POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Spring Boot and Spring WebFlux: experimental evaluation of Rest-API implementations

Supervisor

Prof. Marco TORCHIANO

Candidate

Loredana FINOCCHIARO

December 2021

Summary

Reactive programming has been around for some time now, but recently has gained a new resonance, simultaneously with the rise of microservices and the ubiquity of multi-core processors: it represents a way of meeting the increasing demands of today, where applications need to have high availability and low response times also during high load. It avoids stateful programs, allowing the programmer to abstract away low-level threading, synchronization, and concurrency issues. However, reactive programming can represent the most appropriate choice only in peculiar scenarios, for instance when concurrency and asynchronicity are absolutely needed, or for handling large streams of data.

The thesis is the result of the internship experience carried out at Technology Reply. The scope of interest regards the Connected Vehicle Technology and the Business Intelligence.

For a well-known automotive company, a system was developed that retrieves and collects data deriving from multiple types of signals, sent in real-time from each connected vehicle (uniquely identified by its VIN, i.e. Vehicle Identification Number), with a very high frequency: as a consequence, the amount of data gathered every day is considerably large (about two billion records per day). These signals are collected, stored and analyzed in a Corporate Data Lake, with the purpose of identifying common patterns (e.g., about driving or charging habits), allowing the business and the data scientist team to carry out deep dive analyses on the phenomena related to the connected vehicle. The necessity of reading and analyzing the collected data efficiently becomes evident and essential.

The objective of the application that was developed is the detailed analysis of a single vehicle or driver in order to categorize it on the basis of the received signals, by integrating them with information retrieved in real-time from external systems.

The application was developed in Java, leveraging the Spring framework, then deployed as a microservice on Oracle Cloud. Two different versions will be analyzed, in order to evaluate and compare different Rest-API implementations, especially aiming to better manage large amounts of data and parallelize calls to an external geocoding service.

- The first version is a traditional Spring Boot application, where Rest-APIs will be implemented both in a synchronous and asynchronous fashion (using *CompletableFutures*);
- The second one is a Spring Boot Reactive aka WebFlux application, fully based on reactive and functional programming.

Finally, performance and load tests were conducted with an open-source testing tool, Gatling, that allowed to create different scenarios by simulating a varying number of users making calls, with the objective to demonstrate how the WebFlux application reveals to be more scalable and performant with respect to the other attempted solutions.

Acknowledgements

In conclusion of my course of study, I would like to express my sincere thanks to those people who guided me throughout this thesis work, contributing significatively to the success of it.

Above all, I thank my supervisor, professor Marco Torchiano, a person I had the possibility to really appreciate both from a human and a professional point of view.

I also wish to thank the whole Technology Reply team, in particular Alessandro Dugo, manager and company tutor, for giving me the opportunity of carrying out my thesis within the company; Cristina Ardito, my constant technical reference, for the idea, the availability and the support never lacking; and finally Davide Giosa, who assisted me a lot during the study and the analysis of the data.

Table of Contents

List of Tables

List of Figures

Acronyms

OS	Operating System
AP	Asynchronous Programming
RP	Reactive Programming
FRP	Functional Reactive Programming
UI	User Interface
DSL	Domain-Specific Language

Chapter 1

The Reactive Landscape

In this opening chapter we're going to present the theoretical background related to the Reactive Programming.

First, we'll introduce some key and preliminary concepts that are fundamental to better understand the whole topic; then, we'll go in depth analyzing what the term "reactive" means in the context of Software Engineering, focusing particularly on what Reactive Programming and Reactive Systems are today, trying to understand the differences among them, once widely described. The chapter ends with a final section devoted to mention and present briefly the reactive libraries available in Java.

1.1 Preliminary concepts

Before going to deepen the main topic of this chapter, we found it necessary to provide and clarify some basic as much as worth definitions that stand behind and together with the idea of Reactive Programming.

1.1.1 Threading models

A program under execution is called *process*. A process usually starts with a single thread and, later on, possibly creates other threads.

Threads are sequences of execution of code that can be executed independently one from another. A thread is a simple flow of instructions and the smallest unit of tasks that can be executed by an OS.
An application can have one among two different types of model (or architecture):

- Single-threaded: only one task is processed at a time;
- Multi-threaded: multiple tasks can be executed at the same time.[1]

1.1.2 Synchronous programming

In *synchronous* (or *blocking*) I/O, a thread starts an I/O operation and immediately enters a wait state until the I/O request has been completed.[2]

In a single-threaded architecture, a thread is allocated to handle the task that is requested to run: if the response to the request involves time-consuming operations (e.g., a database query), the thread will remain idle until the operation completes. Meanwhile, if other tasks arrive, these will have to wait for the previous I/O to complete before being served.

In a multi-threaded architecture, a new thread is allocated to a new request or task: if there is a large number of concurrent tasks, the program may run out of threads, putting new tasks to wait until a thread is available. Following this idea, we are not able to make the best advantage of the hardware.[1]

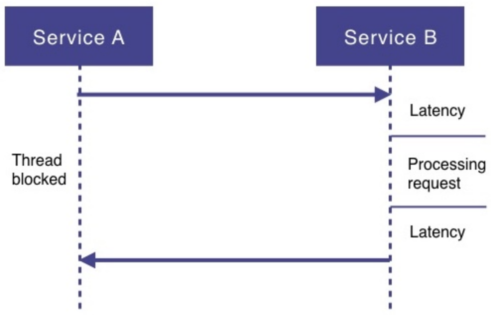


Figure 1.1: Thread blocked waiting for response.[3]

1.1.3 Asynchronous programming

The Oxford Dictionary defines the adjective asynchronous as "not existing or happening at the same time"[4], which in our context means that "the processing of a message or event is happening at some arbitrary time, possibly in the future"[5].

Asynchronous programming is not a new concept: indeed, it exists since the very early days of computing, when it searched for strategies to guarantee an optimal usage of the hardware; recently, though, it has become a standard programming paradigm.

In a single-threaded architecture, a single thread will handle several tasks without waiting for their results; when results are ready, the thread will collect and present them to the caller.

In a multi-threaded architecture, multiple threads will handle several tasks *concurrently*, without waiting for their results that, once ready, will be taken and returned to the caller.[1]

AP supports non-blocking execution: threads in execution competing for a shared resource don't need to wait by blocking, that would prevent the running threads from performing other work until current work is done, rather they can execute other tasks while a specific resource is owned.[5]

As an evidence, there are numerous benefits to choosing AP, such as improved application performance, enhanced responsiveness and better usage of hardware resources.[6]

We exploit AP in order to improve the user experience: we would like to deliver a smooth user experience to our users without freezing the main thread, slowing them down, but in order to keep the main thread free we need to do a lot of heavy and time-consuming work in background. We also want to do heavy work and complex calculations on our servers as mobile devices are not so powerful to do the heavy lifting.[7]

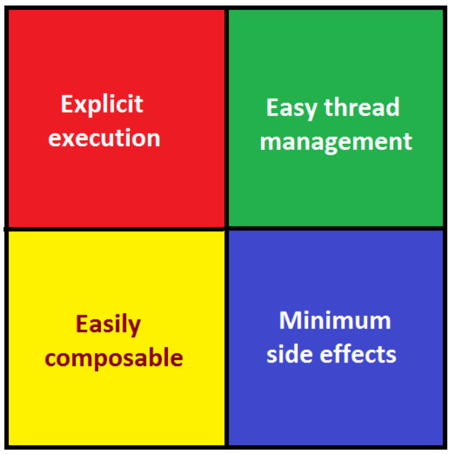


Figure 1.2: Evaluation matrix for the asynchronous library.[7]

Let's analyze what we should need from the library that handles all the asynchronous work:

- **Explicit execution**: we should be able to control the execution of a task on a new thread.
- **Easy thread management**: it is the key in such a context. We should be able to switch a thread easily and transfer the work to another thread if and when needed.
- **Easily composable**: the asynchronous library should be easily composable enough and provide less room for the error.
- **Minimum side effects**: while working with multiple threads, one thread should experience minimum side effects from another thread. This makes the

code easily readable and understandable and improves the traceability of the errors.[7]

Pitfalls of Asynchronous Programming

Talking about drawbacks, we should mention two problems related to the use of AP, also important for further discussions concerning the Reactive Programming.

- The difficulty of using callbacks: in AP, callbacks are used to handle the results derived from asynchronous computations that, in large applications, can be really numerous. This can generate the so called *Callback hell* or *Callback spaghetti*, that could make the flow of the application really complex, contributing even more to the difficulty of not only debugging but also of writing code. This results in longer development time. Nevertheless, programmers cannot do without callbacks, whose use is necessary to gain the possibility to execute code which depends on results from asynchronous computations.

- The use of OS threads: applications that implement asynchronous computations are thread intensive, and OS threads are expensive, since they allocate system resources. For such applications, a threading model that is not one-to-one with OS threads should run without problems, while a thread model of one-to-one with OS threads would waste excessively CPU and memory.

  We usually refer to OS threads by calling them *processes*, and are expensive to create because each of them has its own set of resources; instead, threads, that reside within a process, are considered sort of *lightweight processes*, since they consume less system resources with respect to processes. When talking about concurrent programming, the focus is mostly on threads.[8, pp. 25-26]

1.1.4 Stream

A *stream* is a sequence of ongoing events (or state changes) ordered in time. It can emit three possible and different things: a value (of some type), an error, or a "completed" signal.

Events are captured asynchronously: therefore, we need to define some function handlers, one for each of the three possible results, that will execute based on what is going to be emitted.

"Listening" to the stream corresponds to *subscribing*. The functions we define are called *observers*. The stream is the subject (i.e., the "observable") being observed.

When dealing with RP, data streams are going to be the spine of the application: you can create data streams for anything, including: variables, user inputs, properties, caches, data structures, and so on. These streams can then be observed and actions can be taken accordingly.[9] RP also provides lots of functions that allow to transform streams (with actions like map, filter, etc.) or combine them together.

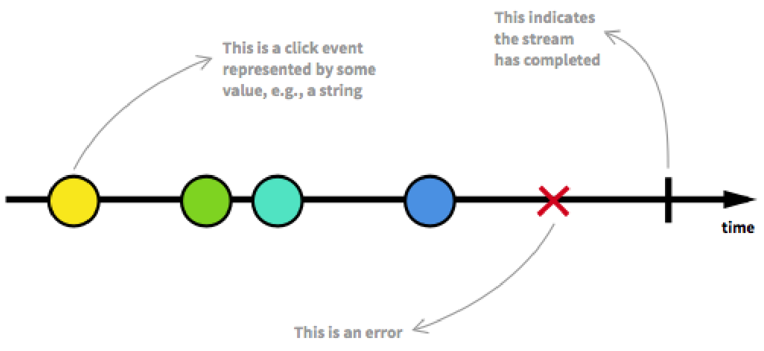

This is a click event represented by some value, e.g., a string
This indicates the stream has completed
time
This is an error

Figure 1.3: A simple stream execution flow.[9]

A notification will be sent when data is emitted in the stream *asynchronously*, meaning independently to the main program flow. By structuring the program around data streams, we are writing asynchronous code: we write code invoked when the stream emits a new item. Threads, blocking code and side-effects are very important matters in this context.[10]

1.2 Reactive Programming

Nowadays, the term reactive has become fairly loaded and sometimes confusing: indeed, it is used to refer to different things, and frequently one may find it coupled with words like “streaming”, “lightweight” and “real-time.”

When people talk about reactivity in the context of software development and design, they generally refer to one of three possible things:

- Reactive Programming (declarative event-based);
- Reactive Systems (architecture and design);
- Functional Reactive Programming (FRP).

Reactive Programming and Reactive Systems, although often used as synonymous, are not exactly the same things. In this section, we’ll examine the concept of Reactive Programming in all its facets, finally mentioning the FRP; in the next section, we’ll concentrate on Reactive Systems, highlighting similarities and differences with the Reactive Programming.

1.2.1 Definitions

It is possible to come across multiple definitions for Reactive Programming. Just to give an idea, we'll report some of them.
Reactive Programming is:

- "A style of micro-architecture involving intelligent routing and consumption of events, all combining to change behaviour"[11];
- "A development model structured around asynchronous data streams"[10];
- "An asynchronous programming paradigm oriented around data streams and the propagation of change, allowing changes to be modeled as they propagate through a circuit"[12];
- "A paradigm in which declarative code is issued to construct asynchronous processing pipelines"[9].

RP has been around for some time (its origins can probably be traced to the 1970s or even earlier), but gained much more interest during the last years, in the current state of web and application design: this new resonance has arrived (not accidentally) at the same time as the rise of microservices, and the ubiquity of multi-core processors. The reason derives from the fact that traditional imperative programming has some limitations and cannot support efficiently the demands of today, where applications need to have high availability and low response times also during high load.[3] Everything follows from the simple principle of incorporating asynchronous data streams into a programming framework, but the implementation can become exceedingly complex.

RP can be seen as a subset of AP and a paradigm where the availability of new information drives the logic forward rather than having control flow driven by a thread-of-execution. It supports decomposing the problem into multiple discrete steps where each can be executed in an asynchronous and non-blocking way, and then be composed to produce a workflow, possibly unbounded in its inputs or outputs.[5]

RP is generally event-driven, in contrast to Reactive Systems, which are message-driven.
The Application Program Interface (API) for RP libraries can be either:

- Callback-based, where anonymous and side-effecting callbacks are attached to event sources, and are being invoked when events pass through the dataflow chain;
- Declarative, through functional composition (usually using well established combinators like map, filter, etc.).

However, most libraries provide a mix of these two styles.[5]
Examples of programming abstractions that support RP are:

- Futures/Promises: containers of a single value, many-read/single-write semantics where asynchronous transformations of the value can be added even if it is not yet available.
- Streams: unbounded flows of data processing, enabling asynchronous, non-blocking, back-pressured transformation pipelines between a multitude of sources and destinations.
- Dataflow Variables: single assignment variables (memory-cells) which can depend on input, procedures and other cells, so that changes are automatically updated. A practical example is spreadsheets, where the change of the value in a cell ripples through all dependent functions, producing new values "downstream".[5]

1.2.2 The Reactive Principles

The Reactive Principles can be considered the follow-up to the *Reactive Manifesto* (described in detail in the next section). The document, written by experienced Reactive practitioners, suggests guidelines and techniques for realizing services, systems and applications. It integrates the ideas, paradigms, patterns and methods of both RP and Reactive Systems into a set of practical principles to be applied by software developers and designers.

The following principles helps in designing and implementing highly concurrent and distributed softwares that will be scalable, performant and resilient, conserving resources when deploying, operating and maintaining them.[13]

I. **Stay Responsive** - *Always respond in a timely manner.*

   Responsiveness is not just about providing low latency and fast response times, but it is also managing changes (in data, usage patterns, context, environment) that should be represented within the application and its data model, right up to its end-user interactions.

   Reactive and responsive applications deal with problems: in the worst-case, they respond with an error message or provide a degraded but still useful level of service, creating the basis for delivering a consistent quality of service. Such consistent behavior makes error handling easier, builds end-user confidence and encourages further interactions.

II. **Accept Uncertainty** - *Build reliability despite unreliable foundations.*

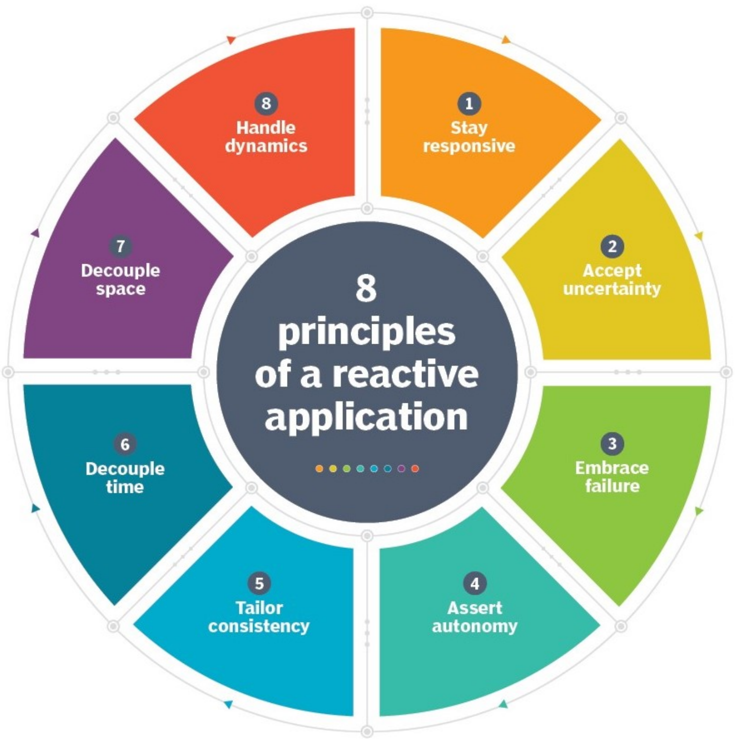


Figure 1.4: The eight principles of a reactive application.[14]

The world of distributed systems is "a world of uncertainty", in which systems could fail in the most spectacular and intricate ways, where information becomes lost, reordered, and corrupted, and failure detection is really challenging. Furthermore, the present is relative and subjective.

As a consequence, our algorithms will lack information due to faulty hardware, unreliable networks, or the plain physical problem of communication latency. Even though well established distributed algorithms exist, those tend to give poor performance and scalability, then we have had to give up most of them in order to achieve responsiveness, and so accepting the higher level of uncertainty that comes from this choice.

We can't always trust time as measured by clocks and timestamps, or order (causality might not even exist). We have to use strategies to cope with uncertainty (for instance: rely on logical clocks, such as vector clocks).

III. **Embrace Failure** - *Expect things to go wrong and design for resilience.*

Reactive applications consider failure as an expected condition that could eventually occur. Therefore, failure must be explicitly represented and handled by a supervisor component, or within the component itself (by using internal redundancy). Requests should be answered whenever possible even in the failure case, even though component autonomy will already ensure that the failure remains contained in as small an area of the application's function as possible.

IV. **Assert Autonomy** - *Design components that act independently and interact collaboratively.*

In reactive applications, autonomy is clearly defined by determining the component boundaries (who owns what data and how the owners make it available) and by designing them so that each party will have the necessary degree of freedom to make its own decisions.

The component, when called, must have the ability to communicate back momentary degradations (e.g., caused by overload or faulty dependencies) and the freedom not to respond when shedding heavy load. This requires the protocol between these components to be asynchronous and event-based.

V. **Tailor Consistency** - *Individualize consistency per component to balance availability and performance.*

Consistency is about guaranteeing the correctness and integrity of your application and user's data.

When possible, design systems for eventual consistency or causal consistency, leveraging asynchronous processing, which tolerates delays and temporary unavailability of its participants, allowing the system to stay available and, in case of failure, automatically recover.

VI. **Decouple Time** - *Process asynchronously to avoid coordination and waiting.*

Even in a reactive landscape, there are still times when we have to communicate and coordinate our actions. The problem with blocking on resources (e.g., I/O) is that the caller, including the thread it is executing on, is blocked while waiting for the resource to become available and so is unavailable for other requests. This can be mitigated or avoided by employing *temporal decoupling*: we give the caller the option to perform other work, asynchronously, rather than be blocked waiting on the resource to become available (this can be achieved by allowing the caller to put its request on a queue, register a callback to be notified later, return immediately, and continue execution).

VII. **Decouple Space** - *Create flexibility by embracing the network.*

A resilient system lives in multiple locations (i.e., its parts are distributed across space): its distributed autonomous components collaborate to make maximal use of the newly won independence from one specific location.

This *spatial decoupling* makes use of network communication to connect the potentially remote pieces again: since all the networks work by passing messages between nodes and since this takes time, spatial decoupling introduces asynchronous message-passing on a foundational level. Spatial decoupling enables replication, which ultimately increases the resilience of the system and availability.

VIII. **Handle Dynamics** - *Continuously adapt to varying demand and resources.*

Workloads can vary drastically and applications need to stay responsive and continuously adapt to the current situation, being elastic and reacting to changes in the input rate by increasing or decreasing the resources allocated to service these inputs (allowing the throughput to scale up or down automatically to meet varying demands).[13]

1.2.3 Benefits and limitations

By using asynchronous data streams, RP can provide a highly dynamic experience: multiple data streams from the same page or form can be evaluated independently one from another with regards to content and time; feeds can update in real-time based on clicks or activity; forms can become interactive instead of statically awaiting completion of a full data set, and much more.[15]

The benefit is more visible in modern web and mobile apps that are plenty interactive with a multitude of UI events related to data events. Up to ten years ago, interaction with web pages consisted basically in submitting a long form to the backend and performing simple rendering to the frontend; nowadays, apps have evolved to be more real-time: modifying a single form field can automatically trigger a save to the backend, or some content could be reflected in real-time to other connected users, and so on.[16]

Once RP techniques are mastered and implemented, we gain benefits that go beyond the end-user experience. Indeed, by listening or subscribing independently to different data streams, the programmer can achieve a higher level of abstraction that is more in-tune with business rules and requirements. Moreover, the code can be more concise, and implementation of new rules or listening methods can be isolated to a discrete data stream, rather than wedged into a single method that has to account for the entire data set. This simplifies the life of the developer by making change and development more agile and more intuitive.

Furthermore, RP allows the developer to work with complex and possibly changing dependencies. With a series of values that all have dependencies on one other, data has to be ingested asynchronously in order to propagate change in the correct manner.[15]

Going into detail, advantages of RP can be summarized as follows:

- Increased utilization of computing resources on multi-core and multi-CPU hardware;[5]
- Optimization in the usage of resources, by reducing for instance the number of generated thread simultaneously working;
- Simpler management of parallel and concurrent working threads, since the need for explicit coordination between active components is usually removed;
- Usage of Functional Programming, that provides a lot of patterns to be reused;[17]
- Softer management of back-pressure, that is crucial to avoid over-utilization, or rather unbounded consumption of resources;[5]
- Provided better control over the response times associated with the processing of events;
- Made the concept of a stream or event flow explicit, improving overall management of compute elements and processing resources by making them more "visual";[14]
- Increased performance, thanks to the possibility to handle huge volumes of data in a quick and stable way;
- Simplified modifications and updates, due to more readable and easier to predict code.[18]

Clearly, while choosing if using RP or not, we should take in consideration also its limitations and drawbacks:

- Steep learning curve: it takes a long time to familiarize with functional programming notations (readability may become tricky for those who have an imperative background);
- It is not easy to debug to figure out which piece of code is not working properly;
- Constant management and monitoring of the various elements of the architecture.[17]

RP is recommended when all the data sources are non-blocking and executable in parallel; otherwise, it is still possible to "wrap" a blocking API within a reactive flow by circumscribing the blocked threads in a thread-pool so as to prevent the application from crashing due to the latter.

One should not choose RP when dealing with most blocking data sources, and wrapping them all is not a good idea, because it would be counterproductive.

Furthermore, it is necessary to exclude applications of an "imperative" type, that is a well-defined sequence of procedures, in which the currently active element needs the output of the previous element in order to start its work.[17]

1.2.4 Use cases

Going reactive provides an elegant solution when it comes to specific types of high-load or multi-user applications:

- Social networks, chats;
- Games;
- Audio and video apps.

And to the following components of any application type:

- Server code that serves highly interactive UI elements;
- Proxy servers, load balancers;
- Artificial intelligence, machine learning;
- Real-time data streaming.[18]

Relevant use cases for leveraging RP could be the following:

- IoT applications, where sensors generate events that control real-world process steps or create business transactions, or both;
- Applications that receive status information from networks or data processing elements by means of software agents that monitor activities or data elements;
- Signaling between applications, especially between the *foreground* and the *background* (or *batch*) applications, that perform statistical analysis and database cleanup;
- Coordination between functional AWS Lambda cloud processing and backend data center processing;
- Applications that require highly interactive user-to-user interface handling.[14]

Here are some examples from an enterprise setting that illustrate general patterns of use:

- **External service calls**: nowadays, many backend services are RESTful, so the underlying protocol is blocking and synchronous; however, the implementation of such services often involves calling other services. This generates a lot of I/O going on, that is a good candidate for optimization.
- **Highly concurrent message consumers**: message processing, particularly if highly concurrent, is a common enterprise use case. Since an event translates nicely into a message, reactive patterns fit naturally with message processing.
- **Spreadsheets**: if cell B depends on cell A, and cell C depends on both A and B, we need to propagate changes in A, ensuring that C is updated before any change events are sent to B. If we can leverage a powerful reactive framework, we don't care about the problem: we just declare the dependencies, and everything will work.[11]

Hundreds of enterprises in every major market around the world have embraced the principles of RP to build and deploy or improve some famous real-world applications. Let us give some well-known examples:

- Netflix API: RP with RxJava library has enabled Netflix developers to leverage server-side concurrency, needed to effectively reduce network chattiness, without the typical thread-safety and synchronization concerns. Without concurrent execution on the server, a single "heavy" client request might not be much better than many "light" requests, because each network request from a device naturally executes in parallel with other network requests. If the server-side execution of a collapsed "heavy" request does not achieve a similar level of parallel execution it may be slower than the multiple "light" requests, even accounting for saved network latency.[19]

  The API service layer implementation has control over concurrency primitives, which enables us to pursue system performance improvements without fear of breaking client code.[20]
- Trivago backend services: trivago's search backend team leveraged RP in Java when designing and implementing one of the many Java backend services. The benefits support them in some key challenges that every engineer is facing with essentially every (micro-) service in today's backend architectures: handling of blocking IO, backpressure, managing highly varying loads as well as message and error propagation. However, this choice required a paradigm and mindset shift in how to craft and handle the flow of the application. This took some time for the team to learn the required techniques and implement them step-by-step in their large and evolving backend architecture.[21]
- Capital One redesigned its auto loan application around Reactive principles to simplify online car shopping and financing. Customers can browse more

than four million cars from over 12,000 dealers and pre-qualify for financing in seconds, without impacting credit scores.

- LinkedIn turned to Reactive principles to build real-time presence indicators (online indicators) for the half billion users on its social network.
- Verizon Wireless, operators of the largest 4G LTE network in the United States, slashed response times in half using Reactive principles in the upgrade of its e-commerce website that supports 146 million subscribers handling 2.5 billion transactions a year.[22]

1.2.5 Functional Reactive Programming (FRP)

By definition, Functional Reactive Programming, or FRP, is a general framework for programming hybrid systems in a high-level, declarative manner. The key ideas in FRP are its notions of behaviors and events: *behaviors* are time-varying, reactive values, while *events* are time-ordered sequences of discrete-time event occurrences.[23]

FRP can be considered as the combination of functional and reactive paradigms; in other words, it is reacting to data streams using the functional paradigm. FRP is not a utility or a library: it changes the way it architects and think about the applications.[24]

Reactive data streams, that are inherently asynchronous, allow to write modular code by standardizing the method of communication between components. The reactive data flow allows for a loose coupling between these components, too. The functional part of FRP provides the tools to work with streams in a sane manner (for instance, functional operators allow to control how the streams interact with each other).[25]

1.3 Reactive Systems

Reactive programming and reactive systems have not the same meaning: although often used interchangeably, the terms are not exactly synonymous and reflect different things.

Reactive Systems, as defined by the Reactive Manifesto, are "a set of architectural design principles for building modern systems that are well prepared to meet the increasing demands that applications face today"[26]. They are designed to "enable applications composed of multiple microservices working together as a single unit to better react to their surroundings and one another, manifesting in greater elasticity when dealing with ever-changing workload demands and resiliency when components fail"[9].

The principles of Reactive Systems are definitely not new, and can be traced back to the 70s; however, it's been only in the last 5-10 years that the technology industry has been forced to reconsider current "best practices" for enterprise system development, that means learning to apply the hard-won knowledge of the Reactive principles on today's world of multi-core, Cloud Computing and the Internet of Things.[5]

The difference between RP and reactive systems is in their applications: RP is applied to the internal logic of components, while reactive systems are applied on an architectural level for systems. Applications implementing RP are highly event-driven, which means that events drive the execution forward instead of a thread-of-execution. Reactive programming facilitates decoupling in time and thereby concurrency. Reactive systems are, instead, message-driven. The systems facilitate decoupling in space and thereby distribution.[8, pp. 26-27]

1.3.1 Event-driven vs Message-driven

As previously said, RP, focusing on computation through ephemeral dataflow chains, is Event-driven, while Reactive Systems, focusing on resilience and elasticity through the communication and coordination of distributed systems, is generally Message-driven.

