 30 we can see that the duration for the entire test was 64 seconds.

Response Time:



Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	1000000	964102	35898	4%	15384.615	0	31	529	1087	7857	13503	466	1294
HelloWebFlux	1000000	964102	35898	4%	15384.615	0	31	529	1087	7857	13503	466	1294

Figure 31. WebFlux /hello - Response time

Figure 31 shows that the mean response time for the request made to the */hello* endpoint was 466 milliseconds, with a standard deviation of 1294 milliseconds. The max response time was 13503 milliseconds and 95% of the requests were fulfilled within 7857 milliseconds.

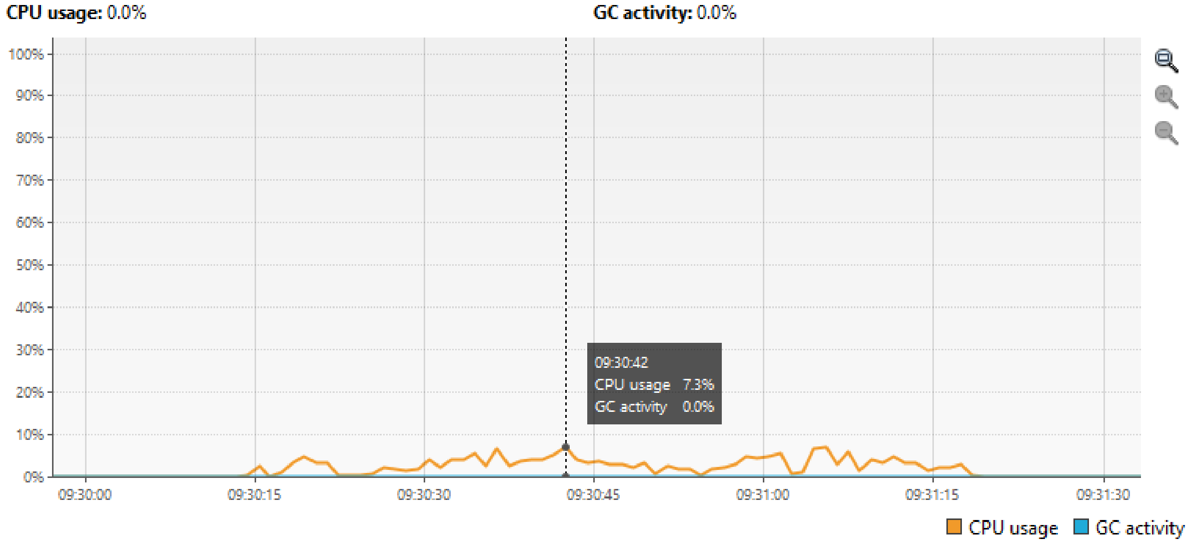


Figure 32. WebFlux /hello - CPU usage

From Figure 32 we notice that the max CPU usage when calling the */hello* endpoint was 7.3%.

