A Comparative Evaluation of Java HTTP Clients for Efficient Backend Service Communication

LIAM LJUNGERUD AND ERIK STJÄRNQUIST
BACHELOR´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

A Comparative Evaluation of Java HTTP Clients for Efficient Backend Service Communication

By

Liam Ljungerud and Erik Stjärnquist

Supervisors:

IKEA: Rand Abou Dan, LTH: Christin Swahn

(Template by Peter Nilsson)

Department of Electrical and Information Technology
Faculty of Engineering, LTH, Lund University
SE-221 00 Lund, Sweden

Abstract

HTTP clients are used to gather data from servers over a network. Several different usage scenarios for HTTP clients exist. This thesis focused on and investigated the usage of three different HTTP clients in a Spring Boot system from the perspective of implementation and performance. This system was a mock-up made by IKEA to mimic their real live Spring Boot system. The research focused on comparing the existing HTTP clients used within the backend-focused IKEA team where the study was conducted, based on Java's built-in client and RestTemplate, with Spring's reactive and asynchronous WebClient. The areas compared were implementation differences and performance consisting of different metrics. For the performance testing two methods were used, local tests with a mock API server and K6 tests. These tests were constructed to mimic real world scenarios and gather the relevant metrics. Further, the local tests were based on SpringBootTest with threads representing users and internal loops to mimic concurrency. Moreover, the K6 tests are configured to use the Spring Boots system's controller to imitate a more real scenario. Following testing results showed implementation could be made shorter and easier since several manually coded methods were integrated in WebClient. However, this could be done easily only if reactive programming was understood prior. Performance was shown to be significantly better for the reactive WebClient in several regards such as response time and throughput if the entire chain for the HTTP responses were changed to be reactive. Otherwise, the Spring Boots systems blocking parts hindered performance.

Keywords

HTTP, HTTP client, Spring, Spring Boot, RestTemplate, WebClient, Reactive

Sammanfattning

HTTP-klienter används för att hämta data från servrar över ett nätverk. Det existerar flera olika användningsscenarier för dessa klienter. Detta examensarbete fokuserar på och undersöker användningen av tre olika HTTP-klienter i ett Spring Boot-system ur ett implementerings- och prestandaperspektiv. Systemet som användes var en mock-up skapad av IKEA för att efterlikna deras faktiska produktionssystem i Spring Boot. Studien fokuserade på att jämföra IKEAs befintliga HTTP-klienter baserade på Javas inbyggda klient och RestTemplate med den reaktiva och asynkrona WebClient från Spring. De områden som jämfördes var implementeringsskillnader samt prestanda baserat på olika mätvärden. För prestandatesterna användes två metoder: lokala tester med en mockad API-server och K6-tester. Dessa tester utformades för att efterlikna verkliga scenarier och samla in relevanta mätvärden. Vidare baserades de lokala testerna på SpringBootTest, där trådar representerade användare och interna loopar användes för att simulera samtidighet (concurrency). K6-testerna konfigurerades för att använda Spring Boot-systemets controller för att imitera ett mer realistiskt scenario. Resultaten från testerna visade att implementeringen kunde göras kortare och enklare, då flera manuellt kodade metoder finns integrerade i WebClient. Detta var dock endast enkelt under förutsättning att det fanns en tidigare förståelse för reaktiv programmering. Prestandan visade sig vara avsevärt bättre för den reaktiva WebClient i flera avseenden, såsom svarstid och genomströmning (throughput), förutsatt att hela kedjan för HTTP-svar gjordes reaktiv. I annat fall utgjorde de blockerande delarna i Spring Boot-systemet ett hinder för prestandan.

Nyckelord

HTTP, HTTP-klient, Spring, Spring Boot, RestTemplate, WebClient, Reaktiv

Acknowledgments

This bachelor's thesis could not have been initiated and completed without the help and knowledge of several people.

To start, we would like to thank our supervisor at our university, Christin, for giving invaluable insight into our work during the entire process. We would also like thank our examiner, Christian, for helping to get the thesis started and completed.

Continuing, we would like to give our utmost thankfulness to Jessica at IKEA for making this thesis possible at all. Also, a big thank you to the entire team at IKEA.

Lastly, our supervisor at IKEA, Rand, has gone out of her way to make sure that the thesis and all our questions were completed and answered, and for this we are very grateful and want to thank Rand for her involvement.

Liam Ljungerud & Erik Stjärnquist

Contents

1. Introduction

This chapter introduces the entire thesis work. The chapter includes a background and the purpose of the thesis. Furthermore, the thesis's problem questions are presented.

1.1. Background

Inter IKEA includes digital departments responsible for developing and maintaining software solutions that support IKEA's digital ecosystem. This thesis was conducted within a backend-focused team in the REX department, which specializes in planning-related digital solutions aimed at improving life at home, including planning tools for areas such as kitchens, storage, and other home furnishing solutions.