The main difference between the two is that Messages are inherently directed, Events are not. Messages have a clear, single destination, while Events are facts for others to observe. Moreover, messaging is preferably asynchronous, with the sending and the reception decoupled from the sender and receiver respectively.

These differences in semantics and applicability have significant implications in the application design, including things like resilience, elasticity, mobility, location transparency and management of the complexity of distributed systems, which will be better explained in the next paragraph.

In a Reactive System, especially one which uses RP, both events and messages will be present, as one is a great tool for communication (messages), and another is a great way of representing facts (events).[5]

1.3.2 The Reactive Manifesto

In 2014, version 2.0 of the Reactive Manifesto was published: this document was drawn up by a group of experienced developers, represented by Jonas Boner, who met in 2013 to define the main principles of a reactive architecture. To date, the manifesto has been signed by almost 30,000 people.

A system is defined as a set of components that cooperate together to provide services for users.

Reactive systems are responsive, resilient, elastic and message driven. The expectations from systems have become considerably higher since a few years ago, when long response times and several hours of maintenance with servers offline were considered acceptable. In contrast, users nowadays consider fast response times and no downtime of servers as a must. The authors of the manifesto believe that, in order to keep up with the current expectations, a coherent approach is needed.

By choosing a reactive system approach, these systems will be easier to develop, and also easily modifiable and resistant to failure, because they will be flexible, loosely-coupled and scalable. At the same time the systems will give users fast interactive feedback.[8, p. 20]

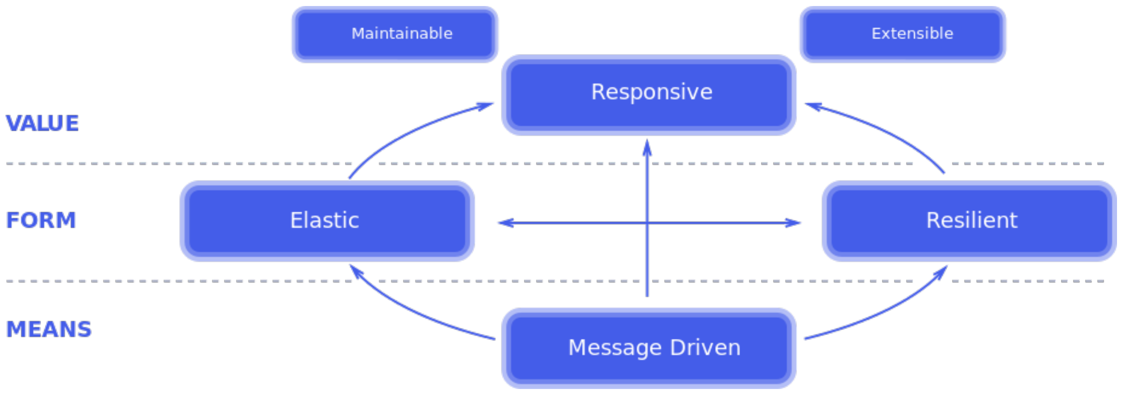


Figure 1.5: Reactive systems' core principles.[26]

The Reactive Manifesto lays out four key high-level characteristics of reactive systems:

- **Responsiveness**: a system should provide fast and consistent response times. It makes error detection simple and errors easier to handle. In return this will create a better user experience and thereby the consumers will continue to use the service.[8, p. 21]
- **Resilience**: it is about Responsiveness under failure and is an inherent functional property of the system, something that needs to be designed for, and not something that can be added in retroactively.[5]

  For a system that should always be available it is necessary that a failure does not affect availability. There are four characteristics for a resilient system: replication, containment, isolation and delegation. By *replication*, a component can be executed simultaneously in for example different threads or network nodes. *Isolation* can be achieved by *decoupling*, which can be in time or space (decoupling *in time* means that a sender and a receiver communicate asynchronously; decoupling *in space* means that the sender and the receiver are running in different nodes of the network). By *delegating* tasks to other components, the original component can oversee the progress and handle failures or perform other tasks during the time.[8, p. 21]

- **Elasticity**: it is about Responsiveness under load, meaning that the throughput of a system scales up or down (i.e. adding or removing cores on a single machine) as well as in or out (i.e. adding or removing nodes/machines in a data center) automatically to meet varying demand as resources are proportionally added or removed. It is the essential element needed to take advantage of the promises of Cloud Computing: allowing systems to be resource efficient, cost-efficient, environmentally-friendly and pay-per-use.[5] Elasticity facilitates dynamic resource allocation thereby effectively removing bottlenecks.
- **Message Driven**: components from a reactive system interact using asynchronous message passing to enable loose coupling, isolation and location transparency.[9]

  Location transparency is defined as the ability to access objects without the knowledge of their location[27]: the user should not see any difference whether the system runs on one or several nodes. In terms of scaling, this means no difference is made between scaling vertically and horizontally. Asynchronous message passing helps with delegation of failures and provides the possibility of elasticity and responsiveness by overseeing the message queues and control of back-pressure. If the pace of incoming messages for a component is too high to process it needs to be communicated in a suitable way to the publisher. It is not acceptable for a component to start dropping messages or to stop working. The component should instead communicate this upstream to decrease the pace of messages. This mechanism of handling pressure is called *back-pressure.*[8, p. 21]

1.4 Reactivity in Java

Actually, Java is not a "reactive language", in the sense that, differently from other programming languages on the JVM (e.g., Scala, Clojure), it doesn't support reactive models natively.[11]

However, asynchronous processing in Java is often necessary to make our systems fast and responsive, especially when we need to deal with huge volumes of data or multi-userness, but it can be tricky to realize, and the code may become hard to understand and maintain.[18]

With its latest releases (starting with Java 8), Java itself has made some attempts to introduce built-in reactivity, but these attempts are not very popular with developers to date yet.[18] Fortunely, Java is a powerhouse of enterprise development, and there has been a lot of activity recently in providing Reactive layers on top of the JDK, particularly appreciated and utilized by Java developers.[11]

Reactive Streams

Reactive Streams is an initiative created as a standard to unify reactive extensions and deal with asynchronous stream processing with non-blocking backpressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols.

Reactive Streams' main objective is to control the exchange of stream data across an asynchronous boundary while ensuring that the receiving side is not forced to buffer arbitrary amounts of data (backpressure is an integral part of this model). The benefits of asynchronous processing would be annulled if the communication of back-pressure were synchronous, therefore care has to be taken to mandate fully non-blocking and asynchronous behavior of all aspects of a Reactive Streams implementation.[28] The reactive streams APIs have been standardized in the JDK starting from version 9.[9]

Reactive Streams offers four interfaces, that are also implemented by other extension frameworks (like RxJava, Reactor, Akka):

- **Publisher**, that represents the data producer/data source; there's one method, *subscribe()*, that lets the subscriber register to the publisher.
- **Subscriber**, that is the consumer and has the following methods:
  - *onSubscribe()*, used to pass a Subscription object from the Publisher to the Subscriber;
  - *onNext()*, used to signal that a new item has been emitted;
  - *onError()*, used to signal that the Publisher has encountered a failure and no more items will be emitted;
  - *onComplete()*, used to signal that all items were emitted sucessfully.
- **Subscription**, that holds methods that makes the client able to control the Publisher's emission of items (i.e., providing backpressure support):
  - *request()*, that allows the Subscriber to inform the Publisher on how many additional elements to be published;
  - *cancel()*, that allows the Subscriber to cancel further emission of items by the Publisher.[3]
- **Processor**: it is an intermediary between Publisher and Subscriber. It subscribes to a Publisher and then a Subscriber subscribes to Processor.[9]

However, the streams typically need to be manipulated by maps, filters, flatMaps and so on, but users are not meant to implement reactive streams APIs directly for Reactive Streams; as a consequence, the implementations have to be provided by third party libraries such as Akka Streams, RxJava, or Reactor.[9]

RxJava

In 2011, the Reactive Extensions (ReactiveX or Rx) library was released by Microsoft for .NET, with the aim to provide an easy way to create asynchronous, event-driven programs. In a few years time, Reactive Extensions was ported to several languages and platforms including Java, JavaScript, C++, Python and Swift, quickly becoming a cross-language standard. The development of RxJava was driven by Netflix and version 1.0 was released in 2014.

ReactiveX is a combination of the best ideas from the Observer pattern and the Iterator pattern from the Gang of Four[1], and functional programming:[3]

- **Observer Pattern**: it is a behavioral design pattern, concerned with communication between Classes/Objects. *Subject* and *Observer* are two concepts strictly related to the observer pattern: an Observer watch or observe a Subject, and get notified and updated automatically whenever this subject change its state.
- **Iterator Pattern**: it is a behavioral design pattern, too, used to access the objects of a collection in a sequential manner without exposing its internal structure, aiming to keep track of the current item and to identify what elements are next to be iterated. It has two methods:
  - *next()*, that returns the next value from the current position;
  - *hasNext()*, that returns true if the current item is the last one.[29]

Rx is a non-blocking I/O and lets no worry about low-level threading, synchronization, thread-safety and much more: as the tasks runs on the background thread, this will allow us to perform smooth and continuous user experience tasks without blocking the main thread.

In Rx, everything is a data stream (that can be events, http calls, variable changes or even errors), and whenever a value is emitted to the stream some actions will take place. Data streams can be observed, or it is possible to iterate over and make some operations on them using operators (like filter, combine, etc.).[29]

Rx is mainly described by three important key concepts:

[1]Gang of Four, often abbreviated in GoF, is used to refer to the authors of the book: *Design Patterns: Elements of Reusable Object-Oriented Software.*

Figure 1.6: Some well-known services adopting ReactiveX.[30]

- **Observables**: it is an object (i.e., a data stream) an Observer can subscribe to, so that it can react to whatever item or sequence of items is emitted.

  It's important to understand the behavior of the Observables in order use the full potential of Rx:

  - *Hot Observables* start emitting data without waiting that a subscription is made. Moreover, data are shared among multiple Observers;
  - *Cold Observables* start emitting data only after a subscription is made, and don't share data among multiple Observers. A cold Observable can become hot by invoking the method *share()*, that will allow to share the stream between multiple Observers.

- **Observers**: also called *Subscriber*, it is connected to the Observable through the method *subscribe()*. This method subsets three other methods: *onNext()*, *onError()*, *onCompleted()* (refer to the previous subsection for further explanations about them).

  A *Subject* is special type of reactive that can be an Observable and an Observer (it can emit values or events, accept subscriptions and add new elements into the sequence).

- **Schedulers** simplifies the process of managing complex threads, allowing to add threading to Observables and Observers. For this purpose, Rx introduces two main operators:

  - *subscribeOn()* allows to indicate on which thread the Observable should operate;
  - *observeOn()* specifies on which Scheduler an Observer should observe the Observable.[29]

Project Reactor

Project Reactor (or just Reactor) is a Java framework from the Pivotal open source team (i.e., Spring developers): it represents the foundation of the reactive stack in the Spring ecosystem and is featured in projects such as Spring WebFlux, Spring Data, and Spring Cloud Gateway.[11] This library helps in building non-blocking applications on the JVM, supporting natively the back-pressure.[3]

It is quite similar to RxJava, but has simpler abstraction, and won popularity due to the possibility to leverage benefits of Java 8.[18] Indeed, it integrates directly with the Java 8 functional APIs, notably *CompletableFuture*, *Stream*, and *Duration*. It offers composable asynchronous sequence APIs: Flux (for N elements) and Mono (for 0 or 1 elements), and extensively implements the Reactive Streams specification.[31] It also provides wrappers around low-level network runtimes, like Netty and Aeron.[11]

Spring Framework 5.0

Spring Framework 5.0 (first milestone: June 2016) has reactive features built into it, including tools for building HTTP servers and clients. It builds on Project Reactor, but also exposes APIs that allow its features to be expressed using a choice of libraries (e.g., Reactor or RxJava); moreover, it introduces a dedicated reactive web framework, called Spring WebFlux. Users can choose from Tomcat, Jetty, Netty (via Reactor IO) and Undertow for the server side network stack.[11]

Being fully reactive and non-blocking, this stack is suitable for event-loop style processing that can scale with a small number of threads.[32]

Ratpack

Ratpack comprises a set of libraries for creating highly performant services over HTTP. It build on Netty and implements Reactive Streams. Furthermore, it supports Spring natively, and can be embedded by Spring Boot users inside a Spring application, bringing up an HTTP endpoint and listening there instead of using one of the embedded servers supplied directly by Spring Boot.[11]

Akka

Akka is an open-source library that helps to easily develop concurrent and distributed applications by leveraging the *Actor Model* in Java or Scala.[33]

The Actor Model was originally created by Carl Hewitt in 1973, but recently has gained popularity for concurrency. An application is broken up into small units of execution, called *actors*.[34] An actor represents an independent computation unit,[33] and encapsulates the behaviour and the state of a part of the application.

Actors in Akka are extremely lightweight, since they are standard objects that only consume a small quantity of memory. The state of an actor is isolated from the rest of the system (no worries about synchronization issues).

Actors communicates with each other via messages, that can be of arbitrary type.[34] One of the advantages of using message instead of method calls is that the sender thread won't block to wait for a return value when it sends a message to another actor.[33] All communication with actors is done through asynchronous message passing (and this is what makes actors reactive).

It is possible to combine actors into a supervisory hierarchy, that allows to make the application fault-tolerant and scalable.[34] Organizing the actors in a hierarchy allows to facilitate the error handling: each actor can notify its parent of a failure, so it can act accordingly (at that point, the parent actor can decide to stop or restart the child actors).[33]

The strong isolation principles, the message-driven model and location transparency typical of this model helps in solving hard concurrency and scalability problems intuitively, allowing the programmer to concentrate on building the application and not the infrastructure.[34]

Chapter 2

Spring Boot Reactive, aka WebFlux

This second chapter will be focused particularly on Spring Boot Reactive, better known as Spring WebFlux.

We will start with a comparison between the two Spring versions, the imperative and the reactive one (used to implement the applications described in detail in the next chapter), trying to understand the differences between the homonymous programming paradigms they refer to and the threading approaches they use. Then, we'll describe Spring WebFlux in full, starting from the Project Reactor library and the interfaces it offers and proceeding with all the peculiar characteristics that make this framework the ideal choice for building scalable and powerful reactive applications.

2.1 Imperative vs Reactive Programming

Imperative and reactive programming can be considered two different programming paradigms.

Imperative programming basically means that the code gets executed in order, statement by statement: the order doesn't imply that we cannot use conditionals, loops or function calls anymore, but if the debugger can clearly point to a statement in the code, it's also clear what will be the next line where it will jump to. Shortly, imperative programming is the way we most likely write the code everyday.[35]

Imperative programming is practically the basis of any programming language and even machine language is imperative. Other programming paradigms incorporates an imperative style internally, abstracting this out for the caller. Moreover, imperative style is highly sequential and so easy to write, understand and debug.[36]

On the other hand, reactive programming, extensively discussed in the previous chapter and often coupled with functional programming, consists in dealing with asynchronous events data streams over time (that can be created, changed or combined along the way) and propagation of changes.[35][36]

Furthermore, in synchronous and imperative code, blocking calls serve as a natural form of backpressure that forces the caller to wait; in non-blocking code, instead, it becomes important to control the rate of events so that a fast producer does not overwhelm its destination.[37]

The applications we will go into in depth in the next chapters have been developed using Spring Boot and Spring WebFlux, that are respectively the imperative and the reactive versions of Spring.

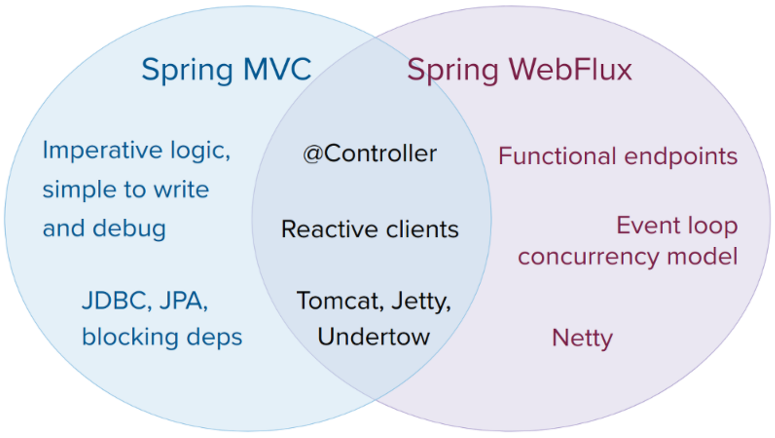


Figure 2.1: Imperative and Reactive technologies in Spring.[37]

In Spring Boot, all the code is written with an imperative approach and calls to external systems are blocking. On the contrary, in Spring WebFlux all the logic is written in a reactive way, guaranteeing a non-blocking end-to-end flow: in particular, we have used Netty for the HTTP part and R2DBC as connector, that is a non-blocking version, opposed to JDBC which instead is blocking.[17]

2.1.1 Thread per Request Model in Spring Boot

Traditional Rest-APIs handle concurrent requests exploiting the *Thread per Request Model*, whose limit depends on the thread pool size (default value for a Tomcat server is 200 connections). For overcoming this limit, we usually opt for spinning up more instances of the application (i.e., horizontal scaling), even if this would imply deploying parts of the app we're not interesting in. Such APIs are blocking and synchronous.

Suppose we make a HTTP GET request to the server: whenever it takes a request, a new servlet thread is allocated for serving that request.[38] This thread will have to serve the task that is assigned to itself throughout its life cycle: indeed, if the task executes a blocking call (e.g., to a db or an external service), the thread is also put on hold, and only when the blocking call terminates the thread will be able to continue the task execution. The thread will be released only when the whole task has finished. What stands out is that we have a 1 to 1 ratio between task and thread.[17]

We will be able to serve 200 concurrent requests at most: the eventual 201th request will have to wait until a thread will be freed and available for that.[38]

Spring Boot adopted a synchronous programming model: blocking requests will block the threads, too, and there's no CPU utilization in this period, that's the reason why this model uses a large thread pool for request processing. This could make applications with a high request rate slow or unresponsive.

Spring Boot usually has Tomcat as its default embedded server which uses the Thread Per Request Model. This model was adopted by Servlet Containers to handle incoming requests, and it increases the resource utilization.

Spring WebFlux allows to use Servlet Containers as well (but only if they implement Servlet 3.1+ APIs). Tomcat is the most commonly used Servlet Container and it supports the reactive programming as well. If we choose to use Spring WebFlux on Tomcat, we will obtain the same situation of synchronous processing with the difference that, in a reactive approach, the user can delegate requests to another pool of worker threads (as will be discussed in the next subsection about Event Loop). In this way, requests processing threads become available and the applications remains responsive.[39]

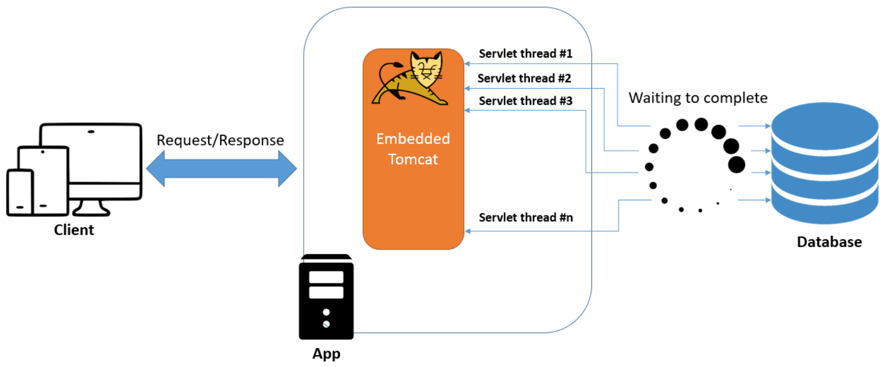


Figure 2.2: Example of Thread per Request Model in Spring Boot.[38]

2.1.2 Event Loop in Spring WebFlux

A considerable advantage would derive from the possibility, for a thread, to process the request avoiding to wait for the response from the I/O operations, rather we would like to get notified once that process is done: this means that the thread will be immediately freed and able to serve the next request. Once the response will be ready, the I/O device (e.g., a db or an external service) will make a callback and any thread that is free will take that data and send back to the user.[38]

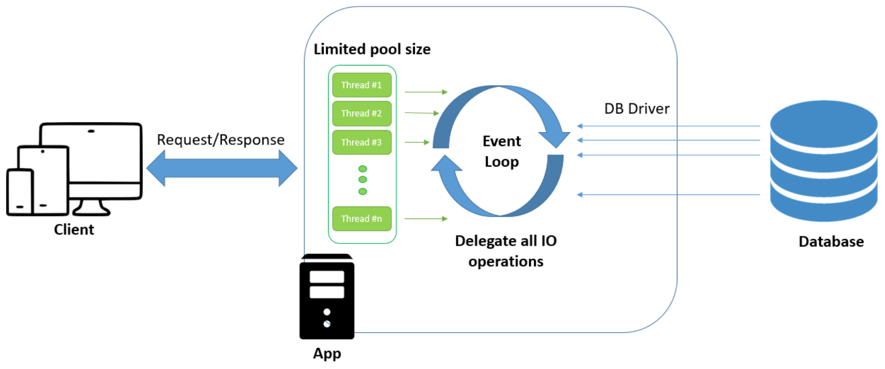


Figure 2.3: Example of Event Loop in Spring WebFlux.[38]

This mechanism is called *Event Loop*, one of the most widely used patterns in reactive programming, used to prevent servlet threads from being blocked.

Event Loop is a Non-Blocking I/O Thread (NIO) which runs continuously and has the burden of managing the input requests and treats each of these as being an event related to a listener. It also has the role of deciding to whom to forward calls traveling in the direction of external systems. In this case, first it will be checked if there is already a listener representing this service to be reused, otherwise a new one will be created.

The event loop decides the thread scheduling based on the needs of the external reactive service or the asynchronous library and will do its job without blocking the loop. In the case of the arrival of a new request, this will enter the event loop that can decide whether to use the same listener created by a previous request or not. Once the external system finishes its processing and returns a response, this will be handled again by the event loop that will know who to send the information to.

The advantage is that there are no threads that remain blocked waiting for the answer but there will be a callback that will be executed as soon as the response will be available to the caller, going to awaken that part of code that had started the asynchronous call. As a consequence, this mechanism allows the number of concurrent requests to grow up and leads to a more efficient CPU usage.[17]

Spring Webflux has been introduced as part of Spring 5 and with this it started to support reactive programming, adopting an asynchronous, non-blocking programming model, similar to the one supported by Node.js, but with multiple event loops, moving away from the Thread per Request blocking model of Spring Boot.[40] The use of reactive programming makes applications highly scalable to support high request load over a period of time.

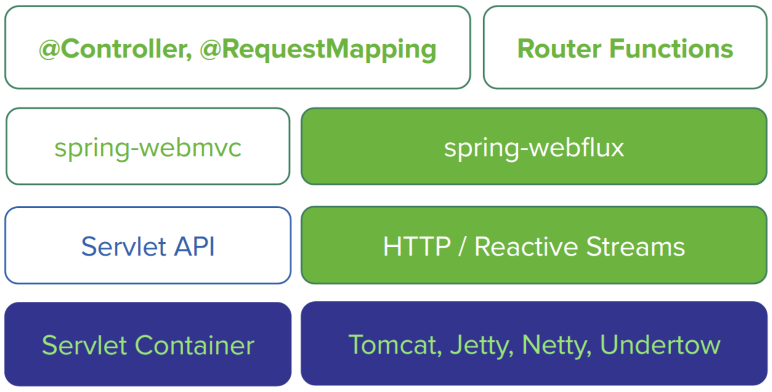


Figure 2.4: The server-side stack of Spring Boot and Webflux.[41]

By default, it has Netty as an embedded server, but it is also supported on Tomcat, Jetty, Undertow and other Servlet 3.1+ containers (note that Netty and Undertow are non-servlet runtimes and Tomcat and Jetty are well-known servlet containers). Netty has been preferred as default which uses the Event Loop Model with backpressure support, providing highly scalable concurrency in a reactive asynchronous manner.[39]

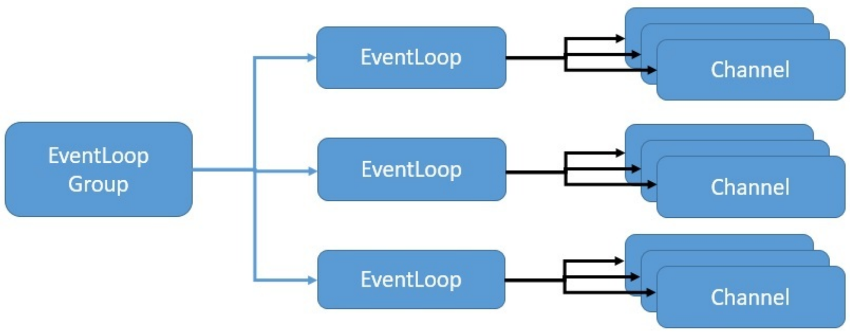


Figure 2.5: Netty threading model.[42]

In Figure 2.5, we can see how EventLoop Group manages one or more EventLoop which must be continuously running (hence, it's not recommended to create more EventLoops than the number of cores available in the machine). The EventLoop Group further assigns an EventLoop to each newly created channel, so, for the lifetime of a channel, all operations are executed by the same thread.[42]

2.2 WebFlux and Project Reactor

Spring Webflux, added starting from Spring 5.0, is a reactive-stack web framework that is fully non-blocking and runs on such servers as Netty, Undertow, and Servlet 3.1+ containers.

The Spring reactive stack is build on Project Reactor, a reactive library that implements backpressure and is compliant with the Reactive Streams specification.[43]

Spring WebFlux allows to more efficiently leverage CPU and network resources, providing a more scalable architecture and a more responsive user experience.[40] Furthermore, it offers the possibility to handle concurrency with a smaller number of threads and to scale with fewer hardware resources with respect to traditional Spring Boot framework, and differently from the latter one it also supports functional programming.[44]

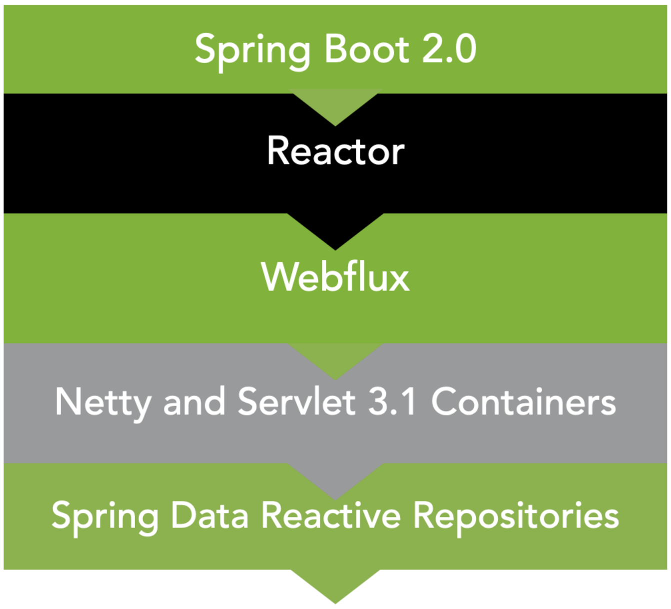

Spring Boot 2.0
Reactor
Webflux
Netty and Servlet 3.1 Containers
Spring Data Reactive Repositories

Figure 2.6: Spring Reactive stack.[40]

Spring WebFlux and Reactor together constitute the perfect combination that allows to build applications that are:

- **Non-blocking**, being able to accept further requests while waiting for a response from a remote component or service related to the current request;

- **Asynchronous**, responding to events from an event stream by generating response events and publishing them back to the event stream;

- **Event-driven**, reacting to events generated by the user or by another service (e.g., mouse clicks);

- **Scalable**, being capable of handling large numbers of events using less computing and networking resources with respect to other application programming models;

- **Resilient**, managing efficiently possible services' failures.[43]

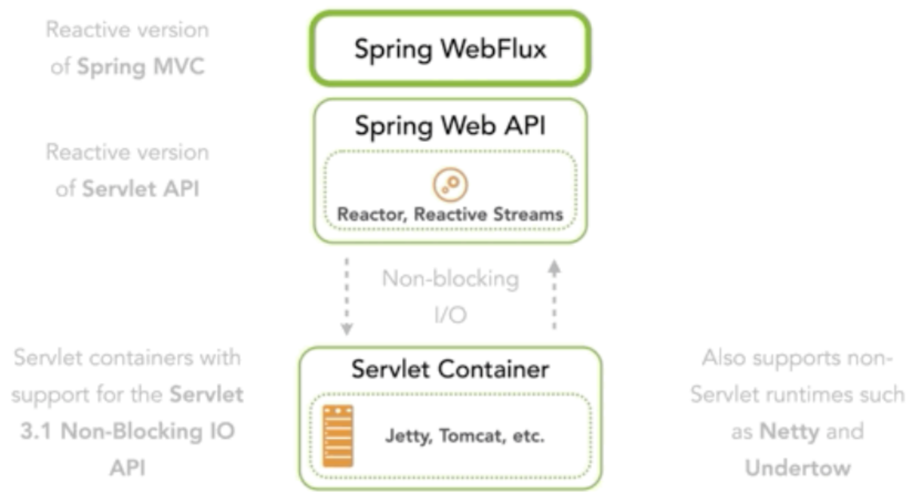

Reactive version of Spring MVC
Spring WebFlux
Spring Web API
Reactive version of Servlet API
Reactor, Reactive Streams
Non-blocking I/O
Servlet containers with support for the Servlet 3.1 Non-Blocking IO API
Servlet Container
Jetty, Tomcat, etc.
Also supports non-Servlet runtimes such as Netty and Undertow

Figure 2.7: Spring WebFlux in action.[40]

2.2.1 Reactive API

As already said, Project Reactor library implements the specifications of the Reactive Streams initiative, whose aim is to provide a standard for asynchronous stream processing with non-blocking backpressure.[45] The reactive API defines four main components (or interfaces): *Publisher*, *Subscriber*, *Subscription*, and *Processor*.