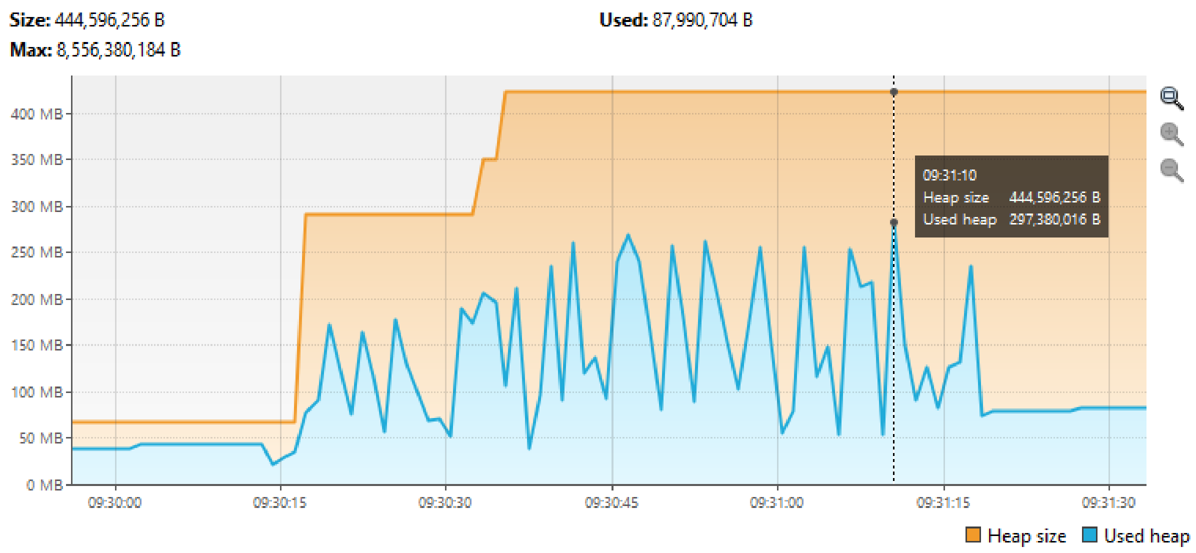


Figure 33. WebFlux /hello - Heap memory

In Figure 33 we see that the heap memory size and utilization converted to megabytes (MB) was:

Max heap size: 444 MB

Max used heap: 297 MB

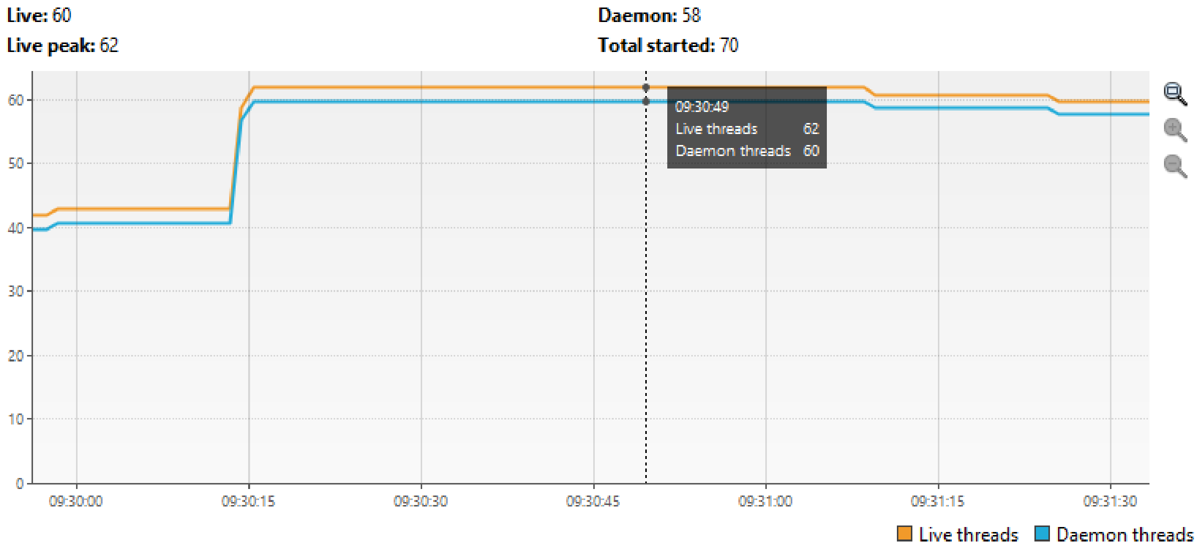
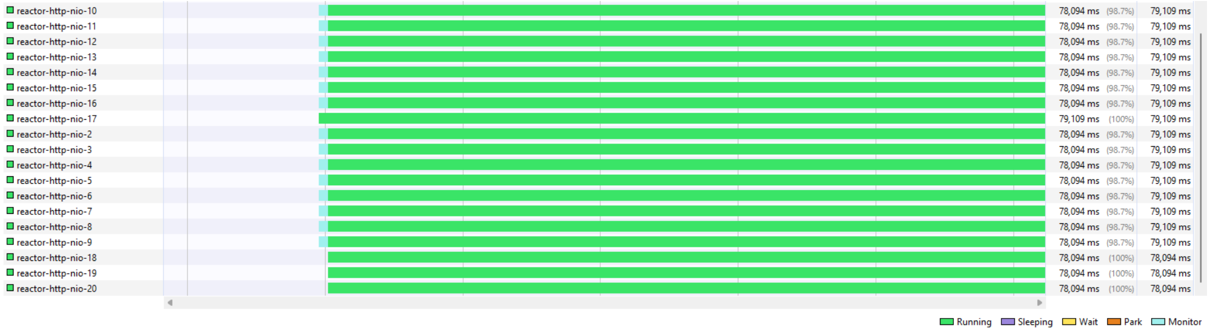


