FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Fully asynchronous Java APIs for web applications

Daniel Pereira da Silva

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Carlos Viegas Martins Bispo

Receiving Enterprise: Konk Consulting - Consultoria Informática S.A.

July 15, 2021

Fully asynchronous Java APIs for web applications

Daniel Pereira da Silva

Mestrado Integrado em Engenharia Informática e Computação

July 15, 2021

Abstract

Applications for cloud environments present several non-functional requirements, such as execution time, responsiveness, and scalability. An increasingly common way of addressing these requirements is making Application Programming Interfaces (APIs) asynchronous, non-blocking, and cancelable.

APIs for cloud environments work as a middleman between the servers/resources, and the calls of client applications. An asynchronous, non-blocking API means that requests to the API are executed concurrently, returning immediately after firing the request and before the work is completed. Cancelable means that we are able to interrupt the work started by a request before it finishes (e.g., in case it takes more than a certain amount of time), freeing its corresponding resources. Resources may be other APIs or databases.

konkconsulting, the company where this dissertation takes place, already has extensive experience with C# and frameworks that address the challenges of cancellation in that ecosystem. However, they now need to develop similar solutions for Java/Kotlin, and the transition proved to be challenging due to the nonexistence of an equivalent standard framework, as the one found in C#.

Kotlin is a relatively new programming language that can be compiled to run on the Java Virtual Machine. It is also interoperable with Java. The objective of this dissertation is to find a Java/Kotlin framework that can better accommodate the implementation of an API with the mentioned capabilities. The framework was selected from the set of available open-source tools, and was studied and tried in different ways to provide the required functionality. Kotlin offers a set of functionalities, such as *coroutines*, which are used in the implementation.

Keywords: Kotlin, Java, coroutines, api, async, webflux, reactive, webclient, jpa

Resumo

As aplicações para ambientes *Cloud* apresentam vários requisitos não funcionais, tais como o tempo de execução, a capacidade de resposta, e a escalabilidade. Uma forma cada vez mais comum de dar resposta a estes requisitos é fazer as API (Application Programming Interfaces) assíncronas, não-bloqueantes e canceláveis.

As API para ambientes *Cloud* funcionam como um intermediário entre os servidores/recursos e os pedidos dos clientes. Entende-se como assíncrona e não-bloqueante uma API em que os pedidos por ela recebidos são processados paralelamente, retornando imediatamente após o lançamento do pedido e antes de o trabalho estar concluído. Cancelável significa que é possível interromper o trabalho iniciado por um pedido antes to seu fim (ex. se demorar mais do que um determinado período), libertando os recursos correspondentes. Estes recursos poderão ser outras API ou bases de dados.

A konkconsulting, empresa onde esta dissertação decorre, já tem bastante experiência com C# e com *frameworks* que lidam com os desafios do cancelamento nesse ecossistema. No entanto, precisam agora de desenvolver uma solução semelhante para Java/Kotlin e a transição mostrou-se complexa devido à inexistência de uma *framework* equivalente à mencionada no contexto de C#.

Kotlin é uma linguagem de programação relativamente recente que pode ser compilada para correr na *Java Virtual Machine*. É também interoperável com Java. O objetivo desta dissertação é encontrar uma *framework* em Java/Kotlin que consiga suportar a implementação de uma API com as capacidades mencionadas. A *framework* foi escolhida de entre as ferramentas de código aberto disponíveis, tendo sido estudada e experimentada de diferentes formas de modo a dar a resposta pretendida. Kotlin tem uma série de funcionalidades, como *Coroutines*, que são usadas nesta implementação.

Keywords: Kotlin, Java, coroutines, api, async, webflux, reactive, webclient, jpa

Acknowledgements

I wish to express my gratitude to Professor João Carlos Viegas Martins Bispo, my supervisor in this Dissertation, who helped me with his guidance and valuable input. I would also like to thank konkconsulting, in the person of Jorge Almeida, my supervisor at the company. Jorge helped me push through many difficulties that could have thwarted this effort whilst empowering me to go beyond what I thought possible in my endeavours.

Naturally, I cannot let go of this opportunity to thank all my colleagues and friends at the IEEE University of Porto Student Branch, with whom I have spent much time exchanging points of view about all subjects, including my Dissertation's. Through the years, they have brightened up my academic life and helped me grow in several domains.

Moreover, I would like to thank my parents, Fernando Silva and Elisabete Silva, and my sister Cristiana Silva who have always been present, giving me all kinds of support. Without them, I would not have arrived at the fruition of this work.

The present LaTeX template is provided by professors João Canas Ferreira and João Correia Lopes, to whom I extend my gratitude as it has dramatically simplified the formatting considerations for this Dissertation. I cannot forget about Dominik Schürmann, the creator of *wireshark2latex*[1], a small utility that helped bring the Wireshark output to form in this document.

Daniel Pereira da Silva

[1]https://github.com/dschuermann/wireshark2latex

“Right now we are at the place where we actually are using Spring Web MVC in a functional way and it’s the place where it’s **very very easy** *to migrate to Spring Webflux.“*

(emphasis added)

Nicolas Fränkel[2]

[2] https://youtu.be/8-6Cd9YemOc?t=1613

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
WWW	*World Wide Web*
JVM	Java Virtual Machine
IDE	Integrated Development Environment
CRUD	Create, read, update and delete
GUI	Graphical User Interface
SLOC	Source lines of code
CD	Continuous Deployment
CR	(Kotlin) Coroutine(s)
PoC	Proof Of Concept
SQL	Structured Query Language
DBMS	Database Management System
DBAL	Database Abstraction Layer
IO	Input-Output
NIO	Non-blocking Input-Output
MVC	Model—View-–Controller
AB	ApacheBench
TES	(Spring) Task Execution and Scheduling
RFC	Request For Comments
Q&A	Question and Answer

Chapter 1

Introduction

1.1 Context

In a world where the speed of replies to API calls is paramount, and the response to an API call can range from a few milliseconds to minutes or even hours, we consider that cancellation of these calls is an important topic due to the high volume of data that can be processed and the high computational effort that can be required to reach an answer or to finish a process. However, it seems the problem of cancellation has not been properly addressed.

konkconsulting, the company that promoted this dissertation, has been using the asynchronous and cancellable features of the .NET framework for C# with great success. konkconsulting is now working in a new project with another company — which has experience with Java. Together, they decided to implement this project using Kotlin. They have started implementing a series of micro-services using Kotlin and now wish to implement an API using that language.

The implementation of a vertical solution, using Kotlin, for an API that can have its requests cancelled is, thus, the objective of this dissertation.

1.2 Motivation

Human operators have time-sensitive objectives when dealing with an API that can be slow, especially when there are no indications of the progress, such as a progress bar or the ETA. Thus, it is common for them to cancel an issued request because it is slow and to request something else. Moreover, human operators are also prone to mistakes, which can result in an API call to a resource that was not the intended one. Disregarding the destructive effects that such lack of attention may have, there are also other consequences.

The resources dedicated to replying to requests that have been abandoned by the client, when those requests take a long time, can be enormous. Additionally, they may be blocking other operations because they have locked a file or a database table, or they are using all the network bandwidth available, etc.

Another reason why tackling this issue is so important has got to do with updates. Continuous deployment has become commonplace in all sorts of environments and is more and more used in agile development to ensure that features and fixes become live as soon as possible. However, there is the issue of long tasks. If these tasks cannot be properly interrupted, the system may have to wait a very long time (during which it cannot receive new requests, otherwise it would be waiting forever, depending on the load) before applying the update. The possibility to gracefully interrupt running tasks speeds up the update process and reduces Lead Time, while keeping the system in a consistent state that can be easily resumed.

1.3 Report Structure

Chapter 1, the present chapter, introduces the document.

In Chapter 2 there are explanations of some concepts pertinent to this dissertation, such as Kotlin and Rest APIs.

Chapter 3 contains the literature review about Cancellation, Reactive Programming, Coroutines and an overview of the tools available for implementing an API using Java/Kotlin.

Chapter 4 describes the problem at hand and its proposed solution, as well as the means to validate and compare the different approaches that were developed during the dissertation.

Regarding the next Chapters, we have a different approach. Unlike regular dissertations, this one focuses on several small themes being studied, and although our desire is to find the best solution to all those and put them together in a single project, they are still rather disconnected to be studied as a whole. For this reason, instead of the regular approach in which there is a chapter regarding the proposed solution, we present several chapters, each regarding one of the subjects studied. There is a special chapter — Chapter 9 — in which we discuss the overall experience of the integration of the new solution. While this is a rather exploratory dissertation, which aims at selecting existing solutions, rather than creating a new one, we are still invested in the scientific and analytic side of the tests that will be executed. For this reason, we present a Results chapter — Chapter 10 — in which we lay out the diverse trials that were run.

Finally, Chapter 11 ends the dissertation presenting its conclusions and expected Future Work.

Chapter 2

Concepts

The novelty of some key aspects of this dissertation force us to make a clear introduction to those as we cannot, in good conscience, expect the common reader to be comfortable with them. These aspects are presented in a way that should be easier to understand than primary sources, and that is more adequate to the context of this dissertation. Nevertheless, we give footnote references of the sources used so that further investigation by the reader is possible. Naturally, some of the subjects presented here may seem irrelevant, but they become important further ahead.

It should be noted that some of our sources are Podcasts, Webinars, recorded workshops, etc (throughout the whole document, not just in this chapter). The target-audience of the technologies involved makes it so. Whenever crucial information is necessary, we quote or paraphrase the original author, which helps the reader as they do not have to search the original piece.

2.1 Kotlin

Kotlin is a programming language usually compared to Java, due to its origin, as explained below. The differences between Java and Kotlin have been extensively studied. Gakis el al. [7] and Gotseva et al. [8] compare them, and have helped shed some light on the topic. This comparison makes sense because Kotlin was created from a Java background and compiles to Java Bytecode. It also compiles to Javascript and machine code. Because it runs on the JVM, Kotlin has the same advantages as Java regarding portability. Moreover, Kotlin is interoperable with Java, which facilitates reusage and integration of legacy code. Kotlin is not the first language developed to run on the JVM after Java, but it is the one that became most popular in such short time. Its development started in 2010, by JetBrains, the developer of the well-known IntelliJ IDEA IDE. Its first stable version was released in 2016, released under the Apache 2.0 license. The source code is available on its GitHub page. In 2017, Google included Kotlin in Android Studio, and in 2019 Google officially announced Kotlin as the preferred language for Android development[1].

Kotlin has a series of advantages over Java:

[1] https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html

- Null safety

  One of the biggest differences between Java and Kotlin, according to Gotseva et al. [8], is the fact that variables in Kotlin can be nullable or non-nullable, in an effort to reduce the occurrence of null-pointer exceptions. Gakis et al. [7] note that developers can access a not nullable variable without checking its value first as they know that a null pointer exception will never be thrown when accessing it.

- Coroutines

  Despite being *similar* to Java threads, coroutines are significantly lighter and perform better than threads when bulk-launched. Gakis et al [7] also mention that coroutines are useful for background computations when developing a GUI as they enable the developer to launch parallel tasks, thus not blocking the main UI. Coroutines are a complex syntactic construct that creates a hidden callback structure in Java Bytecode.

- Smaller language features

  Everlönn et al. [7] recognize this as a minor feature, but still find it relevant: two equivalent programs in both Java and Kotlin are much smaller (in terms of Source Lines of Code) when Kotlin is used.

  Type inference was a feature provided by Kotlin, but has since become available in Java 10.

  Kotlin also features "smart casting", which reduces the number of calls to `instanceof`. This is also available since Java 14.

  Everlönn et al. and Chakraborty[7, 8] both mention the fact that Kotlin supports named arguments when calling methods, which greatly reduces the amount of definitions for methods. This also allows for easier reading of the code as the name of each parameter is in the code itself.

Chakraborty [8] also notes the fact that functions can be marked as inline, as well as as tail recursive, in Kotlin.

There are a series of special mechanisms in Kotlin that are used in this dissertation, or that temporarily thwarted our efforts. Both cases are listed below:

2.1.1 Data Classes

Data Classes [2] are a special kind of classes in Kotlin used for classes whose main purpose is to hold data. Such classes can be marked with the `data` keyword, as follows:

```
data class Book(val name: String, val year: Int)
```

This makes the compiler automatically implement several functions for said class:

- `equals()` and `hashCode()`

[2] https://kotlinlang.org/docs/data-classes.html

- `toString()` of the form `Book(name=Eragon, year=2002)`
- `copy()` which receives as named parameters the fields to be changed, e.g.: `val cor = eragon.copy(name="Coraline")`
- `componentN()` allowing to extract variables as follows: `val (name, year)= cor`, keeping the variable order as specified in the class declaration. [3]

Most ORM libraries work well with data classes, the exception being JPA, which is not designed to work with immutable classes. [4]

2.1.2 Coroutine Context & Dispatchers

Each Kotlin Coroutine runs in a context that the documentation calls CoroutineContext, which contains the running Job, its dispatching mechanism, and other related elements.

The coroutine dispatcher determines on which thread the coroutine will run. According to the documentation[5], the "coroutine dispatcher can confine coroutine execution to a specific thread, dispatch it to a thread pool, or let it run unconfined":

- Thread: a specific thread is used for this coroutine. Threads can easily be created using `newSingleThreadContext()`. However, this is an expensive resource. The documentation advises closing the thread when it is no longer needed, or alternatively reusing it throughout the application.
- Thread Pool: the default option — Kotlin offers a dedicated thread pool for coroutine execution. Dy default, the thread-pool size is the same as the number of cores available, with a minimum of two. However, a custom-sized thread pool may be provided. An example is provided in Listing 2.1.
- Unconfined: the coroutine will be ran in the calling thread until the first suspension. After that, it will be resumed in a thread defined by the suspending function that was invoked.

```kotlin
import kotlinx.coroutines.asCoroutineDispatcher
import java.util.concurrent.Executors

val dispatcher = Executors.newFixedThreadPool(128).asCoroutineDispatcher()

CoroutineScope(coroutineContext).async(dispatcher) {
    // code block
}
```

Listing 2.1: Creating a thread pool for running coroutines.

[3] https://stackoverflow.com/a/43158972/1469991
[4] https://github.com/spring-guides/tut-spring-boot-kotlin#persistence-with-jpa
[5] https://kotlinlang.org/docs/coroutine-context-and-dispatchers.html

2.1.3 Performance Analysis

Everlönn et al. [7] have done extensive benchmarks using several programs implemented in both Java and Kotlin, and running on a Windows machine and an Android phone. In the Windows simulations, the speed of execution on both languages was very similar, with differences ranging from 1% to 4% both ways.

Memory usage, however, yielded unexpected results, with Kotlin always using more RAM than Java. Of the 4 benchmarks that hey performed, two had a difference under 5%, but other two had a 17% and 340% difference. The authors do not present any studies trying to explain this difference.

2.1.4 Java Fibers & Kotlin Coroutines

Java Fibers are to be a native JVM construct defined by Warski [23] as "light-weight threads". Paluch describes this technique as "offloading blocking calls to a Fiber-backed Executor". Considering the apparent similarity between Java Fibers and Kotlin Coroutines, the second point presents an interesting prospect: the possibility of wrapping the JDBC in an IO Coroutine-based Executor, so that it can be used in with Spring-Webflux. Further investigation has shown, however, that this shall not be possible: Warski [23], in his blog post about Java Fibers, presents a side note where he explains the differences between those and Kotlin Coroutines. While Java Fibers, part of the Loom project, are implemented on the JVM itself, and therefore have native access to the OS layer, Kotlin Coroutines, as magical as they may seem, are a mere syntactic construct that create a hidden callback structure in Java Bytecode. Other members of the development community have suggested the possibility of Kotlin making use of JVM Fibers, but that is not a reality as of yet, and its implementation did not take place in useful time for this dissertation. Despite Project Loom being currently in active development[6], there have not been any news on it for some time. Nevertheless, its development remains active on GitHub[7].

2.2 REST APIs

REST commonly relies on HTTP(S). HTTP is a protocol created in *CERN* in 1991 which is the foundation for the WWW as we know it today. When it was created, HTTP was not designed to be cancellable. This means that once a request is fired, there is no way to cancel that particular request, because no state is kept by either party of the communication. It may be possible, however, to close the underlying TCP socket. If the server can detect that the socket was closed, it may then choose to cancel the resolution of the request. [8]

Alternatively, HTTP codes 102 Processing and 202 Accepted may be used for long tasks.

[6] https://wiki.openjdk.java.net/display/loom/Main

[7] https://github.com/openjdk/loom/commits/fibers

[8] https://softwareengineering.stackexchange.com/questions/362187/can-a-caller-abort-an-execution-of-code-invoked-by-http-request

REST is an architectural style which defines the way messages are to be exchanged between computers on the web. Some characteristics of REST are:

- **Stateless:** the meaning of each message is the same regardless of the messages previously exchanged. A counter-example would be the *get <file>* command on FTP: if a relative path is provided, the previous *cd* command (e.g.: cd /public) affects the result.
- **Client to Server architecture:** the client performs a CRUD (Create, Read, Update, Delete) operation on a resource and the server issues a reply. The server cannot initiate an operation to a client.

REST makes use of the HTTP methods (verbs) to indicate what kind of request is being issued [16]. The most common are GET, POST, DELETE, PUT. It is possible to implement custom methods, despite this being undiplomatic:

- Frameworks rarely offer support for custom HTTP methods.
- Additional methods change the protocol and turn it into a superset of HTTP, which may hinder communication with other systems.

2.3 Blocking & Non-Blocking IO in the Java Environment

The concept of thread is a considerably complicated subject when compared with the early non-time-sharing Operating Systems.

The first computers were able to execute simple tasks and were programmed by setting the instructions directly on physical binary switches. Nowadays, computers have several cores (sometimes even several CPUs), and multi-threaded programming takes advantage of this to provide much faster results when parallelisation is possible.

Threads exist within processes and share most of the memory of that process, effectively being a lighter version of a child process.

However, while launching a handful of threads is a lightweight operation — much more lightweight than forking, the process which creates child processes — launching a thousand threads, for example, can be a heavy operation: especially considering the constant context switch necessary to accommodate every running thread.

In Java applications, usage of threads is very commonplace as those allow offloading of heavy operations off the main thread, keeping applications responsive and allowing for controlled parallelisation. Thread-pools are used as an abstraction in order to manage the available threads in a simple fashion. A regular use case is launching a thread for each incoming request in a server application.

According to Elizarov [3], creator of Kotlin, "Threads are expensive", and blocking a thread should be avoided. While blocking is unavoidable when a CPU-bound operation is taking place, such as a heavy mathematical calculation, the same cannot be said for IO-bound operations.

Elizarov defends that "if you block because of IO, then you can (...) avoid blocking by using non-blocking (...) IO libraries that do not block threads at all." While IO-blocked threads do not consume CPU resources, they are still taking up space in the thread-pool and their creation, scheduling, and destruction become heavier as the number of threads increases, according to Thirunavukkarasu [21].

Karabyn [11] explains that non-blocking IO (NIO) libraries work by implementing buffers from and to which the main thread can read/write respectively. The buffer processing is done by a specialized thread (usually just one) — even for several concurrent files or other resources — which reduces the amount of open threads at a time and also reduces the combined sum of time that threads in a program are waiting for IO. This behaviour is observed in our trials ahead, e.g. in Chapter 10.2.3.

2.4 Reactor Schedulers

Reactor Schedulers are an "abstraction that give the user control about threading"[9]. While Reactor operators are usually concurrent agnostic, time-dependent operators are not. Those must run on a scheduler, the `Schedulers.parallel()` scheduler by default. These Schedulers are important for us because of the way Spring Webflux deals with and uses them. There are several schedulers available[10], as shown in Table 2.1.

Scheduler	Description
Parallel	Runnable, non-blocking executions
Single	low-latency, Runnable, single-shot executions
Elastic	longer executions, blocking tasks, growing # of tasks
Bounded Elastic	longer executions, blocking tasks, capped # of tasks
Immediate	do not schedule, "null-scheduler"
(custom)	`fromExecutorService(ExecutorService)`

Table 2.1: Reactor Schedulers

2.5 Webflux Threading Model

Webflux and Netty are important parts of this dissertation, as we explain in section 5.2.4. The Webflux documentation delves into the threading model [11] explaining why it may seem so unusual for newcomers: explaining the blocking calls dilemma, the reactive pipeline call structure, and the actual threads that one may expect to see in a Webflux Application (considering that Java Reactor and Netty are used):

[9] https://spring.io/blog/2019/12/13/flight-of-the-flux-3-hopping-threads-and-schedulers

[10] https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#boundedElastic

[11] https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-concurrency-model}

- **server** thread, launched by the main application.
- **reactor-http-nio-** threads, usually as many as the available cores on the CPU. These handle incoming requests and take care of Reactor-related operations.
- Reactor Schedulers threads, which exist when the running Reactor code has defined a Scheduler. The thread names are as found in the first column of Table 2.1, e.g.: boundedElastic-12
- Other Libraries' threads, depending on which external libraries are used. e.g.: Hibernate.
- Language-specific threads, depending on the Language used (Kotlin, Groovy, Java) and the constructs that the developer crafted for thread management.

However, the documentation does not explain how it deals with blocking code on the request handlers. Fortunately, Piotr [20] studied extensively the Webflux Threading Model. He reports that the worker threads (reactor-http-nio-*) delegate long-running tasks to threads available in the boundedElastic pool, which is a Reactor Scheduler as described in Chapter 2.4. This is compatible with what we have observed.

2.6 Idempotence and Safety

Idempotence is a property of certain operations characterised by the possibility of being applied several times without changing the result beyond the initial execution.

Safety is a property of certain operations characterised by the no change of the accessed resource, regardless of the number of times it is accessed.

This is applicable to all calls/requests done between two parties, as for example in:

- HTTP requests
- SQL requests
- Websocket requests/messages

Idempotence does not mean that the call does not change the state. It merely means that one or more requests to the same operation always yield the same result in the system — not necessarily the same result to the requesting party. Another way of putting it, according to T. Yuan et al [24] is that a method being idempotent "means duplicate actions cause no effect". Safety, on the other hand, means that nothing is changed when executing that method.

For example, in HTTP, a DELETE operation is idempotent because, in the first run, the resource is deleted and a `200 Ok` response is returned. Additional calls to the same method may yield different responses (such as `404 Not Found`), but the state is preserved (the resource is still deleted). Another example is doing a PUT, where the first call will create the content, usually returning a `201 Created` response. Subsequent identical requests can return whatever response, but the status of the server will not be changed.

Conversely, subsequent POST requests will create several resources, each changing the state of the database, which makes it an non-idempotent method.

Studying idempotence and safety is relevant because this dissertation focuses on cancellation, and in many cases it is not possible to know what state the cancelled operation stayed in. The matrix below can help us understand what may be necessary:

	Idempotent	**Not Idempotent**
Safe	Cancellation is always safe.	Impossible combination.
Unsafe	Changes can be rolled back carefully, or redone if cancellation was unintended.	Redoing changes is dangerous. There should be a way to check what was changed in order to react to that.