Publisher

The *Publisher* is the data source that provides a finite (or potentially infinite) sequence of elements or events to its Subscribers (they can be more than one). In order to support backpressure, it emits events according to the demands of the Subscriber. It has only one method, *subscribe()*, invoked by Subscribers in order to subscribe to the Publisher.[38][46]

```java
package org.reactivestreams;

public interface Publisher<T> {

    public void subscribe(Subscriber<? super T> s);
}
```

Listing 2.1: Publisher interface.

Subscriber

The *Subscriber* is, shortly, the consumer: it represents the receiver and controller of the reactive communication, since it decides when and how many elements it is able and willing to receive from the Publisher. A Subscriber subscribes to a Publisher by invoking the method *onSubscribe(Subscription)*.

```java
package org.reactivestreams;

public interface Subscriber<T> {

    public void onSubscribe(Subscription s);

    public void onNext(T t);

    public void onError(Throwable t);

    public void onComplete();
}
```

Listing 2.2: Subscriber interface.

- Demand: *request(long)* is called by the Subscriber in order to receive notifications from the Publisher, by specifying the number of events it wants to process (whenever it wants to demand more events, it has just to call the same method multiple times);
- Receive: *onNext(object)*, invoked one or more times, is used to receive events;

```
package org.reactivestreams;

public interface Subscription {

    public void request(long n);

    public void cancel();
}
```

Listing 2.3: Subscription interface.

- Error: *onError(Throwable)* is called in order to receive exception details in case of errors;
- Finish: *onComplete()* is invoked at the end of the event sequence.[46]

Subscription

The *Subscription* can be intended as a unique relationship between a Subscriber and a Publisher[38]: indeed, it represent a one-to-one life cycle between the two. If a Publisher has multiple Subscribers, there will be a separate Subscription of each of them.

A Subscriber uses a Subscription for signaling demand for events using the method *request(long)*; it uses *cancel()*, instead, for cancelling an existing demand.[46]

Processor

The *Processor* represents a processing stage, which is in the middle ground between Subscriber and Publisher and must obey the contracts of both.[38] It extends both Subscriber and Publisher and has to provide implementations for them.[46]

```
package org.reactivestreams;

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> { }
```

Listing 2.4: Processor interface.

Reactive Streams Data Flow

When dealing with reactive streams, the data flow is as follows:

1. The *subscribe()* method is called on a Publisher instance;

2. A Subscription object is created, and then it is passed to the *onSubscribe()* method of the Subscriber;

3. A Subscriber calls the *request()* method specifying the number of objects it can process (if not specified, the Publisher will return all the data by default). It is a simple example of backpressure, because the consumer controls the size of the response;

4. The Subscriber can receive objects through the *onNext()* method. It also has the possibility of cancelling the data flow through the *cancel()* method, after which the Publisher won't emit any more events;

5. Once all the data is retrieved, the Publisher sends an *onComplete()* event, except if there is an error, case in which it calls instead the *onError()* method.[45][38]

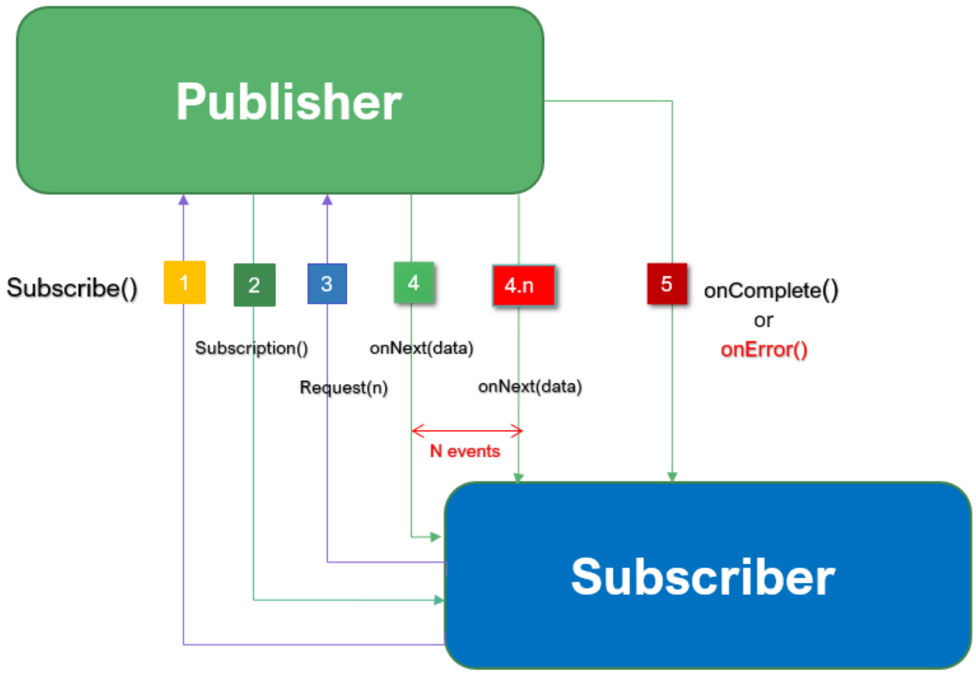


Figure 2.8: Reactive Streams Data Flow.

2.2.2 Mono and Flux

In reactive-core module, we find *Mono* and *Flux*, that represent Reactive Stream Publishers, i.e. Project Reactor's Publisher implementations.[38]

A **Mono** is a Publisher that publishes a single element (or zero): it will signal *onNext()* to deliver the event and then call *onComplete()* to signal the termination. In case of error, it will signal *onError()* without sending any event.[46]

Mono is used when you want to retrieve just one item from the data source (e.g., retrieving the user's information by id).[38]

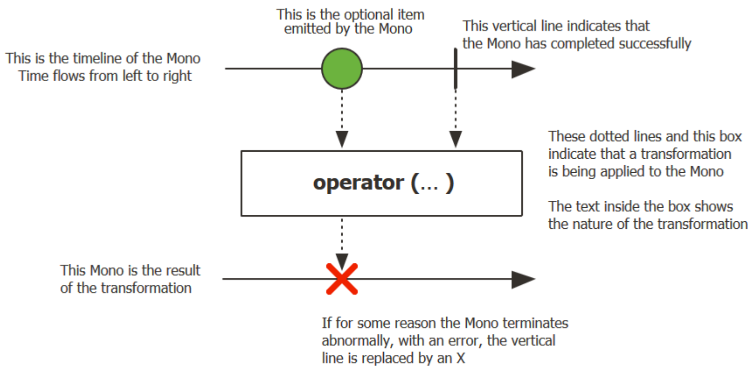

This is the optional item emitted by the Mono
This vertical line indicates that the Mono has completed successfully
This is the timeline of the Mono Time flows from left to right
These dotted lines and this box indicate that a transformation is being applied to the Mono
operator (...)
The text inside the box shows the nature of the transformation
This Mono is the result of the transformation
If for some reason the Mono terminates abnormally, with an error, the vertical line is replaced by an X

Figure 2.9: Mono's timeline.[38]

A **Flux** is a Publisher that can emit from 0 to N elements. As Mono, it can complete successfully or with an error.[38]

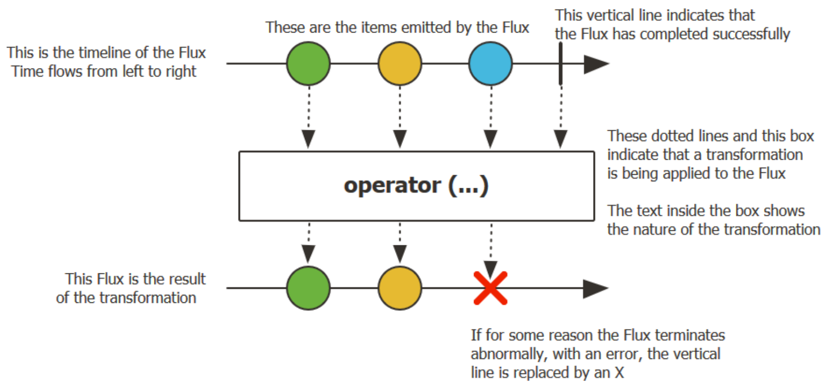

These are the items emitted by the Flux
This vertical line indicates that the Flux has completed successfully
This is the timeline of the Flux Time flows from left to right
These dotted lines and this box indicate that a transformation is being applied to the Flux
operator (...)
The text inside the box shows the nature of the transformation
This Flux is the result of the transformation
If for some reason the Flux terminates abnormally, with an error, the vertical line is replaced by an X

Figure 2.10: Flux's timeline.[38]

Based on the their emission behavior, we can distinguish two types of publishers:

- **Cold Publisher**: it doesn't produce any value unless at least one Subscriber subscribes to it. A dedicated data producer is created for each new subscription.[47] In this case, subscription generates data: as Reactor says, "*nothing happens until you subscribe*".[48]

- **Hot Publisher**: there is one single data producer for all the Subscribers listening to the data produced by it, so getting the same data.[47] Hot Publishers can publish data all the time without caring whether any Subscriber is there or not.[48]

2.2.3 Backpressure

In Reactive Streams, *backpressure* defines how to regulate the transmission of stream elements (i.e., how to control how many elements the recipient can consume).

Spring WebFlux works with asynchronous non-blocking flows of Reactive Streams and leaves the responsibility of handling backpressure to Project Reactor, that internally uses Flux functionalities in order to apply control mechanisms to the events produced by the emitter.

WebFlux adoperates TCP flow control to adjust the backpressure in bytes, but this flow control is still for bytes rather than for logical elements, so it doesn't handle the logical elements the consumer can receive. Let's analyze the process in detail:

- WebFlux must convert events into bytes in order to transfer/receive them through TCP;

- The subscriber could start a long-running job before requesting the next logical element;

- While the receiver is processing the events, WebFlux enqueues bytes without acknowledgment, because there is no demand for new events;

- Due to the nature of the TCP protocol, if there are new events, the publisher will continue sending them to the network.[49]

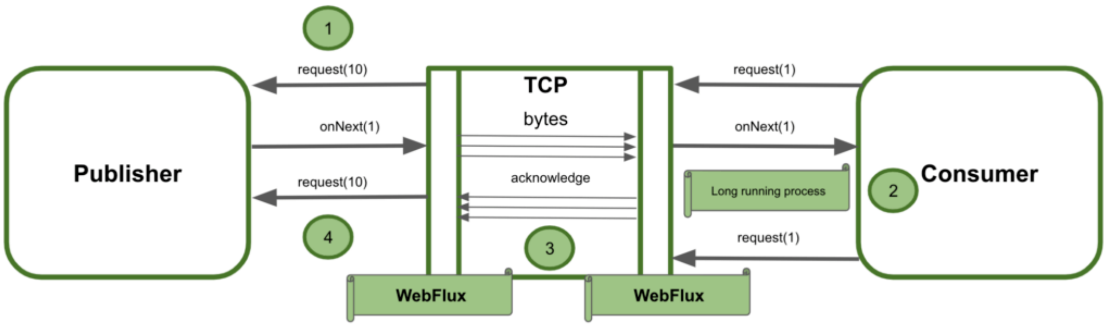


Figure 2.11: Backpressure handling in WebFlux.[49]

In Figure 2.11, we can note that the demand in logical elements may be different for the subscriber and the publisher. WebFlux, differently from what we could expect, ideally doesn't manage the backpressure between the services that are interacting as a unique system, rather it handles that with the subscriber and the publisher independently, without taking into account the logical demand between the two services.[49]

In order to limit the number of items being processed is through the implementation of a custom *BaseSubscriber*, that will deal with requesting data as necessary.[48] The basic idea could be as follows:

```java
public class BackpressureReadySubscriber<T> extends BaseSubscriber<T> {

    public void hookOnSubscribe(Subscription subscription) {
        request(1); //requested the first item on subscribe
    }

    public void hookOnNext(T value) {
        //process value
        //processing...
        //once processed, request a next one
        //(implement specific logic to slow down processing here)
        request(1);
    }
}
```

Listing 2.5: Example of a custom Subscriber for better handling backpressure.[48]

Then, it is possible to subscribe to a source of this custom tipe.

When manually creating a Subscriber, be aware of requesting enough data so that the Flux does not get stuck: it's necessary to have at least one *request()* being called from the *hookOnNext()* method.[48]

2.2.4 Programming models

Spring WebFlux supports two different programming models:

- **Annotation-based Controllers**: it will look quite familiar, since it uses the same annotations from the Spring Web module that is used in Spring Boot. The only difference is that the methods here return the reactive types Mono and Flux.[50] WebFlux also supports reactive *@RequestBody* arguments.[44]

```
@RestController
@RequestMapping("/students")
public class StudentController {

    @Autowired
    private StudentService studentService;

    @GetMapping("/{id}")
    public Mono<ResponseEntity<Student>> getStudent(@PathVariable long
   id) {
        return studentService.findStudentById(id)
                .map(ResponseEntity::ok)
                .defaultIfEmpty(ResponseEntity.notFound().build());
    }
}
```

Listing 2.6: Example of a RestController using the annotation-based model.[50]

- **Functional Endpoints**: it is a lambda-based, lightweight and functional programming model, that leaves the application in charge of the full request handling. It can be considered as a small library used for routing and handling requests. The big difference with annotated controllers is that the application is in charge of request handling from start to finish versus declaring intent through annotations and being called back.[44]

  It is based on the concepts of *HandlerFunctions* and *RouterFunctions*:[50]

  - **HandlerFunctions** are used to generate a response for a given request.
  - The **RouterFunction** is used to route the requests to the HandlerFunctions. It has a similar purpose as a *@RequestMapping* annotation, except for an important distinction: with the annotation your route is limited to what can be expressed through the annotation values, and the processing of those is not trivial to override; instead, with RouterFunctions, the processing code can be overridden or replaced quite easily.[51]

```java
@Component
public class StudentHandler {

    @Autowired
    private StudentService studentService;

    public Mono<ServerResponse> getStudent(ServerRequest serverRequest)
   {
        Mono<Student> studentMono = studentService.findStudentById(
                Long.parseLong(serverRequest.pathVariable("id")));
        return studentMono.flatMap(student -> ServerResponse.ok()
                .body(fromValue(student)))
                .switchIfEmpty(ServerResponse.notFound().build());
    }
}
```

Listing 2.7: Example of HandlerFunction.[50]

```java
@Configuration
public class StudentRouter {

    @Bean
    public RouterFunction<ServerResponse> route(StudentHandler
   studentHandler) {

        return RouterFunctions
            .route(
                GET("/students/{id:[0-9]+}")
                    .and(accept(APPLICATION_JSON)),
                    studentHandler::getStudent);
    }
}
```

Listing 2.8: Example of RouterFunction.[50]

2.2.5 Operators

Working with Reactive Streams, it will surely happen to deal with *operators* in order to modify and transform them. Let's try to understand some of the most common operators provided by Project Reactor and analyze how to use them through examples.

Filter

It evaluates each element of the Flux against a given predicate, returning a new Flux containing only values that pass the predicate test.[31]

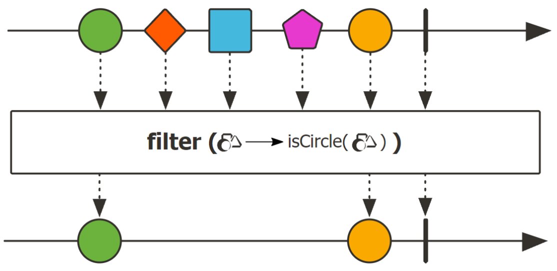


Figure 2.12: Filter operator.[31]

```java
@Test
public void testFilterOperatorFlux() {
    /* Create a Flux from a list of Integer numbers from 1 to 8 */
    Flux<Integer> initials = Flux.just(1, 2, 3, 4, 5, 6, 7, 8);
    /* Apply filter for even numbers only */
    Flux<Integer> finals = initials.filter(i -> i % 2 == 0);
    StepVerifier
            .create(finals)
            .expectNext(2, 4, 6, 8)
            .verifyComplete();
}
```

Listing 2.9: Example of usage of the Filter operator.[52]

Map

It transforms the items emitted by the Flux by applying a synchronous function to each element.[31]

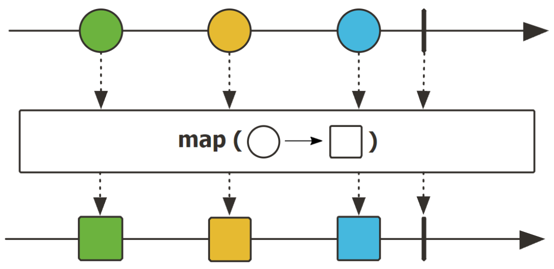


Figure 2.13: Map operator.[31]

```java
@Test
public void testMapOperatorFlux() {
    /* Create a Flux from a list of random Strings */
    Flux<String> initials = Flux.just("FirstString", "SecondString");
    /* Apply map returning a flux made by the lengths of the Strings */
    Flux<Integer> finals = initials.map(i-> i.length());
    StepVerifier
            .create(finals)
            .expectNext(11, 12)
            .verifyComplete();
}
```

Listing 2.10: Example of usage of the Map operator.[52]

FlatMap

It transforms the elements emitted by the Flux asynchronously into Publishers, then flatten these inner Publishers into a single Flux through merging, which allow them to interleave.[52]

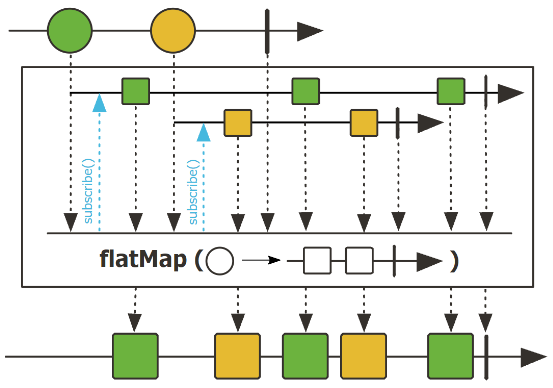


Figure 2.14: FlatMap operator.[31]

```java
@Test
 public void testFlatMapOperator() {
    /* Create a Flux from a list of Integer numbers from 1 to 3 */
    Flux<Integer> initials = Flux.just(1, 2, 3);
    // Range emits a sequence of incrementing Integers starting from i
    // Then flatMap converts all the Flux into a single Flux
    Flux<Integer> finals = initials.flatMap(i -> Flux.range(i, 10));
    StepVerifier
            .create(finals)
            .expectNextCount(27)
            .verifyComplete();
}
```

Listing 2.11: Example of usage of the FlatMap operator.[52]

Merge

It merges data from Publisher sequences contained in an Iterable into an interleaved merged sequence. Unlike *concat*, inner sources are subscribed to eagerly.[52]

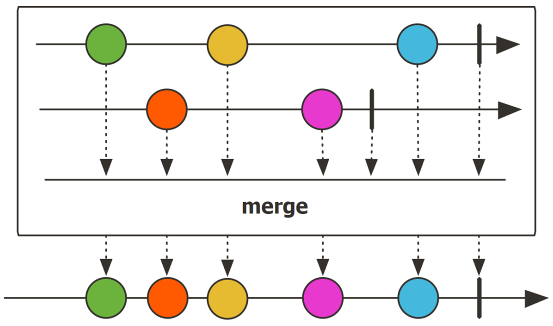


Figure 2.15: Merge operator.[31]

```
@Test
public void testMergeOperator() {
    /* Create a first Flux from a list of Integer odd numbers */
    Flux<Integer> odds = Flux.just(1, 3, 5, 7);
    /* Create a second Flux from a list of Integer even numbers */
    Flux<Integer> evens = Flux.just(2, 4, 6, 8);
    /* Apply merge returning a merged Flux */
    Flux<Integer> finals = Flux.merge(odds, evens);

    StepVerifier
            .create(finals)
            .expectNextCount(8)
            .expectComplete()
            .verify();
}
```

Listing 2.12: Example of usage of the Merge operator.[52]

Buffer

It collects all incoming values into a single List buffer that will be emitted by the returned Flux once it completes.[52]

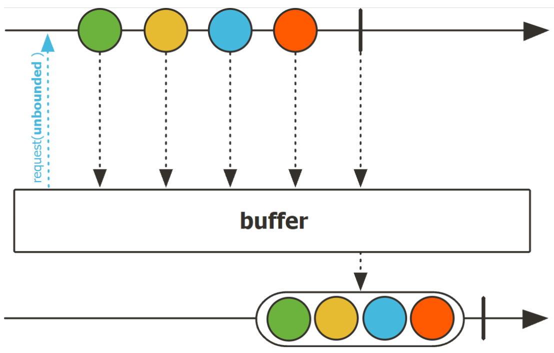


Figure 2.16: Buffer operator.[31]

```
@Test
public void testBufferOperator() {
    /* Create a Flux from a list of Integer numbers */
    Flux<Integer> initials = Flux.just(1, 2, 3, 4, 5, 6, 7);
    /* Apply buffer that returns a Flux of lists */
    /* Each list will contain two elements */
    Flux<List<Integer>> buffer = initials.buffer(2);

    StepVerifier
            .create(buffer)
            .expectNext(Arrays.asList(1, 2))
            .expectNext(Arrays.asList(3, 4))
            .expectNext(Arrays.asList(5, 6))
            .expectNext(Arrays.asList(7))
            .verifyComplete();
```

Listing 2.13: Example of usage of the Buffer operator.[52]

2.2.6 Schedulers

Reactive Programming really shines in a completely non-blocking environment with just a few threads, but if there were indeed a part that is blocking, it will result in far worse performance, since a blocking operation can freeze the event loop entirely. Spring WebFlux proposes a way to switch processing to a different thread pool in between a data flow chain, that can provide the user precise control over the scheduling strategy for each task.[42]

Even with dealing with a modern framework such as Project Reactor, there is no doubt that the work has to execute on some specific thread.[48] Reactor operators generally don't impose a particular threading model and just run on the thread on which their *onNext()* method was invoked.

In Reactor, a *Scheduler* is an abstraction that gives the user control about threading:[53] indeed, it is possible to choose which Worker threads have to be allocated (or which thread pool) for a certain task.

Reactor comes with several default Scheduler implementations:

- *Schedulers.immediate()*, used to keep the execution in the current thread;
- *Schedulers.single()*, that is a single reusable thread for all the callers;[54]
- *Schedulers.newSingle()*, that is a single, dedicated thread;
- *Schedulers.elastic()* and *Schedulers.boundedElastic()* are good for more long-lived tasks (such as network calls): the first one spawns threads on-demand without a limit, while the recently introduced *boundedElastic* does the same with a ceiling on the number of created threads;
- *Schedulers.parallel()*, good for CPU-intensive but short-lived tasks, can execute N such tasks in parallel (by default N corresponds to the number of CPU cores of the machine).[53]

While Schedulers offer several execution contexts, Reactor also provides different ways to switch the execution context through the convenient methods *publishOn* and *subscribeOn*.[42]

These methods accept any of the above Schedulers to change the task execution context for the operations in a reactive pipeline. While *subscribeOn* forces the source emission to use specific Schedulers, *publishOn* changes Schedulers for all the downstream operations in the pipeline as shown below.[54]

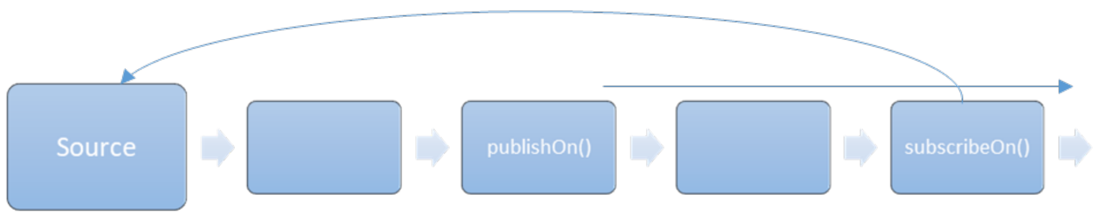


Figure 2.17: The effects of publishOn() and subscribeOn() in a pipeline.[54]

2.2.7 WebClient

Spring WebClient is a non-blocking and reactive client to perform HTTP requests, supporting Reactive Streams. WebClient has been added in Spring WebFlux, replacing the existing *RestTemplate* client, which was simple but blocking, soon to be deprecated in near future in favor of the new WebClient alternative.[55]

Using WebClient, it's possible to make either synchronous or asynchronous HTTP requests with a functional fluent API that can integrate directly into your existing Spring configuration and the WebFlux reactive framework:[56] indeed, even though it is a non-blocking client belonging to the Spring WebFlux library, the solution offers support for both synchronous and asynchronous operations, making it suitable also for applications running on a Servlet Stack (the code will wait for the request to finish before progressing further).[57] Having support to non-blocking Reactive Streams, the WebClient can decide whether to wait for the request to finish or proceed with other tasks.[55]

There are different ways for building a WebClient instance:

- Using the *create()* factory method, where it is possible to specify the base url of the upstream service;
- Employing the *WebClient.Builder* that, once created, can be reused to instantiate multiple WebClient instances, avoiding reconfiguring all the client instances.

Once the WebClient instance is created, it is possible to make HTTP requests (like GET, POST, etc.) through that. Remember the following points:

- Specify the URI of the target point;
- Add headers to the request and, if needed, also a request body;
- Finally, execute the request and check server response or errors. This can be done simply using the *retrieve()* method and then convert the response to a Mono or a Flux. Furthermore, it is possible to handle any server or client errors through the method *onStatus()*. Alternatively, for a more controlled response handling, we can use *exchange()* while executing the request[55]

```
WebClient webClient = WebClient.create("http://localhost:3000");

Employee createdEmployee = webClient.post()
                            .uri("/employees")
                            .header(HttpHeaders.CONTENT_TYPE,
                            MediaType.APPLICATION_JSON_VALUE)
                            .body(Mono.just(empl), Employee.class)
                            .retrieve()
                            .bodyToMono(Employee.class);
```

Listing 2.14: Example of creation and use of a WebClient instance for a POST API.[58]

```
@Autowired
WebClient webClient;

public Flux<Employee> findAll() {
    return webClient.get()
        .uri("/employees")
        .retrieve()
        .bodyToFlux(Employee.class);
}

public Mono<Employee> findById(Integer id) {
    return webClient.get()
                .uri("/employees/" + id)
                .retrieve()
                .onStatus(httpStatus ->
                    HttpStatus.NOT_FOUND.equals(httpStatus),
                    clientResponse -> Mono.empty())
                .bodyToMono(Employee.class);
}
```

Listing 2.15: WebClient GET API examples with error handling.[58]

Chapter 3

Rest-API implementations

In this third chapter we're going to analyze in detail the application that has been developed during this thesis work, focusing on each of the two versions: the first one is a traditional Spring Boot application, where Rest-APIs will be implemented both in a synchronous and asynchronous way; the second one is a Spring WebFlux application, fully based on reactive and functional programming.