Figure 34. WebFlux /hello - Number of threads

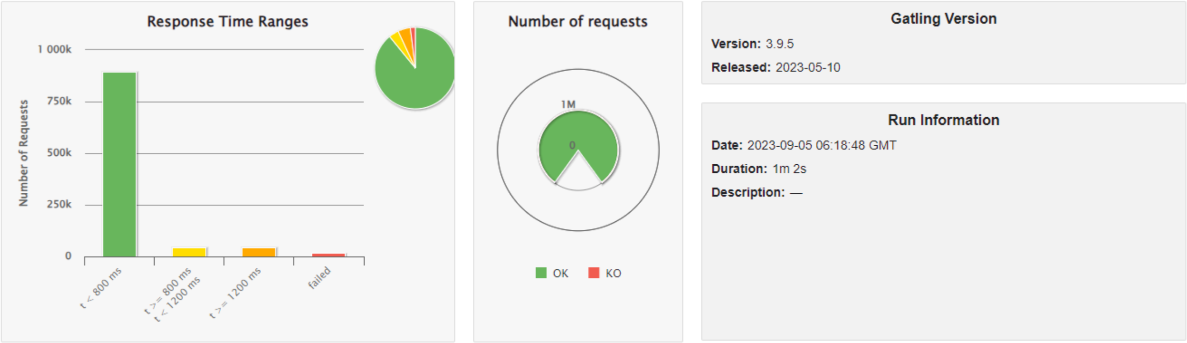


reactor-http-nio-10	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-11	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-12	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-13	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-14	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-15	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-16	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-17	79,109 ms	(100%)	79,109 ms
reactor-http-nio-2	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-3	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-4	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-5	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-6	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-7	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-8	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-9	78,094 ms	(98.7%)	79,109 ms
reactor-http-nio-18	78,094 ms	(100%)	78,094 ms
reactor-http-nio-19	78,094 ms	(100%)	78,094 ms
reactor-http-nio-20	78,094 ms	(100%)	78,094 ms

Figure 35. WebFlux /hello - The reactor threads created

A total of 62 threads were running during the test as shown in Figure 34, and in Figure 35 we can see the 20 *reactor-http-nio* threads that were used during the test. Likewise for */add* there were 22 threads used mainly for database calls and the remaining 20 threads are distributed across JVM, Spring, and Netty.

5.3.4 */hello* endpoint performance in the Spring MVC application



Gatling Version

Version: 3.9.5

Released: 2023-05-10

Run Information

Date: 2023-09-05 06:18:48 GMT

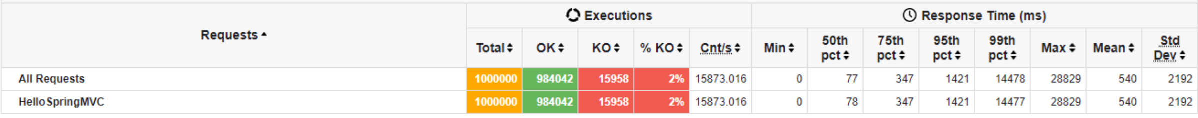
Duration: 1m 2s

Description: —

Figure 36. WebFlux /hello - Test overview

In Figure 36 we can see that the duration of the entire test was 62 seconds.