Table 2.2: Cancellation behaviours matrix

2.6.1 HTTP

Idempotence is a big theme in HTTP. The HTTP Specification [6] is clear about which methods should be idempotent and safe:

Method	**Idempotent**	**Safe**
GET	Yes	Yes
POST	No	No
PUT	Yes	No
DELETE	Yes	No
HEAD	Yes	Yes
OPTIONS	Yes	Yes

Table 2.3: Idempotent and safe methods' classification in HTTP

It is clear that only POST requests are supposed to be idempotent. However, it is not uncommon to see implementations where multiple GET requests result multiple in changes to state (e.g. some websites have visitor counts which do not require a POST to be increased). For this reason, evaluating idempotence of API calls should not be solely based on the HTTP method used: the context and implementation must be taken into account. The same goes for Safety evaluation.

2.6.2 SQL

Like in HTTP, SQL operations can also be studied regarding their idempotence and safety properties. However, this study is much harder as there is the possibility of sub-queries, if-clauses and triggers. Assuming only the simplest constructions, we can classify the different SQL functions as follows:

Again, note that errors and exceptions are not taken into account when studying idempotence. Only the (final) state of the database is considered.

Function	**Idempotent**	**Safe**
SELECT	Yes	Yes
INSERT (with ID as default value)	No	No
INSERT (with a defined ID)	Yes	No
UPDATE	Yes	No
DELETE	Yes	No

Table 2.4: Idempotent and safe functions' classification in SQL

2.7 SpEL

Spring Expression Language[12] is a simple language that Spring executes at runtime. It can be used in several annotations, such as `@PreAuthorize`. SpEL offers support for calling functions, getting a specific element from an array, a list, a map, or an object's properties, among others. Its complete documentation explains the ways it can be used.

2.8 Code sources

Throughout this dissertation, we will be using several sources of code and tutorials:

- GitHub — an online repository where developers from all around the globe contribute to public and private projects. Public repositories (also called repos) are searchable across the platform and can be scoured for code examples, configuration files, etc.
- StackOverflow — a Q&A website where users ask questions and get answers from the community. It is considered one of the biggest sources for code examples and quick solutions for numerous problems.
- Baeldung — a blog where Spring contributors publish tutorials explaining how to do a task, e.g. using the WebClient library.

2.9 Summary

Kotlin is a novel language which shall facilitate the present work via its many quality-of-life mechanisms, namely the Coroutines[13]. Additionally, the REST standard, well laid out as it is, seems to lack support for cancellation. This shall be studied further during the dissertation.

As any computer tool, the present dissertation work shall interact with many external platforms and protocols, such as databases, the OS, other (REST) APIs. However, those were not mentioned in this report as it is impossible, at this stage, to predict what kind of knowledge will be required to achieve the goals of the present dissertation.

[12] https://docs.spring.io/spring-framework/docs/4.3.10.RELEASE/spring-framework-reference/html/expressions.html
[13] https://kotlinlang.org/docs/reference/coroutines/basics.html

Chapter 3

Related Work

The study of the related work is an important step before starting work on the solution. Here, we present the state of the art regarding the cancellation of tasks, asynchronous programming, and reactive programming.

We have been unable to find literature correlating Kotlin (see section 2.1) with cancellable APIs, despite great effort. Further research yielded no literature results relating cancellation with languages such as Java or C#.

For this reason, we will be focusing on the same subject — cancellation, asynchronicity and parallel programming —, but mostly in a language-agnostic way.

3.1 Literature Review

3.1.1 Reactive Programming & Cancellation

Several of the tools described in the next section mention the term "reactive programming" or "reactive streams" in their websites. Redhat defines reactive programming as "a development model structured around *asynchronous* data streams". [4] Asynchronous programming is a way of parallel programming that allows a task to run separately from the primary application thread. When the task is complete (or interrupted), it notifies the main thread, usually via a callback. It is, thus, characterized by the presence of callbacks, function references, etc.

Reactive programming is a subset of this, and refers specifically to when code runs *reactively* to an event, usually the reception of data streams, and which propagates changes to interested parties automatically. Chakraborty, R. [2] gives the example of spreadsheets which update cells as the formulas they contain get up to date data from other cells. Reactive programming has been being used at least in Javascript[10] and Kotlin[2], in both contexts usually via specialized libraries/frameworks.

Kambona et al. [10] defend that the major problem with Asynchronous Programming which makes Reactive Programming an interesting proposition is the presence of nested callbacks. These are used in event-driven programming and are used to call a function to handle the success or insuccess of an asynchronous task. They highlight that using callbacks extensively ends up in "complex flows" that require the programmer to be omnisciently aware of all the program, especially when several events may fire up at the same time. Chakraborty agrees, calling this phenomenon "callback hell". Belson et al. [1] also note this problem and employ the same term.

Reactive programming solves this issue by implementing *observables*, a tool which helps programmers catch data stream events and their contents. Escoffier [4], from Redhat, lists a series of operations that can be performed on the observed variables: "combine, merge, filter, transform (...)". Kambona et al. [10] also mention the merging of streams, which utility they exemplify with the scenario of a file being auto-saved periodically (periodic event) or when the user clicks the save button (stochastic event), both of which do the same operation: saving the file. Without observables, implementing the same functionality would involve catching both events and calling the same function for both, or possibly copying the code from one to the other.

3.1.2 Cancellation

One of the goals of this dissertation will be the cancellation of tasks. For this reason, we have done a review of the literature on this matter.

Kolesnichenko et al. [12] have documented the cancellation of tasks in a client/server (supplier) environment and have classified tasks cancellation techniques as:

- Client-based cancellation — where the client is the sole responsible for requesting the cancellation. Client-based cancellation can be further classified as:
  - Forceful cancellation — where the client asks the server to stop. Forceful cancellation can be:
    - *Abortive cancellation* — the server immediately stops execution of the task.
    - *Interruptive cancellation* — the server can reach a safe point before cancelling the task.
  - Passive cancellation — the client merely stops waiting for the reply and does not inform the server that the request has been canceled. HTTP is a typical example of this.
- Supplier-based cancellation — the server itself takes the initiative to cancel the task. This is common when the server finds an error and cannot proceed.
- Client/supplier cancellation — the server and the client negotiate the cancellation of the task.

Kolesnichenko et al. [12] call *safe points* to the places in the execution of a task where it can be cancelled safely. These are usually added by the programmers as checks that the task has not been cancelled, and they are added specifically in places where it is possible to revert the tasks effects, if desired, or release the allocated resources. Kolesnichenko et al. also note that "this work is the first to attempt a comprehensive classification and evaluation of task cancellation techniques". This does not seem to have changed in the meantime.

Kumar, V. [13] presents a different approach regarding these *safe points*, calling them *yield-points*, defending that these points can be generated on compile-time and thus there is no need for manual verification of the cancellation status. The author relied on this to implement Featherlight, "a new programming model for speculative task parallelism ". This approach is novel as it does not require any explicit cancellation checks throughout the code. This way, developers can focus on the issue they are solving, thus increasing productivity. However, its purpose is for task sets in which most tasks will fail (i.e., only one needs to succeed), such as computations with graphs (e.g. finding the best move in a game tree or route planning in GPS systems). Moreover, Kumar does not focus on the issue of cancelling external requests gracefully, which is paramount in this dissertation.

Kangwood Lee et al. [14] have studied an interesting theme: cancellation in distributed data storage. They study the idea that it may be possible to reduce job latency by scheduling the same job at several servers and waiting for the first reply, cancelling the rest after that. The problem, according to the authors, is that the models that study the cancellation of said tasks do not take into account the fact that cancelling a task can be a heavy task, i.e., cancellation is not instantaneous. In some cases, cancellation is not even possible (e.g. DNS or HTTP requests).

Solution/Method	**Cancellation technique**	**Safe Points**	**Overhead Analysis**
Featherlight[13]	Supplier-based cancellation (non-suited for client-server)	Compiler-made	no overhead because cancellation is not graceful
Distributed tasks cancellation[14]	Some form of Forceful cancellation (depending on the task)	not studied by the authors	The central point of this work. Overhead is taken into account before launching parallel tasks

Table 3.1: Comparison between cancelling solutions

Cancellation technique as defined by Kolesnichenko et al. [12] and showed previously in this chapter.

3.2 Tools

It is clear that the development of this dissertation will implicate the selection of a Kotlin microservices framework. While that study is extensive, and a part of the dissertation itself, we believe it makes sense to provide an overview of the available tools as an entry point for the rest of this dissertation.

After some searching, the following tools were found:

Framework	**Supported Languages**	**Build Tools**
Helidon SE	Java, Kotlin	Maven
Ktor	Kotlin	Maven, Gradle
Micronaut	Groovy, Java, Kotlin	Maven, Gradle
Quarkus	Java, Kotlin, Scala	Maven
Spring Boot	Groovy, Java, Kotlin	Maven, Gradle

Table 3.2: Tools overview

Scala and Groovy are languages that, like Kotlin, compile to the JVM.

3.2.1 Helidon SE[1]

Helidon SE is part of *Project Helidon*. The project's slogan, as displayed on their website, is *Lightweight. Fast. Crafted for Microservices*. Helidon is split in two parts:[2]

- Helidon SE – designed to be a microframework that supports the reactive programming model.
- Helidon MP – an Eclipse MicroProfile runtime for running microservices portably.

3.2.2 Ktor[3]

Ktor is a Kotlin framework developed by JetBrains. Its home page title is *Ktor: Build Asynchronous Servers and Clients in Kotlin*. Its website features Documentation, Tutorials and Samples for developers.

3.2.3 Micronaut[4]

Micronaut introduces itself as the *Polyglot Framework*, as it natively supports several languages: Java, Groovy and Kotlin. It integrates easily with several databases, such as Redis, MongoDB, SQL databases, etc.

Micronaut can be used both for serverless applications, such as AWS Lambda or regular server APIs. Its website also mentions that it can be used to "build fully reactive and non-blocking apps".

3.2.4 Quarkus[5]

Quarkus' value proposition is what has become known as "live reload". Its website advertises the "Developer Joy" of having code reloaded on save. Its focus is mainly on Java, and while it does support Kotlin, there is no mention to it on their page.

[1]https://helidon.io/
[2]https://www.baeldung.com/microservices-oracle-helidon
[3]https://ktor.io/
[4]https://micronaut.io/
[5]https://quarkus.io/

3.2.5 Spring Boot[6]

Spring Boot is the oldest player in the field. In its website, there are several highlighted aspects: Microservices, Reactive, Cloud, Web Apps, Serverless, Event Driven and Batch. Spring Boot comes bundled with an integrated webserver (such as Tomcat).

A complete setup guide for Spring Boot and Kotlin is provided in their blog. [7]

Spring Boot is part of the Spring family. Spring Boot is an extension of the Spring framework that speeds up the setup of a Spring application by reducing the necessary boilerplate configurations [8]. Guntupally et al. [9] reeinforce this idea, saying that is has "definite advantages over other spring-based frameworks", allowing for "simpler dependency management, auto configuration, built-in CRUD handling, and flexible project startup".

Bootstrapping

As of January 2021, all the aforementioned frameworks' websites provide what is called a *Web Starter*. This helps programmers bootstrap the application, allowing them to explore and select the packages that they would like to import. The final result is a zip file with the application and also the configuration file(s) for the selected build tool. Alternatively, some frameworks also support CLI tools for the same process, such as Micronaut.[9]

Such bootstrapping platforms simplify the deployment of testing environments, creating of PoC projects, among others.

3.3 Summary

Despite our best efforts, it was not possible to find literature correlating Kotlin with cancellation. This may be due to the fact that the people engaged in the development of this subject are not integrated in the creation of literature. This theory is further supported by the fact that the amount of literature about cancellation of tasks as a whole is very diminute. The revised literature pertains to cancellation in a language and protocol-agnostic way. Nevertheless, the literature review process provided noteworthy results, such as the classification of cancellations and the notion of automatic cancellation. The idea of launching the same task on different servers and waiting for the first result is also notable, but out of scope for this project.

In Chapter 4.2, we explain which kinds of cancellation are expected to be used in the proposed solution, as per the technique classification by Kolesnichenko et al. [12].

The fact that most of the tools and related technology that this dissertation focuses on are mostly used in enterprises makes it understandable that there is so little literature on the subject. Concepts like Reactive Programming or even Task Cancellation, as primary as they may sound, are not as studied as other fields of computation — which could be due to the newness of the

[6]https://spring.io/projects/spring-boot
[7]https://spring.io/guides/tutorials/spring-boot-kotlin/
[8]https://www.baeldung.com/spring-vs-spring-boot
[9]https://docs.micronaut.io/1.3.3/guide/index.html#buildCLI

first or the apparent irrelevance of the latter. Regardless, it is clear for us that this dissertation can become a great addition to libraries, as there are no documents regarding the subject at hand.

Kotlin is indeed a recent language that has not had its full potential reviewed in the literature, and while this dissertation does not focus on it, it does cover some of its aspects.

We note that the research done has shed some light on the ins and outs of cancellation of tasks and parallel programming. This dissertation does not approach cancellation in this structured manner, focusing instead on its low-level complication and the way it is accomplished, as well as the study of its side-effects.

Moreover, the tools review that has been done in this stage greatly aided in the selection of the framework for the development of this dissertation.

Chapter 4

Problem & Proposed Solution

In this Chapter, there can be found an explanation of the problem at hand, the proposed solution for that problem, albeit in a descriptive and generic way, and lastly the validation and evaluation methods for the solutions found during the dissertation.

4.1 Problem Statement

The cancellation of API calls is an uncommon subject to discuss, in a world where these calls should become increasingly faster in terms of response time. However, in the case of konkconsulting, some calls are immutably slow: they access huge amounts of data spread through several microservices and involve heavy computations. The ability to cancel such calls is crucial for konkconsulting. Cancellation should be as quick as possible, it must keep all the resources in a graceful state (databases, files, etc.) and should free all allocated resources so that further calls can be accommodated. konkconsulting has started its endeavours on the creation of a solution using Spring Boot, so this tool is preferred.

4.2 Proposed Solution

konkconsulting has already been integrating some of the newly developed APIs using Spring Boot. Their request, however, is to select one of the tools (section 3.2) based on how they are designed regarding the support for coroutines and cancellation[1], and to extend it so that calls can be cancelled. In the beginning, the protocols to support cancellation had not yet been determined, as we kept the scenery open for whatever solution seemed possible. *HTTP* and *WebSocket* were the first possibilities we considered. Previous use of Spring MVC by konkconsulting means that Spring (or any tool of the Spring family, such as Spring Webflux) is preferred. Cancellation will have to be implemented across all layers of the application, i.e., each layer (call reply, database access, other APIs access) must have its own cancellation code, pertaining to its special needs in that regard.

[1]https://kotlinlang.org/docs/reference/coroutines/cancellation-and-timeouts.html

For example, the cancellation of a database request must be made in such a way that the query is cancelled, if possible, while keeping the connection open for further requests.

Depending on the way cancellation will take place, it can be classified as follows (according to the technique classification by Kolesnichenko et al. [12]:

- **Client-based cancellation > Forceful cancellation > Interruptive Cancellation**
  Whenever a cancellation request is received from a client, pertaining an ongoing request.

- **Supplier-based cancellation**
  Upon receiving an update request from Continuous Deployment, the application should decide which tasks to cancel and carry that out. Not all tasks will need to be cancelled if there is a study about cancellation overhead. Only the cancelled tasks will fall into the *Supplier-based cancellation* category.

Figures 4.1 and 4.2 show cancellation across layers and in the deepest layer(s), respectively.

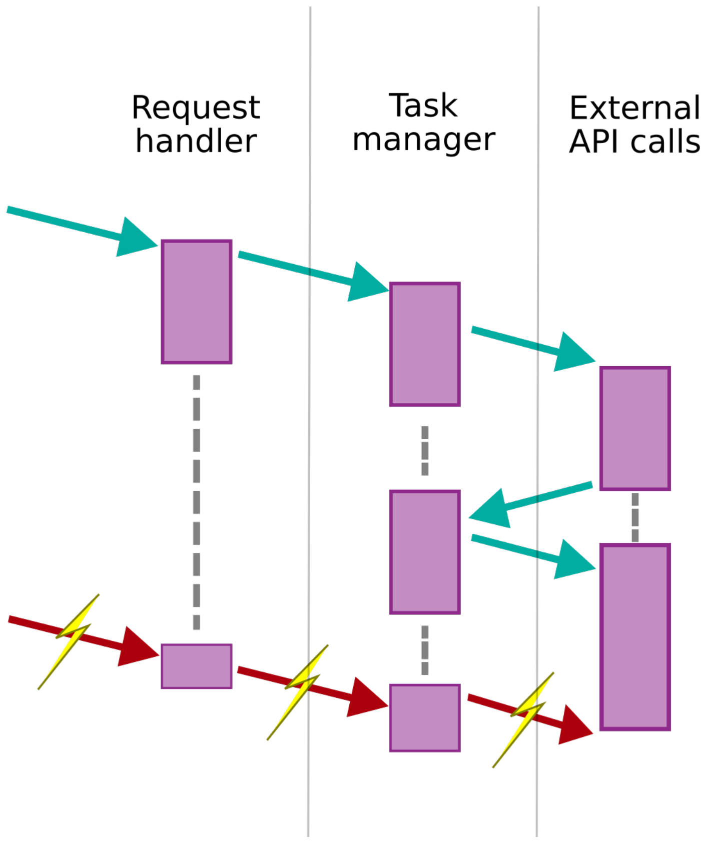


Figure 4.1: Cancellation overview diagram - A cancellation request, be it issued by the system or by an external entity, should be handled across all layers of the application. *External API calls* is given as an example.

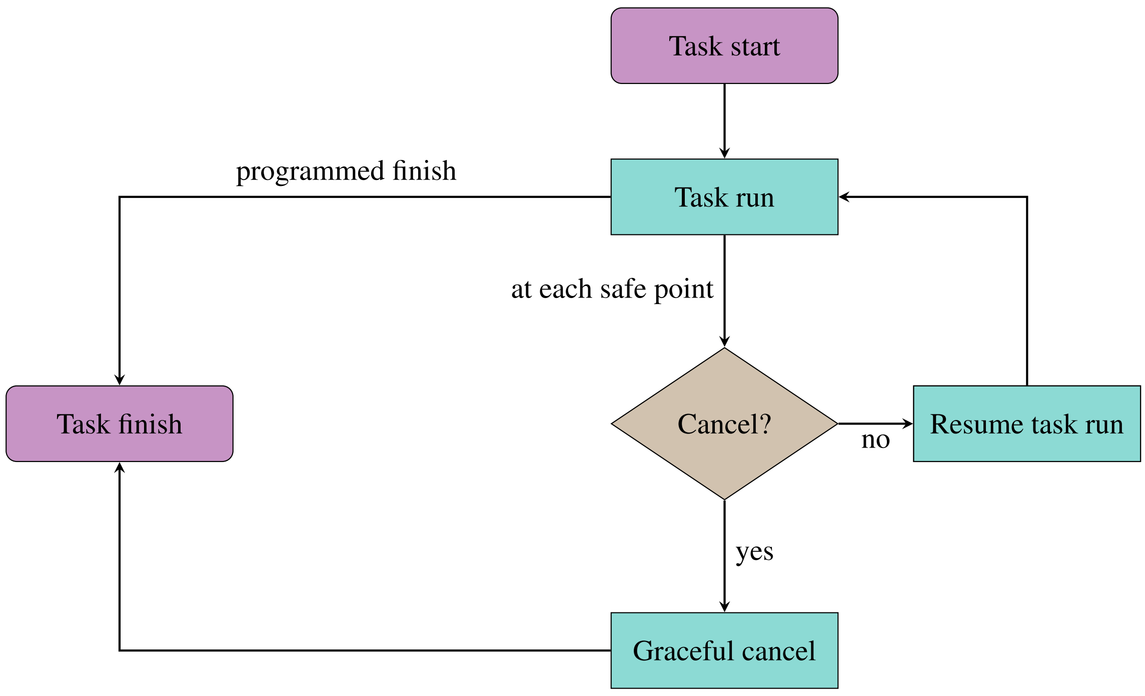


Figure 4.2: Inner layer low-level cancellation flowchart - The inner most layer(s) should contain *safe points*, which allow code execution to be interrupted in a graceful manner. Alternatively, *Exceptions* may be used, but that is not planned in order to keep the consistency with konkconsulting's present C# solution, which is desired.

4.3 Validation & Results Evaluation

Validation of the achieved results shall be done with proofs of concept, in an initial phase. Posterior validation and comparison of the different alternatives with different frameworks and cancellation techniques shall be done taking into account the variables that have proved themselves to be the most relevant. It was hard to anticipate these. We considered the following:

- Cancellation time
- Amount of wasted resources
- Maximum amount of tasks supported
- Interoperability with the already developed code

4.4 Expected Results

The ultimate goal of this dissertation is the creation of a proof of concept, in Kotlin and using a suitable web framework (preferably Spring Webflux), of cancellation in the requested contexts: third party API access, database access, Websockets. The literature review has shown how necessary this is, and that it will be a breakthrough in modern times for Kotlin and Web development. Optionally, and if time allows, the developed work shall be integrated in an actively used real-world application developed by konkconsulting which assists in the synchronisation of data between SAP systems.

4.5 Summary

The problem that the present dissertation aims at solving — cancellation of asynchronous API calls, in a reactive environment — is a challenging matter that shall have several approaches in order to solve it. Careful evaluation of each alternative, the results it yields and its drawbacks is paramount to ensure that the best solution is chosen for production. Such integration shall make systems with long running tasks where cancellation is commonplace much more responsive and update-friendly.

Chapter 5

Cancellation of Requests to the Server

As previously explained, konkconsulting is looking for ways of bringing their Java/Kotlin products to the same level of their C# products in terms of cancellation capabilities. For this reason, we figured the best place to start would be the selection of a web framework capable of cancellation. However, just before looking at that, we started with simpler examples.

5.1 Simple tasks cancelling

We quickly realised that cancellation is an intricate subject and that studying cancellation in simpler scenarios was preferable for starting, and thus we did so using threads and similar structures.

5.1.1 C# Implementation

konkconsulting was kind enough to provide examples of code and their explanation regarding graceful cancellation in C#, both in a Console Application, and in a Web Application. The latter is explained in section 5.2.1.

Regarding the Console Application, two implementations are possible: using Threads or [1] using Tasks[2].

In both cases, the algorithm to be run asynchronously is implemented in a function, which receives an instance of a CancellationToken[3]. This cancellation token is used to check if the execution is to be interrupted at each safe point, as seen on Listing 5.1.

The CancellationToken can be used in several ways. Instead of doing a loop of tasks, it can be mingled in the task itself if that is wished, as displayed on Listing 5.2.

Notice the simplicity of the implementation, possible due to the language natively supporting `CancellationToken`. The rest of the code is shown in listing B.4.