The goal is to analyze and compare three different Rest-API implementations, in order to understand which one would be more suitable for better managing large amounts of data, also providing a more efficient thread scheduling in order to parallelize some computations.

3.1 Application overview

Before proceeding with the discussion about the specific implementations, let's present the scenario to which the application is related, highlighting also objectives and motivations; details about the architecture of the application, the database and the external geocoding service involved will follow.

3.1.1 Context, motivations, objectives

The application that has been designed and developed is placed in the context of the Connected Vehicle Technology and the Business Intelligence.

The company I worked for, Technology Reply, has realized for a well-known automotive company a system for collecting data sent from Connected Vehicles in a unique Corporate Data Lake, giving the possibility to process a large amount of data effectively and efficiently. Connected Vehicles data can now also be crossed with different data sources, such as the geographical position of the workshops and assistance interventions, thus allowing to detect the activities and conditions of the vehicles, enabling new paradigms of data analysis either descriptive or predictive, and providing added value to the analytics world.

This solution favors new business opportunities: the gathered information provides visibility on the use of a real vehicle in terms of time and activity and, through the use of Machine Learning techniques, this data can be used to conduct advanced analyses (predictive maintenance activities, customer segmentation, analysis of driving behavior) and, by exploiting the technical data of the vehicle, it is possible to anticipate the needs of the customers by offering them services tailored to real needs.

The realization of the solution envisaged the reception of multiple heterogeneous flows, coming from sources, both in real-time and batch mode. In particular, in addition to the flows of the system that collects data from the vehicle's control unit when it is switched on and off (fuel level, tire pressure, odometer value, etc.), many other sources containing the data are involved, collected during the real use of the vehicle, such as the status of the battery or the vehicle itself, data from remote operations carried out by the user and from subscriptions to services.

More than 100 GB of data from the Connected Vehicles are integrated into the Corporate Data Lake every day. The company adopted an innovative logical compression solution on the ingested data, in order to make it possible to process data in real-time mode with updating frequencies of business indicators several times a day.

Figure 3.1: A connected vehicle.

Each Connected Vehicle, uniquely identified by its VIN (i.e., Vehicle Identification Number), sends lots of different signal, some with particularly high frequency (about one per second). As a consequence, the amount of data collected every day is considerably large.

These signals are stored with the purpose of identifying common patterns (e.g., about driving or charging habits), allowing the business and the data scientist team to carry out deep dive analyses on the phenomena related to the connected vehicle. The necessity of reading and analyzing the collected data efficiently becomes evident and essential.

The objective of the application is the detailed analysis of a single vehicle or driver in order to categorize it on the basis of the received signals, by integrating them with information retrieved in real-time from external systems.

In particular, for the purpose of the thesis, it has been decided to consider only four types of signals: the vehicle speed retrieved from the GPS, the altitude, the latitude and the longitude.

3.1.2 The architecture

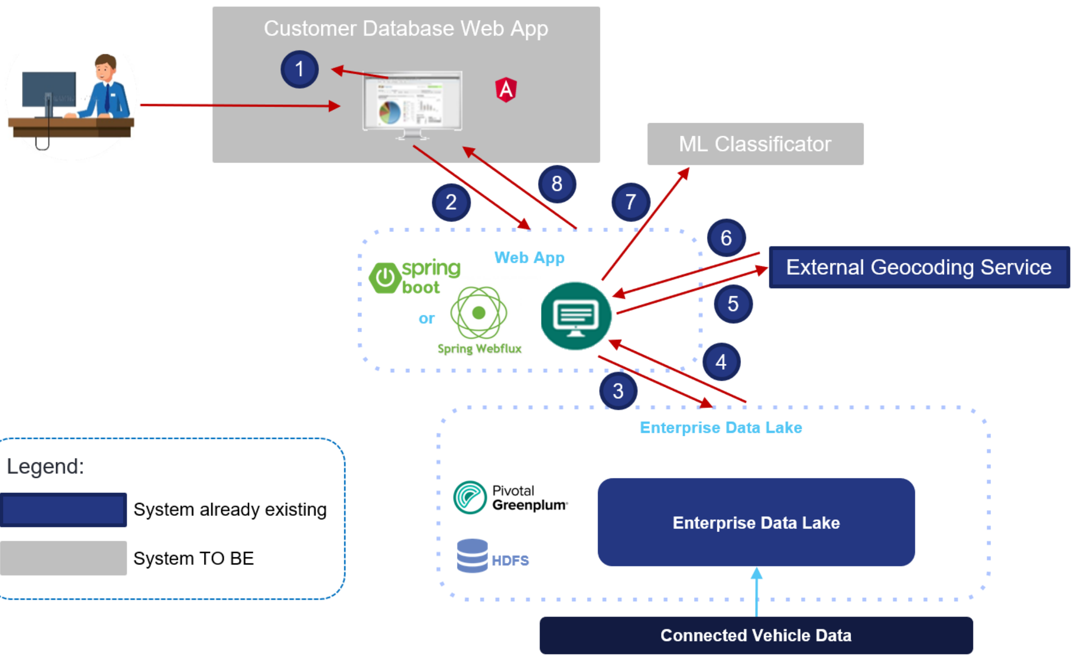


Figure 3.2: Application's complete architecture.

In Figure 3.2, it is represented a potential complete architecture of the application: indeed, there are some components, defined as *ML Classificator* and *Customer Database Web App*, that don't exist yet, but that are still part of the hypothetical design of the whole system.

For the scope of the thesis, I focused entirely on implementing the central component, named *Web App*, and on its interactions with the *Enterprise Data Lake*, from which the data are read, and the *External Geocoding Service*, used to retrieve information in real-time given a specific GPS position as input.

Following the numbers identifying each step, we can try to rebuild the workflow of the application:

1. The user interacts through the interface of the *Customer Database Web App*, requesting for instance the profile of a client or a specific vehicle;

2. The request of the user calls a Rest-API from the *Web App* component (developed either in Spring Boot or WebFlux);

3. The Rest-API needs to read data from the database;

4. The database returns the requested data;

5. The *Web App* manipulates and aggregates the data and calls the *External Geocoding Service* for retrieving information related to some GPS positions;

6. The *External Geocoding Service* returns the information related to the GPS position given as input (steps 5-6 are repeated lots of times depending on the number of GPS positions, possibly in parallel);

7. The information obtained both from the database and the *External Geocoding Service* are then sent to a *Machine Learning Classificator*, that will make some interesting calculations;

8. Finally, the complete response is returned to the *Customer Database Web App*, and the user will be able to show it, being that eventually equipped with intuitive graphs to better show statistics and relevant details.

3.1.3 Connected Vehicle database

The analyzed signals (speed, altitude, latitude and longitude) belong to one of the main data flows, called *UBI*, which collects data during normal travel, from the ignition of the machine panel (key-on) to the shutdown of the vehicle (key-off). All the data are linked to a Trip-ID, permitting to reconstruct the vehicle's route.

Below the recap of the three main UBI datasets:

- *Minimum Set of Data (MSD)*, mainly related to position, speed, acceleration, rotation angle, etc., sampled at a frequency of 1 Hz;

- *Additional Data*, regarding gear, ABS, driving style, odometer, doors, tires, etc., sampled each "key on" and on signal change;

- *Event Data*: data on event (e.g., a crash) concerning acceleration, rotation angle, throttle, braking position, etc., sampled at high frequency (about 400 Hz) on a [-20, +20] seconds window across the event.

Signal-ID	Information	Sampling Frequency	Default
5A	Latitude	0 Hz - 2 Hz	1 Hz
6A	Longitude	0 Hz - 2 Hz	1 Hz
9A	GPS speed	0 Hz - 2 Hz	1 Hz
11A	Altitude	0 Hz - 2 Hz	1 Hz

Table 3.1: Details about the MSD analyzed signals.

For the scope of the application, a data extraction was performed from the Greenplum platform, where the Corporate Data Lake resides, to a MySQL database. The data extraction involved four types of signals (see Table 3.1) related to three distinct VIN codes and two fiscal codes, collected over a period of about 10 days. Obviously, before the effective usage, these data have been appropriately anonymized.

Target Field	Datatype	Key	Description
dt_ingestion_timestamp	timestamp		Ingestion timestamp of the row
cd_userid_code	int		Unique User Identification Number
cd_vin_code	int		Unique Vehicle Identification Number
cd_device_type_code	varchar(10)		Type of device sending the signal
cd_device_code	int	1	Unique Device (ECU) Identification Number
id_trip_id	bigint	2	A progressive number (incremented by 1 unit by the device for each trip)
id_message_id	int	3	Unique Message Identification Number (a progressive number of the message sent by the device within the trip)
cd_target_consumer_code	varchar(10)		Partner Identification String (the code of the commercial partner or target consumer that receives the signal)
dt_message_timestamp	timestamp		Timestamp of sending the message

Table 3.2: ENT_SIGNAL_HEADER table fields.

Target Field	Datatype	Key	Description
dt_ingestion_timestamp	timestamp		Ingestion timestamp of the row
cd_device_type_code	varchar(10)		Type of device sending the signal
cd_device_code	int	1	Unique Device (ECU) Identification Number
id_trip_id	bigint	2	A progressive number (incremented by 1 unit by the device for each trip)
id_message_id	int	3	Unique Message Identification Number (a progressive number of the message sent by the device within the trip)
id_data_collection_id	int	4	Group identifier of the set of signals sent together
cd_signal_code	varchar(5)	5	Identifier code of the signal supplied by the control unit
dt_start_signal_timestamp	timestamp	6	Timestamp of the signal starting from which has the current value
dt_end_signal_timestamp	timestamp		Timestamp of the signal starting from which the signal value is changed inside the same trip
gn_signal_trigger_type_name	varchar(20)		Trigger condition for which the signal was sent. The trigger can be PERIODIC (sent periodically) or CONDITION (sent at specific conditions, such as key-on/off, crash)
gn_signal_trigger_name	varchar(20)		Trigger extension condition for which the signal was sent (possible values are NOTIFICATION, REPORT, CRASH, etc.)
nm_signal_value_number	decimal		Numeric value of the signal
gn_signal_value_text	varchar(20)		String value of the signal (may be empty)
fl_signal_value_flag	varchar(10)		Translation of the Boolean value of the signal into flag (may be empty)
dt_signal_value_timestamp	timestamp		Timestamp value of the signal (may be empty)

Table 3.3: ENT_SIGNAL_DETAIL table fields.

Target Field	Datatype	Key	Description
cd_vin_code	int	1	Unique Vehicle Identification Number
fiscal_code	varchar(16)		Fiscal code associated to the owner of the vehicle

Table 3.4: VIN_FISCAL_CODE table fields.

The database consists of three different tables:

- **ENT_SIGNAL_HEADER**, where the headers of the signals are stored (extracted about 2K rows). There is a one-to-many relationship between this table and ENT_SIGNAL_DETAIL;

- **ENT_SIGNAL_DETAIL**, containing the details of the signals, such as the numeric value (extracted about 200K rows);
- **VIN_FISCAL_CODE**, that stores the correspondence between the VIN code and the fiscal code of the vehicle's owner. There is a one-to-many relationship between this table and ENT_SIGNAL_HEADER.

3.1.4 External geocoding service

Latitude and longitude are two among the four signals taken into consideration for the study. Given a latitude and a longitude related to the same timestamp, it could be interesting to go back to the original address in order to discover details about the surrounding area, such as the road type or the nearest point of interest. This analysis can be useful for better outlining the driver's habits.

Nominatim is the geocoding software that powers the official OSM site `www.openstreetmap.org`. It uses OpenStreetMap data to find locations on Earth by name and address (geocoding), but can also do the reverse, finding an address for any location on the planet.

Nominatim API offers different endpoints for querying the data: the */reverse* one, that generates a detailed address from a latitude and longitude, is just right for us. It does not work by exactly computing the address for the coordinate it receives, rather by finding the closest suitable OSM object and returning its address information.
The main format of the reverse API is:
`https://nominatim.openstreetmap.org/reverse?lat=<value>&lon=<value>`
where *lat* and *lon* are latitude and longitude of a coordinate in WGS84 projection. The API returns exactly one result or an error if the coordinate is in an area with no OSM data coverage. Additional parameters are accepted, like the format (xml, json, geojson, etc.), the language of results, the output details and the result limitation.[59]

```
{
    "place_id": 116739679,
    "licence": "Data OpenStreetMap contributors, ODbL 1.0
                https://osm.org/copyright",
    "osm_type": "way",
    "osm_id": 58789053,
    "lat": "45.04991424070042",
    "lon": "7.424878539864067",
    "place_rank": 26,
    "category": "highway",
    "type": "residential",
    "importance": 0.09999999999999998,
    "addresstype": "road",
    "name": "Via Santa Rita",
    "display_name": "Via Santa Rita, Reano, Frazione Benna, Reano,
                     Torino, Piemonte, 10051, Italia",
    "address": {
        "road": "Via Santa Rita",
        "residential": "Reano",
        "hamlet": "Frazione Benna",
        "village": "Reano",
        "county": "Torino",
        "state": "Piemonte",
        "postcode": "10051",
        "country": "Italia",
        "country_code": "it"
    },
    "boundingbox": [
        "45.0496864",
        "45.050354",
        "7.4243111",
        "7.4251725"
    ]
}
```

Listing 3.1: Example of Nominatim reverse API response in json format.

By looking at the example of Nominatim reverse API response in Listing 3.1, we can point out two important attributes, *category* and *type*, that are key-value pairs, useful to identify relevant features characterizing the position given as input (e.g., type of the road, nearest facilities, etc.).[60]

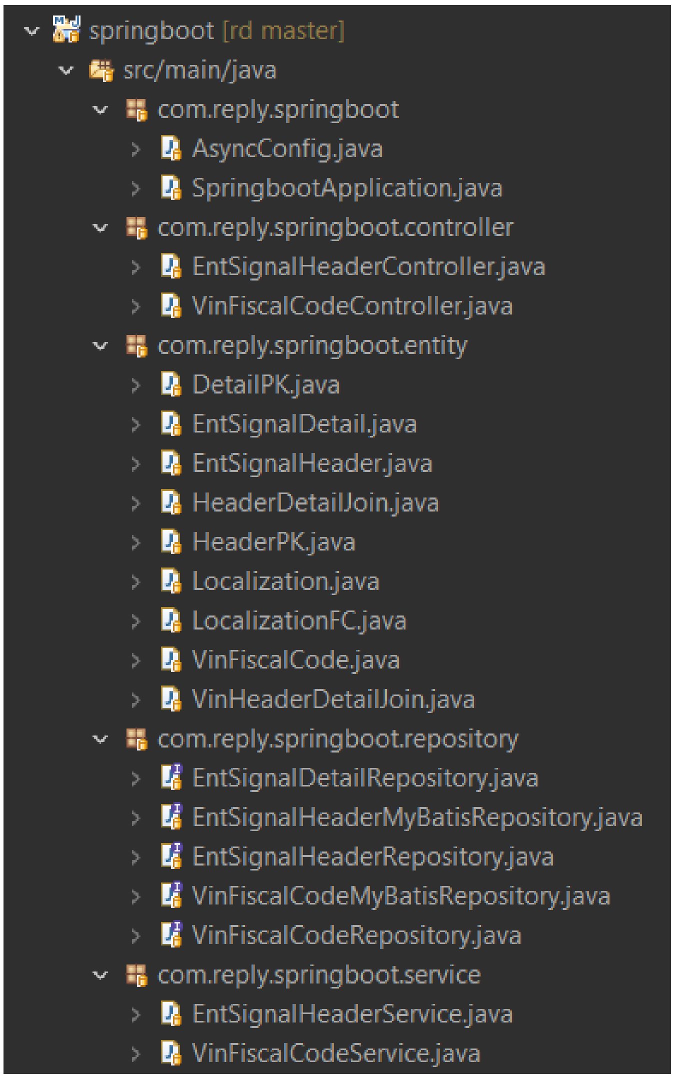
3.2 Spring Boot application

The first version of the application is a Spring Boot application, running on a Servlet blocking stack. The required APIs aim to retrieve information through the analysis and elaboration of the signals related to a specific vehicle (identified uniquely by its VIN) or driver (identified by the associated fiscal code). The code snippets that will be provided will always regard only the API accepting the VIN as input.

Two different solutions will be proposed for the Rest-APIs: a synchronous and an asynchronous one (using CompletableFutures).

3.2.1 Project structure and dependencies

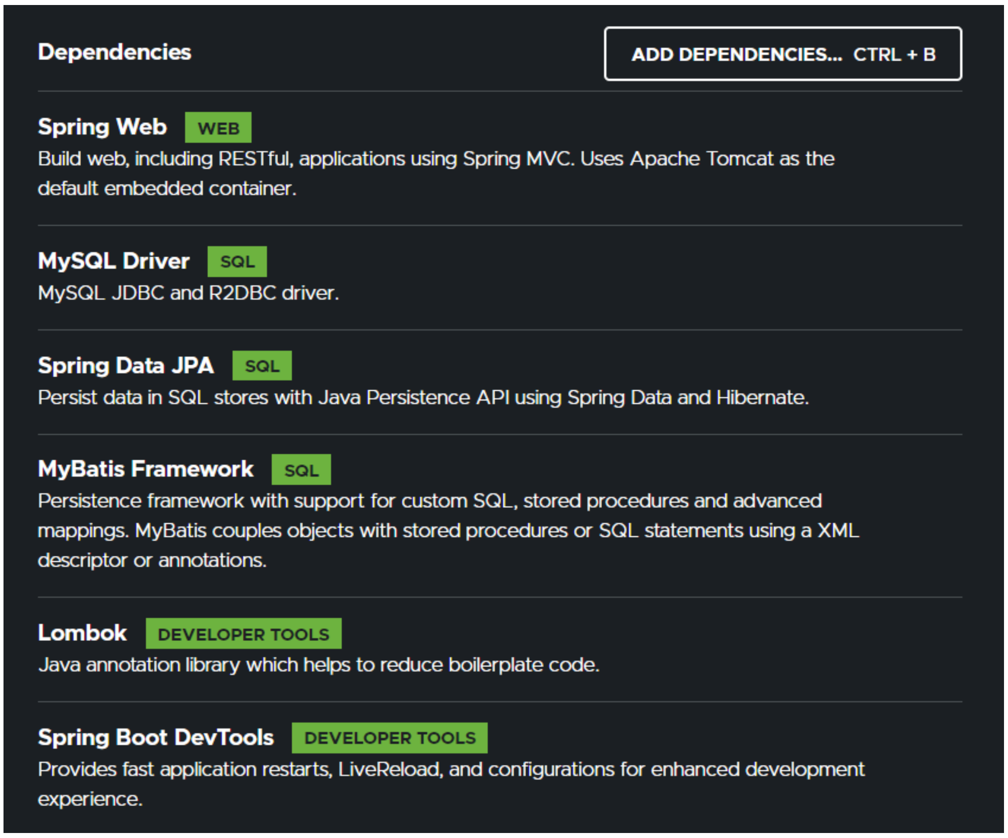
Figure 3.3 shows the package structure of the project. The main packages are Entity, Repository, Service and Controllers. Furthermore, together with the main class, there is an extra configuration file, called *AsyncConfig*, useful for implementing the asynchronous solution of the Rest-APIs (it will be better explained in the next sections).



```
springboot [rd master]
  src/main/java
    com.reply.springboot
      AsyncConfig.java
      SpringbootApplication.java
    com.reply.springboot.controller
      EntSignalHeaderController.java
      VinFiscalCodeController.java
    com.reply.springboot.entity
      DetailPK.java
      EntSignalDetail.java
      EntSignalHeader.java
      HeaderDetailJoin.java
      HeaderPK.java
      Localization.java
      LocalizationFC.java
      VinFiscalCode.java
      VinHeaderDetailJoin.java
    com.reply.springboot.repository
      EntSignalDetailRepository.java
      EntSignalHeaderMyBatisRepository.java
      EntSignalHeaderRepository.java
      VinFiscalCodeMyBatisRepository.java
      VinFiscalCodeRepository.java
    com.reply.springboot.service
      EntSignalHeaderService.java
      VinFiscalCodeService.java
```

Figure 3.3: Spring Boot project structure.

In Figure 3.4, it is possible to have a look at the necessary dependencies included in the project.



Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

MySQL Driver SQL
MySQL JDBC and R2DBC driver.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MyBatis Framework SQL
Persistence framework with support for custom SQL, stored procedures and advanced mappings. MyBatis couples objects with stored procedures or SQL statements using a XML descriptor or annotations.

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

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Figure 3.4: Dependencies included in the Spring Boot project.

3.2.2 Entity

In the Entity package, it is possible to find:

- Entity classes that are directly mapped to the database tables (i.e., EntSignalHeader, EntSignalDetail and VinFiscalCode) or to the tables' composite primary keys (i.e., HeaderPK, DetailPK). These are the "real" entities for excellence: as we will see in the next code snippet, Spring JPA (i.e., a Java specification for managing relational data in Java applications) and its annotations will be applied for these classes in order to clearly specify primary keys and possible relationship between tables;

- Classes used for mapping the objects resulting from the join operation between two or more tables (i.e., HeaderDetailJoin, VinHeaderDetailJoin). JPA annotations won't be necessary in this case;

- Other classes useful for mapping the API responses (i.e., Localization, LocalizationFC). Even in this case, JPA annotations will not be present.

```java
@Entity
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@Table(name = "ent_signal_header")
public class EntSignalHeader implements Serializable {

   @EmbeddedId
   private HeaderPK headerPk;

   @Column(name = "dt_ingestion_timestamp")
   private Timestamp dtIngestionTimestamp;

   @Column(name = "cd_userid_code")
   private Integer cdUseridCode;

   @Column(name = "cd_vin_code")
   private Integer cdVinCode;

   @Column(name = "cd_device_type_code")
   private String cdDeviceTypeCode;

   @Column(name = "cd_target_consumer_code")
   private String cdTargetConsumerCode;

   @Column(name = "dt_message_timestamp")
   private Timestamp dtMessageTimestamp;

   @OneToMany(targetEntity = EntSignalDetail.class, cascade =
    CascadeType.ALL)
   @JoinColumns({
      @JoinColumn(name="cd_device_code",
      referencedColumnName="cd_device_code"),
      @JoinColumn(name="id_trip_id",
      referencedColumnName="id_trip_id"),
      @JoinColumn(name="id_message_id",
      referencedColumnName="id_message_id")
   })
   private List<EntSignalDetail> details;

}
```

Listing 3.2: EntSignalHeader entity class.

In Listing 3.2, there is an example of an entity class, EntSignalHeader, that maps the ENT_SIGNAL_HEADER database table. Above the class definition, we find some *Lombok* annotations that automatically generate the all-args constructor, the no-args constructor, getters and setters for each attribute.
It is also possible to identify many Spring JPA typical annotations:

- *@Entity*, indicating that the class is correlated with a table in the database;
- *@Table* and *@Column*, where the first is used for mapping the entity class name to the database table name, and the second for specifying the mapped column for a persistent property or field;
- *@EmbeddedId*, useful for embedding the composite primary key (mapped by another class, properly annotated with *@Embeddable*) in the corresponding entity;
- *@OneToMany*, specifying the relationship that subsists between this table and EntSignalDetail, together with the involved join columns.

```java
// Previously described Lombok annotations are omitted for saving space
@JsonIgnoreProperties(ignoreUnknown = true)
@JsonInclude(JsonInclude.Include.NON_DEFAULT)
public class HeaderDetailJoin {

   private Integer cdVinCode;

   private String cdSignalCode;

   private Timestamp compactTimestamp;

   private Double signalValue;

   public HeaderDetailJoin(Integer cdVinCode, String cdSignalCode,
    Double signalValue) {
        this.cdVinCode = cdVinCode;
        this.cdSignalCode = cdSignalCode;
        this.signalValue = signalValue;
    }
}
```

Listing 3.3: HeaderDetailJoin class.

In Listing 3.3, we find an example of a class, HeaderDetailJoin, taken among the ones used for mapping the objects resulting from a join operation, that in this case subsists between EntSignalHeader and EntSignalDetail tables.

Above the class definition, there are two Jackson annotations, useful at deserialization of JSON to Java object. The class contains, apart from the attributes, a custom constructor, necessary for a SQL query we will see later on in the Repository section.

```java
// Previously described Lombok annotations are omitted for saving space
@Builder
@JsonIgnoreProperties(ignoreUnknown = true)
@JsonInclude(JsonInclude.Include.NON_DEFAULT)
public class Localization {

   private String category;

   private String type;

   private List<HeaderDetailJoin> averages;

}
```

Listing 3.4: Localization class.

Listing 3.4 shows a class that is employed for mapping the API response. Lombok *@Builder* annotation helps in producing the code automatically using the Builder pattern.

3.2.3 Repository

The Repository package contains either JPA repository interfaces, that extend *CrudRepository* and allow to work directly with objects rather than with tables, or MyBatis mappers, that provide direct access to SQL data. The configuration attributes required to connect to MySQL database have been defined in the *application.properties* file. Overall, two queries have been written:

- The first query selects the signal code, the average value of speed and the average value of altitude given as input the VIN code (sited in *EntSignalHeaderRepository*) or the fiscal code (sited in *VinFiscalCodeRepository*). These queries were implemented in JPA repository interfaces, exploiting the SQL-like syntax offered by JPA.
  The complete query by VIN code can be found in Listing 3.5: note that other where clauses were added in order to discard signal values lower than 0 (it may happen due to possible GPS errors, that could invalidate the average) and, relatively to the speed signal, to exclude from the average values of speed lower than 10.

```
public interface EntSignalHeaderRepository extends
    CrudRepository<EntSignalHeader, HeaderPK> {

   @Query(
   "SELECT new
    com.reply.springboot.entity.HeaderDetailJoin(h.cdVinCode, "
   + "d.detailPK.cdSignalCode, "
   + "AVG(d.nmSignalValueNumber) as signalValue) "
   + "FROM EntSignalHeader h JOIN h.details d "
   + "WHERE h.cdVinCode = :cdVinCode "
   + "AND (d.detailPK.cdSignalCode = '11A' "
   + "OR (d.detailPK.cdSignalCode = '9A' "
   + "AND d.nmSignalValueNumber >= 10)) "
   + "AND d.nmSignalValueNumber > 0 "
   + "GROUP BY h.cdVinCode, d.detailPK.cdSignalCode "
   + "ORDER BY d.detailPK.cdSignalCode ")
   List<HeaderDetailJoin> getSignalsAverageByVinCode(Integer cdVinCode);

}
```

Listing 3.5: EntSignalHeaderRepository interface.

- In the second query we can note the usage of complex SQL functions for manipulating timestamps that made necessary the use of MyBatis mappers, since JPA syntax don't support them. It selects the signal code, and the signal value of respectively latitude or longitude extracting only one timestamp every 10 minutes (i.e., 600 seconds), given as input the VIN code (sited in *EntSignalHeaderMyBatisRepository*) or the fiscal code (sited in *VinFiscalCode-MyBatisRepository*).