Response Time:



Requests ▴	Executions					Response Time (ms)							
	Total ⬍	OK ⬍	KO ⬍	% KO ⬍	Cnt/s ⬍	Min ⬍	50th pct ⬍	75th pct ⬍	95th pct ⬍	99th pct ⬍	Max ⬍	Mean ⬍	Std Dev ⬍
All Requests	1000000	984042	15958	2%	15873.016	0	77	347	1421	14478	28829	540	2192
HelloSpringMVC	1000000	984042	15958	2%	15873.016	0	78	347	1421	14477	28829	540	2192

Figure 37. SpringMVC /hello - Response time

Figure 37 shows that the mean response time for the request made to the */hello* endpoint was 540 milliseconds, with a standard deviation of 2192 milliseconds. The max response time was 28829 milliseconds and 95% of the requests were fulfilled within 14477 milliseconds.

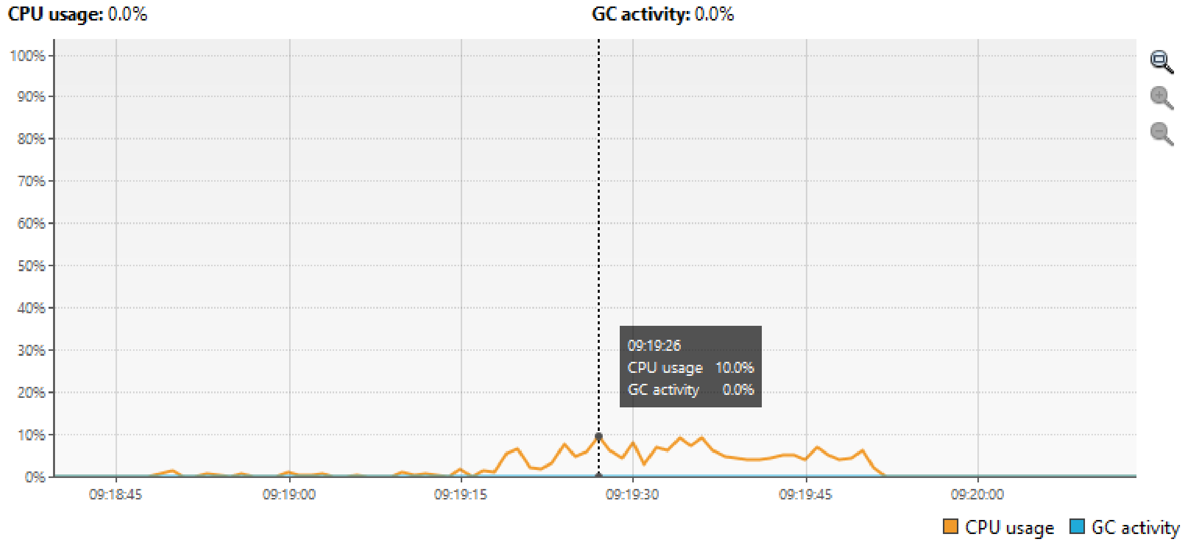
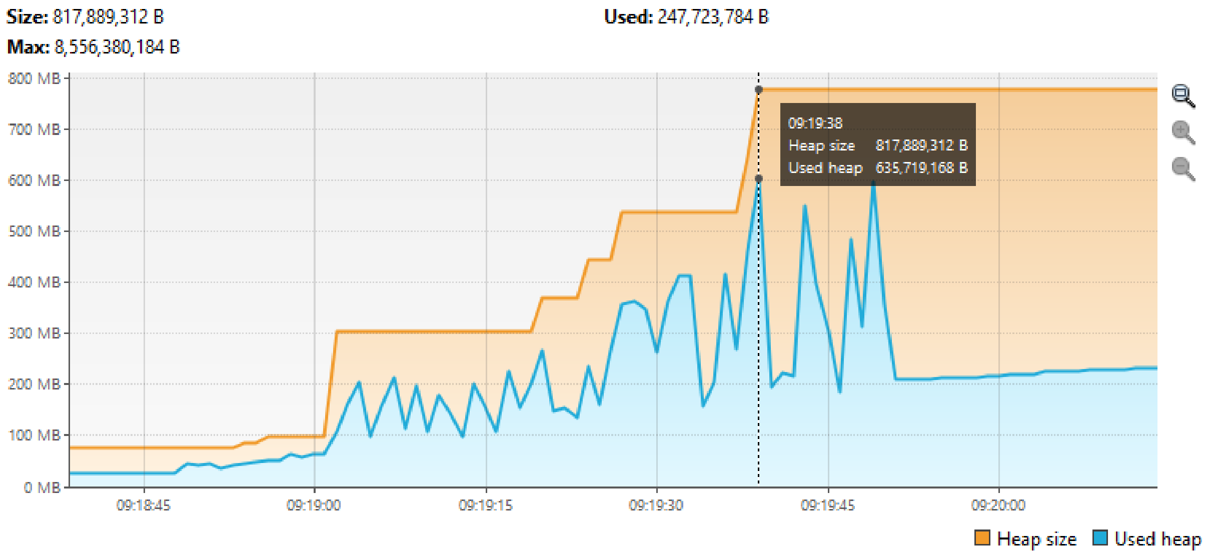