[1] https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread
[2] https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
[3] https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken

```
private static void Do(object cancellationToken)
{
    CancellationToken token = (CancellationToken)cancellationToken;
    while (!token.IsCancellationRequested)
    {
        //heavy computation
    }
    return;
}
```

Listing 5.1: CancellationToken usage in while loop

```
private static void Do(object cancellationToken)
{
    CancellationToken token = (CancellationToken)cancellationToken;

    // "atomic operation 1"
    if (token.IsCancellationRequested) doRollback();
    // "atomic operation 2"
    if (token.IsCancellationRequested) doRollback();
    // "atomic operation 3"

    // no need to check cancellation at end of task.

    return;
}
```

Listing 5.2: CancellationToken usage within task code

5.1.2 Kotlin Implementation

As introduced in Chapter 2.1, Koltin has native support for Coroutines[4]. Their parallel nature is highly corresponded by the reactive, non-blocking nature of current web frameworks. This evidentiates both the strengths and weaknesses of coroutines, which of the latter we can highlight the fact that a blocking function running in a coroutine scope will make all coroutines in that scope block when said function hangs waiting for some event. Naturally, **coroutines were not made for running blocking functions**.

5.1.2.1 Using Coroutines (Job)

When a coroutine is launched, a Job is returned so that the programmer can control the newly launched coroutine. The `cancel` method marks the coroutine for interruption at its convenience. However, a coroutine may be abruptly interrupted if the `cancel` call is executed while the coroutine execution is running a `delay` call, unless said call is done in a try catch block. This may make it look like Kotlin has no support for graceful cancellation of running coroutines. Listing 5.3 shows an example of cancellation using coroutines where the cancellation event takes place a certain amount of time past the creation of said coroutine.

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = launch {
        repeat(1000) { i ->
            println("job: I'm sleeping $i ...")
            try {
                delay(500L)
            } catch (e : CancellationException){
                println(e.message)
            }
            if(!isActive)
                return@launch
        }
    }
    delay(1300L) // delay a bit
    println("main: I am tired of waiting!")
    job.cancel() // cancels the job
    job.join() // waits for completion of the job
    println("main: Now I can quit.")
}
```

Listing 5.3: Cancellation in Kotlin using Coroutines

[4] https://kotlinlang.org/docs/coroutines-overview.html

5.1.2.2 Using Threads

An alternative to coroutines is the use of Threads, as implemented by the JVM. Kotlin offers native support for running threaded code, and can be used in a way which is very similar to C#. Listing 5.4 shows an example of graceful cancellation using Kotlin threads, where a Boolean variable `ct` is made available to the thread which it can use to check the cancellation status.

```kotlin
import kotlin.concurrent.thread

fun main() {
    var ct = false
    val t1 = thread {
        val tid = Thread.currentThread().id

        repeat(1000) { i ->
            println("thread $tid: I am sleeping $i ...")
            Thread.sleep(1000)
            if (ct) {
                return@thread
            }
        }
    }

    val tc = thread {
        val tid = Thread.currentThread().id
        println("thread $tid: hit ENTER to cancel...")
        readLine()
        ct=true;
    }

    t1.join()
    tc.join()
}
```

Listing 5.4: Cancellation in Kotlin using Threads

5.2 Cancellation in HTTP via underlying TCP layer

It is well established that HTTP does not support cancellation. Some web servers, namely Kestrel, do a careful handling of the underlying TCP layer in order to determine when a request has been "cancelled". If the requesting party (browser, API, etc.) cancels the request by sending a `FIN` TCP packet, the server could detect this and, without having to write to the socket (in the hope of retrieving an `ACK` from the client), cancel the request on its end. Terminating a TCP connection requires that both parties send a `FIN` TCP packet [22].

There are three parts in this section:

- Cancellation of requests in C# web App (with Kestrel).
- Failed attempt at cancellation of requests in Spring Boot with Netty and Kotlin regular functions.
- Successful cancellation of requests in Spring Boot with Netty and Kotlin suspend functions.

The second part denotes the initial approach at cancellation using JVM technologies and the mistakes and misassumptions taken during that attempt. The third and last part presents the solution to those mistakes and concludes this chapter with a successful implementation.

5.2.1 Cancellation of requests in C# web App (with Kestrel)

C# ASP.NET web applications use the built-in web server Kestrel by default.[5] This web server supports cancellation of HTTP requests via the underlying TCP layer. In order to understand this behaviour, we started by studying a regular connection using Wireshark[6] in order to study the TCP packets being exchanged between the client and the server, which are shown in Figure 5.1.

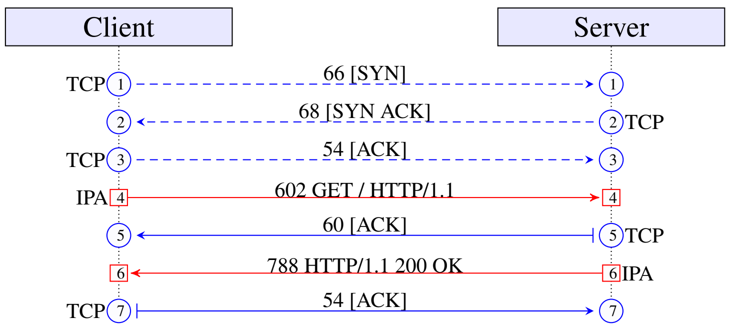


Figure 5.1: Regular request flow in C# with Kestrel web server

The first three packets {1-3} that are exchanged are what is known as the TCP three-way handshake [22]. It is initiated by the client. The server had passively opened its socket and was waiting for connections. After the TCP connection is established, an HTTP GET request is sent by the client {4}, i.e., the browser. The server sends an `ACK` (acknowledged) {5} back to the browser 60ms after having received the request. Usually the `ACK` would be sent in the TCP packet which includes the HTTP reply, but because this request is taking longer than that to be processed, the server lets the browser know that the previous packet arrived. Eventually, the HTTP reply is ready and is sent to the client {6}, which `ACK`s the response {7}. The TCP channel is left open for further requests.

It is relevant to see what happens when the HTTP request is cancelled while the server is preparing the response. This is shown in Figure 5.2.

The first three packets {1-3} are the TCP three-way handshake. After that, an HTTP GET request is sent by the client {4}. Like before, the server sends an `ACK` {5} back to the browser 60ms

[5]https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel
[6]https://www.wireshark.org/

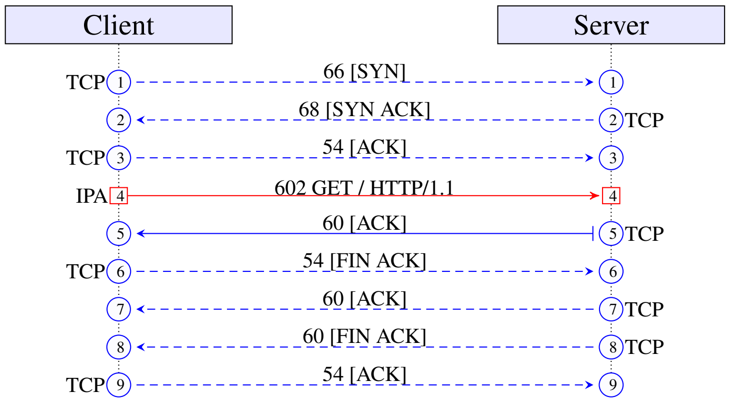


Figure 5.2: Cancellation in C# with Kestrel web server

after having received the request. Differently than before, however, the browser's user presses cancel, and the browser sends a `FIN ACK` TCP packet to the server {6}, which the server `ACK`s {7}. Note that the `ACK` was unnecessary, but it is common to send an `ACK` with a `FIN` just in case the previous `ACK` was lost.[7]. In turn, the server sends a `FIN ACK` TCP packet to the client {8}, which the client `ACK`s {9}. The TCP connection is closed and no further packets are exchanged.

[7] https://cs.stackexchange.com/questions/76393/tcp-connection-termination-fin-fin-ack-ack

5.2.2 Failed attempt at cancellation of requests in Spring Boot with Netty and Kotlin regular functions

It was necessary to find an environment suitable for testing cancellation in Spring. Considering konkconsulting's preference for the Spring environment, a new Spring Application was created using the web starter[8] using Spring Webflux. The endpoint created to test cancellation was served by the method shown in listing 5.5:

```
@GetMapping(value= ["/"], produces= ["application/json"])
fun index() : String {

    logger.info("got request")
    Thread.sleep(1000)
    logger.info("replying to request")
    return "{\"threadid\": " + Thread.currentThread().id + "}"
}
```

Listing 5.5: Spring fun for testing cancellation

The first observation made was that when using Spring with the Netty web server, hitting the cancel button in the browser would not interrupt the request, as the message "replying to request" was being logged. The app would finish its response and only then, when trying to flush it back to the client, learn that it was no longer needed. [9] This is shown in Figure 5.4. Moreover, when receiving the `RST` packet from the browser, the server would throw an `IOException`.

Several sources suggested that cancellation detection, or listening for client channel closed events in server side, must be first implemented on the web server ran by Spring[10] [11]. Spring supports several web servers:

- Netty - NIO (Non-blocking I/O) server engine.
- Tomcat - Spring's preferred web server engine.
- Jetty - Similar to Tomcat, but more lightweight.
- Undertow - NIO and IO server engine.

In order to understand the underlying behaviours, we used Wireshark[12] to study the TCP packets being exchanged between the client and the server. This is shown in Figure 5.3.

[8] https://start.spring.io
[9] https://stackoverflow.com/a/64042877
[10] https://stackoverflow.com/questions/8785949/netty-channel-closed-detection
[11] https://stackoverflow.com/a/2962511/1469991
[12] https://www.wireshark.org/

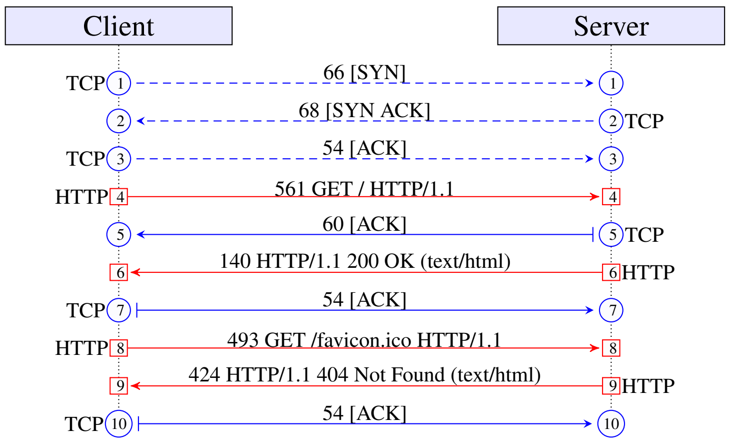


Figure 5.3: Regular request flow in Spring with Netty web server

The first three packets {1-3} are the TCP three-way handshake. After the TCP connection is established, an HTTP GET request is sent by the client {4}, i.e., the browser. The server sends an `ACK` (acknowledged) {5} back to the browser 50ms after having received the request. Usually the `ACK` would be sent in the TCP packet which includes the HTTP reply, but because this request is taking longer than that to be processed, the server lets the browser know that the previous packet arrived. Eventually, the HTTP reply is ready and is sent to the client {6}, which `ACK`s the response {7}. The following TCP packets exchanged pertain to the HTTP request for the favicon {8-10}. The TCP channel is left open for further requests.

It is relevant to see what happens when the HTTP request is cancelled while the server is preparing the response. This is shown in Figure 5.4.

In both cases, the method returns, not knowing if the user is still interested in the response.

5.2.3 Echo Service for Link Detection

Echo Service for Link Detection is an idea we conceived which consists in flushing some data to the client with a well-defined frequency. Clients which have not closed the connection will receive the data and `ACK` it. However, clients which have closed the connection will not be expecting a TCP packet from the server, effectively returning a TCP `RST` packet, which can in turn be caught by the server and used to throw an exception of some sort. It is not clear how user code would have access to that framework information.

Summary

The aforementioned tests were repeated for all 4 supported web servers, with all showing the same behaviour. After having (mistakenly) concluded that it would be necessary to change/extend one of these solutions in order to achieve a behaviour similar to Kestrel's, and discarding the possibility of implementing an echo service for link detection, we decided to move on to cancellation in the other layers of the application.

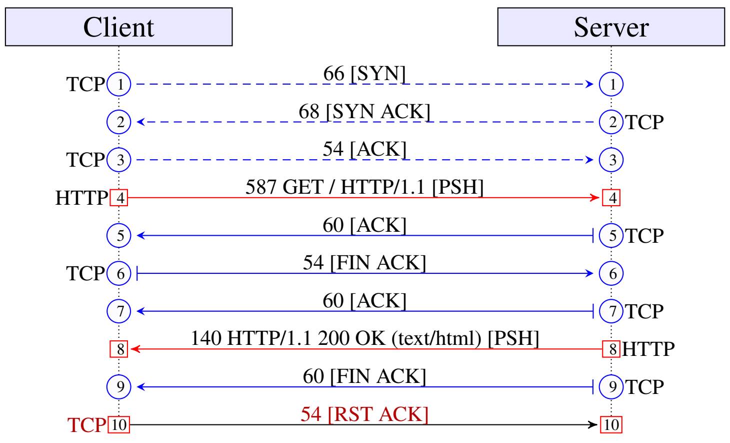


Figure 5.4: Cancellation in Spring with Netty web server

The first three packets {1-3} are the TCP three-way handshake. After that, an HTTP GET request is sent by the client {4}. Like before, the server sends an `ACK` {5} back to the browser 50ms after having received the request. Differently than before, however, the browser's user presses cancel, and the browser sends a `FIN` TCP packet to the server {6}, which the server `ACKs` {7}. The `FIN` flag merely means that the sender will not send any more data. For this reason, it is acceptable for the server to send the reply {8}, and only then send its `FIN` {9}. However, in an HTTP scenario, sending the `FIN` flag is a *hack* used to cancel a request, which both Google Chrome and Mozilla Firefox implement. After sending it, and getting an `ACK`, they clear their TCB (Transmission control block, the data structure that holds all information pertaining to one specific TCP connection). After this, no further replies are expected, which is why the browser replies with a `RST` {10}, signaling the server to immediately destroy its own TCB and send no more data.

5.2.4 Successful cancellation of requests in Spring Boot with Netty and Kotlin suspend functions

Shortly after having given up on HTTP requests' cancellation, another approach regarding the definition of the method for handling requests was experimented with in order to use Flow objects. This alternative methodology yielded a radically different and unexpected result which was **graceful cancellation** on HTTP.

Spring's integration with Kotlin features has been carefully developed to make use of all the language's features. When defining the handler method as a `suspend fun` (suspending function), it is ran in a coroutine context, and is cancelled when the request's transport layer is closed. Listing 5.6 shows an example of method implementation.

`isActive` is a property of the `coroutineContext` variable. This variable in only available if suspending functions are used, and these functions can only be called by a coroutine. Had not Webflux handlers been thought of in a way that would support suspending functions, an error would be thrown, as is the case with other Webflux-related functions (such as configuration and Webfilter functions).

```kotlin
@GetMapping("/coroutine")
suspend fun cancellable(): String {
    try {
        delay(1000)
    } catch (e: Exception) {
        log.warn(e.message)
    }
    log.warn("cr is active: " + coroutineContext.isActive)
    return "Ok"
}
```

Listing 5.6: Reactive Spring fun for testing cancellation

`isActive` — which indicates whether the coroutine has been cancelled — is a property of the `coroutineContext` variable. This variable in present in any coroutine execution scope.

When calling the `/coroutine` endpoint, without cancelling, the log will show `cr is active: true` However, if the request is cancelled, the output is `cr is active: false`.

In terms of Wireshark output, in the context of a cancelled request, the result is shown in Figure 5.5, which is similar to Figure 5.2.

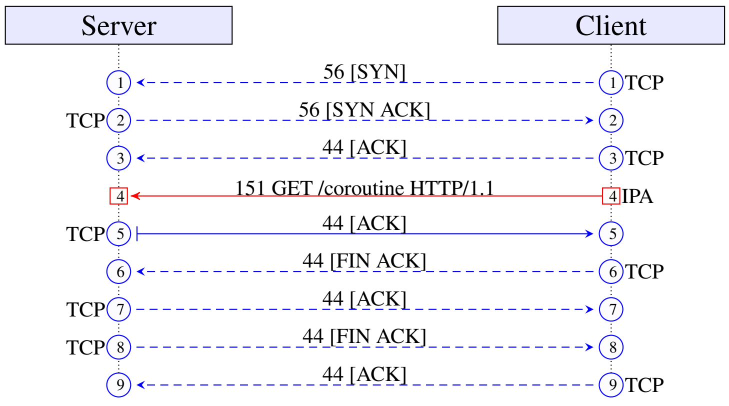


Figure 5.5: Cancellation in Spring with Netty web server using Coroutines

Summary

What was initially proving to be impossible and motivated so much effort and learning regarding the use of Wireshark, the TCP protocol and the efforts that the community has made to achieve cancellation on HTTP became much more straightforward when using suspend Kotlin functions. The novelty of these methods has complicated searching for an adequate solution. The presented solution was found while investigating something else, in this case. This breakthrough opens doors both for HTTP cancellation, this section's topic, and passing the cancellation to the next layers of the application, one of the objectives in terms of implementation of this dissertation.

It is also worth noting that the `suspend` modifier was tested in Ktor, the framework written in Kotlin (and presumably most compatible with its functionalities), using diverse configurations.

However, it failed to provide graceful cancellation — the suspending function handler, ran via a coroutine, would never have its status set to cancelled.

Unfortunately, this behaviour that enables us to do so much in terms of cancellation, allowing for reduction of wasted time and resources such as energy and component's wear and tear, is not documented anywhere (or at least, we could not find it, despite our best efforts). It is our firm conviction that something as useful as this should not only be available in the documentation, but should also be publicised and used as a selling point.

Other webflux-supported web servers — Jetty, Tomcat and Undertow — were also tested with the Kotlin `suspend fun` construct. In all these tests, none of these managed to present graceful cancellation like Netty did. This sealed our choice for Spring Webflux (which was the preferred solution by konkconsulting), and that had consequences, as seen in section 9, which focuses on the mix of Spring MVC and Spring Webflux, and the transition from one to another, while using different Web Servers.

Chapter 6

SQL, Cancellation & Reactive Streams

After concluding a PoC (Proof of Concept) regarding HTTP cancellation, and choosing Spring Webflux on Netty as our solution, we focused on SQL. konkconsulting has been using PostgreSQL but has kept its implementation SQL-agnostic, and that is a hard constraint regarding the technologies available for usage, which must support that database and allow for keeping the implementation SQL-agnostic.

Soft constraints set by konkconsulting are twofold: first, that SQL operations such as `JOIN` need not be manually specified, i.e., an Object–Relational Mapping (ORM) is dealing with the SQL mechanics needed for accessing properties, retrieving and updating values of associated entities, etc., and secondly, that there is the possibility of defining models in the code so that the database can be generated by the library, using the model mentioned above.

Database abstractions are used to support several Database Management Systems (DBMS), though not more than one at each given time. Essentially, this means that different instances of the application can be launched using different DBMSes, which has several advantages:

- migration of DMBS will not involve changing the source-code.
- the choice of the DMBS belongs to the client, when the application is licensed to a third party.
- Database Abstraction Layers (DBAL) are usually deployed with ORM, which allow for easy construction and maintenance of the database using that layer.

However, there are also downsides to using DBALs, namely:

- reduced support for DMBS-specific features which could be used if that DMBS was explicitly used.
- If the selected DBAL does not allow for manually specifying some queries, some form of abstraction inversion may be necessary to code more complex queries, ultimately leading to significant performance losses. Alternatively, queries could be manually specified for each DBMS, if that is deemed necessary.

- When DBALs are deployed with ORM, where the data constraints are defined, these may end up not being defined on the database level (depending on the ORM), which complicates sharing the database with several systems, especially when more than one writes to it — an uncommon, but realistic scenario.

Database Abstraction in the Java environment is a mature issue that has been addressed by several different solutions. There is a specification, JDBC[1] (Java Database Connectivity). The implementation of this specification is usually provided by the DBMS vendors.

The most well-established ORM is specified by JPA[2] (Jakarta Persistency, formerly Java Persistence API). There are several implementation of this, such as OpenJPA or Oracle TopLink.

6.1 Reactive Streams

SQL data retrieval is the first subject that we studied that entails retrieving data from a third party. For this reason, we decided it would be a perfect opportunity to use reactive programming, the paradigm on which Spring Webflux is built upon and that is preferred for this framework. The reactive streams website defines the technology in a sentence:

> "Reactive Streams is an initiative to provide a standard for *asynchronous stream processing* with *non-blocking back pressure*."[3]

Stream processing is the processing of data as a *Flow*, this is, as it is made available by the *producer* to be processed by the *consumer*. Flows can be either a *Flux* — 0 or more elements — or a *Mono* — 0 or 1 elements.

Non-blocking back pressure means that the consumer has the ability to signal the producer that the rate of emission is too high (or not as high as it could be). One can think of back pressure as the resistance the software offers to the flow and transformation of data through and by it. If a user generates 30 requests per second, but the server can only handle 15 requests per second, some form of back pressure mechanism is necessary. Phelps explains this well [19]. He presents 3 strategies to deal with back pressure:

- Control the producer: using reactive mechanisms, the Subscriber requests that the Publisher reduce the rate of emission of data.
- Buffer — used when the flow from the producer is expected to diminish, so that the buffer will not become full. This solution is not scalable.
- Drop — when control is impossible and the producer's throughput is too high to handle — e.g. user input data.

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/
[2] https://javaee.github.io/javaee-spec/javadocs/javax/persistence/package-summary.html
[3] https://www.reactive-streams.org/

Part of dealing with back pressure entails the right of the consumer to abort or *cancel* the flow. This is part of what makes Reactive Streams an attractive possibility in this scenario.

Reactive Streams are comprised of two key elements, the Publisher and the Subscriber. The relation between the two is called a **Subscription**. When an element is both a Subscriber and a Publisher it is called a **Processor**. Processors typically subscribe to one or more Publishers (as a Subscriber), and emit data to a Subscriber (as a Publisher). The emitted data is a transformation/mapping/etc. of the original flow(s).

When doing reactive programming, one creates what is called a reactive pipeline. This pipeline is the description of the operations to be applied to the Flow. A Flow is a set of elements emitted by the Publisher. Only when the flow is *subscribed* will the pipeline run.

The Reactive Specification does not mention any security considerations regarding abuse from users. For example, a malicious user could subscribe a big stream and only request one item (similar to a SYN Flood[4]). However, from a reactive perspective, this should not create additional load for the server because it will only request (from the source) the pieces that the client has requested. It can, however, be problematic if the server is loading the data to be emitted from a non-reactive source, as it must buffer the payload until it is fully emitted.

6.2 Reactive Streams in SQL: R2DBC

From a reactive perspective, present database drivers and ORM solutions are unable to provide results as a subscribable flow. For this reason, **R2DBC**[5] has emerged.

R2DBC is natively supported by Spring Data and has special integration capabilities with Kotlin. With the shortcomings of other solutions, this has become the *de facto* standard in the Java environment for reactive database access. However, no major release has yet been launched, with paramount functionalities like SQL `JOIN` still missing. Moreover, it is not production ready yet. R2DBC is founded on the Reactive Streams specification, and this means that its usage implies that the developer is familiar with one of the supported implementations, either Project Reactor[6], RxJava[7] or Smallrye Mutiny[8]. As Spring has integration with Project Reactor, that is the one we used in our investigation.

6.2.1 Model Definition and Data Retrieval

Defining models while using R2DBC is a simple task. The fact that there is no support for relations makes it much cleaner than other alternatives, but also much more limited. Due to this limitation, we chose to model a User (shown in Listing 6.1), instead of the common Posts & Tags model (shown in Appendix A.1).

[4]https://en.wikipedia.org/wiki/SYN_flood
[5]https://r2dbc.io/
[6]https://projectreactor.io/
[7]https://github.com/ReactiveX/RxJava
[8]https://smallrye.io/smallrye-mutiny/