  The choice of extracting one sample value every 10 minutes derived from the following reasoning. For each couple of latitude and longitude, we need to call an external geocoding service in order to retrieve additional information about that specific position. New values of these signals are received every second (unless a small percentage that goes lost), but it does not make sense to call the external service at each minimal variation: if the driver is going at a very high speed, he's probably driving on a highway, so the external service, for how it works, will respond by indicating the type of road, that will be always the same; on the contrary, if he's going at a very low speed, the variation of the position may be really insignificant, and also in this case the external service will possibly give similar or even identical responses, assuming its intrinsic imprecision.
  In the Listing 3.6, it is possible to find the complete query by VIN code.

```
@Mapper
public interface EntSignalHeaderMyBatisRepository {

    @Select(
    "SELECT h.cd_vin_code, d.cd_signal_code,
   MAX(d.nm_signal_value_number) as signal_value, "
    + "DATE_FORMAT(from_unixtime(unix_timestamp "
    "(d.dt_start_signal_timestamp) -
   unix_timestamp(d.dt_start_signal_timestamp) mod 600),"
    + " '%Y-%m-%d %H:%i:00') as compact_timestamp "
    + "FROM ent_signal_header h JOIN ent_signal_detail d "
    + "ON h.cd_device_code = d.cd_device_code "
    + "AND h.id_trip_id = d.id_trip_id "
    + "AND h.id_message_id = d.id_message_id "
    + "WHERE h.cd_vin_code = #{cdVinCode} "
    + "AND (d.cd_signal_code = \"5A\" OR d.cd_signal_code = \"6A\") "
    + "GROUP BY h.cd_vin_code, d.cd_signal_code, compact_timestamp "
    + "ORDER BY compact_timestamp, d.cd_signal_code ")
    @Results({
        @Result(property="cdVinCode", column="cd_vin_code"),
        @Result(property="cdSignalCode", column="cd_signal_code"),
        @Result(property="signalValue", column="signal_value"),
        @Result(property="compactTimestamp", column="compact_timestamp")
    })
    List<HeaderDetailJoin> getLatLonByVinCode(Integer cdVinCode);

}
```

Listing 3.6: EntSignalHeaderMyBatisRepository interface.

3.2.4 First approach: synchronous endpoints

The objective of the Rest-APIs to be implemented is to retrieve information about a VIN code or a fiscal code given as input by analyzing four types of signals: more specifically, the response should consist of a list of objects (beloging to the class called *Localization*), where the first one contains the average values of speed and altitude, while the other ones have two attributes, *category* and *type*, extracted from the complete response returned by the external geocoding service, called as many times as the number of pairs of latitude and longitude signals grouped by timestamp. Since the two APIs are very similar, further discussions will regard only the one that accepts the VIN code as input.

The first approach is sequential and synchronous: all the operations inside the Service method *getInfoByVinCode()* are executed by one single thread.

The steps inside the Service method can be summarized as follows:

- First of all, a list of Localization object, called *results*, is created;
- The first object to be added to the results list is obtained by calling the repository method *getSignalsAverageByVinCode()*, that returns the average values of speed and altitude signals;
- At this point, we call the mapper method *getLatLonByVinCode()*, that returns latitude and longitude signal values with timestamp. The timestamp attribute here is fundamental, because it is the key that allows to pair latitude and longitude. For this specific case, we exploit Java 8 Streams, that offers interesting methods that allowed to perform the group by operation;
- For each pair of latitude and longitude, we make an external api call to the geocoding service, and save each result by adding it to the results list. The external api call is a synchronous HTTP GET request, executed thanks to *RestTemplate*;
- Finally, the complete results list is returned.

The detailed code of the method just described can be found in the following Listing 3.7.

```
@Service
public class EntSignalHeaderService {

   @Autowired
   EntSignalHeaderRepository entSignalHeaderRepository;

   @Autowired
   EntSignalHeaderMyBatisRepository entSignalHeaderMyBatisRepository;

   // External geocoding service base url
   String baseURL = "https://nominatim.openstreetmap.org";

   public List<Localization> getInfoByVinCode(Integer cdVinCode) {
      // Create a list to store results
      List<Localization> results = new
      ArrayList<Localization>();

      // Query for getting the averages of speed and altitude
      List<HeaderDetailJoin> averageByVinCode =
      entSignalHeaderRepository.getSignalsAverageByVinCode(cdVinCode);
      Localization averages =
      Localization.builder().averages(averageByVinCode).build();
```

```
        // Add Localization object containing averages to results list
        results.add(averages);
        // Query for getting lat and lon signals
        List<HeaderDetailJoin> infoByVinCode =
        entSignalHeaderMyBatisRepository.getLatLonByVinCode(cdVinCode);
        // Group lat and lon signals by timestamp
        Map<Timestamp, List<HeaderDetailJoin>> groupedSignals =
        infoByVinCode.stream()
                     .collect(Collectors.groupingBy(w ->
                                   w.getCompactTimestamp()));
        // For each pair of lat and lon, make an external call
        for (List<HeaderDetailJoin> entry : groupedSignals.values()) {
           String latitude = "", longitude = "";

           if (entry.size() == 2) {
              latitude = entry.get(0).getSignalValue().toString();
              longitude = entry.get(1).getSignalValue().toString();

              String uri = baseURL + "/reverse?format=jsonv2&lat=" +
                           latitude + "&lon=" + longitude;
              RestTemplate restTemplate = new RestTemplate();
              // Make external call to geocoding service
              Localization result = restTemplate.getForObject(uri,
                                   Localization.class);
              // Add the result to results list
              results.add(result);
           }
        }
        return results;
    }
     /* This class has other methods that we'll see later */
}
```

Listing 3.7: Synchronous implementation of getInfoByVinCode() service method.

The service method is called by the homonymous controller method, that exposes the endpoint. The related code can be seen in the following Listing 3.8.

```
@RestController
@RequestMapping(value = "signals",
produces = {MediaType.APPLICATION_JSON_VALUE, "application/hal+json"})
public class EntSignalHeaderController {

    @Autowired
    EntSignalHeaderService entSignalHeaderService;
```

```
   @GetMapping("/getByVinCode")
   public ResponseEntity<List<Localization>>
    getInfoByVinCode(@RequestParam Integer cdVinCode) {
       return new
    ResponseEntity(entSignalHeaderService.getInfoByVinCode(cdVinCode),
                   HttpStatus.OK);
   }
}
```

Listing 3.8: GetInfoByVinCode() controller method.

3.2.5 Second approach: asynchronous endpoints

The second and asynchronous approach leverages the Java 8 *CompletableFutures* class, that provides an easy way to write asynchronous, non-blocking and multi-threaded code. It was built as an advancement to Future interface, which had limitations in terms of what can be done with a result of an asynchronous call though, while CompletableFuture provides an array of APIs for composing, combining and error handling in asynchronous situations.

First of all, it was necessary to create a configuration class, called *AsyncConfig*, in order to to enable and configure asynchronous method execution.

```
@Configuration
@EnableAsync
public class AsyncConfig {

    @Bean (name = "taskExecutor")
    public Executor taskExecutor() {
        final ThreadPoolTaskExecutor executor = new
        ThreadPoolTaskExecutor();
        executor.setCorePoolSize(2);
        executor.setMaxPoolSize(4);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("SignalThread-");
        executor.initialize();

        return executor;
    }
}
```

Listing 3.9: Async configuration file.

The *@EnableAsync* annotation enables the possibility to run *@Async* methods in a background thread pool. The bean *taskExecutor* helps to customize the thread executor (i.e., configuring the number of threads for the application,the queue limit size, etc.). In our case, we want to limit the number of concurrent threads to two and limit the size of the queue to 100. Spring will specifically look for this bean when the server is started.[61]

Service asynchronous methods will have the *@Async* annotation above. In Listing 3.10, we find the code related to the asynchronous version of the service method *getInfoByVinCode()*, already described in detail in the previous section.

Apparently, the steps are really similar to the ones already seen for the synchronous version, but actually there is an important difference: there is no more a unique thread that executes the whole code, rather there are many worker threads (the maximum number depends on the CPU cores of the machine), each one in charge of executing one or more tasks. Specifically, the tasks that have been parallelized are the computation of the average values of speed and altitude and the external geocoding api calls (see the methods *getAvgByVinCode()* and *reverseGeocodingApi()* in Listing 3.11).

Furthermore, note that threads don't wait for the complete results of the tasks they're responsible for: they will get only *CompletableFutures*, so the real values will be available later on and only at that point it will be possible to retrieve them by applying the *get()* method, that is blocking though. Since we're interesting in running tasks in background that return some value, we use *CompletableFuture.supplyAsync()*, instead of *CompletableFuture.runAsync()*, which would not return any value.

```
@Async
public CompletableFuture<List<Localization>>
    getInfoByVinCodeAsync(Integer cdVinCode) {
     // Method for getting the averages of speed and altitude
     CompletableFuture<Localization> averages =
     CompletableFuture.supplyAsync(()->{
          return getAvgByVinCode(cdVinCode);
       });
     // Query for getting lat and lon signals
   List<HeaderDetailJoin> infoByVinCode =
    entSignalHeaderMyBatisRepository.getLatLonByVinCode(cdVinCode);

   // Group lat and lon signals by timestamp
   Map<Timestamp, List<HeaderDetailJoin>> groupingByTimestamp =
   infoByVinCode.stream()
                 .collect(Collectors.groupingBy(w ->
                               w.getCompactTimestamp()));
   // Convert previous map to list of lists
   List<List<HeaderDetailJoin>> listOfLatLon =
    groupingByTimestamp.values()
```

```
                      .stream()
                      .collect(toList());

   // Create a list to store CompletableFutures results
    List<CompletableFuture<Localization>> localizationFutures = new
   ArrayList<CompletableFuture<Localization>>();

    // For each pair of lat and lon, call the function
   reverseGeocodingApi
    for (List<HeaderDetailJoin> l : listOfLatLon) {
      if (l.size() == 2) {
         CompletableFuture<Localization> loc =
            CompletableFuture.supplyAsync(()->{
               return reverseGeocodingApi(l);
            });
         localizationFutures.add(loc);
      }
    }

    // Create a list to store final results
    List<Localization> results = new ArrayList<Localization>();

    // Retrieve value of averages and add to results list
    results.add(averages.get());
    // Retrieve each value of localizationFutures and add to results
    for (CompletableFuture<Localization> s : localizationFutures) {
      results.add(s.get());
    }
    // return the final results list as a new CompletableFuture
    return CompletableFuture.completedFuture(results);
}
```

Listing 3.10: Asynchronous implementation of getInfoByVinCode().

```
public Localization getAvgByVinCode(Integer cdVinCode) {
   // Query for getting the averages of speed and altitude
   List<HeaderDetailJoin> averageByVinCode =
   entSignalHeaderRepository.getSignalsAverageByVinCode(cdVinCode);
    Localization averages =
   Localization.builder().averages(averageByVinCode).build();

    return averages;
}
```

```
public Localization reverseGeocodingApi(List<HeaderDetailJoin> input) {

   String uri = baseURL + "/reverse?format=jsonv2&lat=" +
    input.get(0).getSignalValue() + "&lon=" +
    input.get(1).getSignalValue();
   RestTemplate restTemplate = new RestTemplate();
   // Make external call to geocoding service
   Localization res = restTemplate.getForObject(uri,
    Localization.class);

   return res;
}
```

Listing 3.11: Methods getAvgByVinCode() and reverseGeocodingApi().

In Listing 3.12, there's the code related to the controller that invokes the asynchronous service method, with a static function aimed to signal possible exception errors that may occur.

```
@GetMapping("/async/getByVinCode")
public @ResponseBody CompletableFuture<ResponseEntity>
    getInfoByVinCodeAsync(@RequestParam Integer cdVinCode) {
   return entSignalHeaderService.getInfoByVinCodeAsync(cdVinCode)
   .<ResponseEntity>thenApply(ResponseEntity::ok)
            .exceptionally(handleGetInfoByVinFailure);
}

private static Function<Throwable, ResponseEntity<? extends
    List<HeaderDetailJoin>>> handleGetInfoByVinFailure = throwable -> {
     LOGGER.error("Failed to read records: {}", throwable);
     return
    ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
};
```

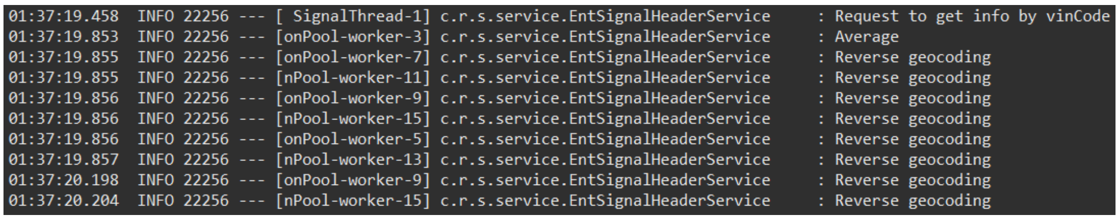
Listing 3.12: GetInfoByVinCode() asynchronous controller method.

Finally, Figure 3.5 shows the console logs printed when the API just been described is called. It helps understanding what is happening at low level.

When the request is sent, we can note that one *SignalThread* (among the two available, as set in the configuration file) starts in order to serve that. The machine where the application run has 4 CPU cores, so, excluding the SignalThread-1, we will have 7 other available threads among the thread pool.

After few milliseconds, the worker threads immediately start executing the assigned task (there will be only one task for average computation and many for reverse geocoding).

There are exactly 7 prints almost at the same instant of time (from millisecond 853 to 857), that demonstrate their parallel execution; the remaining tasks will be assigned to the fastest threads (worker-9 and worker-15, in this specific case).



```
01:37:19.458  INFO 22256 --- [ SignalThread-1] c.r.s.service.EntSignalHeaderService     : Request to get info by vinCode
01:37:19.853  INFO 22256 --- [onPool-worker-3] c.r.s.service.EntSignalHeaderService     : Average
01:37:19.855  INFO 22256 --- [onPool-worker-7] c.r.s.service.EntSignalHeaderService     : Reverse geocoding
01:37:19.855  INFO 22256 --- [nPool-worker-11] c.r.s.service.EntSignalHeaderService     : Reverse geocoding
01:37:19.856  INFO 22256 --- [onPool-worker-9] c.r.s.service.EntSignalHeaderService     : Reverse geocoding
01:37:19.856  INFO 22256 --- [nPool-worker-15] c.r.s.service.EntSignalHeaderService     : Reverse geocoding
01:37:19.856  INFO 22256 --- [onPool-worker-5] c.r.s.service.EntSignalHeaderService     : Reverse geocoding
01:37:19.857  INFO 22256 --- [nPool-worker-13] c.r.s.service.EntSignalHeaderService     : Reverse geocoding
01:37:20.198  INFO 22256 --- [onPool-worker-9] c.r.s.service.EntSignalHeaderService     : Reverse geocoding
01:37:20.204  INFO 22256 --- [nPool-worker-15] c.r.s.service.EntSignalHeaderService     : Reverse geocoding
```

Figure 3.5: Asynchronous getInfoByVinCode() printed logs.

3.3 WebFlux application

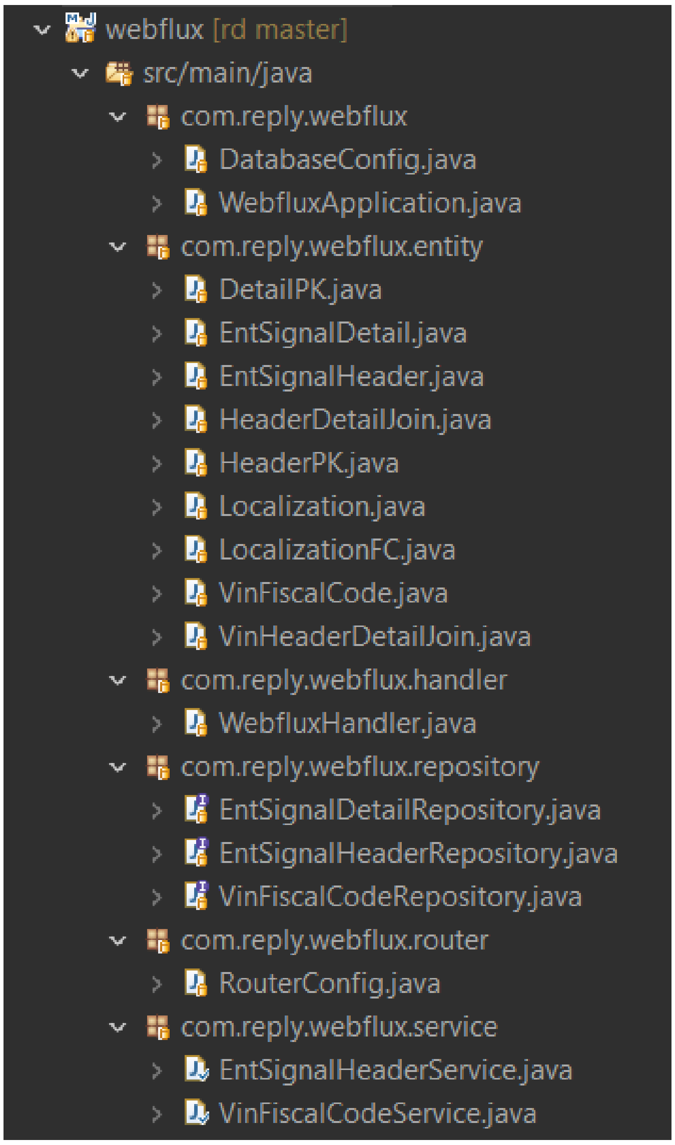
The second version of the application is a Spring WebFlux application, running on a fully reactive stack, with Netty as runtime server.

Differently from Spring Boot, where all the code is generally written with an imperative approach and call to external systems are always blocking, here all the logic is written in a reactive and functional style, guaranteeing a non-blocking end-to-end flow (WebClient was used as the new and non-blocking alternative to RestTemplate to make the downstream calls).

The required APIs and their purposes have already been widely described.

3.3.1 Project structure and dependencies

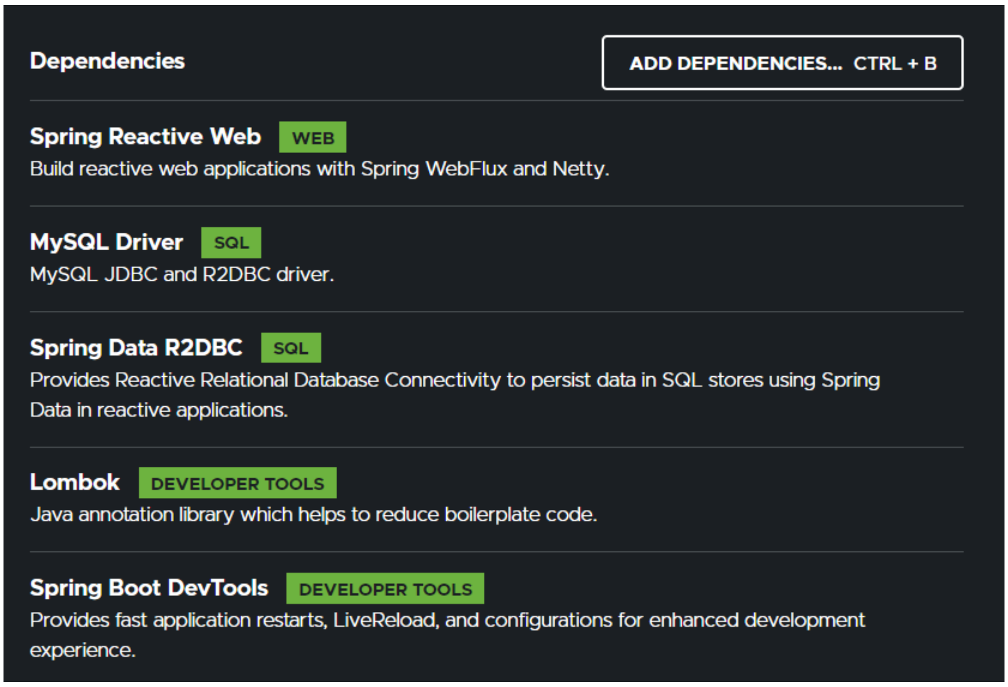
Figure 3.6 shows the package structure of the project. The main packages are Entity, Repository, Service, Handler and Router. Furthermore, together with the main class, there is the datasource configuration file, establishing the connection with MySQL database.



```
webflux [rd master]
  src/main/java
    com.reply.webflux
      DatabaseConfig.java
      WebfluxApplication.java
    com.reply.webflux.entity
      DetailPK.java
      EntSignalDetail.java
      EntSignalHeader.java
      HeaderDetailJoin.java
      HeaderPK.java
      Localization.java
      LocalizationFC.java
      VinFiscalCode.java
      VinHeaderDetailJoin.java
    com.reply.webflux.handler
      WebfluxHandler.java
    com.reply.webflux.repository
      EntSignalDetailRepository.java
      EntSignalHeaderRepository.java
      VinFiscalCodeRepository.java
    com.reply.webflux.router
      RouterConfig.java
    com.reply.webflux.service
      EntSignalHeaderService.java
      VinFiscalCodeService.java
```

Figure 3.6: Spring WebFlux project structure.

In Figure 3.7, it is possible to have a look at the necessary dependencies included in the project.



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

MySQL Driver SQL
MySQL JDBC and R2DBC driver.

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.

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Figure 3.7: Dependencies included in the Spring WebFlux project.

3.3.2 R2DBC repository and entities

The Entity package basically corresponds to the one already seen and analyzed for the Spring Boot project, with the important difference that here JPA annotations are not allowed, just as the related dependency misses in *pom.xml*. Indeed, the classes that map tables' composite primary keys exist but their usefulness is relative, since it is not possible to embed them in the corresponding entities as we did in the previous case.

The reason why JPA is not applicable here is due to the fact that it uses blocking I/O, which means each request needs a dedicated thread: in highly concurrent systems this may easily lead to scaling problems.

Spring Data R2DBC, that stands for Reactive Relational Database Connectivity, is a specification to integrate SQL databases using reactive drivers, making it easy to implement R2DBC based repositories using relational data access technologies in a reactive application stack.

Spring Data R2DBC aims at being conceptually easy. In order to achieve this, it does not offer caching, lazy loading, write-behind, or many other features of ORM frameworks. This makes Spring Data R2DBC a simple, limited, opinionated object mapper.

R2DBC uses non-blocking I/O, which means it is able to handle requests with only a fixed, low number of threads, which makes scaling easier and cheaper.[62]

In Listing 3.13, we can have a look at the configuration code of a R2DBC connection to MySQL database.

```
@Configuration
public class DatabaseConfig {

   @Bean
   public ConnectionFactory connectionFactory() {
      ConnectionFactory connectionFactory = ConnectionFactories.get(
      "r2dbcs:mysql://admin:*********@130.61.233.42:3306/reactive");

         // Creating a Mono using Project Reactor
         Mono.from(connectionFactory.create());

         return connectionFactory;
   }

   @Bean
   DatabaseClient databaseClient(ConnectionFactory connectionFactory) {
       return DatabaseClient.builder()
           .connectionFactory(connectionFactory)
           .namedParameters(true)
           .build();
   }
}
```

Listing 3.13: Configuration of MySQL R2DBC connection.

In Spring WebFlux project, there is only one repository interface, extending *ReactiveCrudRepository*, for each database table: MyBatis mappers were not needed in this case, since R2DBC allows to write manually defined queries.
An example of EntSignalHeaderRepository can be found in Listing 3.14.

```
@Repository
public interface EntSignalHeaderRepository extends
   ReactiveCrudRepository<EntSignalHeader, HeaderPK> {

    @Query(
    "SELECT h.cd_vin_code, d.cd_signal_code,
   MAX(d.nm_signal_value_number) as signal_value, "
    + "DATE_FORMAT(from_unixtime(unix_timestamp "
    + "(d.dt_start_signal_timestamp) -
   unix_timestamp(d.dt_start_signal_timestamp) mod 600),"
    + " '%Y-%m-%d %H:%i:00') as compact_timestamp "
    + "FROM ent_signal_header h JOIN ent_signal_detail d "
    + "ON h.cd_device_code = d.cd_device_code "
    + "AND h.id_trip_id = d.id_trip_id "
    + "AND h.id_message_id = d.id_message_id "
    + "WHERE h.cd_vin_code = :cdVinCode "
    + "AND (d.cd_signal_code = \"5A\" OR d.cd_signal_code = \"6A\") "
    + "GROUP BY h.cd_vin_code, d.cd_signal_code, compact_timestamp "
    + "ORDER BY compact_timestamp, d.cd_signal_code ")
    Flux<HeaderDetailJoin> getLatLonByVinCode(Integer cdVinCode);

    @Query(
    "SELECT h.cd_vin_code, d.cd_signal_code,
   AVG(d.nm_signal_value_number) as signal_value "
    + "FROM ent_signal_header h JOIN ent_signal_detail d "
    + "ON h.cd_device_code = d.cd_device_code "
    + "AND h.id_trip_id = d.id_trip_id "
    + "AND h.id_message_id = d.id_message_id "
    + "WHERE h.cd_vin_code = :cdVinCode "
    + "AND (d.cd_signal_code = \"11A\" OR (d.cd_signal_code = \"9A\" "
    + "AND d.nm_signal_value_number >= 10)) "
    + "AND d.nm_signal_value_number > 0 "
    + "GROUP BY h.cd_vin_code, d.cd_signal_code "
    + "ORDER BY d.cd_signal_code ")
    Flux<HeaderDetailJoin> getSignalsAverageByVinCode(Integer
   cdVinCode);

}
```

Listing 3.14: EntSignalHeaderRepository interface.

3.3.3 Third approach: reactive endpoints

The third and last version of Rest-API implementations proposed in this thesis follows the complete reactive approach typical of Spring WebFlux, leaning on the Reactive Streams specification through Project Reactor library, together with functional programming paradigm.

The final result will be a Flux made by the *merge* of a Mono, i.e. a Localization object containing the averages of speed and altitude, and another Flux, that includes many Localization objects, each one having *category* and *type* as attributes, extracted from the external geocoding service responses.
Let's try to describe the steps (i.e., the operators chain) that allow to obtain each of the two parts of the final result:

- Flux<Localization> externalApiResults:

  1. Latitude and longitude signals with related timestamps are read from the

database through *getLatLonByVinCode()* repository method, that returns a Flux of many elements;

2. The *bufferUntilChanged()* operator is then applied: it collects subsequent repetitions of elements as compared by a key extracted through a provided function (in this case it is *getCompactTimestamp()*) into multiple list buffers that will be emitted by the resulting Flux;[31]
3. The operator *filter()* discards list buffers with less than two elements (that would mean incomplete pair, with latitude or longitude missing);
4. The *flatMap()* operator takes each list buffer of two elements, calls the external geocoding service and convert the body response to a Mono. The external api calls are parallelized thanks to the final subscribe on *Schedulers.parallel()*.[63]

- Mono<Localization> averages:

  1. Average values of speed and latitude are retrieved from the database through *getSignalsAverageByVinCode()* repository method, returning a Flux of two elements;
  2. The *collect()* operator collects all the elements emitted by the Flux into a list that is emitted by the resulting Mono when the sequence completes;[31]
  3. Finally, the *map()* operator transform the emitted Mono to a Localization object applying a synchronous function.

The complete code implementation *getInfoByVinCode()* service method can be found in Listing 3.15.

```
@Service
public class EntSignalHeaderService {

   @Autowired
   private EntSignalHeaderRepository entSignalHeaderRepository;

   // Create a WebClient instance for the external geocoding service
   WebClient webClient = WebClient
         .builder()
         .baseUrl("https://nominatim.openstreetmap.org")
         .build();

   public Flux<Localization> getInfoByVinCode(Integer cdVinCode) {

      Flux<Localization> externalApiResults =
      entSignalHeaderRepository
            .getLatLonByVinCode(cdVinCode)
            .bufferUntilChanged(HeaderDetailJoin::getCompactTimestamp)
            .filter(l -> l.size() == 2)
            .flatMap(l -> webClient.get()
                  .uri("/reverse?format=jsonv2&lat="
                  + l.get(0).getSignalValue() + "&lon="
                  + l.get(1).getSignalValue())
                  .accept(MediaType.APPLICATION_JSON)
                  .retrieve()
                  .bodyToMono(Localization.class))
            .subscribeOn(Schedulers.parallel());

      Mono<Localization> averages =
      entSignalHeaderRepository.getSignalsAverageByVinCode(cdVinCode)
                        .collectList()
                        .map(list -> {
                           Localization avg = new Localization();
                           avg.setAverages(list);
                           return avg;
                        });
      // Final result is the merge of the two
      return Flux.merge(averages, externalApiResults);
   }
}
```

Listing 3.15: Reactive implementation of getInfoByVinCode() service method.