Figure 38. SpringMVC /hello - CPU usage

From Figure 38 we notice that the max CPU usage when calling the */hello* endpoint was 10% at the beginning of the test.



Size: 817,889,312 B

Max: 8,556,380,184 B

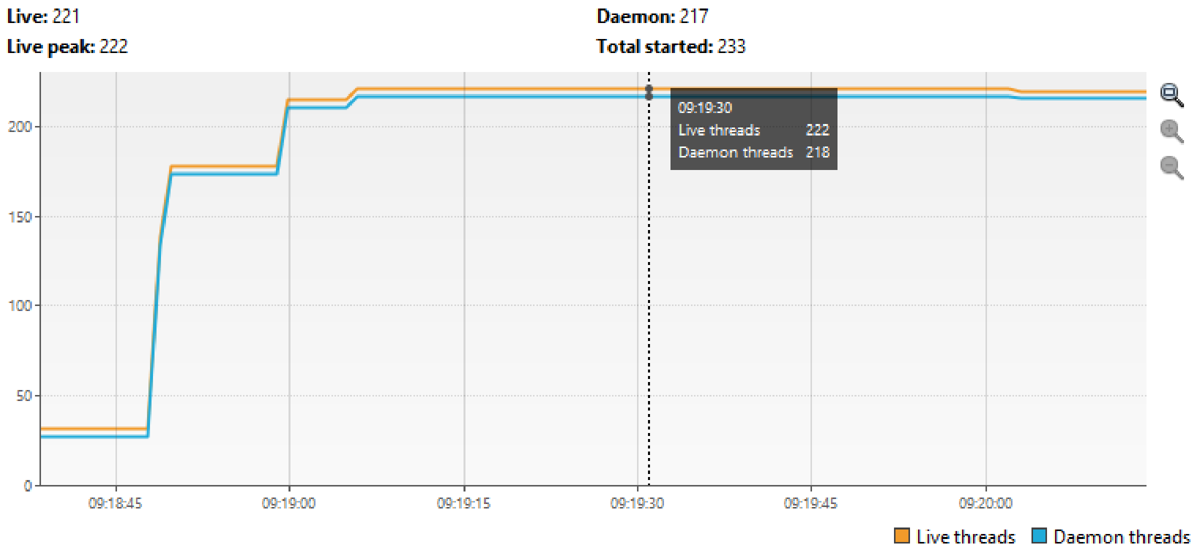
Used: 247,723,784 B

Figure 39. SpringMVC /add - Heap memory

In Figure 39 we see the heap memory size and utilized converted to megabytes (MB) was:

Max heap size: 817 MB

Max used heap: 635 MB



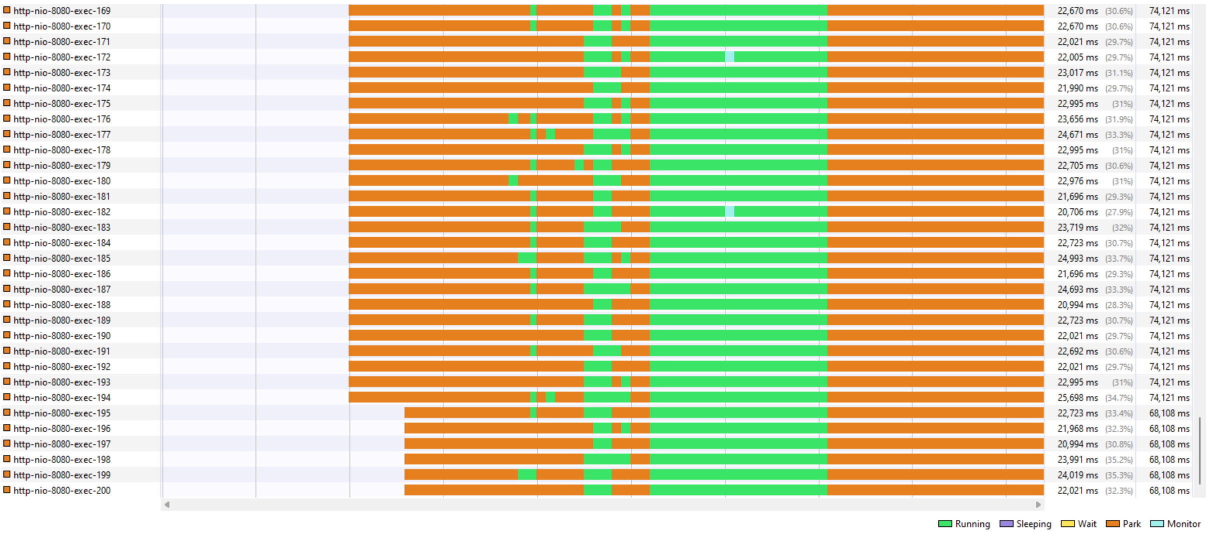
Live: 221

Live peak: 222

Daemon: 217

Total started: 233

Figure 40. SpringMVC /hello - Number of threads



http-nio-8080-exec-169	22,670 ms	(30.6%)	74,121 ms
http-nio-8080-exec-170	22,670 ms	(30.6%)	74,121 ms
http-nio-8080-exec-171	22,021 ms	(29.7%)	74,121 ms
http-nio-8080-exec-172	22,005 ms	(29.7%)	74,121 ms
http-nio-8080-exec-173	23,017 ms	(31.1%)	74,121 ms
http-nio-8080-exec-174	21,990 ms	(29.7%)	74,121 ms
http-nio-8080-exec-175	22,995 ms	(31%)	74,121 ms
http-nio-8080-exec-176	23,656 ms	(31.9%)	74,121 ms
http-nio-8080-exec-177	24,671 ms	(33.3%)	74,121 ms
http-nio-8080-exec-178	22,995 ms	(31%)	74,121 ms
http-nio-8080-exec-179	22,705 ms	(30.6%)	74,121 ms
http-nio-8080-exec-180	22,976 ms	(31%)	74,121 ms
http-nio-8080-exec-181	21,696 ms	(29.3%)	74,121 ms
http-nio-8080-exec-182	20,706 ms	(27.9%)	74,121 ms
http-nio-8080-exec-183	23,719 ms	(32%)	74,121 ms
http-nio-8080-exec-184	22,723 ms	(30.7%)	74,121 ms
http-nio-8080-exec-185	24,993 ms	(33.7%)	74,121 ms
http-nio-8080-exec-186	21,696 ms	(29.3%)	74,121 ms
http-nio-8080-exec-187	24,693 ms	(33.3%)	74,121 ms
http-nio-8080-exec-188	20,994 ms	(28.3%)	74,121 ms
http-nio-8080-exec-189	22,723 ms	(30.7%)	74,121 ms
http-nio-8080-exec-190	22,021 ms	(29.7%)	74,121 ms
http-nio-8080-exec-191	22,692 ms	(30.6%)	74,121 ms
http-nio-8080-exec-192	22,021 ms	(29.7%)	74,121 ms
http-nio-8080-exec-193	22,995 ms	(31%)	74,121 ms
http-nio-8080-exec-194	25,698 ms	(34.7%)	74,121 ms
http-nio-8080-exec-195	22,723 ms	(33.4%)	68,108 ms
http-nio-8080-exec-196	21,968 ms	(32.3%)	68,108 ms
http-nio-8080-exec-197	20,994 ms	(30.8%)	68,108 ms
http-nio-8080-exec-198	23,991 ms	(35.2%)	68,108 ms
http-nio-8080-exec-199	24,019 ms	(35.3%)	68,108 ms
http-nio-8080-exec-200	22,021 ms	(32.3%)	68,108 ms

Figure 41. SpringMVC /hello - The http threads created

A total of 222 threads were running during the test as shown in Figure 40, and in Figure 41 we can see a part of the 200 different *http-nio-8080-exec* threads that were used during the test.

5.3.5 Summary of the result