```kotlin
 @Table("users")
data class User(
        @Id
        val id: Long? = null,
        val name: String,
        val login: String,
        val email: String,
        val avatar: String? = null
)
```

Listing 6.1: User Model for R2DBC.

6.2.2 Constraints

- R2DBC offers no support for relations — It is possible to declare a `@Transient` property (which means it will not be persisted by R2DBC) and manually fetch and match the related object(s).[9] This simple (but laborious) hack merely allows accessing a relation property as it had been automatically retrieved, but requires that the user does it manually, which is hardly acceptable.

- The way R2DBC is built makes is unfeasible to add relation support. Moreover even if support is added, in the near future and after solving the issues that prevent it from happening, plans are for ***read**-relation mapping*, which is insufficient for konkconsulting.[10]

- R2DBC is the reactive solution for simple APIs. Complex environments should use Spring Data JPA and execute calls to repositories in a separate thread using `subscribeOn`.[9]

6.2.3 Craftiness

As with almost every other situation, it is possible find a workaround and make things work in a way that is not the envisioned one.

Some guides, such as "You Don't Need Hibernate With Spring WebFlux and R2DBC"[11], by Yuri Mednikov, claim that R2DBC is relation-ready and even present a working solution. However, they work by manually defining the SQL queries and collecting data as a flow. While this may be a good solution for the common PoC, it will not work for our needs as we've explained in the introduction of this chapter.

Another alternative is the specification of Views on the DBMS, joining the desired tables. These views are interpreted as tables by R2DBC and allow for fetching results from the database in a cancellable flow, provided that the view is defined as a model in the R2DBC configuration. However, there are several limitations to this idea that make it unfeasible:

[9] https://stackoverflow.com/a/60249642/1469991
[10] https://github.com/spring-projects/spring-data-r2dbc/issues/99#issuecomment-610783285
[11] https://dzone.com/articles/you-dont-need-hibernate-with-spring-webflux-and-r2

- Most views combining tables are read-only (depending on the DBMS and the viewed tables), i.e., no writes are possible.

- All combinations of used relations would have to be defined using Views, which, if not done programmatically, would become extremely time-consuming.

- There is a possibility that this solution may have a weak performance due to usage of sub-optimal views when optimal views are not available.

6.3 JDBC: an old Cornerstone

Considering the large limitations of R2DBC, we have decided to look into the possibility of using JDBC in a way that, while not being reactive, is at least asynchronous. There are two aspects to consider:

- How well does the library fit to our aforementioned constraints.

- If the previous item is satisfied, how much can we obtain in terms of bandwidth/response time?

The analysis of the second item is relevant for several reasons, namely the fact that the switch from blocking synchronous to blocking asynchronous only makes sense if the current system is constrained by the amount of users using the application concurrently, and also that the jump to reactive becomes more necessary as the quality of our results decreases (though there are alternatives that may be put in place before that, such as a data cache [e.g. Redis]). Results for these trials are presented in section 10.1.

Developers and authors dwell on a solution for preventing regular DBALs from blocking the entire application when using them in a reactive environment. There are two main ideas:

- Using thread pools: a heavy solution, according to Paluch, in the Spring Blog [18]. Reactive programs usually limit the number of threads to the number of cores available, in a process called "thread limiting". "Additional threads introduce overhead and reduce the effect of thread limiting", says Paluch.

- Using Java Fibers: Paluch suggests combining the well-established JDBC with Project Loom's Java Fibers. Unfortunately, they are not yet available. A comparison of Java Fibers and Kotlin Coroutines is presented in section 2.1.4.

Naturally, only the first of the above is feasible. With this in mind, we started looking for libraries using JDBC. Several caught our attention, namely: ADBA, jOOQ, Ebean and Hibernate.

6.3.1 ADBA

ADBA was an effort by Oracle to provide a non-blocking database access API, to be integrated as a Java Standard.[12] However, work on ABDA abruptly came to an end in September 2019[13] as Oracle prepared to launch Project Loom featuring Java Fibers. While the developed efforts have been made public and licensed in a way that allows further development by third parties, no one has continued the effort. This is not a viable option considering that it is unfinished and the lack of support for what is currently available.

6.3.2 jOOQ

jOOQ Object Oriented Querying (jOOQ) is a database-mapping software library. Its fluent API is one of its banners and that initially drew our attention. However, another of its banners is the ability to "generate Java code from your database". While this is interesting from a theoretical point of view, konkconsulting's environment is currently prepared in a way that the database is generated by the code, not the opposite. jOOQ revolves around this feature, providing no documentation for users wanting to manually define their own models.

Conversely, jOOQ is the first tool we have found during this dissertation that offers paid support plans. This is something that companies appreciate as it guarantees proper support while using that product. However, the lack of the required functionalities in jOOQ is *not* compensated by this advantage.

It is also worth noting that, while the jOOQ blog has some posts regarding the use of the library in Kotlin, there is no official support. Additionally, the lack of an advanced ORM is commonly frowned upon by the community.

For these reasons, we have concluded that jOOQ is not a viable option for defining data structures in the Java/Kotlin code, which is konkconsulting's case.

6.3.3 Ebean

Ebean came across our radar due to its capability to return a Java Future when doing SQL queries. Ebean offers support for Kotlin and Java and has an extensive documentation. Its banners are Type safe queries, ORM and Database migrations, among others. Ebean's Futures look promising for our Asynchronous SQL effort and are used in our experiments.

Ebean Models are well explained in their documentation, and are similar to JPA specification, which allows for easier migration. The Ebean model definition is shown in Appendix A.1.

```
suspend fun getAllPosts(): List<Post> {
    val future = QPost().orderBy("id").findFutureList()
    // other calls to database or other services
```

[12]https://blogs.oracle.com/java/jdbc-next:-a-new-asynchronous-api-for-connecting-to-a-database

[13]https://mail.openjdk.java.net/pipermail/jdbc-spec-discuss/2019-September/000529.html

```
4       return future.get()
5   }
```

Listing 6.2: Obtaining a Java Future using Ebean

Listing 6.2 shows a request to the database which returns a Java `Future`[14]. This means that the call immediately returns, and only when calling the object's `get()` method will there we block for the response, if necessary (depending on whether the result has yet been fetched or not). When running this code, some early testing using Apache Bench (AB) quickly revealed that this approach was actually worse than calling the regular `.findList()` function, which immediatelly blocks for the result. While the reason why this happens in not clear, there is a blocking call on line 4, which is unacceptable in a coroutine context. Thus, we started looking into the possibility of asynchronous, Kotlin-friendly way of waiting for a Java `Future` to be `done`, albeit without success. Webflux cannot handle receiving a Java `Future`, like it does with `Flow`, for example.

```kotlin
suspend fun getAllPostsv3(): List<Post> {
    return GlobalScope.async {
        val future = QPost().orderBy("id").findFutureList()
        while(!future.isDone){
            delay(10)
        }
        return@async future.get()
    }.await()
}
```

Listing 6.3: Using Java Futures and the async method in Kotlin, we can use a special coroutine for asynchronous polling of the Future.

Listing 6.3 displays polling for a `Future`. This approach runs the call in a separate coroutine, which is acceptable. While this looks terrible, it did evidentiate something expectable, but unseen thus far: the PostgreSQL tests server, with `show max_connections;` returning `100`, reached its limit when benchmarking the application with over 100 concurrent connections. Upon further investigation, we have determined that Ebean does not offer any control and queuing mechanisms, which also deters its selection as a viable choice.

In order to assess Ebean's usability as an ORM framework, we developed a query with a certain degree of complexity (Listing A.1.2.1) and attempted to code its ORM equivalent. The core code is shown in Listing 6.4. Running this code is only possible if we declare a model class for the relation, which is not desired, and if that class has a dummy property used for accessing the result of the select as shown on Listing 6.5, which is also not desired.

[14] `https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/concurrent/Future.html`

```
val o = QPostTag._alias
val res = QPostTag().select(o.post_id, o.totalCount).having().totalCount.
    greaterThan(5).findList()
```

Listing 6.4: Example of Ebean COUNT HAVING

```
@Aggregation("count(*)")
var totalCount: Long? = null
```

Listing 6.5: Dummy property needed for querying an aggregation

Throughput Analysis

Further tests were ran regarding Ebean's throughput. Results for these trials are displayed, compared and studied in section 10.1.1.

Summary

Ebean shows lots of potential and makes accessing relational databases relatively easy, especially when simple lookups/inserts are done. The use of Futures also potentiates the cancellation of queries and makes it much simpler to mix database access with other resources' access, by taking advantage of Java Futures. The fact that it automatically introduces some optimizations makes it look better than the alternatives, but ends ups hiding the potential for further tweaking, which is not a good thing in our opinion.

However, the need to define a class for a relation, and even to define a property for each special query that we may wish to retrieve makes it a hurdle in terms of maintenance. This is one of the main reasons why we cannot use it. Another strong motive to discard it is the fact that it does not have a thread-pool management system, which would mean that requests would be dropped after a certain number of concurrent requests. This is not the type of back-pressure response that we intend.

It is worth noting that Ebean development is facilitated by the usage of an Intellij IDEA plugin[15]. The documentation states that it can be used so that "entity beans and transactional methods are enhanced when run develop and run tests in the IDE". Note that it is only available for the mentioned IDE.

Moreover, using Ebean requires the inclusion of a Gradle/Maven plugin, Kapt, which is necessary for generating Q-classes. These are generated from the provided models and used for creating queries. It is easy to miss this caveat, which is why we include a full `build.gradle.kts` in Listing B.1.

[15] https://ebean.io/docs/getting-started/intellij-idea

The documentation is vast, but simple, and seems to have grown a lot recently, as the community frequently complained about its shortcomings. The small community, the reduced support (due to the small community) and a lack of a guarantee that the project will not be abandoned soon force us to remove this option from the table.

6.4 Hibernate (via Spring-Data-JPA)

Having reviewed the remaining alternatives with little success, we moved to JPA, namely its interface with Spring — `spring-data-jpa` — which uses the Hibernate implementation by default. This interface offers no support for futures or any form of cancellation, making it a simple, blocking library. Nevertheless, Hibernate is the *de facto* solution for relational database access in the Java Ecosystem[16], and it is also the one that konkconsulting has been using.

`spring-data-jpa` has its own thread-pool management library, HikariCP[17], which means that unlike Ebean, we are not taking the risk of dropped requests when the database limit is reached. Moreover, as connections are kept open, even though they might be idle, we reduce the overhead of connecting to the database. Hence, when using a thread-pool, requests are queued and allotted to a thread as soon as one is available. HikariCP allows the developers to tweak the minimum and maximum number of threads in the thread-pool. This means that HikariCP does an elastic management of the available threads, which is something positive in our opinion.

6.4.1 Query Tweaking

When defining models, it is possible to set the way objects are retrieved from the database. These parameters influence the speed of the retrieval, which is important because we want to minimize blocking and waiting times.

FetchType

The JPA documentation [17] mentions FetchType in several circumstances, but always with the same meaning: it is a parameter for setting up relationships which defines the way the related objects are fetched:

- LAZY — fetch related data from the database as it is necessary (e.g. because it is being accessed).
- EAGER — fetch related data from the database together with the primary data.

Default values are shown on table 6.1.

Trials with this parameter on the Test Data A.1 had no notable results.

[16] `https://www.jrebel.com/blog/best-java-frameworks`
[17] `https://github.com/brettwooldridge/HikariCP`

Context	**Default Value**
Basic	`EAGER`
ElementCollection	`LAZY`
ManyToOne	`EAGER`
ManyToMany	`LAZY`
OneToMany	`LAZY`
OneToOne	`EAGER`

Table 6.1: Tools overview

@Cacheable

This annotation specifies if an **@Entity** should be cached, as long as caching is enabled and the configuration value `shared-cache-mode` is `ENABLE_SELECTIVE` or `DISABLE_SELECTIVE`. However, some caching mechanism must be available. Spring makes it easy to do that by setting the `spring-boot-starter-cache` dependency.

@BatchSize

This annotation is not specified in the JPA specification. However, Hibernate implements it. When using Lazy Fetching, this defines the number of entities to be fetched from the database per each query. Without this setting, each entity is fetched at once. We should note that Ebean implements a similar mechanism with a default value of 10 entries per request.

@Fetch

This is another Hibernate-specific annotation. It is applied to relations, and sets the way that the related entities shall be retrieved. Its possible values are displayed on Table 6.2.

As this setting is Hibernate specific and actually overrides the JPA FetchType parameter, we focused on it for our trials.

Value	**Description**
SELECT	A secondary **SELECT** will be used for retrieving each requested related entity. This is equivalent to **FetchType.LAZY**.
JOIN	An **OUTER JOIN** will be used to fetch the related entities. This is equivalent to **FetchType.EAGER**.
SUBSELECT	A single secondary **SELECT** will be used to load all the related entities.

Table 6.2: Tools overview

6.4.2 Threads & Blocking

As previously explained, blocking code should not be mixed with reactive code: it breaks the reactive pipeline and thwarts the effort of going reactive, making the whole application block (not just the thread or request). Tests shown in the Results Chapter helped us realise that Spring Webflux can detect that a Spring JPA query may/will take place in a handler and thus uses something we had not seen before: boundedElastic Threads. After persevering our our search, we found what these are. Their usage is documented in section 2.4. Considering the way we have read and heard about blocking code being a hazard, it is good to learn that no special care is needed when using Hibernate with Spring Webflux, as it is specially coded to handle it. It is not clear, however, how other blocking libraries would be handled.

Throughput Analysis

Further tests were ran regarding JPA's throughput. Results for these trials are displayed, compared and studied in section 10.1.2.

6.5 Summary

As much as we would have liked using a reactive access layer, the lack of support for pivotal functionalities, such as relations, made us forego that possibility. We believe that in a few year's time it might be possible to redo this exercise with a different outcome.

That said, access to databases is done mainly using the JPA specification, and there are several implementations, such as Hibernate or Ebean. Of all the analysed possibilities, JPA proved to be the most reliable and customisable. It is also the one konkconsulting has experience with. As Kotlin makes it simple to run queries in a Thread Pool, its usage is acceptable and it is our choice. The management of connections to the database offered by HikariCP, part of `spring-data-jpa`, is rather complete, allowing developers to tune the minimum and maximum number of open connections.

The comparison of the performance of these libraries was not one of our objectives, but became relevant when we realised that there were major performance differences between them. This exercise can be found in section 10.1.

We do an overall positive balance of this effort: it is good to have learned that there are several libraries for different people with different use cases and objectives; however, there is one solution with clear advantage in terms of functionality and developer preference: Hibernate.

Not all languages are like this. For example, the PHP documentation — which is built with support for users' comments, showing on top the most up-voted ones — is ambivalent in terms of MySQL/MariaDB access (for example): users can use the mysqli library[18], or PDO[19] (PHP Data Objects). Many developers discuss and debate which one is best, but there is no clear winner, with both having their pros and cons; we feel the ORM access environments in Java lack this variety.

[18] `https://www.php.net/manual/en/book.mysqli.php`
[19] `https://www.php.net/manual/en/book.pdo.php`

Chapter 7

Third-party HTTP Requests: Asynchronous & (sometimes) Cancellable

The rise of container orchestration tools, such as Docker, contributed to the decoupling and increasing of the number of services that compose an application, by simplifying the deployment of the development environment and the production version. The usage of such tools greatly simplifies both of these operations. As the number of services increases, so increases the number of messages exchanged between them.

Beside messages being exchanged between containers running on the same machine, in which latency is hardly a problem, there can also be communication with third-party APIs, such as government data/ID sources, currency exchange ratio providers, weather providers, etc. These may be slow to respond, may respond with an error, or may even not respond at all. This means that implementations should carefully handle the obtained response, and make a senseful management of timeouts — these are just as important as keeping everything non-blocking.

Considering both latency and the Time to First Byte (TTFB), it is important that applications do not block while waiting for an external API: as previously explained, whenever threads block on something, more threads are required to keep the application running, which may degrade performance after a certain point.

For all the reasons stated above, the study of an asynchronous, non-blocking way of doing HTTP calls is paramount in the search for the best tools for developing Fully Asynchronous APIs for Java Web Applications.

7.1 Setting up an API

In order to conduct tests, we implemented a simple API returning a JSON file. For this, we used Node.js. The implemented API code is shown in Listing 7.1.

```
const http = require('http');
const fs = require('fs');

let f = fs.readFileSync(<file.json>)

//create a server object:
http.createServer(function (req, res) {
    setTimeout(function () {
        res.write(f)
        res.end(); //end the response
    }, <sleep_time>)
}).listen(8081); //the server object listens on port 8080
```

Listing 7.1: Simple HTTP responder implemented in Node.js

7.2 The simplest HTTP request

For reference, we started by implementing a simple HTTP GET request to our API, which is shown in Listing 7.2. It is based on a Baeldung guide.[1].

```
@GetMapping("/3prest/1")
fun rest1(): String
{
    val url = URL("http://localhost:8081")
    val con = url.openConnection() as HttpURLConnection
    con.requestMethod = "GET"

    val input = BufferedReader(InputStreamReader(con.inputStream))
    var inputLine: String?
    val content = StringBuffer()
    while (input.readLine().also { inputLine = it } != null) {
        content.append(inputLine)
    }
    input.close()
    con.disconnect();
    return content.toString()
}
```

Listing 7.2: Example of simple HTTP GET request

[1] https://www.baeldung.com/java-http-request

If the handler function is marked with `suspend`, IntelliJ IDEA shows warnings saying:

`Inappropriate blocking method call`

However, the general behaviour seems to be consistent regardless of `suspend` being present.

The result is that the issued request completes successfully and the retrieved content is displayed in the browser — as expected.

7.3 Netty as an HTTP Client

The simplest solution works — both with a dedicated thread-pool and without (see section 10.2). However, thread-pools are always present, and they are known for not scaling well. Naturally, we continued our study onto the field of non-blocking HTTP libraries. Netty was an easy consideration, as we were already using it in the project. However, implementation was not easy for a number of reasons, from the lack of up-to-date documentation to the cryptic name of the classes involved. Netty allows for low-level configuration and that has some advantages, but also disadvantages. HTTPS is not supported in our current implementation, part of which is shown in Listing 7.3.

```
    @GetMapping("/3prest/3")
    suspend fun rest3(): String {
        val url = System.getProperty("url", "http://run.mocky.io/v3/3e1e2a69-93e8
            -41e7-9870-602531b08d45")
        val uri = URI(url)

// Configure the client.
        try {
            val handler = HttpSnoopClientHandler()
            val initer = HttpSnoopClientInitializer(handler)
            val b = this.bootstrap.clone()
            b.handler(initer)

// Make the connection attempt.
            val ch: Channel = suspendCoroutine<Channel> { continuation ->
                b.connect(host, port).addListener {
                    continuation.resume((it as ChannelFuture).channel())
                }
            }

// Prepare the HTTP request.
            val request: HttpRequest = DefaultFullHttpRequest(
                HttpVersion.HTTP_1_1, HttpMethod.GET, uri.rawPath
            )
            request.headers().set(HttpHeaders.HOST, host)
            request.headers().set(HttpHeaders.CONNECTION, "close")

// Send the HTTP request.
```

```
            ch.writeAndFlush(request)

// Wait for the server to close the connection.
            suspendCoroutine<Unit?> { continuation ->
                ch.closeFuture().addListener {
                    continuation.resume(null)
                }
            }
            ch.close()
            return handler.output
        } catch (e: Exception) {
            println(e.message)
            e.printStackTrace()
        }
        return "error"
    }
```

Listing 7.3: Example of HTTP GET request implemented using Netty

Naturally, the usage of this solution would entail creating a wrapper class for simplifying access to the functionality.

While running tests for this implementation, we realised that it is faulty: although it works for some requests per second, bench testing it in volume, like done previously, resulted in a silent failure in which the response body came out empty. During our investigation, in order to solve this issue, we discovered Spring's WebClient. This became our choice, partly due to time constraints, and also due to its simplicity when compared to a Netty Implementation.

7.4 Spring WebClient

In retrospect, this should have been our first choice. Unfortunately, it did not present itself to us when searching for ways to make third-party web requests. Baeldung explains that "WebClient is an interface representing the main entry point for performing web requests"[2]. Its reactive and non-blocking nature makes it an exceptionally ideal candidate for this kind of task. However, the fact that it is bundled with Spring Webflux makes it harder to use it in other frameworks.

The implementation is remarkably simple when compared to previous approaches. It is shown in Listing 7.4.

```
@GetMapping("/3prest/wc/0")
fun rest4(): Flux<Todo>{
    WebClient.create("http://localhost:8081")
        .get()
        .uri("/")
        .retrieve()
```

[2]https://www.baeldung.com/spring-5-webclient

```
        .bodyToFlux(String::class.java)
}
```

Listing 7.4: Example of simple HTTP GET request using WebClient.

As the reader has certainly noticed, we always implement handler endpoints using the `suspend` keyword, so that we can catch the HTTP cancellation and handle it, as shown in Chapter 5.2.4. This solution, however, omits that. That is because when using that keyword we were getting the following output on the browser, instead of the contents emitted by the server:

```
{
    scanAvailable: true,
    prefetch: -1
}
```

In fact, we were getting the `Flux<*>` object instead of its content. Transforming the output into a Kotlin Flow fixes the issue, but creates another, as explained in Section 9.9.2. Regardless of fixes, this is highly unnecessary because the function immediately returns the Publisher which is in turn subscribed by Webflux. Thus, supporting cancellation (by checking the `coroutineContext.isActive` property) in this kind of function is unnecessary. Nevertheless, this quick fix is shown in Listing 7.5.