3.3.4 Functional endpoints: Router and Handler

Functional endpoints represent another way of writing Rest-API endpoints, opposing to the widely known annotation-based. In this case, the functional web framework uses two different functions to route and handle the requests.

The HandlerFunction is where all the business logic is present: it accepts *ServerRequest*, from which it receives information from the server, and returns *Mono<ServerResponse>*, used for sending information to the server.

The RouterFunction routes an incoming request to the proper HandlerFunction. It basically substitutes the annotation *@RequestMapping*, providing data in a similar way and behavior. Usually, router functions are created using the RouterFunction utility class, where it is possible to list out the endpoints needed.[64]

HandlerFunction and RouterFunction code related to the Rest-APIs implemented in the Spring WebFlux application can be found respectively in Listing 3.16 and 3.17. Note that the content type of the response is set to *TEXT_EVENT_STREAM* in the HandlerFunctions: this means that data contained in the response will be returned gradually as they are ready (i.e., as a stream).

```
@Component
public class WebfluxHandler {

   @Autowired
   EntSignalHeaderService entSignalHeaderService;


   @Autowired
   VinFiscalCodeService vinFiscalCodeService;

   public Mono<ServerResponse> getInfoByVinCode(ServerRequest request) {
      Integer cdVinCode =
      Integer.valueOf(request.pathVariable("cdVinCode"));
      Flux<Localization> results =
      entSignalHeaderService.getInfoByVinCode(cdVinCode);

      return ServerResponse.ok()
            .contentType(MediaType.TEXT_EVENT_STREAM)
            .body(results, Localization.class);
   }

   public Mono<ServerResponse> getInfoByFiscalCode(ServerRequest
    request) {
      String fiscalCode =
      String.valueOf(request.pathVariable("fiscalCode"));
      Flux<LocalizationFC> results =
      vinFiscalCodeService.getInfoByFiscalCode(fiscalCode);

      return ServerResponse.ok()
            .contentType(MediaType.TEXT_EVENT_STREAM)
            .body(results, LocalizationFC.class);
   }
}
```

Listing 3.16: WebfluxHandler methods.

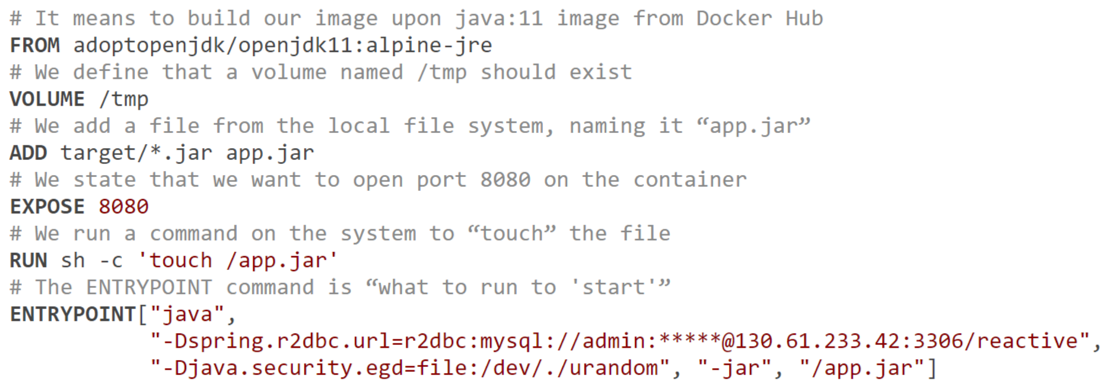
```
@Configuration
public class RouterConfig {

   @Autowired
   private WebfluxHandler handler;
   @Bean
   public RouterFunction<ServerResponse> routerFunction() {
      return RouterFunctions.route()
            .GET("/signals/getInfoByVin/{cdVinCode}",
            handler::getInfoByVinCode)
            .GET("/signals/getInfoByFiscalCode/{fiscalCode}",
            handler::getInfoByFiscalCode)
            .build();
   }
}
```

Listing 3.17: Router configuration.

3.4 Deployment on Oracle Cloud

The application described so far, in both its versions, has been deployed as a microservice on Oracle Cloud. In such a way, it has been possible to conduct performance and load testing without being limited by the machine where test scripts were run.

First step in deploying consisted in building the Docker image of the application: it was necessary to define a proper *Dockerfile*, i.e. a file, placed in the root directory of the application, containing a sequence of commands to be executed for generating the image.
Figure 3.8 shows the Dockerfile of the Spring WebFlux application; the one related to the Spring Boot application is really similar (changes regard the database configuration, the second string argument of the ENTRYPOINT square brackets, that would be jdbc instead of r2dbc).



```
# It means to build our image upon java:11 image from Docker Hub
FROM adoptopenjdk/openjdk11:alpine-jre
# We define that a volume named /tmp should exist
VOLUME /tmp
# We add a file from the local file system, naming it “app.jar”
ADD target/*.jar app.jar
# We state that we want to open port 8080 on the container
EXPOSE 8080
# We run a command on the system to “touch” the file
RUN sh -c 'touch /app.jar'
# The ENTRYPOINT command is “what to run to 'start'”
ENTRYPOINT["java",
           "-Dspring.r2dbc.url=r2dbc:mysql://admin:*****@130.61.233.42:3306/reactive",
           "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app.jar"]
```

Figure 3.8: Spring WebFlux application's Dockerfile.

Once the images have been obtained, these have been tagged and "pushed" to the Oracle Cloud Registry.

At that point, we defined a YAML file for the Kubernetes objects that had to be provided: the first Kubernetes resource is a *Deployment*, which creates and runs containers and keeps them alive (in our case, the desired number of replicas was set to 3); the second is a *LoadBalancer*, i.e. a service that allows the application to be reachable from outside of the cluster, balancing the incoming requests between the available Pods.

In Listing 3.18 it is possible to have a look to the YAML file related to the Spring WebFlux application, that shows the required fields and objects specifications for the Kubernetes Deployment; the one related to the Spring Boot application is almost identical (changes regard the label names and ports). The YAML file was passed as an argument to the *kubectl apply* command, in such a way to apply the desired configuration.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: webflux-deployment
  name: webflux-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webflux-deployment
  strategy: {}
  template:
    metadata:
      labels:
        app: webflux-deployment
    spec:
      containers:
      - image: eu-frankfurt-1.ocir/fr99gki0bywn/docker-osmrg/webflux
        name: webflux
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: webflux−service
  labels:
    app: webflux−deployment
spec:
  ports:
  - port: 8083
    protocol: TCP
    targetPort: 8080
  type: LoadBalancer
  selector:
    app: webflux−deployment
```

Listing 3.18: Spring WebFlux application's YAML file.

In Figure 3.9, it has been represented the architecture of the application running as a microservice in a Kubernetes cluster.

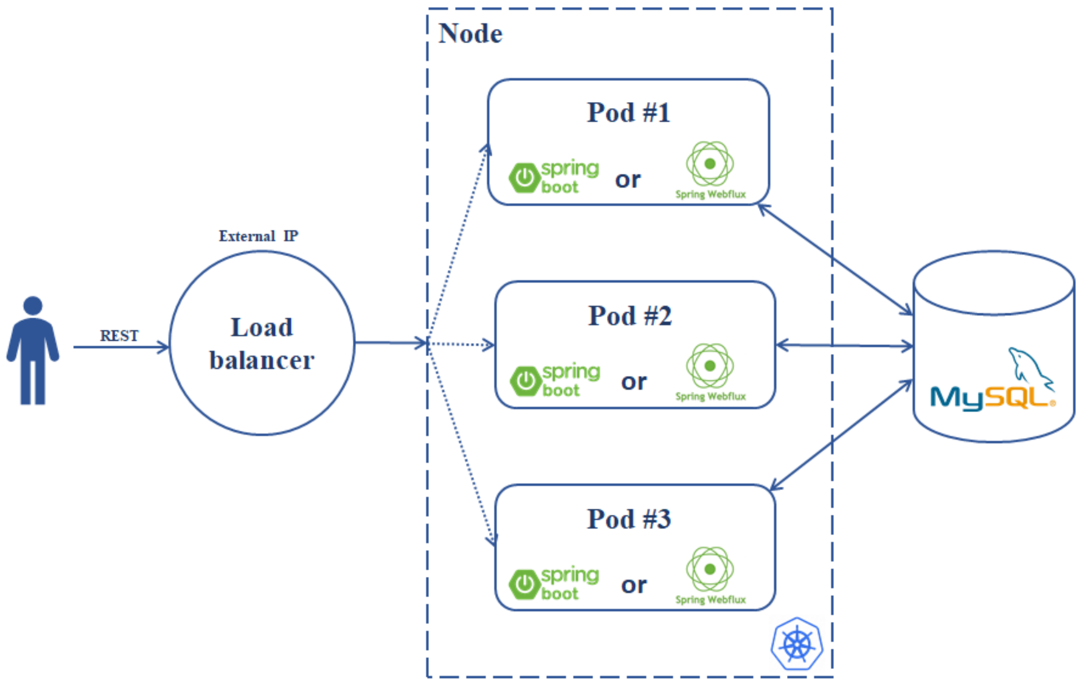


Figure 3.9: Micro-service architecture on cloud.

The application, in both its versions, has been made reachable from outside, since two IP addresses (one for each of the two versions) have been exposed to outside of the cluster.

In Listing 3.19 we provide an example of API response from the Spring Boot application when requesting information by giving the VIN code as input.

```
[
   {
      "averages":[
         {
            "cdVinCode":3,
            "cdSignalCode":"11A",
            "signalValue":208.282
         },
         {
            "cdVinCode":3,
            "cdSignalCode":"9A",
            "signalValue":27.7069
         }
      ]
   },
   {
      "category":"office",
      "type":"moving_company"
   },
   {
      "category":"amenity",
      "type":"school"
   },
   {
      "category":"shop",
      "type":"bakery"
   },
   {
      "category":"sport",
      "type":"tennis"
   },
   {
      "category":"building",
      "type":"hospital"
   },
   {
      "category":"amenity",
      "type":"fuel"
```

```
    },
    {
        "category":"highway",
        "type":"motorway"
    },
    {
        "category":"sport",
        "type":"golf"
    }
]
```

Listing 3.19: Example of Rest-API response.

When requesting information to the Spring WebFlux application, instead, the API response will contain data fields whose value corresponds to json objects like the ones we read in Listing 3.19 (e.g., data:{"category":"amenity","type":"arts_centre"}).

Chapter 4

Performance testing

This last chapter will be devoted to show and describe in detail the results obtained from the performance and load tests that were conducted, in order to evaluate which of the three API implementations was the most scalable and performant. Tests were written through Scala scripts, leveraging the Gatling open-source testing tool.

Note that load testing, aiming to examine how the system behaves during normal and high loads in order to determine if it can handle high loads given a high demand of end-users, has been carried out for its own sake, regardless of how the application will be used in the future.

4.1 Simulation of the external geocoding service

For carrying out the tests, it was not possible to exploit the Nominatim API for reverse geocoding, because of the usage policy of the service: indeed, heavy uses are not allowed (the limit is of 1 request per second and the requests should be limited to a single thread), therefore making parallel calls to the external service would not have been feasible.
Possible alternatives would have been:

- Installing an own Nominatim server;
- Build an own geocoding application service by downloading and using the OSM database (how to deal with data updates?).

Since both solutions would have required a lot of time to be implemented, and as it didn't matter for testing purposes to have a fully functional geocoding service, it has been decided to bypass the obstacle by creating a simple Spring Boot application, deployed as a micro-service on Oracle Cloud, that exposes a "fake" reverse geocoding API: it still receives a latitude and a longitude as input, but actually it returns a couple of *category* and *type* selected *randomly* from a MongoDB, where all the possible key-value correspondences are stored, in such a way simulating the Nominatim API behaviour.

4.2 Gatling, an open-source testing tool

Performance testing aims to determine how a system performs in terms of responsiveness and stability under a particular workload, allowing to discover the upper bound limit a system could handle.

Gatling is an open-source performance testing framework, which uses Scala, Akka and Netty as a technology stack and as its backbone. Since its first release in January 2012, Gatling has had a couple of major releases almost every year, as well as a pretty extensive popularity growth across the performance engineering community.[65]

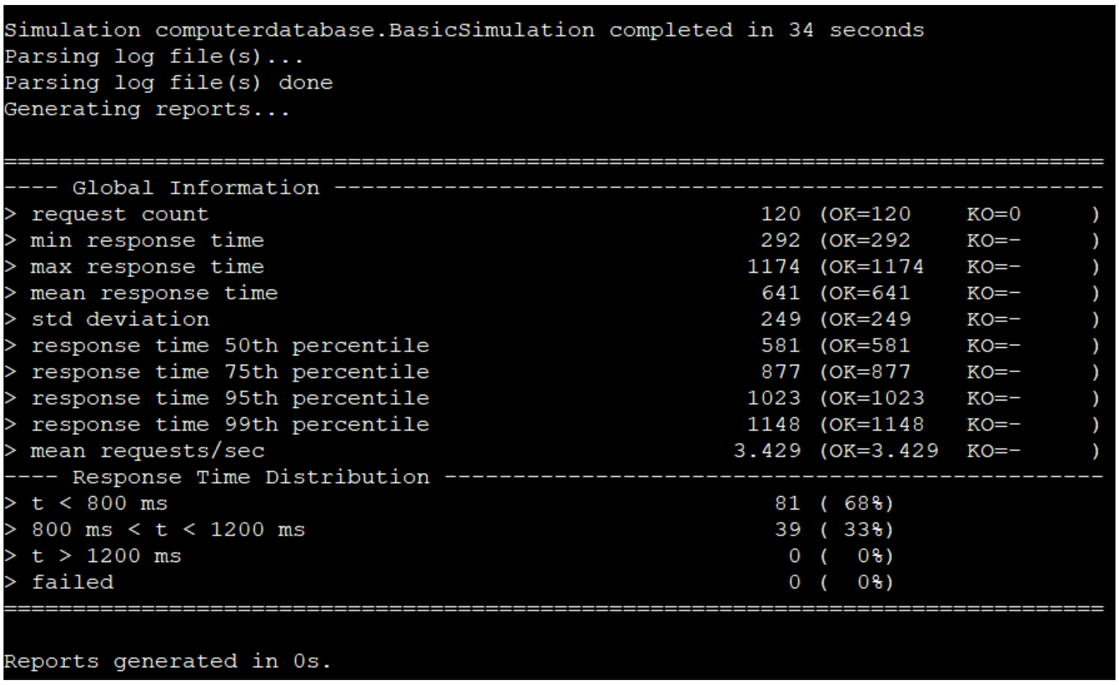
It allows us to create and conduct high-performance, low-maintenance load tests on local and cloud computers.

Advantages of Gatling:

- Being a Scala-based testing platform with a human-readable DSL, it makes writing and running tests easy.
- Possible to add versioning to the source code, in such a way to improve team cooperation and monitor past modifications.
- Support for multithreading: indeed, it makes use of the Netty framework and Akka toolkit, distributed and fully asynchronous by design, that avoid allocating a new thread for each user (a single thread can simulate a list of user trips).
- Support for CI/CD Integration and Reporting (easy connection with real-time tracking systems, e.g., Grafana).
- Built in assertions API allow to conduct various sorts of functional tests alongside speed testing.
- It generates load testing reports that make the information easier to digest, thanks to detailed graphs.

Disadvantages of Gatling:

- Only supports HTTP, WebSockets, server-side events, and JMS (Java Message Service).
- Lots of scripting (it is purely code-based) and lack of support material.[66]



```
Simulation computerdatabase.BasicSimulation completed in 34 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================
---- Global Information --------------------------------------------------------
> request count                                        120 (OK=120    KO=0     )
> min response time                                    292 (OK=292    KO=-     )
> max response time                                   1174 (OK=1174   KO=-     )
> mean response time                                   641 (OK=641    KO=-     )
> std deviation                                        249 (OK=249    KO=-     )
> response time 50th percentile                        581 (OK=581    KO=-     )
> response time 75th percentile                        877 (OK=877    KO=-     )
> response time 95th percentile                       1023 (OK=1023   KO=-     )
> response time 99th percentile                       1148 (OK=1148   KO=-     )
> mean requests/sec                                  3.429 (OK=3.429  KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                            81 ( 68%)
> 800 ms < t < 1200 ms                                  39 ( 33%)
> t > 1200 ms                                            0 (  0%)
> failed                                                 0 (  0%)
================================================================================

Reports generated in 0s.
```

Figure 4.1: Example of console logs printed at the end of a test.

4.2.1 The script

In Listing 4.1, it is reported the Gatling script, in Scala, used for conducting tests on the application under study. The script was developed using the Visual Studio Code IDE and Maven.

The test class extends the basic Simulation class to run the test, where there can be multiple scenarios. It is possible to distinguish three sections:

1. First, we need to provide the HTTP configurations, for which it is necessary to define the base URL that we want to test. In this case, there will be two HTTP configurations: one for the Spring Boot application, the other for the Spring WebFlux application.
2. Then, simulations (one for each API implementation) and scenarios are defined, by specifying the number of repetitions, the API, the header and the pause between a repetition and the following one.
3. Finally, there's the Load Scenario phase. The Gatling simulation base method is *setUp()*: it calls the scenario object and passes the number of users using different injection profiles. There are the different methods which can be injected in the setUp() method to create the scenario of number of users as per our requirement.[65]

Two alternatives have been written (you should comment the one that is not being used):

- A - *rampUsers(numUsers).during(duration)*: injects a given number of users distributed evenly on a time window of a given duration.
- B - *atOnceUsers(numUsers)*: injects a given number of users at once.[65]

```scala
package computerdatabase

import io.gatling.core.Predef._
import io.gatling.http.Predef._
import scala.concurrent.duration._
import scala.util.Random

class BasicSimulation extends Simulation {

    // 1) HTTP configurations
    // Spring Boot configuration
    val httpConfSpringBoot = http.baseUrl("http://193.122.63.101:8082/")
    // WebFlux configuration
    val httpConfWebFlux = http.baseUrl("http://129.159.252.247:8083/")

    // 2) Scenarios definition
    // Spring Boot simulation (num of repetitions may change)
    val simSpringBoot = repeat(3) {
        exec(http("Get information by VIN code with SpringBoot")
            .get("/signals/getByVinCode?cdVinCode="+vinCode)
            .header("Content-Type", "application/json"))
            .pause(1 second, 2 seconds)
    }
    // Spring Boot Async simulation (num of repetitions may change)
    val simSpringBootAsync = repeat(3) {
        exec(http("Get information by VIN code with SpringBoot Async")
            .get("/signals/async/getByVinCode?cdVinCode="+vinCode)
            .header("Content-Type", "application/json"))
            .pause(1 second, 2 seconds)
    }
    // Spring WebFlux simulation (num of repetitions may change)
    val simWebFlux = repeat(3) {
        exec(http("Get information by VIN code with Webflux")
            .get("/signals/getInfoByVin/"+vinCode)
            .header("Content-Type", "text/event-stream"))
            .pause(1 second, 2 seconds)
    }
    // Each scenario consists in executing the simulation just been
   defined
    val scnSpringBoot = scenario("Spring Boot").exec(simSpringBoot)

    val scnSpringBootAsync = scenario("Spring Boot
   Async").exec(simSpringBootAsync)

    val scnWebFlux = scenario("Spring WebFlux").exec(simWebFlux)

    // 3) Load Scenario (use A or B alternatively)
    // A (users growing gradually) - scn and httpConfig may change
    setUp(
        scnSpringBoot.inject(
          rampUsers(numUsers).during(30.seconds)
        ).protocols(httpConfSpringBoot)
    )