The thesis was done in collaboration with Inter IKEA Technology Services AB in Malmö with the specific team DEXF, this team will henceforth only be called IKEA. However, it is the DEXF team and only the DEXF team that is referred to.

IKEA currently operates a variety of systems for its planning tools and backend infrastructure. For the current thesis project, it is IKEA's choice of HTTP client that will be investigated. One of the current solutions inside their Spring Boot system was a custom configured client developed with Javas built in HTTP client. Another solution that is being used is Springs RestTemplate.

Spring Boot is a framework for developing Java applications with support for autowired dependency injection. RestTemplate is part of Spring and is used to perform synchronous http requests. A blocking client is a client that waits for an http answer after a request without sending other requests in the meantime. In the custom Java client, IKEA has modified and added functionality to achieve some asynchronous behavior. Furthermore, other modifications have also been made and are needed for the current backend to work. These modifications were unknown to the thesis workers at the start of the thesis work and will be explained as part of the complete thesis.

IKEA believes that their HTTP client solution should be improved in the future, by both simplifying and increasing the performance of their Spring Boot system. Therefore, they want the thesis work to focus on how performance, error handling and general ease of use can be improved through exchange of HTTP clients. IKEA have constructed a mock-up backend for the thesis work that imitates one of IKEA's web planner backends. In this mock-up the thesis work will be carried out and that includes the adaptation of another HTTP client to match the current one.

The HTTP client that will be tested is WebClient since it is the currently recommended client from Spring Boot. WebClient is also of interest to IKEA because of its non-blocking features and because of its newer release.

1.2. Purpose

The purpose of this thesis was to investigate and compare IKEA's current HTTP client solutions against an implementation of Spring Boot's native asynchronous HTTP client, built to act similarly in place of their current ones, in performance regarding different measurable categories (see Fig. 38). The expected result was to be able to present results in the form of a study to compare the reactive HTTP client to IKEA's current solutions. No prototype was delivered to IKEA. However, an internal mock spring boot application was to be modified to accommodate testing of the different clients.

1.3. Goal Formulation

This thesis aims to evaluate the impact of adopting a new asynchronous HTTP client on system performance and reliability. The evaluation is based on measurements, for example response time, error handling effectiveness, number of requests, resource utilization, and the complexity of implementing WebClient (see also Fig. 38).

The thesis also investigates whether the size of the packages in JSON format sent over an HTTP client has an impact on the results.

1.4. Problem Statement

The following problem questions have been answered;

1. How do the current HTTP clients work in the mock-up backend?
2. What modifications are needed in the mock-up backend for the WebClient HTTP client to be integrated?
3. Which modifications with the new HTTP client are needed for compatibility with IKEA's provided system?
4. What measurement data should be collected to be able to compare the performance of HTTP clients?
5. What is the performance of the current clients according to problem statement 4.
6. What is the performance of the alternative client according to problem statement 4.

1.5. Motivation for the Thesis Project

The motivation for this thesis is to improve IKEA's backend systems and thereby make a positive and meaningful impact on their Spring systems. Carrying out our thesis at IKEA provides us with valuable opportunities. It offers the possibility to apply and expand knowledge in backend development within a truly large corporate setting, as well as the chance to attend professional meetings in person. This direct contact with experienced engineers provides a great opportunity to grow professionally and offers a clear perspective on the daily technical challenges of running Spring Boot systems on a global scale.

For IKEA, the motivation behind this thesis is to obtain a WebClient implementations impact along with an objective comparison against their current HTTP client solutions. To see if it is practical and beneficial to switch to WebClient within their current or future systems. This work will also result in recommendations for improving this aspect of their planning tools.

From a societal perspective this thesis is motivated by the goal of making backend systems run more efficiently, through the utilization of the reactive architecture in Spring. This benefits individuals who are designing their homes online, by providing a faster and more stable experience. Furthermore, these findings are not limited to IKEA alone. The results and architectural insights gathered can be useful for any organization that uses Spring Boot, helping to inform future architectural decisions based on objective measurements.

1.6. Delimitations

The thesis focused only on the current and a single alternative HTTP client. No tests with IKEA's real system were to be performed.

1.7. Division of Labor

Here the division of labor is presented (see Table I) for the thesis workers and the respective activities. Values are between 0-100, i.e. percentage.

TABLE I. LABOR DIVISON OF THESIS WORKERS

Activity	Ljungerud. Liam	Stjärnquist. Erik
Planning	40	60
Implementation	60	40
Testing	50	50
Writing	50	50
Presentation preparation	50	50
Poster	50	50

2. Technical Background

This chapter includes Spring Framework components and connected concepts, as well as testing tools relevant to this thesis.