```
@GetMapping("/3prest/wc/1")
suspend fun rest4(): Flow<Todo>{
    WebClient.create("http://localhost:8081")
    .get()
    .uri("/")
    .retrieve()
    .bodyToFlux(Todo::class.java)
    .asFlow()
}
```

Listing 7.5: Example of simple HTTP GET request using WebClient.

Both implementations yielded very similar results. We analysed the second one, as this allowed to further understand the implications of returning `asFlow()`. Results are available in Chapter 10.2.3.

7.5 Cancelling HTTP requests

The cancellation of requests to the server must entail the cancellation of requests created by the server to other APIs. For this reason, we found it necessary to study how WebClient handles cancelled requests. Previously created endpoint shown in Listing 7.5 is perfect for trials, so we will be using it.

We started by studying the behaviour without any kind of cancellation. Figure 7.1 shows this behaviour.

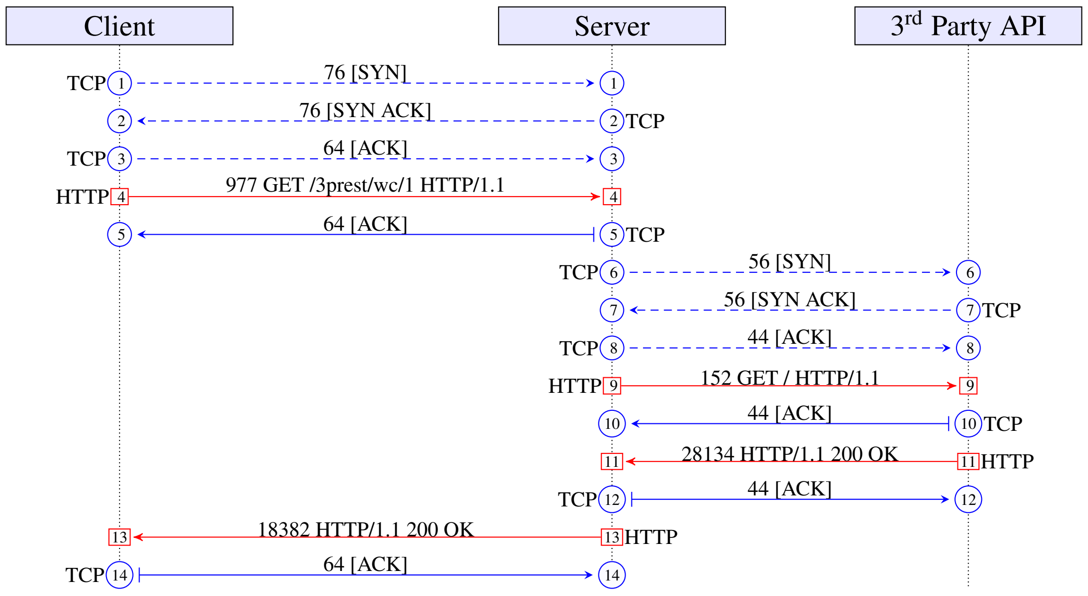


Figure 7.1: Regular request flow when calling 3rd party API.

The first three packets {1-3} are the TCP three-way handshake. After the TCP connection is established, an HTTP GET request is sent by the client {4}, i.e., the browser. The server sends an `ACK` (acknowledged) {5} back to the browser, letting it know that the previous packet arrived.

After that the request to the third party API is started. The first three packets {6-8} are the TCP three-way handshake. After the connection is established, an HTTP GET is sent by the server {9} to the third party API, which `ACK`s the request {10}. When it is ready, the third party API sends the response to the server {11}, which in turn `ACK`s the response {12}.

Lastly, the HTTP reply is ready and is sent to the client {13}, which `ACK`s the response {14}.

The results are as expected, so we repeat the same exercise, but cancelling the HTTP request before it is completed. Figure 7.2 shows this behaviour.

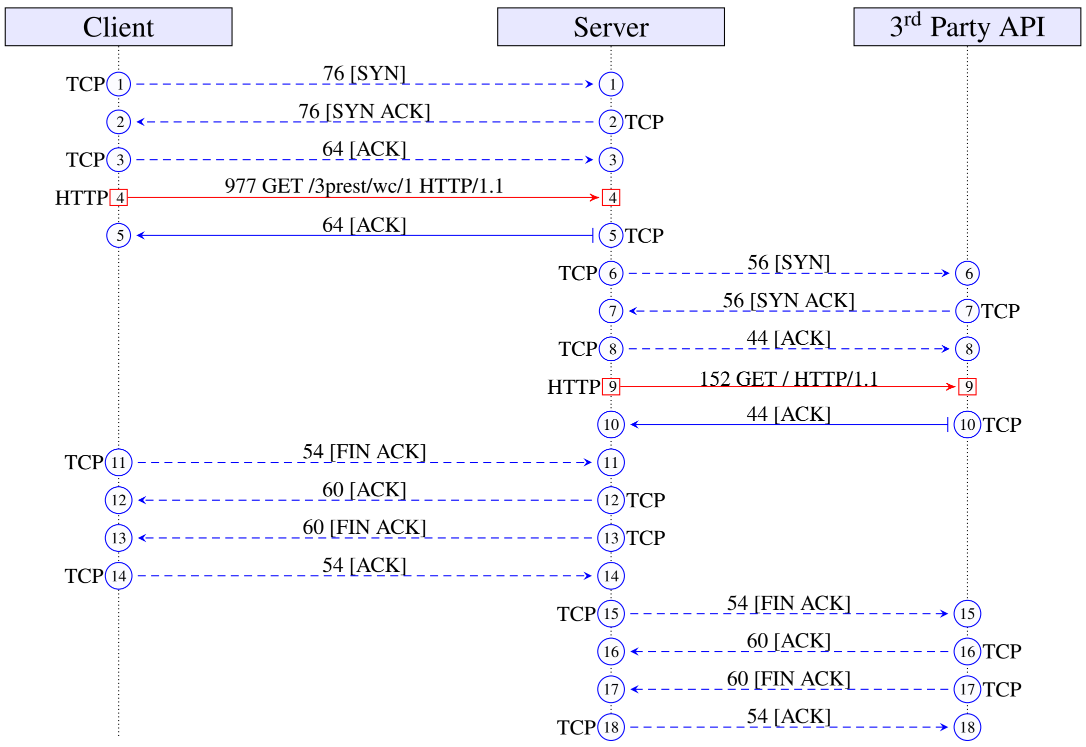


Figure 7.2: Cancelled request flow when calling 3rd party API.

Steps {1} through {10} as the same as explained in Figure 7.1.

This time, however, before the third party API manages to get its response ready to flush back to the server, the server receives a `FIN ACK` TCP packet from the browser {11}, signaling the cancellation of the request, which the server `ACK`s {12}. Note that the `ACK` was unnecessary, but it is common to send an `ACK` with a `FIN` just in case the previous `ACK` {5} was lost[3]. In turn, the server sends a `FIN ACK` TCP packet to the client {13}, which the client `ACK`s {14}. The TCP connection is closed and no further packets are exchanged between the server and the client.

Lastly, the server attempts to close the connection to the third party API, to which it sends a `FIN ACK` TCP packet {15}, which the third party API `ACK`s {16}. In turn, the third party API sends a `FIN ACK` TCP packet to the server {17}, which the server `ACK`s {18}. The TCP connection is closed and no further packets are exchanged between the server and the third party API.

The same tests were repeated for the other WebClient endpoint, shown in Listing 7.4, which yielded the same results.

We drew several conclusions from these trials:

- The less relevant conclusion, albeit most interesting: Node.js' `http` library which allows us to create a server via `http.createServer()` supports cancellation of the likes of Kestrel

[3]https://cs.stackexchange.com/questions/76393/tcp-connection-termination-fin-fin-ack-ack

and Spring Webflux on Netty. Naturally, we did not delve into detecting cancellation in that platform, but a quick search seems to indicate it is possible[4].

- Repeating the tests with a `POST` request instead of a `GET` resulted in the same behaviour. This is problematic because `POST` is not an idempotent operation, and no feedback is given on whether the operation was:
  - cancelled and rolled back
  - cancelled and left mid-way
  - not cancelled, and finished
- When getting a cancellation request, Webflux first cancels with the client, and only after that will it cancel with the 3rd party API, favouring the client, and providing a smoother experience. The process is quite quick anyway: from the browser's `FIN ACK` {11} to the server, to the third Party API's `ACK` to the server {18}, it takes about 0.55ms (on *localhost*, where latency is negligible, and without any load).
- When returning a Publisher — a `Flux` or a `Mono` — in a Controller handler function, it is Webflux itself that *subscribes* to that publisher in order to send the response back to the request's origin, e.g. the browser. While we knew this, we had not conceived that non-suspending functions could have any kind of cancellation behaviour associated. Moreover, there is actually a way to treat a cancellation request when a Flux is returned, which is explained in the next section.

7.6 Handling cancellation when using WebClient

As we have seen, Webclient performs cancellation regardless of the method used. POST, the only non-idempotent method in HTTP, should, in our opinion, not be cancelled. Because it *is* cancelled we must develop a strategy to ensure that there is a way to either:

- enforce the execution until the request finishes gracefully, catch the cancellation, and then take some action, depending on the situation.
- catch the cancellation on-the-flux and handle it there, taking some action, depending on the situation.

The first alternative is quite simple: just use one of the blocking alternatives presented in this chapter. However, this introduces inconsistency regarding the use of WebClient. An alternative is to `block()` the Publisher and check for cancellation afterwards, but this must be run in a special thread or Kotlin `async` block, because calls to `block()` are not supported in the threads running the controllers, `reactor-http-nio-*`.

[4] `https://stackoverflow.com/questions/35198208/handling-cancelled-request-with-express-node-js-and-angular`

The second alternative is simpler, in our opinion. However, depending on the nature of the code to be run (blocking or non-blocking), it may not be the best option. Thanks to Kotlin's SAM conversions[5], it is possible to pass an Anonymous Runnable directly to the `doOnCancel()` function. An example of this is shown in Listing 7.6.

```
@GetMapping("/wc/2")
fun rest6(): Flux<Todo>? {
    return WebClient.create(url)
        .post()
        .uri("/")
        .body(Mono.just("a body"), String::class.java)
        .retrieve()
        .bodyToFlux(Todo::class.java)
        .doOnCancel {
            println("http request was cancelled")
            // handle cancellation here
        }
}
```

Listing 7.6: Example of simple HTTP POST request using WebClient, featuring a cancellation handler in the Reactor pipeline.

Unfortunately, there is no way to prevent cancellation in this scenario (or if there is, we could not find it). However, one can detect if cancellation took place (by providing a handler to `.doOnCancel()`) and deal with that according to the *safety* of the cancelled request.

Summary

The research of HTTP requests to third-party APIs has proven to be a great opportunity to use the reactive model and to understand it in a deeper way. Unfortunately, we only addressed this subject very late in our work. Nevertheless, we learned that it is usually a good idea to look for a way to do a task in a Webflux environment, and not simple in a Java or Spring environment. The differences in the architecture and the fact that reactive solutions are preferred makes it much simpler to use the solutions that have been specifically tailored for Webflux.

The need to set up a testing environment and the fact we intuitively decided to use Node.Js yielded this conclusion: that Express.js, the commonly used Web framework in the JavaScript environment[6], *does* support cancellation. It was to be expected, but despite that, it is a ground-breaking information.

We learned that not everything on Baeldung is worth our time: our first attempt was taken from Baeldung, but was hardly a good choice. The main problem is that the website does not

[5] https://kotlinlang.org/docs/java-interop.html#sam-conversions
[6] https://nodejs.org/en/user-survey-report/#profile2

differentiate between Spring MVC and Spring Webflux articles, which confuses developers who erroneously think they found what they were looking for.

Despite all these positive aspects, there are two details that we consider to be negative outcomes. First, we consider that using Netty as an HTTP client is harder than it should be. Online guides are non-existent, and tutorials are poor, at best. We found the class and function names complex, possibly because we do not have all the knowledge necessary to use it correctly. Nevertheless, its implementation on WebClient seems to be more than sufficient, which is positive. Secondly, how WebClient works better than antecipated, with a low learning curve despite its reactive nature. It is an amazing piece of software, which only works on Spring Webflux. Its usage may be undesired if one wants to keep their team fluent with one library that can be used and reused across several projects.

All in all, achieving cancellation was our greatest objective and despite it being mostly an undocumented feature, we managed to achieve it. Table 7.1 below sums up the pros and cons of each solution.

Solution	**Viable under stress**	**Cancellable**	**First-requests' overhead**
HttpURLConnection	Yes if using a thread-pool	No	Negligible
Netty Client	No, responses are lost	No	n/a
WebClient	Yes	Yes	Measurable (Section 10.2.3)

Table 7.1: Comparison between third-party HTTP request libraries in terms of usefulness.

Further conclusions regarding loading times and performance benchmarks can be found in section 10.2.

Chapter 8

WebSockets

This is the last subject that we will be studying. The Websocket Protocol, which introduced bidirectional communication in the browser environment, seems recent but it will actually be 10 years old by December 2021[1]. Nevertheless, web applications featuring bidirectional communication existed before Websocket, and were usually based on what has become known as HTTP Polling — regularly repeating a request to the server asking for updates.

The Websocket Protocol Specification [5] explains that it is highly versatile, working for applications such as "games, stock tickers, multiuser applications with simultaneous editing, user interfaces exposing server-side services in real time, etc."

Despite its great advantages in bidirectional communication, Kharraz et al. [15] note that Websocket adoption is low, with HTTP polling remaining "significantly more common than WebSockets". Their studies uncovered that from the most popular websites, according to Tranco Top 1M, approximately 6% use Websockets and 15% use HTTP Polling.

Another interesting point raised by Kharraz et al. is the fact that Websocket, as a more complex and less well-established protocol, is usually ill-implemented — security-wise. They suggest improvements to the protocol, to its documentation, and a general push for responsible usage of this alternative to HTTP Polling which can have significative bandwidth and power savings.

The Webflux documentation puts forward that "restrictive proxies" outside of the programmer's control may close Websocket connections because they perceive them as idle. Based on this argument, they propose that shifting from HTTP (polling) to Websockets is something that should be more easily done in-doors, where firewalls are controllable.

Ben Hale, the creator of R2DBC, said in an Oracle conference[2] that one of the advantages of WebSockets over HTTP is the possibility of bidirectional communication, which enables full-fledged reactive subscriptions, including support for back-pressure.

It should be noted, however, that Kharraz et al. focus on WebSockets when used for communication between the browser and the web server. While this use case is important, and possibly the one more easily thought of, there are other use cases, such as inter-systems communication, which

[1] https://datatracker.ietf.org/doc/html/rfc6455

[2] https://www.youtube.com/watch?v=WVnAbv65uCU

is the scenario that konkconsulting has been using for WebSockets. However, we should note that konkconsulting uses Websocket in a request-response structure, not unlike HTTP. For this reason, this is what we will be focusing on.

To avoid trying to use a library that is not adequate because it is from outside the Spring ecosystem (as we did before), the first place we looked for a way to implement Websocket functionally was within the ecosystem it self.

The documentation has a chapter about Websocket, and another about RSocket. RSocket is a "binary protocol for use on byte stream transports" providing support "for Reactive Streams semantics between client-server, and server-server communication"[3].

While RSocket does sound interesting from a features point-of-view, its usage requires that both parties (in the communication) use it. Despite it being available in several programming languages and ecosystems, we still have difficulty embracing it as there is no interest in migrating all other konkconsulting's microservices. For this reason, we will focus our efforts on the simple Websocket implementation available on Webflux. It consists in extending the `WebSocketHandler` class appropriately.

8.1 Service processor setup

Despite not being our objective, we started by implementing a *processor* (something that would receive an object, process it, and return an answer), since it can be easily implemented in a completely reactive way. However, one of the greatest challenges ahead of us is that this library is fully reactive, which makes usage of blocking libraries, such as `spring-data-jpa`, more complex. Listing 8.1 shows a handler. The remaining code blocks are shown in Appendix B.3.

```kotlin
import com.google.gson.Gson

class MainHandler : WebSocketHandler {
    private val gson = Gson()
    override fun handle(session: WebSocketSession): Mono<Void> {
        val out = session.receive().log().map {
            val pl = gson.fromJson(it.payloadAsText, MyPayload::class.java)
            if (!pl.processed && pl.number != null) {
                pl.number = sqrt(pl.number!!.toDouble()).toInt()
                pl.processed=true
            }
            session.textMessage(gson.toJson(pl))
        }
        return session.send(out)
    }
}

class MyPayload {
```

[3] `https://rsocket.io/`

```kotlin
    var processed: Boolean = false
    var number: Int? = null
}
```

Listing 8.1: Websocket handler receiving and processing a MyPayload object.

Despite the Websocket Protocol RFC not defining a format for the payload, it is common to use JSON. Thus, we used `Gson` by Google to serialise/deserialise the `MyPayload` Objects.

8.2 Doing a request to another server

Having successfully tested this processor with a simple graphical client, we moved on to the implementation of an HTTP Webflux handler that would to a Websocket request and wait for an answer, an then return that answer via HTTP. However, due to time constraints, we were unable to complete it. Regardless, Listing 8.2 shows the code which opens the Websocket and sends the request, only missing the response handling.

```kotlin
val gson = Gson()

@GetMapping("/ws")
suspend fun ws0(): Flow<Void> {
    val client = ReactorNettyWebSocketClient()
    val url = URI("ws://localhost:8081/")
    client.execute(url) { session ->
        val msg = session.textMessage(MyPayload(49).asJson())
        session.send(Flux.just(msg))
    }
    return emptyFlow()
}

class MyPayload(var number: Int?) {
    var processed: Boolean = false

    fun asJson(): String {
        return gson.toJson(this)
    }
}
```

Listing 8.2: HTTP handler sending a request via Websocket. Receiving and processing the response is missing.

Summary

Our incursion in the Websocket reality was short, but concise. After our experience with HTTP requests, we considered that it would be a better approach to use the built-in tools. Despite not reaching our objective in full, it was clear that the functionality is present and in a reactive way. Better code examples and a more complete documentation would have helped us tremendously to finish this specific PoC.

Chapter 9

Spring MVC to Spring Webflux: An *attempt* at a smooth transition

konkconsulting's present solution runs on Spring MVC. Having realised that Spring Webflux *is* the best environment currently available for reactive applications in Java/Kotlin, it became relevant to study how to transition from Spring MVC to Spring Reactive. We could do it by rewriting the application from scratch, or incrementally, module by module. Regardless of this choice, transitioning a system under development can be very complex, considering the fact that teams are actively developing, closing issues, and have a schedule to respect. Nevertheless, it is still better than doing it after the development is finished or when the system is live. Moreover, it should be simpler to transition a system from a solution to another in the same ecosystem, as compared to changing ecosystems (e.g. Spring MVC to Ktor).

Due to the nature of reactive applications, a soft transition is undesired as blocking code can block the whole application if not handled carefully. Moreover, it is not recommended to parallelly use the servlet API and the Netty API — Spring MVC and Spring Webflux core functionalities, respectively — despite it being allegedly possible[1].

This leaves the absolute transition, which is actually the most recommended in many webinars about this subject. This means that no blocking code/libraries can stay working like before — and preferably they're removed altogether. This focus on transitioning from Spring MVC — as opposed to an approach focused on newcomers which would help those who want a fresh start — is probably the one that targets the biggest audience, but makes the approach to Webflux require an approach to Spring MVC first, which is not ideal.

We should note that during the course of this dissertation there were two[2,3] webinars presented by *Jetbrains*[4] about Spring Webflux. In the last one, we asked several questions, which did not receive an answer. The questions asked were the following:

[1]https://github.com/bigpuritz/netty-servlet-bridge
[2]https://www.youtube.com/watch?v=FcwR34DFqIc
[3]https://www.youtube.com/watch?v=8-6Cd9YemOc
[4]https://www.youtube.com/channel/UCP7uiEZIqci43m22KDl0sNw

- How is support for relations when using R2DBC? Is it as robust as JPA?
- Do suspending functions support HTTP request cancellation via TCP FIN?

Previous chapters of this dissertation allow the reader to understand that these are paramount questions which could have been answered in a clear way, but they were not picked for answering.

9.1 Spring MVC

Spring MVC[5] is the primary building block in the Spring ecosystem. It is a framework following the Model—View-–Controller design pattern, which does not, however, oblige the developer to use it in that way, which makes it highly flexible.

Spring MVC is a request-driven framework based on blocking web servers. Before 2009, this meant that for each request received, a new thread was launched — or taken from a thread pool — by the underlying server. As previously explained, while threads enable lightweight concurrency, as the number of threads increases, so does the computational effort required to schedule them and the memory requirements necessary to accommodate their proliferation. For this reason, it is important to reduce the amount of time that threads take to finish their job — assuming that there is no control over the number of requests received, and, thus, the number of threads launched. However, since 2009, and Spring 3.0, it has been possible to code asynchronously, returning control of a thread to the server in periods of io-blocking or while waiting for a third-party thread. More recently, Spring MVC has come to support some reactive constructions: e.g. it is possible to have a Spring MVC controller return a Flux. The thread will be decoupled from the servlet container, reactively flushing the Flux output as it comes.

While these changes allow Spring to improve scalability, some internal server functionalities, such as `WebFilter` or `WebHandler` continue having a blocking nature. In a podcast, Rossen Stoyanchev[6] notes that there are many hidden processes that are blocking and can easily be overlooked, such as parsing the request parameters (a blocking operation which involves parsing the body of the request).

It is widely accepted that the thread/blocking scheme is a simple solution that has served us well for a long time, and that continues to serve us, but it is becoming dated due to the way we interact with other systems and the amount of information and sources that are used nowadays.

When using a thread to deal with a request, said thread may need to do things that are, can be, or must be, asynchronous, such as:

- calling databases
- calling REST services
- reading/writing from/to files

[5]https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html

[6]https://open.spotify.com/episode/4RGWIfpx4HhqiAsuieXOcz

The most common solution is using libraries that obfuscate these processes by allotting requests to threads in a thread pool, increasing the number of threads necessary to run the application.

Regarding web servers, Spring MVC comes embedded with Tomcat web server, but can also be used with Jetty or Undertow.

9.2 Spring Webflux

Spring Webflux[7] is a framework in the Spring ecosystem targeting reactive development, as per the Reactive Manifesto.

Unlike Spring MVC, is not based on a blocking structure, but on an event-loop. A simple way of looking at it, according to Rossen Stoyanchev, is comparing with Node.js. In Node.js, there are no thread pools, which imposes the asynchronous model on developers using it. This means that developers must solve their issues in an asynchronous way.

Stoyanchev defends that asynchronous, event-driven programming is a trend that should be addressed; it becomes relevant in situations with increased amounts of latency like calls to third-party services, which should be declared in a way that no unnecessary calls are made, while still respecting precedence of calls (e.g. the result of call A is necessary to make call B, but call C can be done concurrently with calls A and B).

Stoyanchev also notes that writing the response back to the client can be a slow process, depending of the ability of the client to receive the payload, the network bandwidth and the network stability. While blocking servlets will write and block until the complete payload can be flushed, asynchronous implementations can use a single thread to reply to many clients simultaneously.

Spring Webflux comes embedded with Netty web server, but can also be used with Jetty or Undertow. However, cancellation as presented in section 5.2.4 is only attainable with Netty.

9.3 Co-existence: Spring MVC & Webflux

Our investigation of the possibility of coexistence of Spring MVC and Spring Webflux started, naturally, at the Spring Boot Starter page, where we have successfully created a Spring application with both modules, which compiles and runs. A major incompatibility between the two modules should result in an error in the starter page, in our opinion, and the fact that it did not makes us believe that it is possible.

Initial tests revealed that, when launching a project with both Spring MVC and Spring Webflux, Tomcat will be used. This means using the servlet API and all the APIs that use it.

Stoyanchev advocates that while mixing blocking and asynchronous code is usually a hard thing to do, it is possible. However, one must be careful not to use blocking code in the same

[7]`https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html`

thread where non-blocking code is running. This makes it necessary to continue having thread-pools in order to run blocking code, which it not ideal in a reactive context. However, with the help of Kotlin constructs such as Coroutines and Dispatchers (vide section 2.1.2), this should not be a problem.

9.4 konkconsulting's Dilemma

Everyone makes transitioning from Spring MVC to Spring Webflux look simple: Kotlin webinars, Baeldung blog posts. The attentive reader may recall the quote at the beginning of this dissertation. While their message seems authentic, it does not take into account the way code evolves and is shaped to the underlying platforms. In konkconsulting's case, the first problem arose when taking out Tomcat off the application: there are several places where the servlet APIs are used, which are not natively provided by Netty. While the use of a Netty servlet container would probably make it possible to integrate these two parts, konkconsulting has decided to proceed with a full transition.

With this situation on our hands, we have delved into the lengthy task of changing the project so that it can run on Webflux. There were several complications, which we explain in the next subsections.

9.5 Filtering

Spring Boot offers filters, a tool that allows requests to be filtered and managed. For example, a filter can analyze the headers of the requests, and issue a redirect based on a specific value, or a filter can make it so that requests over a certain threshold (in terms of requests per amount of time) can be stopped without ever launching the controller, an operation which must inject dependencies and may be heavy.

The way these filters are implemented in Spring MVC is totally different from the way they are implemented in Spring Webflux. As konkconsulting is using filters, they have to be redone. However, Webinars advocating for developers to switch to Webflux do not mention filters whatsoever.

9.6 Websockets via STOMP

Websockets in Spring MVC are ideally implemented using STOMP[8], the *simple text-orientated messaging protocol*. STOMP is a specification defining the way messages are to be structured, and works even when using other communication channels, such as telnet. STOMP implementations are available for many platforms, such as Java, Python, JavaScript and PHP. konkconsulting was using STOMP, however, due to its reactive nature, Webflux does not support the STOMP implementation that Spring MVC provides. In April 2018, a concerned user asked about this on GitHub and a Spring contributor replied that "There is a general intent to provide higher-level, messaging

[8] `https://stomp.github.io/`

support aligned with WebFlux" but there were not yet any solutions. However, he mentioned efforts to add support for gRPC and RSocket. The latter is already integrated and well supported. However, it is incompatible with STOMP. This means that all Websocket functionality will have to be re-implemented. In the context of this dissertation, no implementation was done. Instead, these functionalities were removed because they were not being used extensively.

9.7 OAuth2

Spring natively supports OAuth2, thus making it easy to setup authentication with third party providers. However, the programmatic interfaces used to access the user data differ significantly, breaking authentication until it is adjusted to the Webflux way.

In the case of konkconsulting, because they are using an internal Identity Provider (IdP) — as opposed to a public, generic one, like Facebook or GitHub —, further configuration is necessary. Thankfully, this configuration need not be changed in order to use the Webflux version of OAuth2 on Spring.

9.8 CORS

CORS (Cross-Origin Resource Sharing) is usually known to be something that breaks applications, but that is extremely important for preventing attacks. When migrating konkconsulting's product, adjusting the CORS configuration as specified on Baeldung[9] did not work, even though we asserted that those lines were being executed. We only managed to get a working application when copying code from an online example on GitHub[10], which — as a development method — is far from ideal.

9.9 Spring-security

Spring MVC APIs such as `SecurityContextHolder` or `AuthenticationManager` are blocking. While this is acceptable in a thread-per-request reality, it cannot be in an event-loop reality like Spring Webflux.

For this reason, such APIs have reactive counterparts: `ReactiveSecurityContextHolder` and `ReactiveAuthenticationManager`, respectively.

These new, Reactive, APIs implement different interfaces and must thus be used in different ways: their reactive nature forces the developers to use reactive programming, as any kind of reactor blocking code will throw an exception and the request will not be fulfilled.

[9] `https://www.baeldung.com/spring-webflux-cors` section 3. *Enabling CORS on the Global Configuration*

[10] `https://github.com/inoutch/inochat/blob/master/server/src/main/kotlin/sample/security/SecurityConfig.kt#L46`