    // B (users at once) - scn and httpConfig may change
    setUp(
        scnWebFlux.inject(atOnceUsers(numUsers))
    ).protocols(httpConfWebFlux)
}
```

Listing 4.1: Scala script used for launching tests.

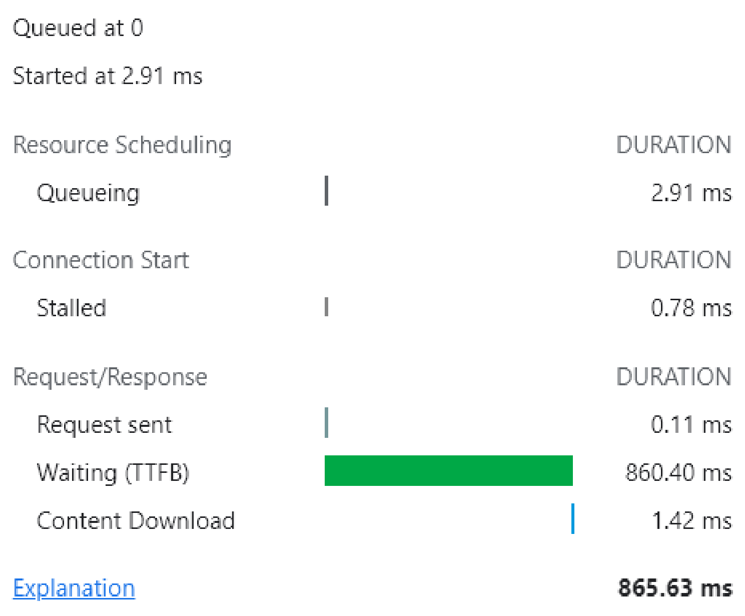
4.3 Results

The analysis on performance testing about the API implementations fully described in the previous chapter starts from looking at the single requests through the timing values recorded by the browser console; then, different scenarios have been created thanks to the Gatling tool in order to simulate a varying number of users making calls.

4.3.1 Single API requests

In order to record a single network request, it was enough to leverage the Chrome DevTools network analysis features: indeed, by default (and so long as it is open), DevTools records all network requests in the Network panel.

In the following three figures, it is possible to find the recorded timings of API requests to the Spring Boot application (synchronous and asynchronous versions) and to the WebFlux one.

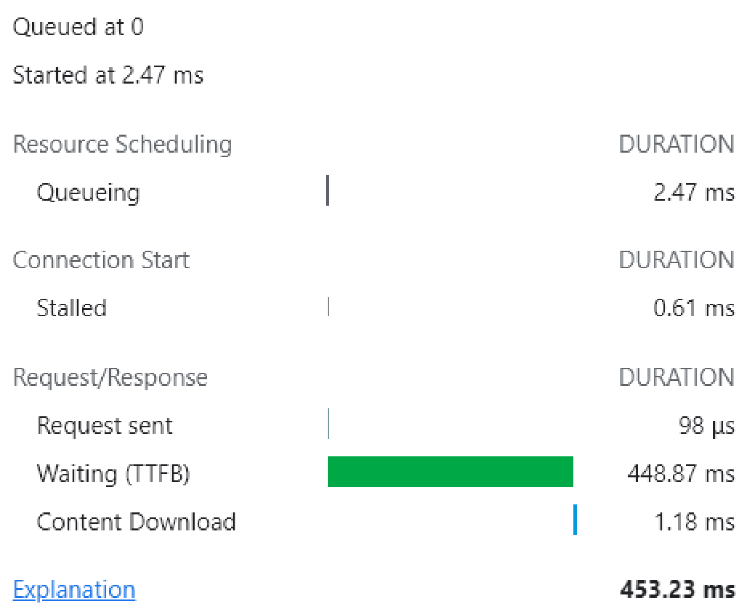


Queued at 0

Started at 2.91 ms

Resource Scheduling	DURATION
Queueing	2.91 ms
Connection Start	**DURATION**
Stalled	0.78 ms
Request/Response	**DURATION**
Request sent	0.11 ms
Waiting (TTFB)	860.40 ms
Content Download	1.42 ms
Explanation	**865.63 ms**

Figure 4.2: Timings of a request to Spring Boot application.

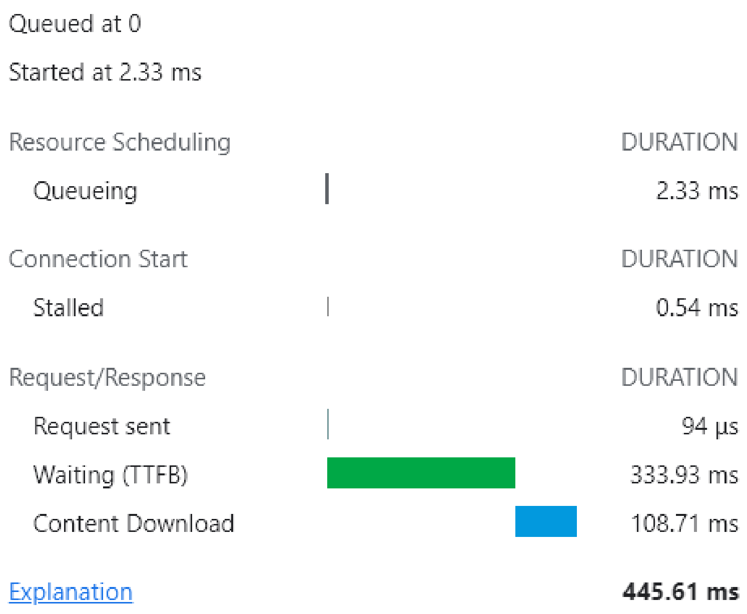


Queued at 0

Started at 2.47 ms

Resource Scheduling	DURATION
Queueing	2.47 ms
Connection Start	**DURATION**
Stalled	0.61 ms
Request/Response	**DURATION**
Request sent	98 µs
Waiting (TTFB)	448.87 ms
Content Download	1.18 ms
Explanation	**453.23 ms**

Figure 4.3: Timings of the async request to Spring Boot application.



Queued at 0

Started at 2.33 ms

Resource Scheduling	DURATION
Queueing	2.33 ms
Connection Start	DURATION
Stalled	0.54 ms
Request/Response	DURATION
Request sent	94 μs
Waiting (TTFB)	333.93 ms
Content Download	108.71 ms
Explanation	**445.61 ms**

Figure 4.4: Timings of the same request to Spring WebFlux application.

The input of the three requests was a VIN code with about 113 pairs of latitude and longitude values (that means 113 downstream calls to the external geocoding service were necessary).

It is evident that the first and synchronous API implementation, as expected, reveals to be the worst (it takes almost twice as much as the other ones), while the asynchronous and the reactive implementations show quite similar final values, but by looking exclusively at the TTFB (i.e., *Time To First Byte*), the reactive one is clearly the fastest.

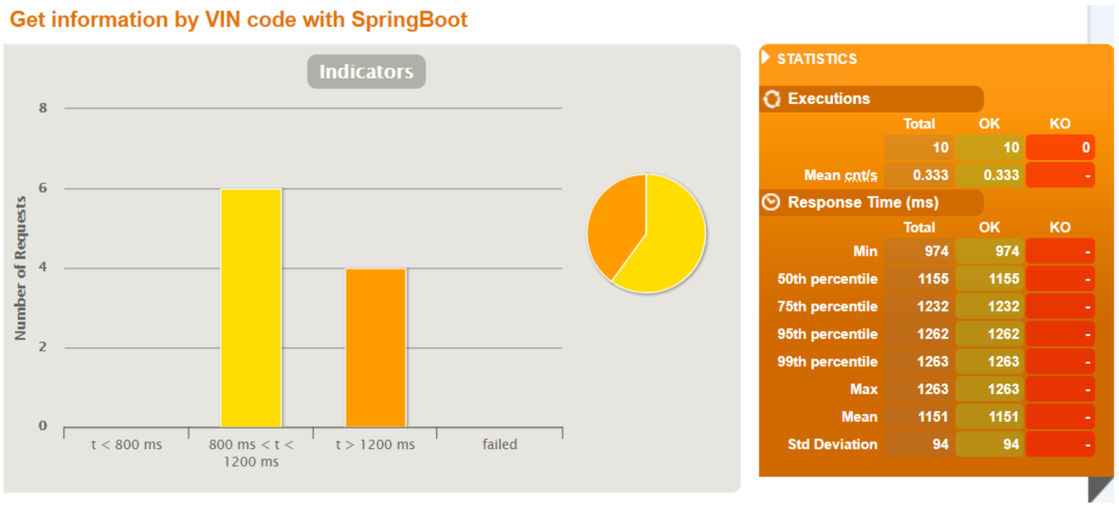
4.3.2 Multiple API requests

This section is devoted to analyze some scenarios created with Gatling, which provides HTML report of the tests once executed. It helps to visualize the scenario deeply. All the information (such as successful requests, failed request, response time for 50th, 75th, 95th and 99th percentage, min and max time) are present in the reports with proper visualization charts.

10 requests from 10 users in 30 seconds

The first scenario consists in simulating 10 users, gradually injected in a time window of 30 seconds, making 10 requests (1 user = 1 request).

As the charts show, Spring Boot Async and Spring WebFlux behave quite similarly and both are able to respond to the requests in less than 800 ms. On the contrary, Spring Boot takes from 800 to 1200 ms to respond to the 60% of the requests, while the remaining ones encounter a delay of more than 1200 ms.



STATISTICS

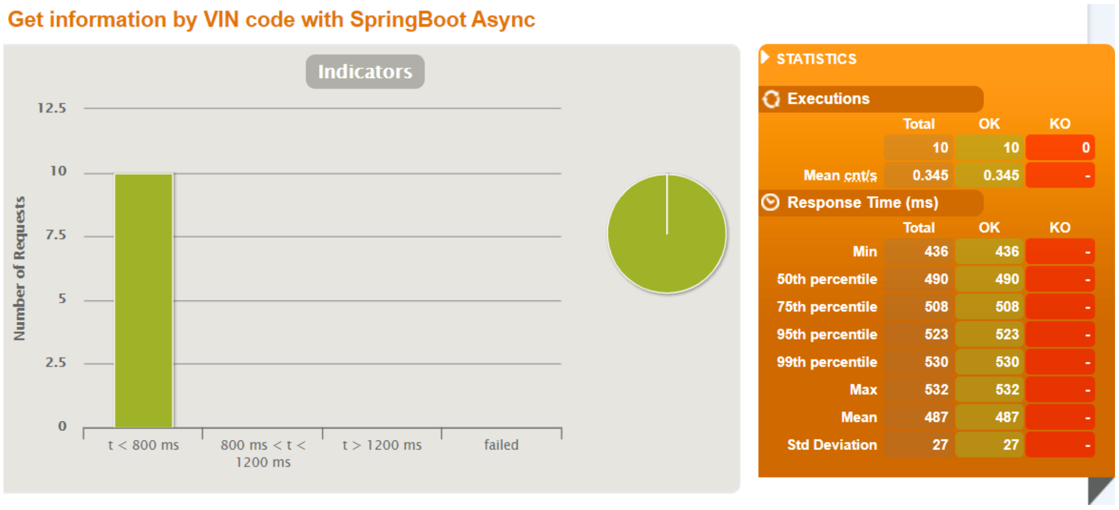
Executions

	Total	OK	KO
	10	10	0
Mean cnt/s	0.333	0.333	-

Response Time (ms)

	Total	OK	KO
Min	974	974	-
50th percentile	1155	1155	-
75th percentile	1232	1232	-
95th percentile	1262	1262	-
99th percentile	1263	1263	-
Max	1263	1263	-
Mean	1151	1151	-
Std Deviation	94	94	-

Figure 4.5: 10 requests by 10 users in 30 seconds to Spring Boot.



STATISTICS

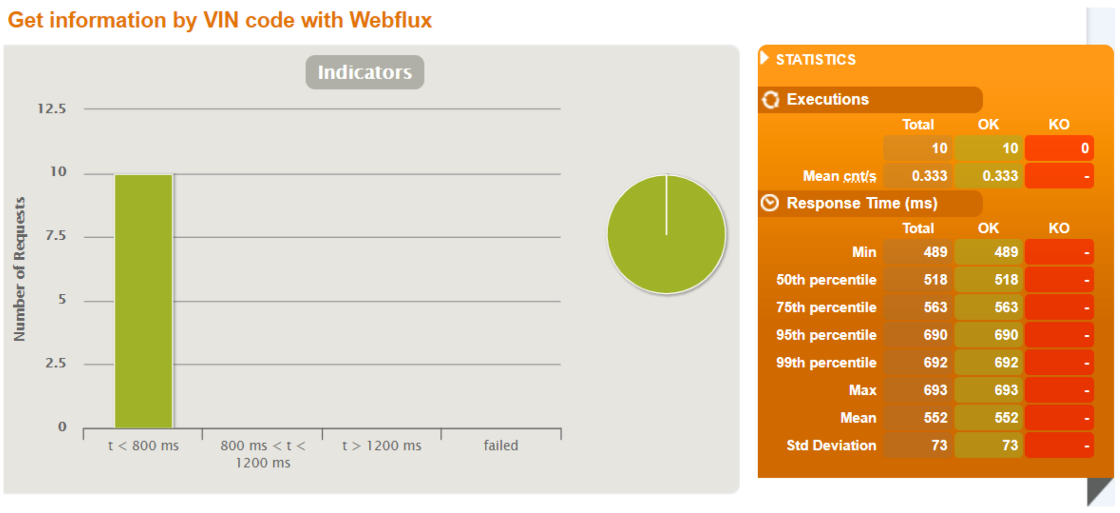
Executions

	Total	OK	KO
	10	10	0
Mean cnt/s	0.345	0.345	-

Response Time (ms)

	Total	OK	KO
Min	436	436	-
50th percentile	490	490	-
75th percentile	508	508	-
95th percentile	523	523	-
99th percentile	530	530	-
Max	532	532	-
Mean	487	487	-
Std Deviation	27	27	-

Figure 4.6: 10 requests by 10 users in 30 seconds to Spring Boot Async.



Executions

	Total	OK	KO
	10	10	0
Mean cnt/s	0.333	0.333	-

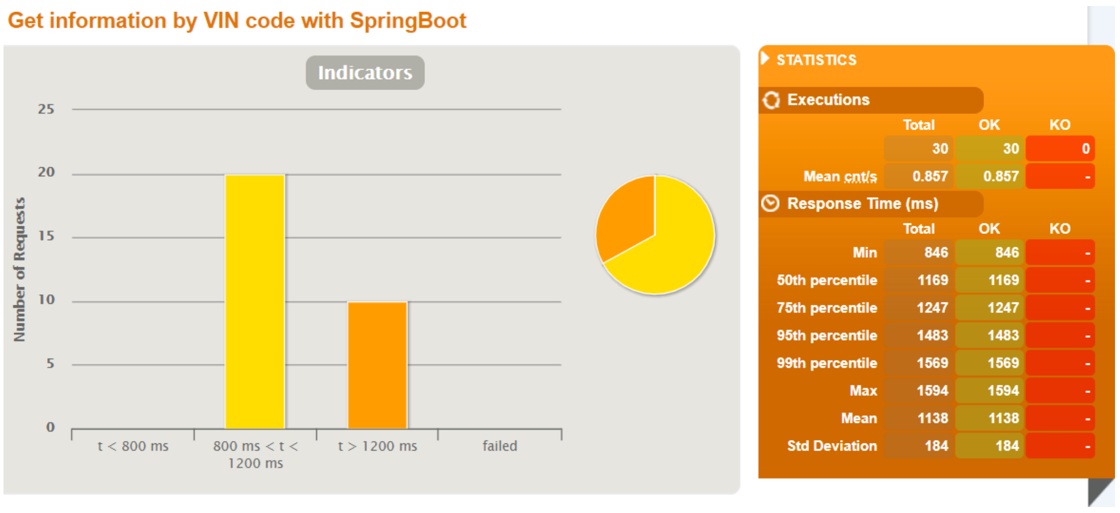
Response Time (ms)

	Total	OK	KO
Min	489	489	-
50th percentile	518	518	-
75th percentile	563	563	-
95th percentile	690	690	-
99th percentile	692	692	-
Max	693	693	-
Mean	552	552	-
Std Deviation	73	73	-

Figure 4.7: 10 requests by 10 users in 30 seconds to Spring WebFlux.

30 requests from 10 users in 30 seconds

The second scenario simulates 10 users, gradually injected in a time window of 30 seconds, making 30 requests (1 user = 3 requests).
As in the previous scenario, similar values can be found for Spring Boot Async and Spring WebFlux (and both, again, are able to respond to all the requests in less than 800 ms). Spring Boot instead takes from 800 to 1200 ms to respond to the 67% of the requests, while the remaining ones encounter a delay of more than 1200 ms.



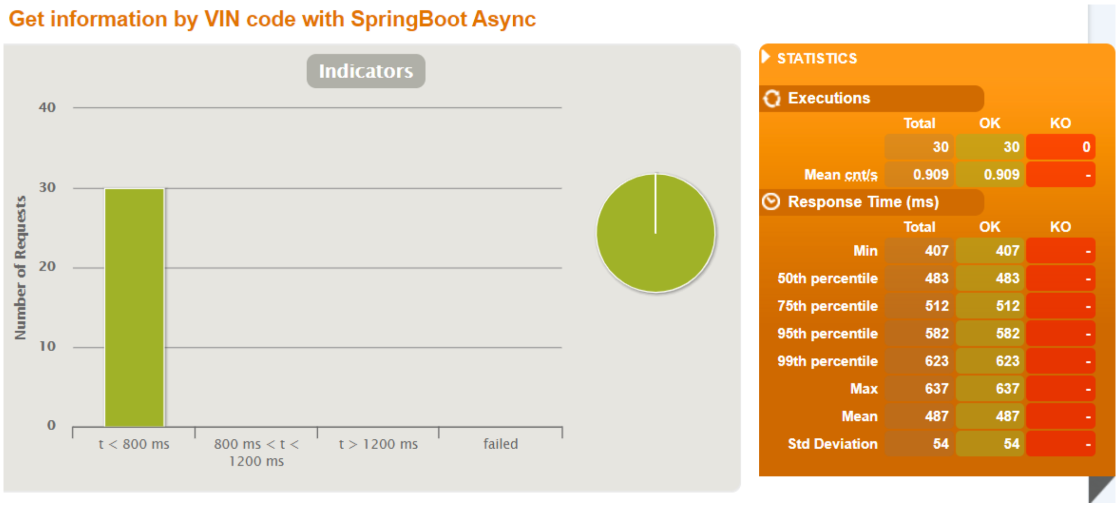
Executions

	Total	OK	KO
	30	30	0
Mean cnt/s	0.857	0.857	-

Response Time (ms)

	Total	OK	KO
Min	846	846	-
50th percentile	1169	1169	-
75th percentile	1247	1247	-
95th percentile	1483	1483	-
99th percentile	1569	1569	-
Max	1594	1594	-
Mean	1138	1138	-
Std Deviation	184	184	-

Figure 4.8: 30 requests by 10 users in 30 seconds to Spring Boot.



STATISTICS

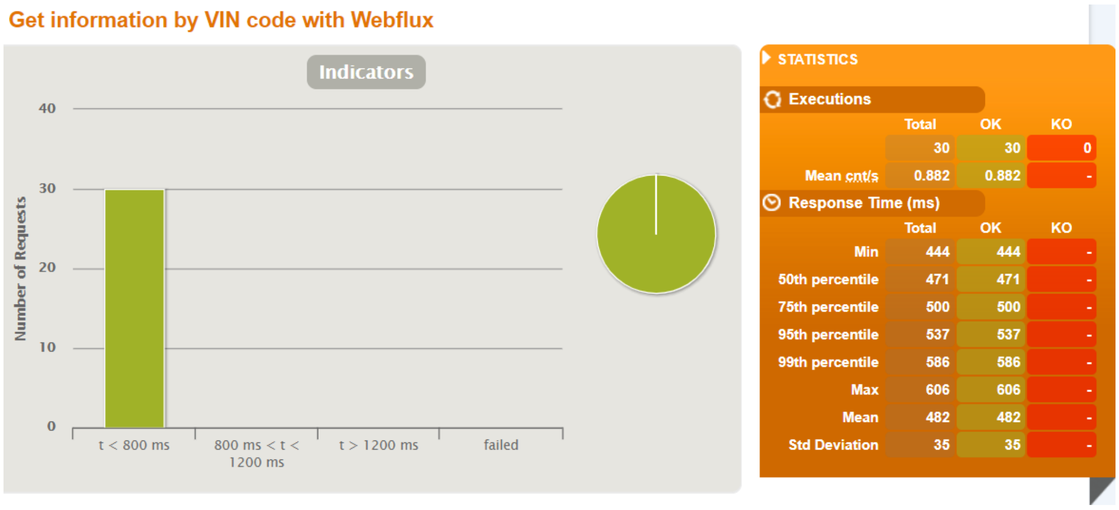
Executions

	Total	OK	KO
	30	30	0
Mean cnt/s	0.909	0.909	-

Response Time (ms)

	Total	OK	KO
Min	407	407	-
50th percentile	483	483	-
75th percentile	512	512	-
95th percentile	582	582	-
99th percentile	623	623	-
Max	637	637	-
Mean	487	487	-
Std Deviation	54	54	-

Figure 4.9: 30 requests by 10 users in 30 seconds to Spring Boot Async.



STATISTICS

Executions

	Total	OK	KO
	30	30	0
Mean cnt/s	0.882	0.882	-

Response Time (ms)

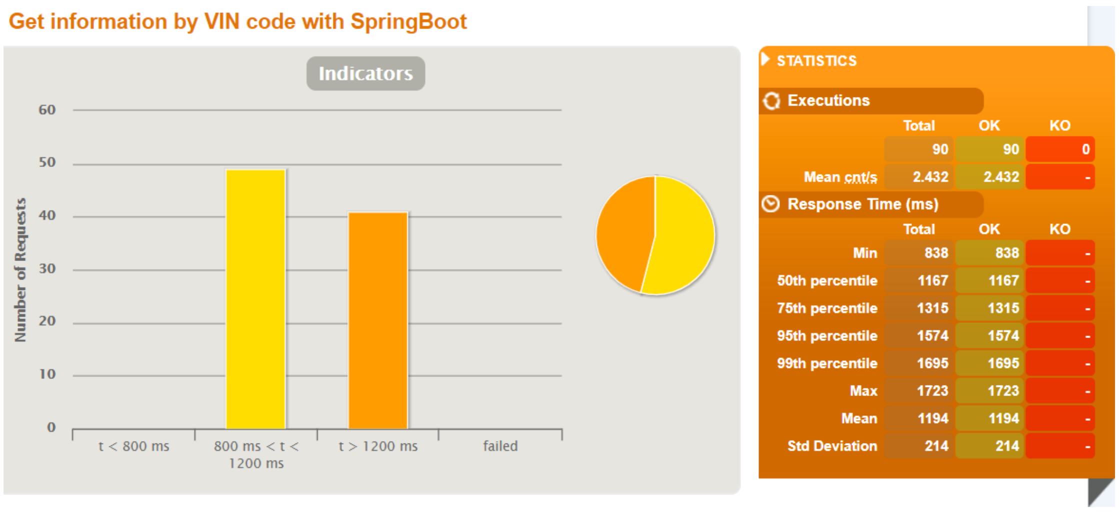
	Total	OK	KO
Min	444	444	-
50th percentile	471	471	-
75th percentile	500	500	-
95th percentile	537	537	-
99th percentile	586	586	-
Max	606	606	-
Mean	482	482	-
Std Deviation	35	35	-

Figure 4.10: 30 requests by 10 users in 30 seconds to Spring WebFlux.

90 requests from 30 users in 30 seconds

The third scenario simulates 30 users, gradually injected in a time window of 30 seconds, making 90 requests (1 user = 3 requests).
Spring Boot here exhibits a behaviour similar to what we've notice in the previous scenarios; on the other hand, we begin to note differences (even not very significant) between Spring Boot Async and Spring WebFlux: indeed, Spring Boot Async responds to the 31% of the requests with a delay between 800 and 1200 ms and to the 4% with a delay greater than 1200 ms, while Spring WebFlux responds with a delay between 800 and 1200 ms only to the 29% of the requests. Anyway, both

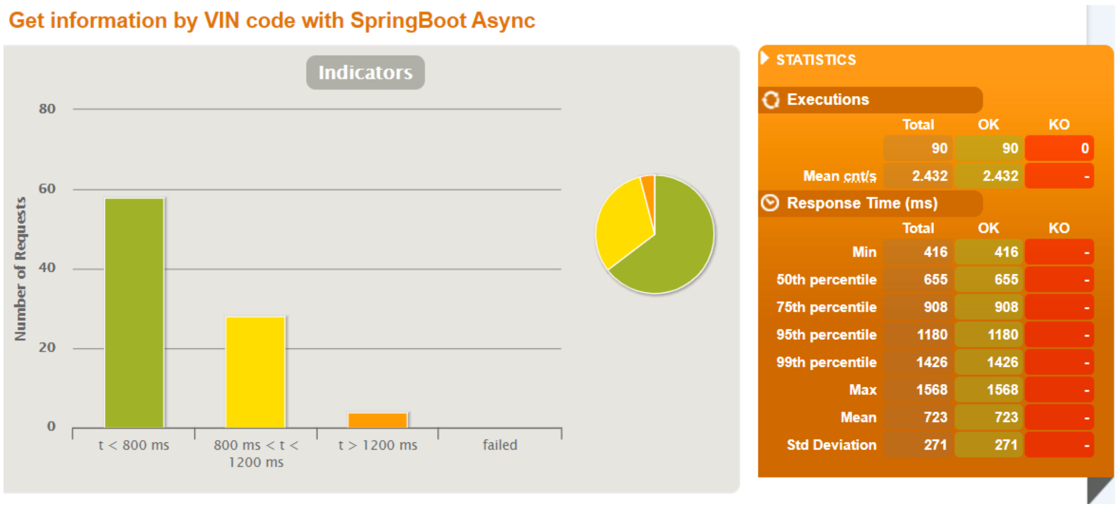
handles most part of the requests in less than 800 ms.



	Total	OK	KO
	90	90	0
Mean cnt/s	2.432	2.432	-

	Total	OK	KO
Min	838	838	-
50th percentile	1167	1167	-
75th percentile	1315	1315	-
95th percentile	1574	1574	-
99th percentile	1695	1695	-
Max	1723	1723	-
Mean	1194	1194	-
Std Deviation	214	214	-

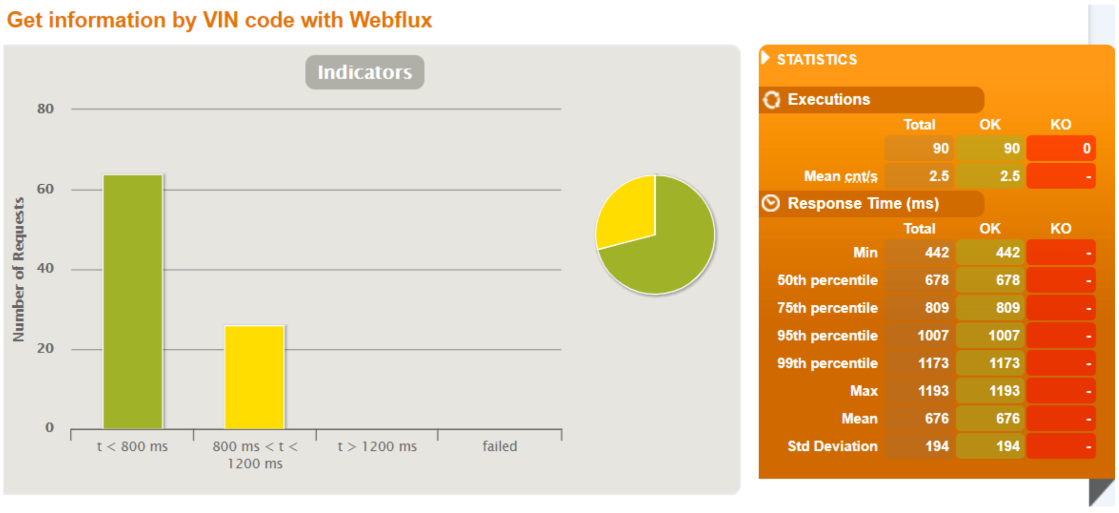
Figure 4.11: 90 requests by 30 users in 30 seconds to Spring Boot.



	Total	OK	KO
	90	90	0
Mean cnt/s	2.432	2.432	-

	Total	OK	KO
Min	416	416	-
50th percentile	655	655	-
75th percentile	908	908	-
95th percentile	1180	1180	-
99th percentile	1426	1426	-
Max	1568	1568	-
Mean	723	723	-
Std Deviation	271	271	-

Figure 4.12: 90 requests by 30 users in 30 seconds to Spring Boot Async.



STATISTICS

Executions

	Total	OK	KO
	90	90	0
Mean cnt/s	2.5	2.5	-

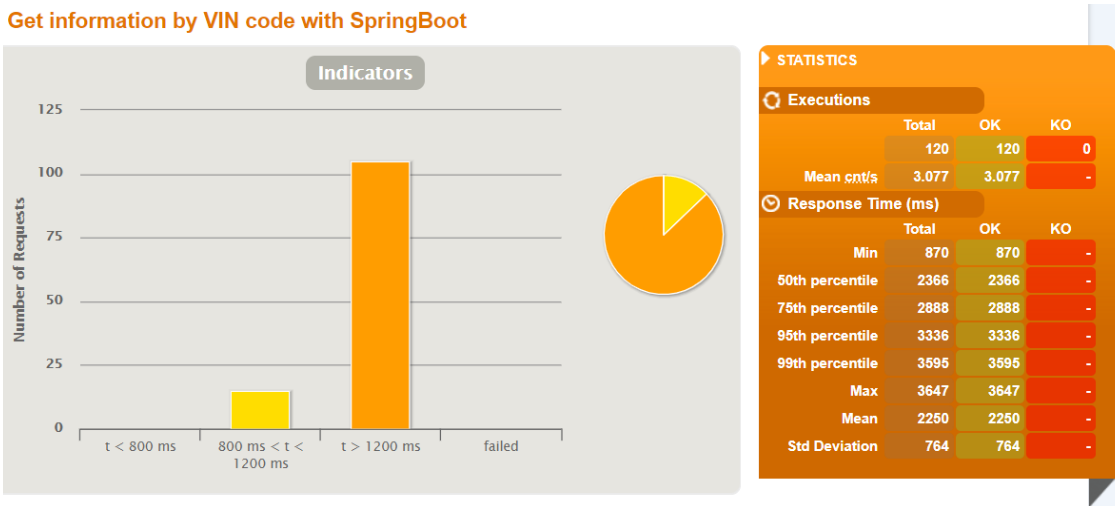
Response Time (ms)

	Total	OK	KO
Min	442	442	-
50th percentile	678	678	-
75th percentile	809	809	-
95th percentile	1007	1007	-
99th percentile	1173	1173	-
Max	1193	1193	-
Mean	676	676	-
Std Deviation	194	194	-

Figure 4.13: 90 requests by 30 users in 30 seconds to Spring WebFlux.

120 requests from 40 users in 30 seconds

The fourth scenario simulates 40 users, gradually injected in a time window of 30 seconds, making 120 requests (1 user = 3 requests).
Spring Boot responds with a small delay only to the 13% of the requests, while the remaining 87% encounters a delay of more than 1200 ms.



STATISTICS

Executions

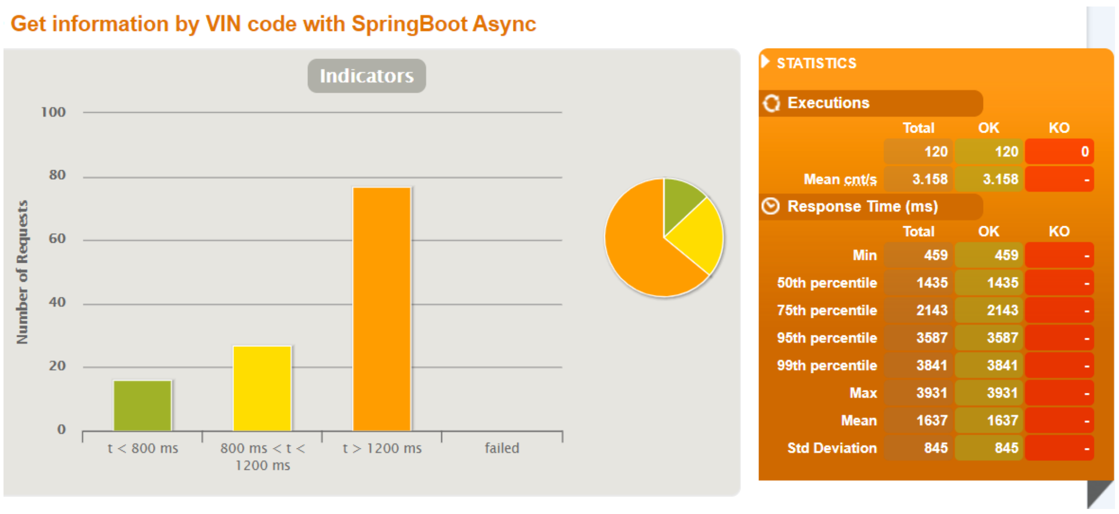
	Total	OK	KO
	120	120	0
Mean cnt/s	3.077	3.077	-

Response Time (ms)

	Total	OK	KO
Min	870	870	-
50th percentile	2366	2366	-
75th percentile	2888	2888	-
95th percentile	3336	3336	-
99th percentile	3595	3595	-
Max	3647	3647	-
Mean	2250	2250	-
Std Deviation	764	764	-

Figure 4.14: 120 requests by 40 users in 30 seconds to Spring Boot.

Spring Boot Async and Spring WebFlux behave clearly differently this time: while Spring WebFlux maintains a similar trend from what we've seen so far (it responds with a small delay only to the 33% of the requests, still managing efficiently the most part of them), the performance of Spring Boot Async downgrades, since the most part of the requests (64%) encounter a delay of more than 1200 ms.



STATISTICS

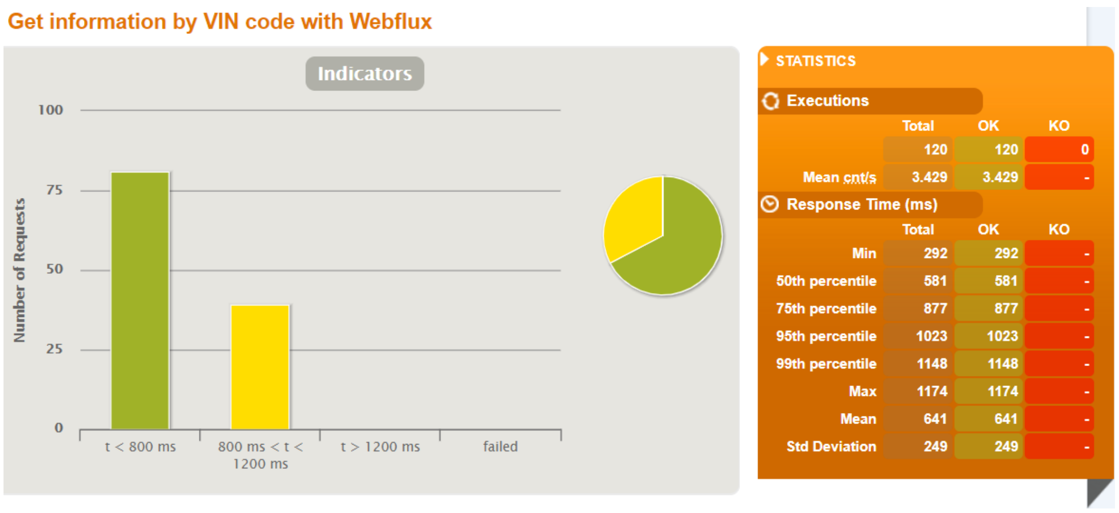
Executions

	Total	OK	KO
	120	120	0
Mean cnt/s	3.158	3.158	-

Response Time (ms)

	Total	OK	KO
Min	459	459	-
50th percentile	1435	1435	-
75th percentile	2143	2143	-
95th percentile	3587	3587	-
99th percentile	3841	3841	-
Max	3931	3931	-
Mean	1637	1637	-
Std Deviation	845	845	-

Figure 4.15: 120 requests by 40 users in 30 seconds to Spring Boot Async.



STATISTICS

Executions

	Total	OK	KO
	120	120	0
Mean cnt/s	3.429	3.429	-

Response Time (ms)

	Total	OK	KO
Min	292	292	-
50th percentile	581	581	-
75th percentile	877	877	-
95th percentile	1023	1023	-
99th percentile	1148	1148	-
Max	1174	1174	-
Mean	641	641	-
Std Deviation	249	249	-

Figure 4.16: 120 requests by 40 users in 30 seconds to Spring WebFlux.

Here below it is possible to find two graphs that show how the number of requests completed varies in the time in this specific scenario.

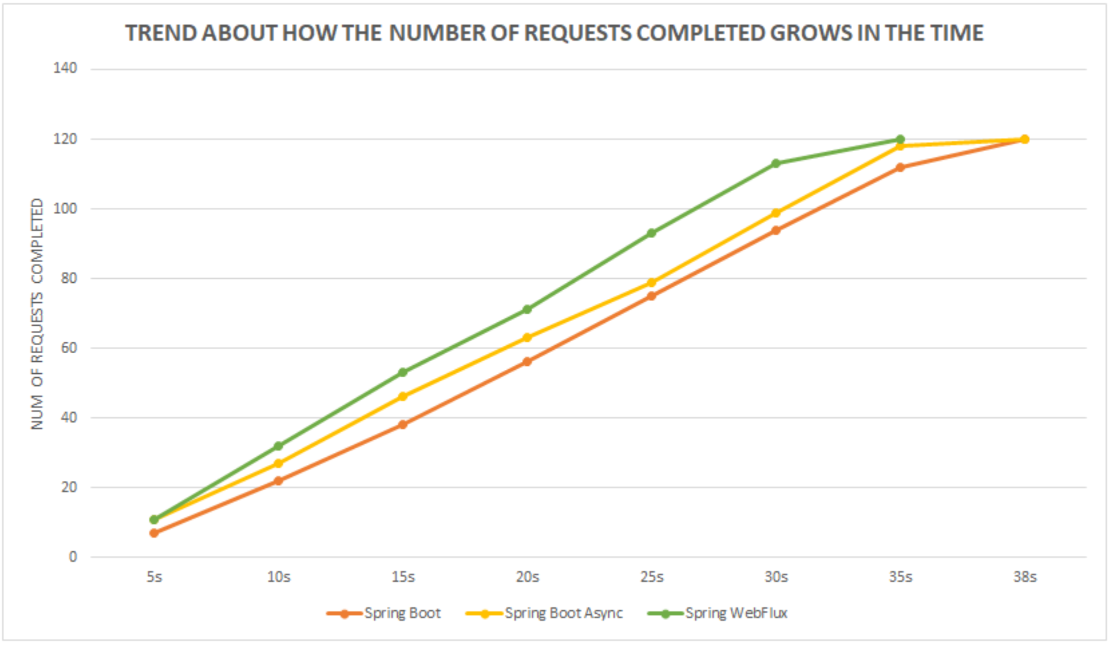


Figure 4.17: Trend about how the number of requests completed grows in the time in the three cases.

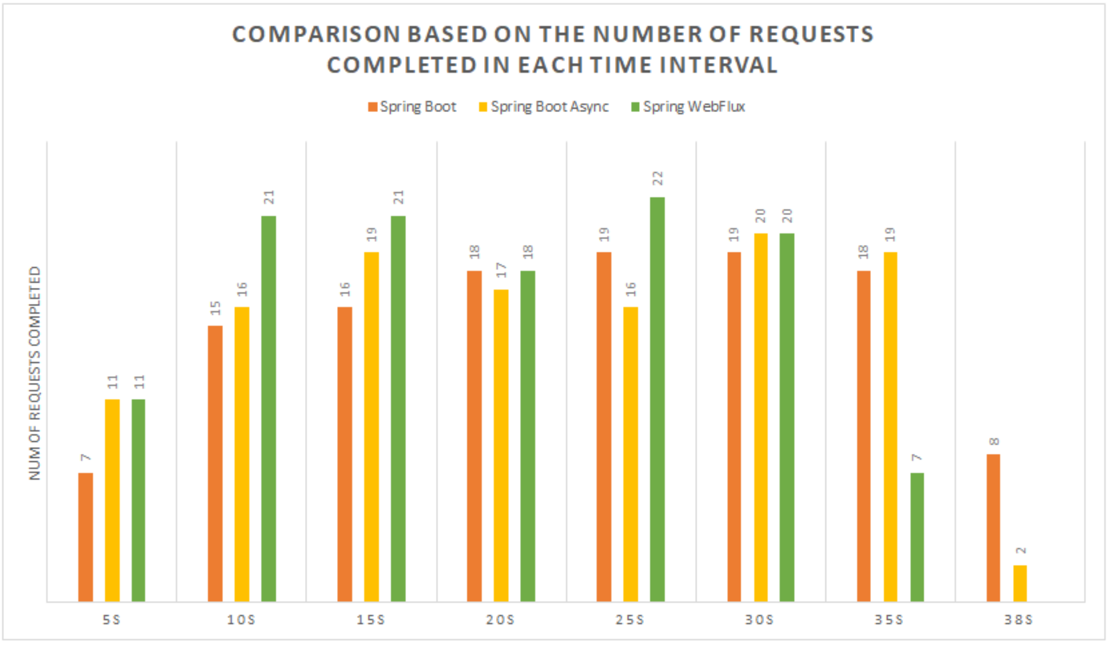
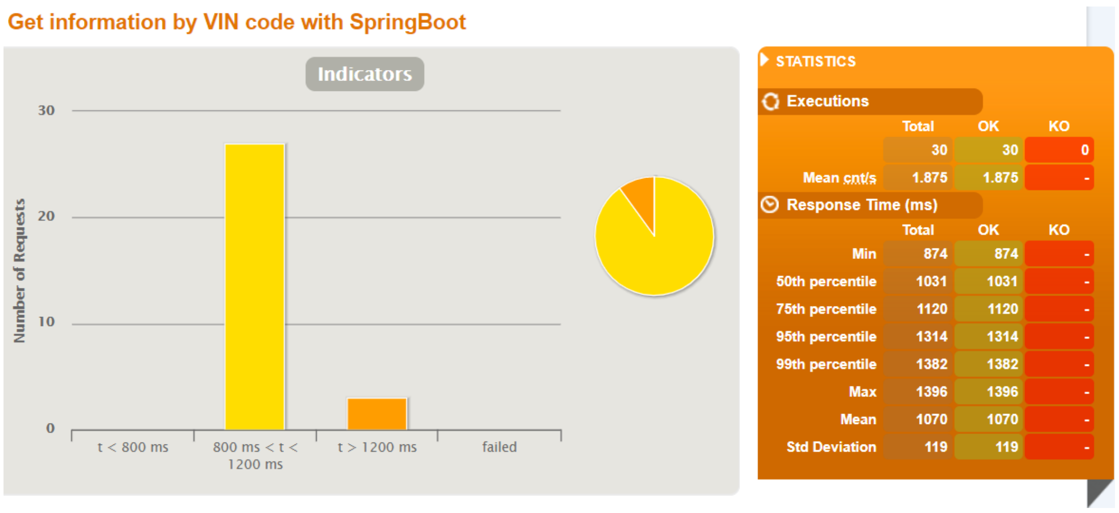


Figure 4.18: Comparison based on the number of requests completed in each time interval.

30 requests from 10 users in 10 seconds

Finally, the last scenario simulates again 10 users, gradually injected in a smaller time window of only 10 seconds, making 30 requests (1 user = 3 requests).
Spring Boot responds to the most part of the requests (90%) with a small delay between 800 and 1200 ms; Spring Boot Async handles a bit more than a half of the requests (53%) in less than 800 ms, while the 27% of them encounter a small delay and the 20% of them a delay of more than 1200 ms; Spring WebFlux, instead, is able to respond to the 80% of the requests in less than 800 ms, while the remaining ones encounter a small (10%) or a high (remaining 10%) delay, in such a way it confirms to be the better of the three also in this case.



STATISTICS

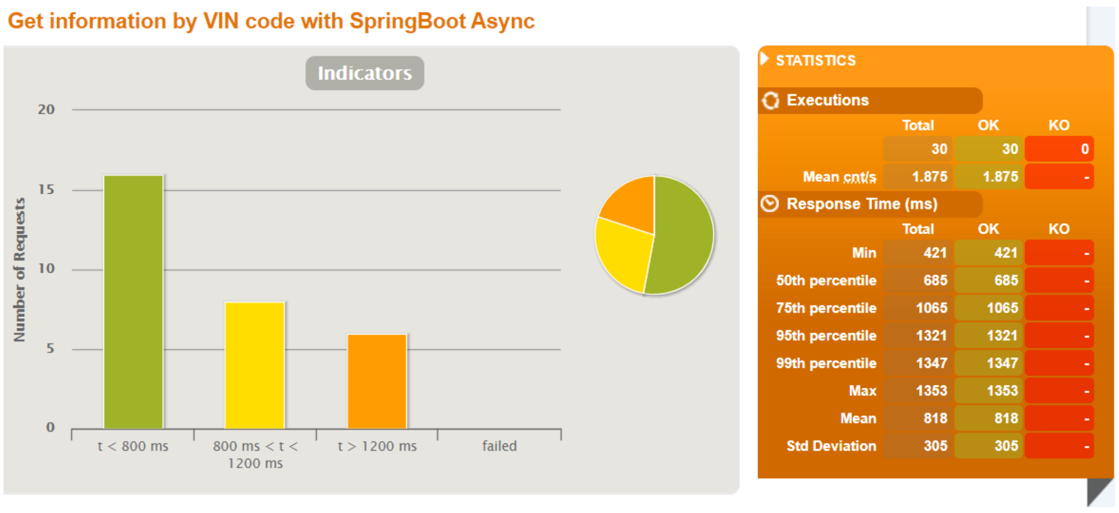
Executions

	Total	OK	KO
	30	30	0
Mean cnt/s	1.875	1.875	-

Response Time (ms)

	Total	OK	KO
Min	874	874	-
50th percentile	1031	1031	-
75th percentile	1120	1120	-
95th percentile	1314	1314	-
99th percentile	1382	1382	-
Max	1396	1396	-
Mean	1070	1070	-
Std Deviation	119	119	-

Figure 4.19: 30 requests by 10 users in 10 seconds to Spring Boot.



STATISTICS

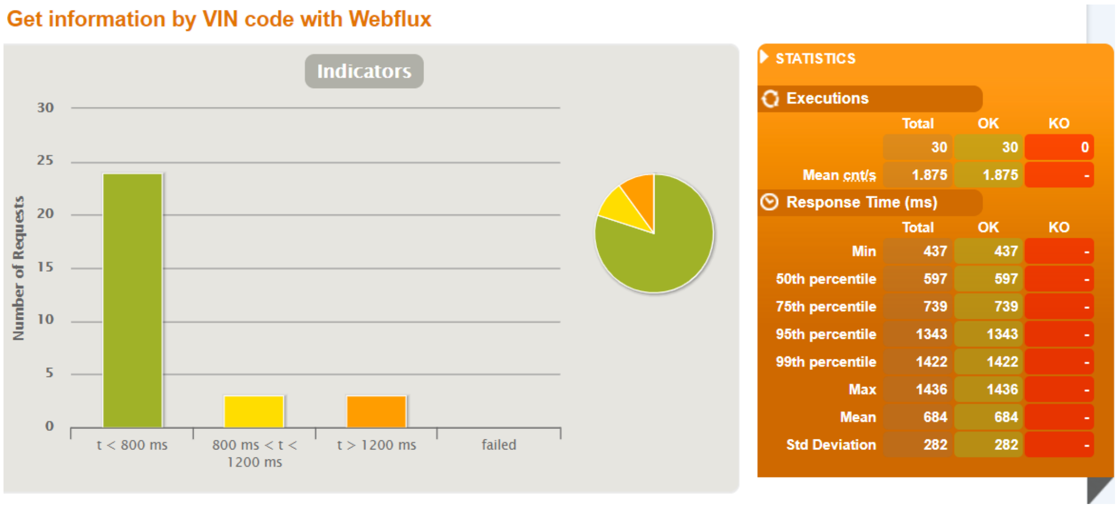
Executions

	Total	OK	KO
	30	30	0
Mean cnt/s	1.875	1.875	-

Response Time (ms)

	Total	OK	KO
Min	421	421	-
50th percentile	685	685	-
75th percentile	1065	1065	-
95th percentile	1321	1321	-
99th percentile	1347	1347	-
Max	1353	1353	-
Mean	818	818	-
Std Deviation	305	305	-

Figure 4.20: 30 requests by 10 users in 10 seconds to Spring Boot Async.



Executions

	Total	OK	KO
	30	30	0
Mean cnt/s	1.875	1.875	-

Response Time (ms)

	Total	OK	KO
Min	437	437	-
50th percentile	597	597	-
75th percentile	739	739	-
95th percentile	1343	1343	-
99th percentile	1422	1422	-
Max	1436	1436	-
Mean	684	684	-
Std Deviation	282	282	-

Figure 4.21: 30 requests by 10 users in 10 seconds to Spring WebFlux.

Chapter 5

Conclusions

5.1 Final evaluations

By analyzing the results obtained with performance and load testing, we can conclude a service written with reactive programming will be able to withstand a much higher load, since the server will be non-blocking with its asynchronous nature, and able to handle more requests with the same amount of hardware. Moreover, Spring WebFlux allows for more scalability, a stack immune to latency (useful for micro services-oriented architecture), and better stream processing capabilities, offering performance improvements, cost reduction, and tools to implement complex threading operations.

The key expected benefit of reactive and non-blocking is the ability to scale with a small, fixed number of threads and less memory. That makes applications more resilient under load, because they scale in a more predictable way. Scaling with a small number of threads may sound contradictory, but if the current thread is never blocked (relying on callbacks instead) there is no need for extra threads, as there are no blocking calls to absorb.

However, what we should take into consideration is that one of the biggest disadvantages of the Webflux is that it is reactive, a fact that brings a lot of additional complexity and needs time to be thoroughly understood and appreciated. To this, it is added the difficulty of debugging: when things are happening in asynchronous manner, keeping track of the sequence of events may become really difficult.