The results of the performance evaluations are summarized in Table 1

Table 1. Summary of the results.

	Duration (seconds)	*Mean response time (milliseconds)*	*OK %*	*Threads used*	*CPU usage (%)*	*Heap used (MB)*	*Application start time (seconds)*
Spring Boot MVC (*/add*)	60	0	100	90	12.8	250	2.3
Spring WebFlux (*/add*)	60	0	100	60	4.6	200	1.5
Spring Boot MVC (*/hello*)	62	540	98	222	10	635	2.3
Spring WebFlux (*/hello*)	64	466	98	62	7.3	297	1.5

5.4 Analysis of the result

In terms of duration, both Spring Boot MVC and Spring WebFlux endpoints were tested for approximately the same amount of time.

Regarding mean response time, for the */add* endpoint, Spring Boot MVC and Spring WebFlux both achieved a low mean response time of under 1 millisecond rounded to 0 milliseconds. However, for the */hello* endpoint, Spring Boot MVC exhibited a slightly higher mean response time of 540 milliseconds compared to Spring WebFlux's 466 milliseconds, indicating that WebFlux was slightly faster in processing */hello* requests.

In the case of the OK percentage, both frameworks achieved a high success rate for the */add* endpoint with 100% OK responses. For the */hello* endpoint, both maintained an approximate 98% OK rate, suggesting effective request handling with only a minor margin of error or failed requests.

In terms of threads used, Spring Boot MVC employed more threads than WebFlux. Specifically, Spring Boot MVC used 90 threads for the */add* endpoint and 222 threads for the */hello* endpoint, whereas WebFlux used 60 threads for */add* and 62 threads for */hello*. This highlights WebFlux's efficiency in thread utilization due to its non-blocking and reactive nature.

In CPU usage, Spring Boot MVC generally consumed more CPU resources, with 12.8% CPU usage for the */add* endpoint and 10% for the */hello* endpoint. In contrast, WebFlux demonstrated greater efficiency with CPU usage, reporting 4.6% for */add* and 7.3% for */hello.* This efficiency can be attributed to WebFlux's event-driven, non-blocking model, which reduces CPU-intensive context switches.