```
java.lang.IllegalStateException: block()/blockFirst()/blockLast() are blocking,
    which is not supported in thread reactor-http-nio-4
```

Listing 9.1: Example of error when using blocking calls in webflux configuration classes.

We find errors like the one in Listing 9.1 harsh on the developer. However, it is absolutely understandable that developers be stopped, at all costs, from blocking the event-loop. This forces good coding practices and helps Webflux not receive a bad reputation due to low performance caused by blocking code.

Rob Winch, Project Lead for Spring Security, did a presentation[11] in 2018 regarding the use of Spring Security in reactive applications. However, not all the topics were covered as this is a rather extensive framework.

9.9.1 SecurityContextHolder

In Spring MVC, the `SecurityContextHolder` uses the `ThreadLocal` data structure in order to store the authentication variables. When using reactive programming, it becomes necessary to let go of this data structure. The reasons for this are two-fold:

- There is no telling if the setting and posterior retrieving of these values takes place in the same thread. Because `ThreadLocal` is attached to a single thread, any thread switch would result in that content no longer being accessible.
- On another hand, there may be several reactor pipelines running on the same thread, causing `ThreadLocal` *collisions*.

For these reasons, the reactive version of `SecurityContextHolder` must be used. The *problem* with `ReactiveSecurityContextHolder` is that it is inaccessible outside of a Reactive context. This means that the security context set in a WebFilter, for example, is not available in the blocking libraries that konkconsulting decided to use for managing database filters, which need to know which user is signed in. Because the database layer is blocking, we could not integrate the new APIs.

9.9.2 @Secured

Spring Security makes use of the `@Secured` annotation. This is used to make sure that only some *users* with certain *roles* can access certain endpoints. In spring WebFlux, this is not possible, and other annotations need to be used, such as `@PreAuthorize` or `@PostAuthorize`. These were already present in Spring MVC. The two annotations in Listing 9.2 have the same behaviour[12].

```
@PreAuthorize("hasRole('ROLE_USER')")
```

[11] https://www.youtube.com/watch?v=YcAufUtfm44
[12] https://stackoverflow.com/questions/3785706/whats-the-difference-between-secured-and-preauthorize-in-spring-security-3

```
public void create(Contact contact) {}

@Secured("ROLE_USER")
public void create(Contact contact) {}
```

Listing 9.2: Two Spring Security annotations showing the same behaviour **in Spring MVC**.

In light of this, migrating from the old `@Secured` to the new `@PreAuthorize` should not be complicated. However, a member of the community migrating a Spring MVC application to Spring Webflux created an issue on Github asking about the possibility of using these annotations[13] and the response from Rob Winch was that **the usage of said annotations requires that all methods return** either a `Mono` or a `Flux`: "this is the way Reactor Context propagates the `SecurityContext`." — he says.

We tested prepending a handler which returned a *Flow* with a `PreAuthorize` annotation, which resulted in requests to that endpoint returning an error 500, with this message on the console:

```
java.lang.IllegalStateException: The returnType class java.lang.Object on public
    java.lang.Object ...HttpController.rest5(kotlin.coroutines.Continuation) must
    return an instance of org.reactivestreams.Publisher (i.e. Mono / Flux) in order
     to support Reactor Context
at org.springframework.util.Assert.state(Assert.java:97) ~[spring-core-5.3.5.jar:5
    .3.5]
...
```

The same happened when testing on a function returning a *String*. This annotation only worked when applied to a handler returning a Publisher, as per the message shown above. However, we should note that merely returning a `Mono.just(ret)` where `ret` is the original return value makes the annotations work. This is only acceptable if if is not necessary to pass context to the handler's body. However, if the handler uses a reactive library that requires the context, then the reactive variable (mono/flow) should be returned directly, otherwise the context flow is broken.

9.9.3 User Roles

As previously explained, `@PreAuthorize` is an annotation which interprets Spring Expression Language (SpEL)[14]. SpEL is explained in section 2.7. The `hasRole(role)` function is the direct translation of the old `@Secured` annotation used by konkconsulting, but the switch to Spring Webflux and the different OAuth2implementation created barriers to the usage of `hasRole(role)`.

`hasRole(role)` works by checking if the passed role is in the `authorities` set, which is a member of the `authentication` object, a part of the `SecurityContext` we analysed earlier. In the specific case of konkconsulting, roles pertaining to an authenticated user's session are gathered from two main sources: the **OAuth2 IdP** and the **database**.

[13] https://github.com/spring-projects/spring-security/issues/5103
[14] https://docs.spring.io/spring-framework/docs/3.0.x/reference/expressions.html

Regarding the OAuth2 IdP's roles, they were being stored under an attribute of the principal object. We had to manually copy them to the authorities object, as shown in Listing 9.3.

```
((sc.authentication.principal as OAuth2User).attributes["authorities"] as JSONArray
    ).forEach {
    updatedAuthorities.add(SimpleGrantedAuthority("ROLE_$it"))
}
sc.authentication = OAuth2AuthenticationToken(
   sc.authentication.principal as OAuth2User,
   updatedAuthorities,
   (sc.authentication as OAuth2AuthenticationToken).authorizedClientRegistrationId)
ReactiveSecurityContextHolder.withSecurityContext(Mono.just(sc!!))
```

Listing 9.3: Adding OAuth IdP roles to the updatedAuthorities set.

As for the database roles, there is a complication which makes the process undesirably, but unavoidably, blocking. For some reason internal to Spring Webflux, — which we have not investigated — Spring WebFilters, the way used to do the aforementioned process of reassigning roles, cannot run on a suspending function. This means that there is no way for Kotlin Coroutines to be used in its context. Because konkconsulting is using Hibernate JPA, a blocking way of doing database calls, there can be problems when authenticating many users simultaneously. Nevertheless, Listing 9.4 shows the code used.

```
val roles = rolesRepository.findUserRoles(userId)
roles.forEach{updatedAuthorities.add(SimpleGrantedAuthority("ROLE_$it"))}
sc.authentication = OAuth2AuthenticationToken(
   sc.authentication.principal as OAuth2User,
   updatedAuthorities,
   (sc.authentication as OAuth2AuthenticationToken).authorizedClientRegistrationId)
ReactiveSecurityContextHolder.withSecurityContext(Mono.just(sc!!))
```

Listing 9.4: Adding database roles to the updatedAuthorities set.

Both code snippets can be combined. The full code is shown in Appendix B.2.

Regarding the database access, we believe it should be possible to use a reactive solution, such as R2DBC (vide section 6.2), which could use the reactive pipeline to make the SQL call non-blocking. However, this should only be equated if the workload reaches a level in which making this call in a blocking manner starts to noticeably degrade the user experience.

9.10 Task Scheduling

konkconsulting makes extensive use of Task Scheduling. Scheduled tasks are used to maintain business constraints, keep databases clean and sync with external systems. konkconsulting uses the Quartz library[15] to manage tasks, which they originally selected as it supports multi-triggered tasks, and they felt it was a robust solution. Moreover, Quartz has a special integration with Spring provided by the Spring team.

This special integration is something that we believed to be a good sign because it should imply that it would be supported when using Spring Webflux. In truth, this was not the case. Trials to integrate this library were unfruitful. Our sole attempt was to use the Spring integration — not the native library — as we believed it to be the best course of action. Baeldung features a blog post[16] explaining how this should be done, which we followed — without success.

We should note that Quartz uses a thread-pool to accommodate Jobs. While this is the standard way of dealing with tasks, it is contrary to the reactive form. Having realised this and faced with difficulties integrating Quartz, we decided to move on and look for alternatives.

Spring natively provides a framework for *Task Execution and Scheduling*[17] (TES). Naturally, this is available when using Spring Webflux. However, we should note that, like with Quartz, support for reactive execution is lacking, with the framework being entirely based on a thread pool that can accommodate tasks. We looked into the possibility of using this with Kotlin Coroutines, with no success. The foundation for Kotlin CR — having a function defined as suspending — is not supported by TES. Execution of these functions fails silently, resulting in Spring crashing. This means that previous efforts regarding cancellation while using Kotlin CR cannot be reused in this scenario (using TES, that is).

Regardless of these complications, we were still able to achieve some cancellation using TES. StackOverflow user *Krishna Prasad* explains that cancelling TES tasks can be done using Spring's `ScheduledAnnotationBeanPostProcessor`[18]. This cancellation is done internally by interrupting the running thread (via Java Thread.interrupt()). In Java, interrupts[19] are a sign sent to a thread signalling that it should stop its current execution. Threads receiving an interrupt are not immediately stopped. Functions which throw `InterruptedExceptions`, such as Thread.sleep(), will throw this exception which, if uncaught, terminates the Thread's execution — exceptions can be caught using a try/catch block. If a thread does not call such functions regularly, it should (manually) verify if the Thread has not been cancelled by checking if `Thread.interrupted()` is false.

An example of a scheduled function in TES supporting cancellation is shown in Listing 9.5.

```kotlin
// every 5 seconds, starting 0.5s after scheduler is launched
@Scheduled(fixedDelay = 5000, initialDelay = 500)
fun deleteFamousPosts() {
    var canceled = false // initialize cancelled as false.
    val famous = postRepository.findFamous(5) // get famous posts to be deleted
```

[15] http://www.quartz-scheduler.org/
[16] https://www.baeldung.com/spring-quartz-schedule
[17] https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/scheduling.html
[18] https://stackoverflow.com/a/50216003/1469991
[19] https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html

```
 6
 7      // check if thread was cancelled in the mean time.
 8      if(Thread.currentThread().isInterrupted)
 9          canceled=true
10
11      // perhaps do something else regardless
12
13      if (canceled) {
14          println("Cancelled.") // not continuing
15          // roll back changes if necessary
16          return
17      }
18      // not cancelled. Continuing
19      postRepository.deleteAll(famous)
20  }
```

Listing 9.5: TES function supporting cancellation mid-execution. The function fetches posts from the database using a specific criteria. After selecting them, it checks the cancellation status (at a safe-point). If it has not been cancelled, it proceeds with the deletion. Note that no roll-back is necessary, so the function can simply return. It could do some rollback on line 15 if necessary.

9.11 Summary

Migrating **a real application** from Spring MVC to Spring Webflux is a complex process. Migrating a simple chat application is not. People from Kotlin, Spring, and other Webflux-friendly projects have not been totally clear about this in the past years, as they have pushed for a narrative of simplicity and ease regarding this subject, with total disregard for the fact that applications done in Spring — used mostly in companies — are done in that manner due to the ease of integration with other Java solutions (or simply because the company is familiar with basic Java libraries/-standards such as JPA). Considering that these other solutions probably do not follow a reactive approach, which makes integration complicated, if not impossible, it is our conviction that those reports of simplicity are exaggerated.

Due to the radical changes between the Imperative and the Reactive model, *migration* is probably not the answer to most problems. A new implementation will undoubtedly yield much cleaner results, both in terms of code and functionality. However, this makes migration a much grimmer outlook and a not-so-attractive prospect, and we understand that. We merely do not believe that the present narrative can produce any long-lasting fruits. Instead, it makes people curious about the new thing — Webflux —, but they soon give up after a while, which is probably why these webinars are so similar every time and most questions avoided. It is our opinion that the first thing one should do, before even thinking of migrating an application to Spring Webflux, is deeply and fully assimilate the underlying knowledge required to build a robust reactive application — and do it. Only after being well versed in this technology does it make sense to continue the effort.

Nevertheless, this exercise has allowed us to understand and document many aspects, from limitations of mixing reactive and blocking code to things that had to be changed or even constructions that can work "as is". This is the final exercise, the place where we transition from PoCs to an actual implementation (or try, at least). The place where we can obtain a real, concrete conclusion about what has become the most resounding question in this dissertation: – How reactive can konkconsulting go? The answer: – up to the database.

Chapter 10

Results

In this chapter, we present the scientific and analytic side of the tests that we ran on several approaches, namely **SQL** and **Third-party HTTP Requests**, in terms of performance and bandwidth. Alongside with the results, other observations and the results' interpretation are presented for easier reading and understanding.

We consider studying these results highly important for two main reasons: firstly, to select between alternatives which seem to be equivalent in theory or which, despite not being equivalent, arose our interest for some specific reason, and secondly, to validate our choices, i.e., to ensure that empirical choices regarding what is the best solution do hold when tested in a real scenario — and understand why other solutions should not be selected.

Results are mainly based on two metrics, **run time** (for a batch of repetitions of the same task), and **thread usage**.

We used *ab - Apache HTTP server benchmarking tool*[1] for launching the requests and collecting the run time, in a detailed way which shows the evolution of the completion of jobs with respect to time.

Considering that we are dealing with JVM applications, all thread-usage readings were taken using *VisualVM*[2], which can show the CPU time for each thread, among other features. Having exported each run report as CSV, we treated those files using a spreadsheet application, which rendered the descriptive statistics that we present.

Naturally, all tests were run in the same environment. Node.JS and Java/Kotlin solutions ran on machine 1, and the PostgreSQL server ran on machine 2. These machines are part of the same LAN (Local Area Network), with an average ping of 0.27ms, and a Gigabit connection. Machine specifications are listed in Table 10.1.

[1] https://httpd.apache.org/docs/2.4/programs/ab.html
[2] https://visualvm.github.io/

Machine	Type	CPUs/Cores	RAM	Disk
1	Desktop/Work PC	1x AMD Ryzen™ 7 3700X@3.6GHz (8 cores/16 threads)	32GB	SSD
2	HPE ProLiant DL360p Gen8	**2x** Intel® Xeon® Processor E5-2650 v2 @ 2.60GHz (8 cores/16 threads each)	64GB	SSD

Table 10.1: Machine specifications for tests run.

10.1 SQL

Comparing loading times in SQL seems unnecessary. The reactive solution (R2DBC) cannot be fairly compared because it offers no support for relations — something we will not let go of — and JPA is our choice. Regardless, we feel it necessary to understand why JPA yielded such terrible results when compared to Ebean, and also to explain how we ended up making JPA better than Ebean in terms of performance.

10.1.1 Ebean

We ran some tests (shown in Figure 10.1) regarding the performance of the Ebean solution when fetching data from a database. Test Data A.1 was used. No JPA tweaking annotations (@Fetch, @Cacheable, FetchType) were used. No special wrappers were used, as shown in Listing 10.1. Results were consistent throughout several runs. The connection pool size was 16 as Spring is not offloading the requests to another thread, meaning that the `reactor-http-nio-*` threads, also called the event-loop threads, did the requests to the database.

```kotlin
@GetMapping("/", produces = [MediaType.APPLICATION_JSON_VALUE])
suspend fun getAllPosts(): List<Post> {
    return QPost().findList()
}
```

Listing 10.1: Simple Eben Query

```
Concurrency Level:      600
Time taken for tests:   5.061 seconds
Complete requests:      600
Failed requests:        0
Total transferred:      37956000 bytes
HTML transferred:       37911600 bytes
Requests per second:    118.55 [#/sec] (mean)
Time per request:       5061.276 [ms] (mean)
Time per request:       8.435 [ms] (mean, across all concurrent requests)
Transfer rate:          7323.53 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       13   38  14.5     38      63
Processing:   144 2522 1470.1   2522    4985
Waiting:      144 2521 1470.1   2522    4985
Total:        207 2560 1457.7   2561    5000
```

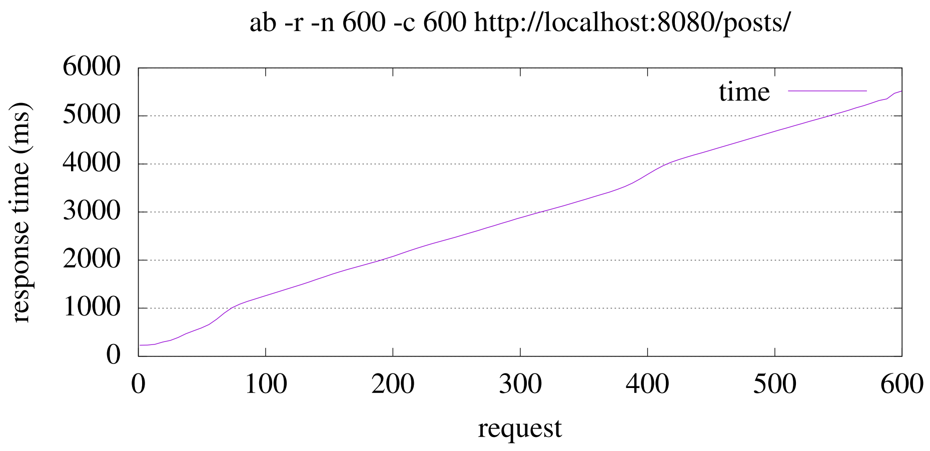


Figure 10.1: AB performance tests of the Ebean solution using Test Data A.1 and no wrapping.

An attentive reader knows that blocking the event-loop threads is *the mistake* that Reactive aficionados warn us about. Let us thus undo that mistake. Listing 10.2 shows an implementation using Kotlin Coroutines which enables the usage of a manually-defined thread-pool. This thread pool has 64 threads, the same as in JPA, which ensures a fair comparison. Figure 10.2 and Table 10.2 show the results of this execution.