Said that, it's clear there are certain use cases where it does not make sense to use it, or where its use brings more problems than it solves. Imperative programming is the easiest way to write, understand, and debug code, and it would still be a good choice in case of no need for asynchronicity.

Let's briefly discuss the main Spring WebFlux components that contributed to the improvement of the application and why they proved to be better than the corresponded counterparts used in Spring Boot, starting from a note about CompletableFuture library.

CompletableFuture: why not?

Java CompletableFuture API was an important step for supporting asynchronous programming in Java, and the API implementation that leverages it performs quite well up to a certain limit, but it doesn't (and couldn't) address the elasticity requirement of the Reactive Manifesto by providing backpressure, as it operates on the level of a single call. Besides, differently from the imperative style, CompletableFuture code may look not so simple to maintain, and the API is neither intuitive nor easy to use.

Java 8 Streams vs Reactive Streams

The powerful of Java 8 Stream library make it easy to describe data transformations and split them across multiple threads to boost speed, but Reactive Streams go beyond: they are streams of data that arrive asynchronously, allowing to process data without blocking. Using Reactive Streams you can easily build non-blocking web servers with dramatically improved throughput and smaller thread pools. Furthermore, a Java Stream is pull-based and single use, while in WebFlux there is a hybrid push/pull model and it is possible to subscribe multiple times to the same Reactive Stream.

WebClient vs RestTemplate

Opposite to RestTemplate, WebClient is asynchronous and non-blocking in nature: using it, the client doesn't need to wait until the response comes back, instead it will be notified through a callback method when there is a response from the server. On the contrary, RestTemplate blocks the request threads while WebClient does not. WebClient can be used to make synchronous requests, too, but the opposite is not true (RestTemplate cannot make asynchronous requests).

JDBC vs R2DBC

R2DBC is the reactive (i.e., non-blocking) alternative to JDBC, and it is often used for the ability to better scale. It is inherently asynchronous, so basically the complete application should be too in order to get the benefits.

At high concurrency, the benefits of using R2DBC and Spring WebFlux instead of JDBC and Spring Boot mainly regard:

- Less CPU and memory are required to process a single request;
- Response times and throughput at high concurrency are better.

On the other hand, there are still some challenges when using R2DBC:

- Limited availability (not all relational databases have reactive drivers available);
- JPA cannot deal with reactive repositories such as provided by Spring Data R2DBC. This means it will be necessary to do more things manually when using R2DBC;
- Application servers still depend on JDBC.[67]

5.2 Future developments

Regarding the application and its specific usage, once analyzed the three different API implementations, the company will definitely opt for the reactive one.

As represented in Chapter 3 (Section 3.1.2), the architecture of the final application will involve adding other important components, as well as planning the development of an own geocoding service.

The analyses carried out so far have taken into consideration only four types of signals, but it is clearly possible to search for strategies to manage many others of them, with the ultimate goal of categorizing the customer in the best possible way.

Bibliography

[1] *Asynchronous Programming.* Oct. 2020. URL: https://medium.com/swlh/asynchronous-programming-296a656fe90d.

[2] *Synchronous and Asynchronous I/O.* 2018. URL: https://docs.microsoft.com/en-us/windows/win32/fileio/synchronous-and-asynchronous-i-o.

[3] Anna Eriksson. *An Introduction to Reactive Programming With Spring.* June 2020. URL: https://dzone.com/articles/an-introduction-to-reactive-programming-with-sprin.

[4] *asynchronous.* In: *Oxford Learner's Dictionaries.* Oxford University Press. URL: https://www.oxfordlearnersdictionaries.com/definition/english/asynchronous?q=asynchronous.

[5] Jonas Bonér and Viktor Klang. *Reactive Programming versus Reactive Systems.* URL: https://www.lightbend.com/white-papers-and-reports/reactive-programming-versus-reactive-systems.

[6] Angela Stringfellow. *When to Use (and Not to Use) Asynchronous Programming.* Sept. 2017. URL: https://stackify.com/when-to-use-asynchronous-programming.

[7] Keval Patel. *What is Reactive Programming?* Dec. 2016. URL: https://medium.com/@kevalpatel2106/what-is-reactive-programming-da37c1611382.

[8] Gustav Hochbergs. «Reactive programming and its effect on performance and the development process». Master's Thesis. Lund University, 2017. Chap. 3.

[9] Grace Jansen and Emily Jiang. *Defining the term reactive.* July 2020. URL: https://developer.ibm.com/articles/defining-the-term-reactive/.

[10] Clement Escoffier. *5 Things to Know About Reactive Programming.* June 2017. URL: https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming.

[11] Dave Syer. *Notes on Reactive Programming Part I: The Reactive Landscape.* June 2016. URL: https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape.

[12] V. Saravanan. *Functional Reactive Programming using Swift.* URL: https://flexiple.com/ios/functional-reactive-programming-using-swift/.

[13] Jonas Bonér et al. *The Reactive Principles. Design Principles for Distributed Applications.* Nov. 2020. URL: https://principles.reactive.foundation/.

[14] Tom Nolle. *Reactive Programming.* Mar. 2021. URL: https://searchapparchitecture.techtarget.com/definition/reactive-programming.

[15] Mohammad Farooq. *Reactive Programming: Using Async Data Streams to Propagate Change.* Sept. 2017. URL: https://dzone.com/articles/reactive-programming-using-async-data-streams-to-propogate-change.

[16] André Staltz. *The introduction to Reactive Programming you've been missing.* 2014. URL: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754.

[17] Mauro Celani and Cristian Bianco. *Reactive VS Imperative.* Feb. 2020. URL: https://techblog.smc.it/it/2020-04-19/reactive-vs-imperative.

[18] Vladimir Sinkevich. *The Essence of Reactive Programming in Java.* July 2018. URL: https://www.scnsoft.com/blog/java-reactive-programming.

[19] Mohit Snehal. *Reactive Programming: Why, What and When?* June 2020. URL: https://blog.devgenius.io/reactive-programming-why-what-and-when-e00495cda9c4.

[20] Netflix Technology Blog. *Reactive Programming in the Netflix API with RxJava.* Feb. 2013. URL: https://netflixtechblog.com/reactive-programming-in-the-netflix-api-with-rxjava-7811c3a1496a.

[21] Stefan Nothaas. *Java Reactive Programming. Effective Usage in a Real World Application.* Mar. 2021. URL: https://tech.trivago.com/2021/03/16/java-reactive-programming-effective-usage-in-a-real-world-application/.

[22] Ekaterina Novoseltseva. *Reactive Architecture Benefits and Use Cases.* May 2020. URL: https://dzone.com/articles/reactive-architecture-benefits-amp-use-cases.

[23] Zhanyong Wan and Paul Hudak. *ACM SIGPLAN Notices. Functional reactive programming from first principles.* Vol. 35. May 2000.

[24] Navdeep Singh. *A quick introduction to Functional Reactive Programming (FRP).* Mar. 2018. URL: https://www.freecodecamp.org/news/functional-reactive-programming-frp-imperative-vs-declarative-vs-reactive-style-84878272c77f/.

[25] Dan Lew. *An Introduction to Functional Reactive Programming.* July 2017. URL: https://blog.danlew.net/2017/07/27/an-introduction-to-functional-reactive-programming/.

[26] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. *The Reactive Manifesto.* Sept. 2014. URL: https://www.reactivemanifesto.org/.

[27] Richard John Anthony. *Systems Programming. Designing and Developing Distributed Applications.* 2016. Chap. 6.

[28] *Reactive Streams.* URL: http://www.reactive-streams.org/.

[29] Mahdi Chtioui. *ReactiveX: Reactive Programming Principles.* Aug. 2019. URL: https://medium.com/@mahdichtioui/reactivex-reactive-programming-principles-dbb1bafa8384.

[30] *ReactiveX.* URL: http://reactivex.io/.

[31] Stephane Maldini and Simon Baslé. *Reactor 3 Reference Guide.* URL: https://projectreactor.io/docs/core/release/reference/index.html.

[32] John Thompson. *What's New in Spring Framework 5?* July 2017. URL: https://dzone.com/articles/whats-new-in-spring-framework-5.

[33] Baeldung. *Introduction to Akka Actors in Java.* May 2020. URL: https://www.baeldung.com/akka-actors-java.

[34] Johan Andrén and Ryan Knight. *Reactive Programming with Akka.* URL: https://dzone.com/refcardz/reactive-programming-akka.

[35] Hunor Marton Borbely. *Imperative vs Reactive.* Sept. 2017. URL: https://codepen.io/HunorMarton/post/imperative-vs-reactive.

[36] R. Niranjan. *Imperative, functional, reactive programming – which one to use when and for what?* Nov. 2018. URL: https://www.webagam.com/pages/2018/11/24/imperative-functional-reactive-programming-which-one-to-use-when-and-for-what/.

[37] *Web on Reactive Stack.* Dec. 2020. URL: https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/web-reactive.html.

[38] Rareş Popa. *Reactive programming with Spring Boot and Webflux.* July 2019. URL: https://medium.com/@rarepopa_68087/reactive-programming-with-spring-boot-and-webflux-734086f8c8a5.

[39] Kaushal Singh. *Spring Webflux: EventLoop vs Thread per Request Model.* Nov. 2020. URL: https://singhkaushal.medium.com/spring-webflux-eventloop-vs-thread-per-request-model-a42d07ee8502.

[40] Christopher Anatalio. *Basic Introduction to Spring WebFlux.* Dec. 2020. URL: https://medium.com/javarevisited/basic-introduction-to-spring-webflux-eb155f501b17.

[41] Rod Johnson et al. *Spring Framework Reference Documentation Part V. The Web.* Chap. 23. URL: https://docs.spring.io/spring-framework/docs/5.0.0.M5/spring-framework-reference/html/.

[42] Kumar Chandrakant. *Concurrency in Spring WebFlux.* May 2021. URL: https://www.baeldung.com/spring-webflux-concurrency.

[43] *Reactive application development guide. Red Hat support for Spring Boot.* Chap. 2. URL: https://access.redhat.com/documentation/en-us/red_hat_support_for_spring_boot/2.3/html/reactive_application_development_guide/.

[44] Bikram Kundu. *Getting started with Spring WebFlux.* May 2020. URL: https://jstobigdata.com/spring/getting-started-with-spring-webflux/.

[45] Vladimir Fomene. *Introduction to Spring WebFlux.* URL: https://auth0.com/blog/introduction-getting-started-with-spring-webflux-api/#What-is-Spring-WebFlux.

[46] Amit Phaltankar. *Introduction to Spring WebFlux and Reactive API.* URL: https://www.amitph.com/introduction-spring-webflux-reactive-api/#WebFlux_Reactive_API.

[47] *Spring's WebFlux / Reactor Parallelism and Backpressure.* Dec. 2019. URL: https://www.vinsguru.com/reactor-hot-publisher-vs-cold-publisher/.

[48] Bartosz Jedrzejewski. *Spring's WebFlux / Reactor Parallelism and Backpressure.* Apr. 2018. URL: https://www.e4developer.com/2018/04/28/springs-webflux-reactor-parallelism-and-backpressure/.

[49] *Backpressure Mechanism in Spring WebFlux.* Apr. 2021. URL: https://www.baeldung.com/spring-webflux-backpressure.

[50] Anna Eriksson. *WebFlux: Reactive Programming With Spring, Part 3.* Apr. 2021. URL: https://dzone.com/articles/webflux-reactive-programming-with-spring-part-3.

[51] Arjen Poutsma. *New in Spring 5: Functional Web Framework.* Sept. 2016. URL: https://spring.io/blog/2016/09/22/new-in-spring-5-functional-web-framework.

[52] Niraj Sonawane. *Project Reactor [Part 2] Exploring Operators in Flux Mono.* Jan. 2020. URL: https://nirajsonawane.github.io/2020/01/01/Part-2-Process-and-Transform-Flux-Mono/.

[53] Simon Baslé. *Flight of the Flux 3 - Hopping Threads and Schedulers*. Dec. 2019. URL: https://spring.io/blog/2019/12/13/flight-of-the-flux-3-hopping-threads-and-schedulers.

[54] *Reactor Schedulers – PublishOn vs SubscribeOn*. Apr. 2020. URL: https://www.vinsguru.com/reactor-schedulers-publishon-vs-subscribeon/.

[55] Amit Phaltankar. *Introduction to Spring WebClient*. URL: https://www.amitph.com/introduction-to-spring-webclient/#What_is_Spring_WebClient.

[56] Tim Perry. *Sending HTTP requests with Spring WebClient*. URL: https://reflectoring.io/spring-webclient/.

[57] *Spring 5 WebClient*. Jan. 2021. URL: https://www.baeldung.com/spring-5-webclient.

[58] *Spring WebClient*. Aug. 2021. URL: https://howtodoinjava.com/spring-webflux/webclient-get-post-example/.

[59] *Nominatim. Open-source geocoding with OpenStreetMap data*. URL: https://nominatim.org/.

[60] *Map features*. Feb. 2021. URL: https://wiki.openstreetmap.org/wiki/Map_features.

[61] Swathi Prasad. *Multi-Threading in Spring Boot using CompletableFuture*. Sept. 2019. URL: https://swathisprasad.medium.com/multi-threading-in-spring-boot-using-completablefuture-a7ca68a0fe48.

[62] *Spring Data R2DBC*. URL: https://spring.io/projects/spring-data-r2dbc.

[63] *Simultaneous Spring WebClient Calls*. Oct. 2021. URL: https://www.baeldung.com/spring-webclient-simultaneous-calls.

[64] Augusto Sopelsa Klaic. *Functional Endpoints*. Nov. 2020. URL: https://medium.com/@augusto.klaic/functional-endpoints-695f3cc87491.

[65] Rushabh Gaherwar. *Performance Test on an API Using Gatling*. Nov. 2020. URL: https://medium.com/swlh/performance-test-on-an-api-using-gatling-85319ee32faa.

[66] *Gatling Load Testing*. URL: https://loadium.com/gatling-load-testing.

[67] Maarten Smeets. *Spring: Blocking vs non-blocking: R2DBC vs JDBC and WebFlux vs Web MVC*. Apr. 2020. URL: https://technology.amis.nl/software-development/performance-and-tuning/spring-blocking-vs-non-blocking-r2dbc-vs-jdbc-and-webflux-vs-web-mvc/.