Heap memory usage showed that Spring Boot MVC consumed more memory, with 250 MB for */add* and 635 MB for */hello.* In contrast, WebFlux proved to be more memory-efficient, using only 200 MB for */add* and 297 MB for */hello.*

6 CONCLUSION

This thesis set out to explore two primary research questions:

1. How does Spring WebFlux compare to traditional synchronous web development in terms of performance and scalability?

The comparative analysis conducted in this study highlights the potential advantages of Spring WebFlux and its reactive programming model. The results indicate that Spring WebFlux shows efficiency gains in terms of thread utilization, CPU usage, and memory consumption, particularly when compared with the traditional Spring Boot MVC framework. These advantages are most pronounced in scenarios characterized by high concurrency and resource-intensive or complex tasks. The non-blocking nature of Spring WebFlux is a key factor in these scenarios. However, it is crucial to note that the suitability of Spring WebFlux over Spring Boot MVC depends on specific project requirements, available infrastructure, and the nature of the application being developed. While the findings lean towards the potential benefits of Spring WebFlux in certain circumstances, they do not universally establish it as the superior choice for all types of web development.

2. What are the key considerations for successfully implementing Spring WebFlux?

The study underscores the importance of aligning the choice of the framework with the specific demands of the project. Choosing between Spring WebFlux and Spring Boot MVC should not be a blanket decision but one that is carefully tailored to the project's unique requirements. Further research is warranted, particularly involving the development of an end-to-end application that encompasses CRUD operations, disk operations, security considerations, and logging. This extended research would provide a more comprehensive understanding of both the capabilities and limitations of Spring WebFlux, offering valuable insights for modern web application development.

In summary, this thesis emphasizes the need for a nuanced approach to framework selection in web development, taking into consideration the specificities of each project. The findings from this study contribute to a more informed decision-making process in the realm of web application development, particularly when choosing between Spring WebFlux and traditional synchronous approaches.

REFERENCES

Chandrakant, K. (2020, August 18). *Concurrency in Spring WebFlux*. Baeldung. https://www.baeldung.com/spring-webflux-concurrency

Dokuka, O., & Lozynskyi, I. (2018). *Hands-On Reactive Programming in Spring 5*. Packt Publishing.

Introduction to spring framework. (n.d.). Retrieved September 5, 2023, from https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/overview.html

Bonér, J., Farley, D., Kuhn, R., & Thompson, M. (2014, September 16). *The Reactive Manifesto*. Reactivemanifesto. https://www.reactivemanifesto.org/

Maldini, S., & Baslé, S. (n.d.). *Reactor 3 Reference Guide*. Retrieved September 5, 2023, from https://projectreactor.io/docs/core/release/reference/

Muscad, O. (2022, November 22). *Synchronous vs. Asynchronous: A guide to choosing the ideal programming model*. DATAMYTE. https://www.datamyte.com/synchronous-vs-asynchronous/

MVC Framework - Introduction. (n.d.). Retrieved September 5, 2023, from https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm

Nolle, T. (2021, March 24). *Reactive programming*. App Architecture; TechTarget. https://www.techtarget.com/searchapparchitecture/definition/reactive-programming

Spring WebFlux Overview. (n.d.). Spring Docs. Retrieved September 8, 2023, from https://docs.spring.io/spring-framework/reference/web/webflux/new-framework.html

Reactive Streams. (n.d.). Retrieved September 5, 2023, from https://www.reactive-streams.org/

Spring framework documentation. (n.d.). Retrieved September 5, 2023, from https://docs.spring.io/spring-framework/reference/

Spring initializr. (n.d.). Spring Initializr. Retrieved September 8, 2023, from https://start.spring.io/

What Is A RESTful API? (n.d.). Retrieved September 5, 2023, from https://aws.amazon.com/what-is/restful-api/