```kotlin
val dispatcher = Executors.newFixedThreadPool(64).asCoroutineDispatcher()

@GetMapping("/v2", produces = [MediaType.APPLICATION_JSON_VALUE])
suspend fun getAllPostsV2(): List<Post> {
    val deferred = GlobalScope.async (dispatcher) {
        return@async QPost().orderBy("id").findList()
    }
    try {
        val n = deferred.await()
        return n
    } catch (e: Exception) {
        deferred.cancel()
    }
    return emptyList()
}
```

Listing 10.2: Simple Eben Query inside Async block

```
Concurrency Level:      600
Time taken for tests:   3.348 seconds
Complete requests:      600
Failed requests:        0
Total transferred:      37956000 bytes
HTML transferred:       37911600 bytes
Requests per second:    179.19 [#/sec] (mean)
Time per request:       3348.452 [ms] (mean)
Time per request:       5.581 [ms] (mean, across all concurrent requests)
Transfer rate:          11069.71 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       18   44  15.1     45      70
Processing:   433 1795 907.0   1722    3262
Waiting:      432 1794 907.8   1721    3262
Total:        503 1839 892.9   1765    3292
```

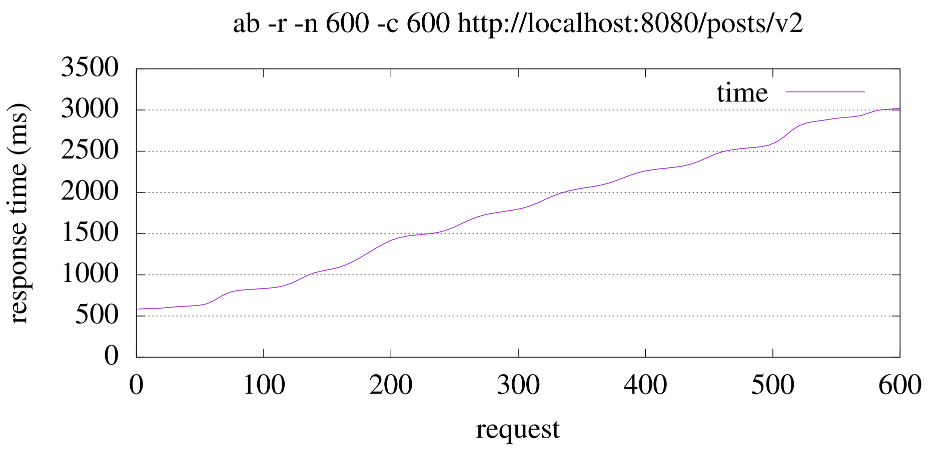


Figure 10.2: AB performance tests of the Ebean solution using Test Data A.1 and wrapping with async.

Summary	**reactor-http**	**pool-1-thread**
Mean	19.775	3092.21875
Standard Error	8.843648851	14.73479962
Median	0	3137
Mode	0	3152
Standard Deviation	35.3745954	117.878397
Range	79.1	698
Minimum	0	2544
Maximum	79.1	3242
Sum	316.4	197902
Count	16	64

Table 10.2: Threads CPU time (ms) statistical analysis grouped by thread type — Ebean SQL requests inside async block

Notice the sum of time used by the `reactor-http-*` threads. Considering we responded to 600 requests, this means each request took approximately $316.4/600 = 0.527$ ms from the event-loop, which is understandable considering that only the query is done in the thread pool, i.e., the event-loop threads are responsible for assigning those jobs to the thread pool and for collecting the result when it is available. Only 4 (four) of the available 16 event-loop threads ran, as they were more than enough to handle this load.

In total, our 64 threads in `pool-1-thread` ran for 197.902 seconds, which makes sense considering the running time was approximately 3.3 seconds ($3.348 * 64 = 214 < 197.902$).

Recapping, using Ebean we did 600 requests to the database, using two different scenarios:

- using the event-loop threads. This took us 5s.
- using a 64-thread thread-pool. This took us 3.3s.

Clearly using the thread-pool yielded better results, but possibly at too high a cost. Perhaps the database resources were maxed out because of the way Ebean is querying it. In the next section, we will focus on how queries can be tweaked so that relations, for instance, can be fetched in a different, quicker way.

10.1.2 JPA

Preliminary tests with `spring-data-jpa` yielded terrible results in terms of fetch time, when compared with other solutions, as shown in Figure 10.3.

```
Concurrency Level:      600
Time taken for tests:   40.308 seconds
Complete requests:      600
Failed requests:        0
Total transferred:      38083800 bytes
HTML transferred:       37911600 bytes
Requests per second:    14.89 [#/sec] (mean)
Time per request:       40308.062 [ms] (mean)
Time per request:       67.180 [ms] (mean, across all concurrent requests)
Transfer rate:          922.67 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       19   43  13.7     43      67
Processing:   847 19967 11306.0  19741   40225
Waiting:      846 19967 11306.2  19741   40224
Total:        910 20010 11293.9  19794   40248
```

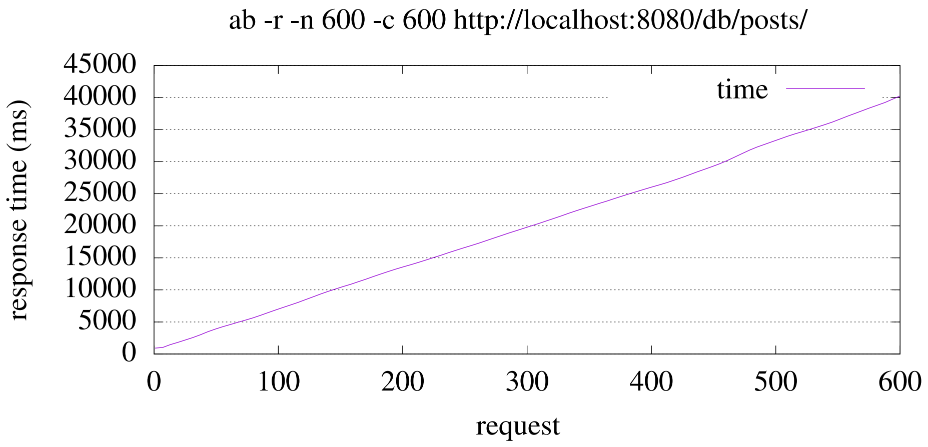


Figure 10.3: AB performance tests of the JPA solution using Test Data A.1.

We do not provide thread data because our collection tool seemed to be making execution much slower. However, we can provide some information that is consistent regardless of execution time: `boundedElastic` threads are used for executing the requests, unlike when we used Ebean. Moreover, CPU time was very low, which makes us believe there is some some mechanism magically managing JPA queries under the hood, without any explicit user configuration.

First trials with `@Cacheable` rendered no result. This is probably because no caching mechanism was available.

In order to learn why Hibernate was taking so long to execute our requests, we tweaked the `application.yaml` file and set the logging level to `DEBUG`.

The first thing noticed was the fact that for each `findPost()` request, **1000** queries were being generated, individually fetching each Post's tags. By comparison, Ebean was generating **100** queries.

@BatchSize was the first setting that yielded visible results. By setting @Batchsize to 10, for example, only 100 extra queries were made, and by setting this to 500, only two were needed, which remarkably cut load times.

Ultimately, it was setting @Fetch to `FetchMode.SUBSELECT` that reduced the execution of the request to two SQL queries. This resulted in the loading times shown in Figure 10.4 (and thread times shown in Table 10.3), as opposed to the previous results show in Figure 10.3.

Summary	**reactor-http**	**boundedElastic**
Mean	29	24.31
Standard Error	29	6.91236514
Median	0	0
Mode	0	0
Standard Deviation	116	87.43527144
Range	464	383
Minimum	0	0
Maximum	464	383
Sum	464	3889.6
Count	16	160

Table 10.3: Threads CPU time (ms) statistical analysis grouped by thread type — JPA SQL requests with @Fetch Mode.

```
Concurrency Level:      600
Time taken for tests:   1.906 seconds
Complete requests:      600
Failed requests:        0
Total transferred:      38083800 bytes
HTML transferred:       37911600 bytes
Requests per second:    314.72 [#/sec] (mean)
Time per request:       1906.432 [ms] (mean)
Time per request:       3.177 [ms] (mean, across all concurrent requests)
Transfer rate:          19508.28 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       17   42  13.8     42      65
Processing:    26  919 535.5    879    1828
Waiting:       25  919 535.4    879    1828
Total:         89  961 522.6    931    1856
```

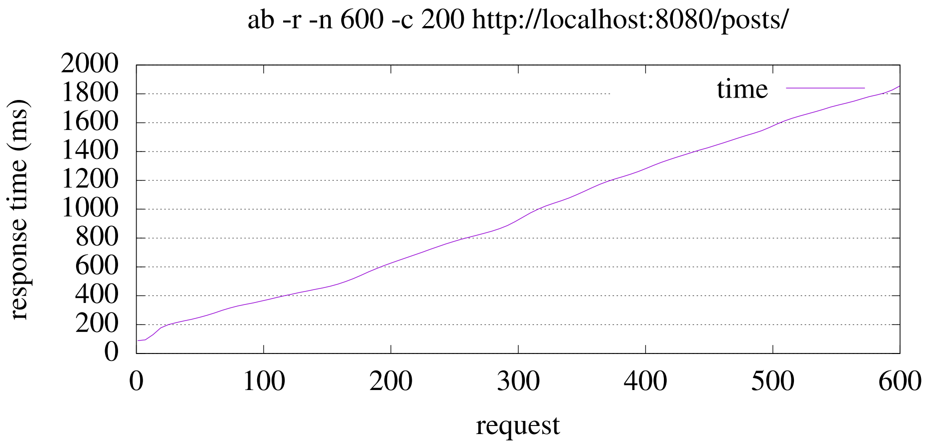


Figure 10.4: AB performance tests of the JPA solution using Test Data A.1 and loading Tags with special @Fetch.

Summary

Table 10.4 shows a comparison between the three trials we ran, where we evidentiate the difference in terms of loading times and CPU times (when available). Notice the fact that *Improved JPA* presents a very low CPU time, as previously explained. We are overall satisfied with our choice: JPA did indeed yield the best results. Moreover, it is the most compatible solution with Spring Webflux — even though it is not supposed to be used, a mitigation of its side-effects has been put into place (which we have been unable to understand fully) and this solves the blocking problem. We did not engage in tweak trials using Ebean considering that our core motivation for the current trials was the huge performance difference between Ebean and untweaked JPA. As we have overcome this difference — and even outperformed what was initially the best loading time — we see little point in further exploring Ebean, especially considering that we have no intention of using it. It is worth noting that Ebean is a great solution for those who are not interested in tweaking their models, as Ebean does a decent job out of the box, and also for those who are using a blocking, servlet API, and who are comfortable with its way of querying, or are willing to learn it.

Run	**Loading Time (s)**	**CPU time (s)**	**# requests**
Ebean	3.348	197.902	600
Naive JPA	40.308	n/a	600
Improved JPA	1.906	3.889	600

Table 10.4: Comparison between SQL solutions. CPU time is the sum of the worker threads for the requests.

10.2 Third-party HTTP Requests

Comparing loading times in HTTP third-party requests is paramount. We have the opportunity to put a reactive solution and a blocking solution to the same test, and finally get values with which we can compare both paradigms.

10.2.1 HttpURLConnection

This was the first method that we tried. We ran some tests using Apache Bench, shown in Figure 10.5 and VisualVM, shown in Table 10.5.

```
Concurrency Level:        1000
Time taken for tests:     7.246 seconds
Complete requests:        1000
Failed requests:          0
Total transferred:        25807000 bytes
HTML transferred:         25512000 bytes
Requests per second:      138.00 [#/sec] (mean)
Time per request:         7246.479 [ms] (mean)
Time per request:         7.246 [ms] (mean, across all concurrent requests)
Transfer rate:            3477.85 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        30   71  22.5     72     108
Processing:   1008 3690 1826.5   4024    7110
Waiting:      1005 3687 1828.6   4023    7109
Total:        1107 3760 1804.3   4092    7142
```

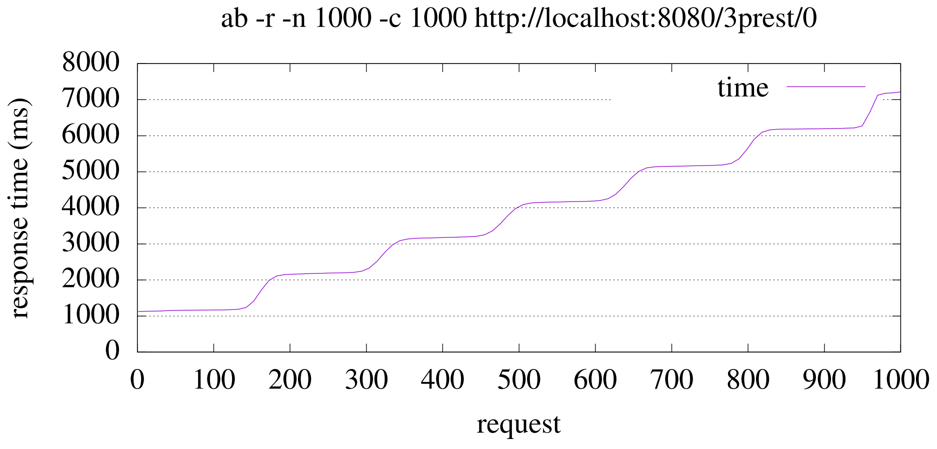


Figure 10.5: AB performance tests of the HttpURLConnection solution.

There is a lot of data to analyse. Let us start with the most obvious things:

- There are 16 `reactor-http-nio-*` threads, because our test environment has 16 cores.

	reactor-http-nio-*	**boundedElastic-***
Mean	287.5625	6043.275
Standard Error	53.77762614	26.77390478
Median	375	5828
Mode	375	5793
Standard Deviation	215.1105045	338.6660839
Range	613	1011
Minimum	0	5793
Maximum	613	6804
Sum	4601	966924
Count	16	160

Table 10.5: Threads CPU time (ms) statistical analysis grouped by thread type — HttpURLConnection

- There are 160 `boundedElastic-*` threads. It is not clear why this is. The documentation does not specify the maximum number of these threads.
- In the request chart, we can see steps, which correspond to each "wave" of requests being handled by the thread pool. This happens because all requests take approximately the same time.
- The sum of the execution time for `boundedElastic-*` threads is smaller than 1000000 ms (1000 requests * 1000 ms delay time for each response from the API). This should not be, and we think that either VisualVM is not showing times correctly, or the requests are not being delayed for 1000 ms at the NodeJS server, but less. Nevertheless, the error is under 5% (3.31%), which is admissible for us.
- Standard deviation for `reactor-http-nio-*` threads is high. Analysis of the original data showed that five of threads did not run, which means the offloading to `boundedElastic-*` threads or adding jobs to this queue is fast, and that is good.

Table 10.5 shows boundedElastic Threads. Their utility is documented in Chapter 2.4.

10.2.2 HttpURLConnection in a Kotlin async

As done in SQL, it is possible to use any standard library and make it run in a thread pool. Our first attempt was running the call embedded in an async block, running on `Dispatchers.Default`. However, the results obtained were poor because the number of threads was very limited — 16, in our test environment. When compared with the 160 boundedElastic threads from the previous trial (10 times more), it made sense that the execution time would also be approximately 10 times higher.

So, we created a custom thread pool with 160 threads and ran the same tests for this scenario. The code is shown in Listing 10.3. Apache Bench results are shown in Figure 10.6 and VisualVM results are shown in Table 10.6.

```kotlin
val dispatcher = Executors.newFixedThreadPool(160).asCoroutineDispatcher()

@GetMapping("/1")
suspend fun rest1(): String {
    return GlobalScope.async(dispatcher) {
        val url = URL(url)
        try {
            val con = url.openConnection() as HttpURLConnection
            con.requestMethod = "GET"

            val input = BufferedReader(InputStreamReader(con.inputStream))
            var inputLine: String?
            val content = StringBuffer()
            while (input.readLine().also { inputLine = it } != null) {
                content.append(inputLine)
            }
            input.close()
            con.disconnect()
            return@async content.toString()
        }catch (e:Exception){
            return@async e.message ?: "error not described"
        }
    }.await()
}
```

Listing 10.3: Example of simple HTTP GET request in an async block

```
Concurrency Level:      1000
Time taken for tests:   7.633 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      25807000 bytes
HTML transferred:       25512000 bytes
Requests per second:    131.00 [#/sec] (mean)
Time per request:       7633.432 [ms] (mean)
Time per request:       7.633 [ms] (mean, across all concurrent requests)
Transfer rate:          3301.55 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       55   92  20.2     93     122
Processing:  1334 4064 1824.4   4394    7480
Waiting:     1325 4061 1825.3   4392    7480
Total:       1449 4156 1810.3   4483    7563
```

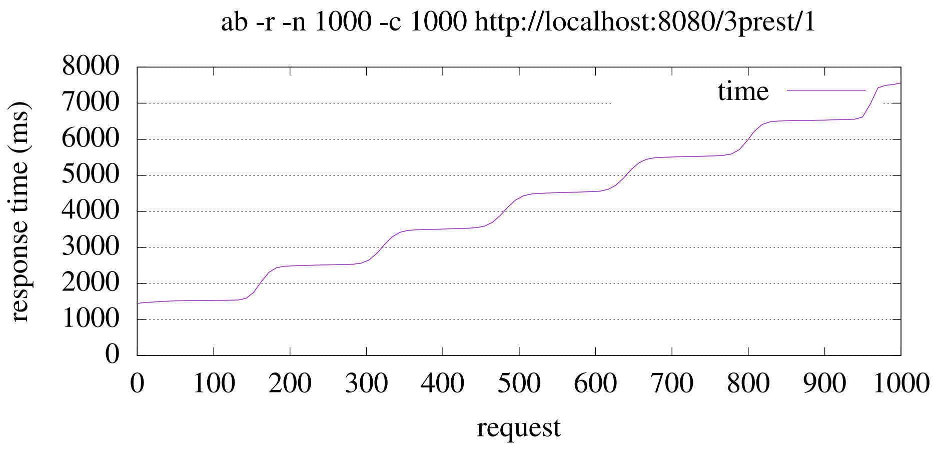


Figure 10.6: AB performance tests of the HttpURLConnection solution inside an async block.

Summary	**reactor-http-nio**	**pool-1-thread**	**boundedElastic**
Mean	66.36875	6316.5	5.983050847
Standard Error	35.89252675	31.27487689	1.817781339
Median	0	6089	0
Mode	0	6089	0
Standard Deviation	143.570107	395.5993781	19.74615966
Range	584	994	70.6
Minimum	0	6089	0
Maximum	584	7083	70.6
Sum	1061.9	1010640	706
Count	16	160	118

Table 10.6: Threads CPU time (ms) statistical analysis grouped by thread type — HttpURLConnection inside an async block.

As before, there are several possible observations to be done:

- There are 16 `reactor-http-nio-*` threads, because our test environment has 16 cores.
- There are 160 `pool-1-thread-*` threads, because that is the number of threads created in our thread-pool.
- In the request chart, we can see the same steps as before.
- The sum of the execution time for `pool-1-thread-*` threads is greater than 1.000.000 ms (1000 requests * 1000 ms delay time for each response from the API), as expected.
- Standard deviation for `reactor-http-nio-*` threads is still high. Analysis of the original data showed **9** of these threads did not run, which means the offloading to `boundedElastic-*` threads or adding jobs to this queue is fast, and that is good.
- Despite there being a custom thread-pool, Spring Webflux still creates the `boundedElastic-*` threads. This is not optimal as an increased number of threads means a bigger overhead for launching and scheduling them. Whether this overhead is or not significant is debatable.

10.2.3 WebClient

As explained in section 7.4, we proceded to try WebClient. Let us analyse the different results obtained this way. Apache Bench results are shown in Figure 10.7 and VisualVM results are shown in Table 10.7.

```
Concurrency Level:      1000
Time taken for tests:   2.554 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      18634000 bytes
HTML transferred:       18323000 bytes
Requests per second:    391.57 [#/sec] (mean)
Time per request:       2553.821 [ms] (mean)
Time per request:       2.554 [ms] (mean, across all concurrent requests)
Transfer rate:          7125.51 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       63  107  22.9    112     144
Processing:  1059 1809 495.1   2014    2393
Waiting:     1057 1797 500.6   2013    2393
Total:       1175 1916 484.8   2091    2527
```

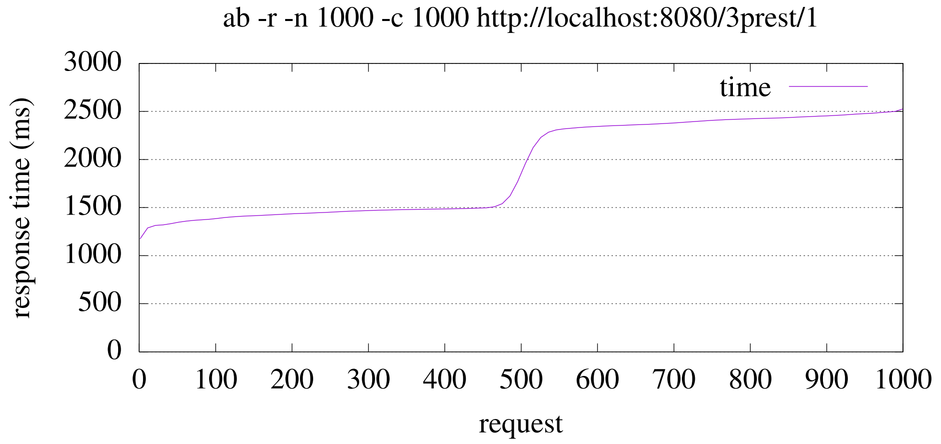


Figure 10.7: AB performance tests of the WebClient.

These results were surprising, let's see what changed:

- There are 16 `reactor-http-nio-*` threads, because our test environment has 16 cores, as before. They sum 3.85s of CPU Time.
- There are 94 boundedElastic threads, which sum 0.6s of CPU Time.

Summary	**reactor-http-nio**	**boundedElastic**
Mean	240.54375	6.404255319
Standard Error	25.3746231	1.587479549
Median	250	0
Mode	167	0
Standard Deviation	101.4984924	15.39118526
Range	400.3	43
Minimum	71.7	0
Maximum	472	43
Sum	3848.7	602
Count	16	94

Table 10.7: Threads CPU time (ms) statistical analysis grouped by thread type — WebClient http requests

- Unlike in previous runs, despite our application having to wait $1000 * 1s = 1000s$ for all requests, the sum of the running threads CPU time is approximately $3.86 + 0.6 = 3.93s$, i.e., **250** times less than before.
- Most of the `boundedElastic-*` threads did not execute anything, and were needlessly created.
- All requests were resolved in under 3 seconds, as opposed to approximately 8 seconds in previous trials.
- There is a step at 500 because of the way that WebClient is configured in Spring Webflux, which allows for a maximum of 500 connections per host, queuing additional requests.[3]
- Resolution of the first requests is not very quick, especially when compared with previous trials, which suggests that while the reactive approach is better for higher volumes, requests that should be as fast as possible and that are not done in volume would probably benefit from using blocking code.

[3] `https://stackoverflow.com/a/57688152`

Summary

Table 10.8 shows a comparison between all three studied solutions, comparing the loading time and the CPU time. Notice how Webclient presents radically different results as the waiting for each response (each with a delay of at least 1 second) is not done by blocking the calling thread, whereas in the blocking solutions (which use HttpURLConnection), the delay between the request being emitted and the response arriving is all spent on one CPU thread per request. NIO libraries use one thread to do all the waiting for several operations (HTTP requests, file access, etc.), but we were unable to find such thread. Perhaps this work is jumping threads, as one would expect in a reactive application. This is in line with the results we obtained, where several pipelines can run on the same thread, and several threads can share the same pipeline as it can jump threads.

Nevertheless, we should note that for the blocking solutions, the size of the selected thread-pool is paramount in affecting the results. For instance, a thread-pool consisting of 1000 threads for *HttpURLConnection w/ Async* (section 10.2.2) can take a mere 1.69 seconds to complete 1000 requests (1 request per thread), effectively becoming faster than the WebClient alternative. Naturally, the creation of many threads will become a slow operation, depending on the hardware where the program is running. We did not study what that number would be, as that was not the focus of our work, and also because it highly depends on the hardware.

Run	**Loading Time (s)**	**CPU time (s)**	**# requests**
HttpURLConnection	7.246	966.924	1000
HttpURLConnection w/ Async	7.633	1010.640	1000
WebClient	2.554	4.450	1000

Table 10.8: Comparison between SQL solutions. CPU time is the sum of the worker threads time for the requests, and does not mean that the CPU was doing intensive work.

Chapter 11

Conclusions & Future Work

We found it hard to dissociate these two subjects: future work is a direct consequence of our conclusions. However, we understand the need to separate them and have thus created two main sections, one for each.

11.1 Conclusions

Because this dissertation focuses on several subjects, it is hard to draw an overall conclusion, especially with such dissonant results. An excellent place to start may be analysing the Expected Results (section 4.3), our most naive expectations regarding this dissertation's progress and outcome.

Our Expected Results are simple: the cancellation of requests to the server, with subsequent cancellation of requests to the actions that may have launched. Optionally, but ultimately, we would like to implement these things in a real application. We had moderate success.

Our first effort, regarding cancellation of requests **to the web server**, was **successful**. It could have been better, however. We are not fully satisfied as only one solution was found — a solution with one string attached which is quite relevant: we *should* use reactive programming.

Trials with **SQL** were mixed. The reactive solution, using R2DBC, did support the cancellation of database requests, despite it not being clear how the safety (as introduced in section 2.6) of the requests was handled. However, as we did not stick with the reactive solution and preferred to keep JPA — with no support for cancellation and no support for context handling —, we consider our endeavours were **unsuccessful**.

Trials regarding **requests to third-party APIs**, i.e., from the server (as an HTTP client) to another server, were **successful**. The selected solution was well studied and yielded solid results. We only found one solution, but considering it is a consequence of a previous choice (Spring Webflux), this makes sense. Its reactive nature makes it easily cancelable, which is in line with our desire.

Trials regarding **Websocket** were **somewhat successful**. Due to time constraints, lack of documentation and lack of examples, we could not fully implement the functionality requested by konkconsulting. However, we could do part of it, which is positive.

All these results culminate in the ultimate objective of this dissertation: the migration of the application provided by konkconsulting. Considering the varying degrees of success, we could think that success on this matter was attainable: especially considering that the WebSocket functionality was dropped (which means our lack of complete success would not be a problem). This only left SQL as a problem. Unfortunately, our difficulties with SQL are a *showstopper*. It was not until nearing the end of this dissertation that we found a late 2018 session of Oracle Code by Ben Hale[1] where he says this:

> *There are still some barriers to using reactive everywhere. (...) What we see is that for reactive programming, there are certain places where it is a really really obvious programming model to adopt: a place like anytime you're to do coordination between multiple micro-services. This is a really obvious thing because today we have huge problems: you're running Tomcat, you have 200 threads, (...), you've just sort of occupied those threads making some sort of outbound calls to other microservices and now all of you sudden you've got a service that's got (...) 1% or 5% CPU usage and yet no more connections can be served. So reactive programming is a real obvious win here because now all of a sudden if we give up those threads while we're waiting for microservices to respond, we can do some more things, right? We can handle more, tens of thousands of simultaneous connections potentially. But if we take a look at some of those barriers, one of the first big ones is data access, so, right now, if you want to access a data store in some sort of asynchronous way we have MongoDB, Apache Cassandra, Redis, sort of the noSQL databases. We don't yet have (...) relational databases although there is this R2DBC project that is also one of my projects that is aiming to rectify that (...).*

If, for one, we would have liked to have found this earlier, it is also worth noting that the fact that we reached the same conclusions as Mr Hale, who has much experience in the field, hints toward the correctness of our discoveries and the worth of our efforts.

Further analysis of what was and what was not successful indicates what the strengths of Spring Webflux are. Mr Hale said that it is adequate for a microservices reality where no relational database access is required, or is available as one of many microservices. It is *possible* to use Spring Webflux without going fully reactive. However, the usage of regular (blocking) libraries must always have two aspects under consideration: how one must be careful not to block the event-loop and how the impossibility of passing context to those libraries will affect their usability. However, keeping these constraints in mind around the clock is so cumbersome that it becomes unfeasible to use anything that is not reactive. We believe the reactive model is highly exclusive.

[1] `https://www.youtube.com/watch?v=WVnAbv65uCU`

Ultimately, one may say that the *problem* with this dissertation is that it was proposed too early. Should we have waited a while longer, we might had gotten utilities such as Java Fibers, or a more complete R2DBC. Perhaps an easier-to-learn version of reactive programming as a paradigm, or even a complex example that could be used to extract methodologies and kickstart the development/migration that we expected. For this reason, it makes sense to expand on these thoughts in the next section, which covers the Future Work.

11.2 Future Work

Even if we had fully achieved the proposed outcome, there would still be something more to add. Considering we have not, there are many things to do. We also collected many insights that we believe are important for those who may have the audacity of retrying what we attempted. We shall split these into two sections.

11.2.1 Community Efforts

There are obvious things that should be finished. For example, the Java Loom Project, with its Java Fibers, would have allegedly changed this dissertation if completed. Experts say Java Fibers will make blocking code non-blocking, which would allow for "dirtier" code to produce good results. This should resolve the problem of thread pools.

Alternatively to Java Loom, having ABDA in working condition would greatly help access relational databases. This, however, is not likely to happen: a company — Oracle — was behind it and shut it down. Unless the community or some other company resumes the effort to develop it, it will not become a reality.

Lastly, on the subject of SQL, there is R2DBC. Suppose it does evolve to a point where it supports the features that have made JPA a significant cornerstone of relational access in Java. In that case, it would become more than the de facto solution for relational database access in reactive: it would become the best solution for it. Not because it is the only solution — be it or not — but because it follows a specification, it is complete and presents an advantage. However, seeing as its development is lead by a few companies, we can expect it to grow as they see fit.

Regarding the cancellation subject, we believe it would be interesting — and it could have been one of our approaches in the early stages of this dissertation — to investigate the possibility of cancellation while using Spring MVC. It should be possible, using the servlet API, to pass cancellation information to the request handler. We believe that would be a major gamechanger for Spring MVC.

Another thing that should be further studied, both academically and in the enterprise context, is cancellation in SQL queries. We have not found any studies trying to classify the different types of queries as cancellable, not cancellable, or others. We realise that transactions are helpful in the context of rolling back changes, but the overhead of embedding every query in a transaction in order to support cancellation is a complication that should be avoidable.

Regarding Kotlin, there is something that could be improved as well. Right now, it is *hard* to use it outside the JetBrains Intellij IDEA IDE. The platform that developers use to code should not affect the technologies available to them, and so we believe there should be an easing of the setup for users using other tools, such as Visual Studio Code, Eclipse, etc. Moreover, it would be good to see more documentation about how Spring interacts with suspending functions. Considering the odd behaviour observed when one of these functions returns a Publisher, and how Spring is all about reactive programming, it becomes confusing that both things are supported, but not together, effectively inviting developers to use non-reactive, suspending algorithms.

11.2.2 Redoing this Exercise

To all those who may reattempt what we did, we believe it is possible to attain different results by changing the way the research is done. First and foremost, a good background of Reactive Programming is advised.

> *This is an advanced feature that is more targeted at library developers. It requires good understanding of the lifecycle of a Subscription and is intended for libraries that are responsible for the subscriptions.*[2]

This is the way the documentation of Project Reactor labels some functionalities, such as Context. It is also the level of proficiency that one should have if to delve into this subject. The Context feature is particularly paramount in these efforts as it is what makes reactive a viable alternative to the current models.

Secondly, assuming the focus on Reactive – as opposed to the previously suggested focus on implementing cancellation on Spring MVC —, we think it is relevant to watch Webinars, be present at Spring Conferences, try to mingle with the community, exchange viewpoints, experiences and ambitions. A conversation with the right person can unlock a lot in an environment where these social gatherings trump documentation.

Despite the norm in the Spring world being the migration of an application from Spring MVC to Spring Webflux, we consider that migrating a complex application is a cruel mission that sane developers should avoid. Spring MVC applications can become overly intertwined with the underlying APIs, and migration becomes unfeasible. An approach considering a new implementation may fare better. Ultimately, one should consider changing more than the code: replacing the database technology, the Websockets communication middleware and other parts should be studied. When going reactive, there is no place for maintenance of old dogmas regarding the tech stack.

[2] `https://projectreactor.io/docs/core/release/reference/#context`

11.3 Note from the author

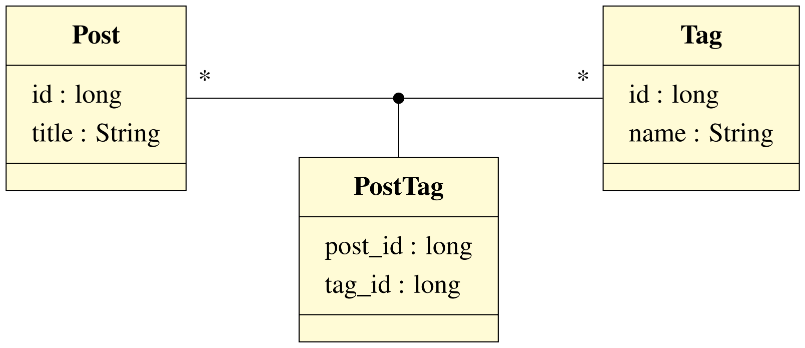
There must be something wrong when one's developing model is copying from examples on GitHub or StackOverflow. On one side, copying means that work is being redone, many times needlessly. On the other hand, it is the failure of the creative process as it exists: humans think of something and create it, or implement it. I understand copying as a learning mechanism — I have done it countless times — but it has to be limited to the learning process. Either I am still learning — a possibility that I do not discard — or there is something missing here. What is missing, in my opinion, is good documentation, good tutorials, real examples. Repeating the same PoCs *ad nauseum* in Webinars does not help the community get any more instructed. Repositories filled with test code that does not apply to a production environment are not a great help. I realise that the Spring ecosystem has the problem of being too Enterprise-oriented, making it hard to find real examples in public repositories. However, Spring MVC is also predominantly used in Enterprise environments, yet there are many examples, excellent documentation, and a strong community. I feel that the Reactive paradigm, which first appeared at the hands of Netflix, Pivotal and Lightbend[3], would have fared better if *its development* was at the hands of the academy, or agenda-free open-source communities. More people might be interested in further developing it, documenting it, finding new ways to use it, ultimately making it more engaging in an agnostic way. Instead, we have a few companies leading the development in the direction that they wish. Spring Webflux has significant contributions from Pivotal (R2DBC[4]), VMWare (Training & Certification[5]), JetBrains (Kotlin push[6]) and Oracle (Webinars[7]). A different — but conceivable — model would be having a company developing it, like Microsoft develops C#. Developing it as a product, and not a mere helper for some internal business logic. However, this might not be as cheap for developers wishing to use it.

[3] https://blog.redelastic.com/a-5ee2a9cd7e29
[4] https://spring.io/blog/2018/12/07/reactive-programming-and-relational-databases
[5] https://spring.io/training
[6] https://www.youtube.com/c/intellijidea/videos
[7] https://www.youtube.com/watch?v=WVnAbv65uCU

Appendix A

TestData

A.1 Posts



Where there are:

- 1000 Posts
- 1000 Tags
- 1000 Post <-> Tag relations

A.1.1 Ebean Model

:

```
@Entity
@Table
class Post (var title:String) : Model() {
    @Id
    @GeneratedValue
    var id: Long? = 0

    @JsonIgnore
    @JsonManagedReference
    @ManyToMany
    @JoinTable(
        name = "post_tag",
        joinColumns = [JoinColumn(name = "post_id")],
        inverseJoinColumns = [JoinColumn(name = "tag_id")]
    )
    var tags: MutableSet<Tag>? = null
}

@Entity
@Table

class Tag(name:String) : Model() {
    @Id
    @GeneratedValue
    var id: Long? = 0
    var name: String? = null

    @ManyToMany(mappedBy = "tags")
    @JsonBackReference
    var posts: MutableSet<Post> = HashSet()

}

@Entity
@Table
class PostTag () : Model() {
    var post_id: Long? = 0
    var tag_id: Long? = 0

    @JsonIgnore
    @JsonManagedReference
    @OneToOne(cascade = [CascadeType.ALL])
    @JoinTable(name = "post_tag",
        inversejoinColumns = [JoinColumn(name = "id", referencedColumnName = "id")
            ],
        JoinColumns = [JoinColumn(name = "post_id", referencedColumnName = "post_id
            ")])
    var post: Post? = null

    @Aggregation("count(*)") var totalCount: Long? = null
}
```

Listing A.1: Model defined in Kotlin for Ebean.

A.1.2 Queries:

A.1.2.1 Famous posts

```sql
SELECT id, title FROM(
    SELECT post_id, COUNT(*) AS c FROM post_tag
    LEFT JOIN post ON post.id = post_tag.post_id
    GROUP BY post_id
    HAVING count(*) > 5
) AS a LEFT JOIN post ON post.id=a.post_id
```

Listing A.2: Select the posts with at least five tags.

Appendix B

Code examples

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins {
  id("org.springframework.boot") version "2.4.4-SNAPSHOT"
  id("io.spring.dependency-management") version "1.0.11.RELEASE"
  kotlin("jvm") version "1.4.30"
  kotlin("plugin.spring") version "1.4.30"

  //custom
  id("org.jetbrains.kotlin.kapt") version "1.4.30"
}

group = "com.example"
version = "0.0.1-SNAPSHOT"
java.sourceCompatibility = JavaVersion.VERSION_15

repositories {
  mavenCentral()
  maven { url = uri("https://repo.spring.io/milestone") }
  maven { url = uri("https://repo.spring.io/snapshot") }
}

dependencies {
  implementation("org.springframework.boot:spring-boot-starter-webflux")
  implementation("com.fasterxml.jackson.module:jackson-module-kotlin")
  implementation("io.projectreactor.kotlin:reactor-kotlin-extensions")
  implementation("org.jetbrains.kotlin:kotlin-reflect")
  implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")
  implementation("org.jetbrains.kotlinx:kotlinx-coroutines-reactor")
  implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core")

  runtimeOnly("org.postgresql:postgresql")

  testImplementation("org.springframework.boot:spring-boot-starter-test")
  testImplementation("io.projectreactor:reactor-test")

  // custom
  implementation("io.ebean:ebean:12.6.2")
  kapt("io.ebean:kotlin-querybean-generator:12.6.2")
}

tasks.withType<KotlinCompile> {
  kotlinOptions {
    freeCompilerArgs = listOf("-Xjsr305=strict")
    jvmTarget = "14"
  }
}

tasks.withType<Test> {
  useJUnitPlatform()
}
```

Listing B.1: Gradle basic configuration for using Ebean with Kapt and Kotlin QueryBean Generator.

```
http.addFilterAt(WebFilter { exchange, chain ->
    val roles = rolesRepository.findUserRoles(userId)

    return@WebFilter chain.filter(exchange).subscriberContext { it ->
        var osc: SecurityContext? = null // outer security context

        // create new holder for Authorities
        val updatedAuthorities: MutableCollection<GrantedAuthority> = mutableListOf
            ()

        roles.forEach{
            // Add previously fetched database Authorities with the `ROLE_` prefix
            updatedAuthorities.add(SimpleGrantedAuthority("ROLE_$it"))
        }

        it.get<Mono<SecurityContext>>(SecurityContext::class.java).subscribe { sc
            ->
            // update the Authorities with previously loaded ones.
            sc.authentication.authorities.forEach {
                if(!updatedAuthorities.contains(it))
                    updatedAuthorities.add(it)
            }

            // also add keycloak claims to the GrantedAuthorities Collection.
            ((sc.authentication.principal as OAuth2User).attributes["authorities"]
                as JSONArray).forEach {
```

```
                updatedAuthorities.add(SimpleGrantedAuthority("ROLE_$it"))
            }

            // Create new Authentication object and store it in the sc.
            val newAuthentication = OAuth2AuthenticationToken(
                sc.authentication.principal as OAuth2User,
                updatedAuthorities,
                (sc.authentication as OAuth2AuthenticationToken).
                    authorizedClientRegistrationId
            )
            sc.authentication = newAuthentication
            osc = sc
        }
        ReactiveSecurityContextHolder.withSecurityContext(Mono.just(osc!!))
    }
}, SecurityWebFiltersOrder.LAST)
```

Listing B.2: Adding database roles to the SecurityContext. To be added to Spring Webflux's springSecurityFilterChain.

```
@SpringBootApplication
class RsocketServerApplication {

  @Bean
  fun handlerMapping(): HandlerMapping? {
    val map: MutableMap<String, WebSocketHandler?> = HashMap()
    map["/"] = MainHandler()
    val mapping = SimpleUrlHandlerMapping()
    mapping.urlMap = map
    mapping.order = Ordered.HIGHEST_PRECEDENCE
    return mapping
  }
}
```

Listing B.3: Setting up a Websocket handler using a HandlerMapping Bean.

```
static void Main(string[] args)
{
    using (
        CancellationTokenSource cancellationSource = new CancellationTokenSource()
    )
    {
        // Task-based Approach
        Task t1 = Task.Factory.StartNew(Do, cancellationSource.Token);
        Task t2 = Task.Factory.StartNew(Do, cancellationSource.Token);
        Task t3 = Task.Factory.StartNew(() =>
        {
            Console.WriteLine("Press any key to stop.");
            Console.ReadKey();
            cancellationSource.Cancel();
        });

        Task.WaitAll(t1, t2, t3);
    }
}
```

Listing B.4: Main function for using CancellationToken

References

[1] Bruce Belson, Jason Holdsworth, Wei Xiang, and Bronson Philippa. A survey of asynchronous programming using coroutines in the internet of things and embedded systems. *ACM Trans. Embed. Comput. Syst.*, 18(3), June 2019. doi:10.1145/3319618. (Cited on page 14)

[2] R. Chakraborty. *Reactive Programming in Kotlin: Design and build non-blocking, asynchronous Kotlin applications with RXKotlin, Reactor-Kotlin, Android, and Spring*. Packt Publishing, 2017. URL: https://books.google.pt/books?id=ZMxPDwAAQBAJ. (Cited on page 13)

[3] Roman Elizarov, Nov 2018. (Accessed on 03/04/2021). URL: https://elizarov.medium.com/blocking-threads-suspending-coroutines-d33e11bf4761. (Cited on page 7)

[4] Clement Escoffier. 5 things to know about reactive programming, Jun 2017. URL: https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming/. (Cited on pages 13 and 14)

[5] I. Fette, Google Inc., A. Melnikov, and Isode Ltd. The websocket protocol, December 2011. URL: https://tools.ietf.org/html/rfc6455. (Cited on page 57)

[6] R. Fielding, UC Irvine, J. Gettys, J. Mogul, DEC, H. Frystyk, T. Berners-Lee, and MIT/LCS. Hypertext transfer protocol version 1.1 (http/1.1), January 1997. URL: https://tools.ietf.org/html/rfc2068. (Cited on page 10)

[7] Stylianos Gakis and Niclas Everlönn. *Java and Kotlin, a performance comparison*. PhD thesis, Kristianstad University, Faculty of Natural Science., 2020. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:hkr:diva-20721. (Cited on pages 3, 4, and 6)

[8] D. Gotseva, Y. Tomov, and P. Danov. Comparative study java vs kotlin. In *2019 27th National Conference with International Participation (TELECOM)*, pages 86–89, 2019. doi:10.1109/TELECOM48729.2019.8994896. (Cited on pages 3 and 4)

[9] K. Guntupally, R. Devarakonda, and K. Kehoe. Spring boot based rest api to improve data quality report generation for big scientific data: Arm data center example. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 5328–5329, 2018. doi:10.1109/BigData.2018.8621924. (Cited on page 17)

[10] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, DYLA ’13, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2489798.2489802. (Cited on pages 13 and 14)

[11] Petro Karabyn. *Performance and scalability analysis of Java IO and NIO based server models, their implementation and comparison.* PhD thesis, Ukranian Catholic University., 2019. URL: https://s3-eu-central-1.amazonaws.com/ucu.edu.ua/wp-content/uploads/sites/8/2019/12/Petro-Karabyn.pdf. (Cited on page 8)

[12] A. Kolesnichenko, S. Nanz, and B. Meyer. *How to cancel a task*, volume 8063 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 61–72. LNCS, 2013. URL: www.scopus.com. (Cited on pages 14, 15, 17, and 20)

[13] Vivek Kumar. Featherlight speculative task parallelism. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 391–404, Cham, 2019. Springer International Publishing. (Cited on page 15)

[14] K. Lee, R. Pedarsani, and K. Ramchandran. On scheduling redundant requests with cancellation overheads. *IEEE/ACM Transactions on Networking*, 25(2):1279–1290, 2017. doi:10.1109/TNET.2016.2622248. (Cited on page 15)

[15] Paul Murley, Zane Ma, Joshua Mason, Michael Bailey, and Amin Kharraz. Websocket adoption and the landscape of the real-time web, Apr 2021. URL: https://kharraz.org/publications/www21.pdf. (Cited on page 57)

[16] A. Neumann, N. Laranjeiro, and J. Bernardino. An analysis of public rest web service apis. *IEEE Transactions on Services Computing*, 2018. doi:10.1109/TSC.2018.2847344. (Cited on page 7)

[17] Inc. Oracle America. Jsr-338 java persistence specification ("specification"), Jul 2017. URL: https://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-spec/JavaPersistence.pdf. (Cited on page 43)

[18] Mark Paluch. Reactive programming and relational databases. https://spring.io/blog/2018/12/07/reactive-programming-and-relational-databases, Dec 2018. (Accessed on 03/04/2021). (Cited on page 39)

[19] Jay Phelps. Backpressure explained — the resisted flow of data through software | by jay phelps | medium. https://medium.com/@jayphelps/backpressure-explained-the-flow-of-data-through-software-2350b3e77ce7, Feb 2019. (Accessed on 02/26/2021). (Cited on page 36)

[20] Mińkowski Piotr. A deep dive into spring webflux threading model, March 2020. URL: https://piotrminkowski.com/2020/03/30/a-deep-dive-into-spring-webflux-threading-model/. (Cited on page 9)

[21] Nilasini Thirunavukkarasu. Java NIO(New I/O) Vs. IO, March 2018. URL: https://medium.com/@nilasini/java-nio-non-blocking-io-vs-io-1731caa910a2. (Cited on page 8)

[22] Information Sciences Institute University of Southern California. Transmission control protocol — darpa internet program — protocol specification — STD 7, Sep 1981. URL: https://tools.ietf.org/html/std7. (Cited on pages 26 and 27)

[23] Adam Warski. Will project loom obliterate java futures?, Jan 2020. URL: https://blog.softwaremill.com/will-project-loom-obliterate-java-futures-fb1a28508232. (Cited on page 6)

[24] Ting Yuan, Yiting Tang, Xi Wu, Yue Zhang, Huibiao Zhu, Jian Guo, and Weijun Qin. Formalization and verification of rest on http using csp. *Electronic Notes in Theoretical Computer Science*, 309:75–93, 2014. Proceedings of the Sixth International Workshop on Harnessing Theories for Tool Support for Software (TTSS). URL: https://www.sciencedirect.com/science/article/pii/S1571066114000917, doi:https://doi.org/10.1016/j.entcs.2014.12.007. (Cited on page 9)