2.1. Spring

Spring is an open-source framework for Java applications. It is used for enterprise applications. These applications are mostly web based. However, Spring consists of multiple modules that allow developers to choose what they need from the framework. This choice allows several types of applications to use Spring. Spring provides autowired dependency injection with components called beans. This allows Spring to offer a container of managed components, also called Spring application context [1], [2].

In this thesis Spring is an integral foundation as it is part of the underlying framework that the HTTP client will be part of.

2.1.1. Spring Boot

Spring Boot is a configuration tool in Spring. It exists to make development with Spring more streamlined. Furthermore, it makes development easier with the help of auto configuring Spring and its modules. Configuring Spring without Spring Boot includes transaction management configuration, Thymeleaf enabling and much more. Spring Boot also handles dependency management [3].

In this thesis Spring Boot has been used to create the application on which all programming and testing is done.

2.1.2. Spring MVC and WebFlux

Spring MVC and WebFlux are web frameworks (see Fig. 1). Spring MVC builds directly on the Servlet API where much is synchronous or blocking. It Circulates around a DispatcherServlet that dispatches requests to handlers, default being based on the @Controller and @RequestMapping annotations. Whereas on Spring Webflux that has the Servlet API behind a low-level adapter is built to be fully non-blocking and use less threads and hardware resources

to scale better. These two frameworks are optional and can be set up together to widen the available options, “for example, Spring MVC controllers with the reactive WebClient.” [4], [5].

The system provided by IKEA is built on Spring MVC and employs the annotations stated above. Spring Webflux had to be added to allow for the use of WebClient in this thesis.

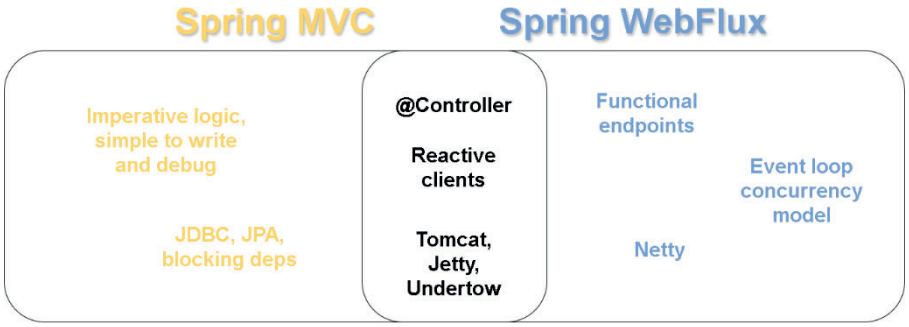


Fig. 1. Ilustration of Spring MVC and Webflux and their differences, inspired by [4]

2.2. ExecutorService

In Java, an executor is a class that provides handling of tasks asynchronously. It automatically provides a pool of threads and the ability to assign tasks to it [6].

2.3. Reactive Programming

Reactive programming is a concept of programming that differs from normal sequential programming in several ways. In programming, if a developer wanted to make a calculation based on several variables, that calculation would only be computed at the time of running that line in the code. For example, if A = B + C is line 42 in a program, A would be based upon the state of B and C at that time. Reactive programming does this differently. By making the program work as if it was using the observer pattern, A would be updated if either B or C changes value at a later time, asynchronously. In this example, A would be a subscriber and B and C publishers. This concept is the fundamental nature of reactive programming,

there are several other features of different implementations of reactive programming, however they are not universal [7].

In this thesis, reactive programming is a fundamental part of the implementation of an async HTTP client.

2.4. Reactor

Reactor is an open-source library to enable reactive programming in java, made by Project Reactor [8]. Reactors underlined base is called Reactor Core. This library offers reactive programming and the ability to handle so-called backpressure when a subscriber is overwhelmed with data from publishers. Reactor Core manages this by making the publisher ask the subscriber how much data it can handle before sending it [7].

In this thesis, Reactor is the base of the async HTTP client being implemented.

2.4.1. Reactor Netty

Reactor Netty builds upon Reactor Core but focuses on providing support for network and HTTP functionality. It provides the ability to set up TCP, UDP and HTTP-servers and clients [9].

For this thesis, Reactor Netty is the default underlying HTTP client used in WebClient and is used as a webserver in as one scenario.

2.4.2. Event Loops in Reactor Netty

At the core of Reactor Netty is its ability to be non-blocking; this is achieved with the use of event loop groups and its event loops. For several channels, client-server connections, Netty has one event loop and thus one thread to oversee and handle HTTP requests.

Furthermore, if more than one request is sent a new channel may be created if the previous channel has not been kept alive. The event loop has an event queue that it places the channels events in, although the event loop group may place a new channel in another event loop entirely.

Moreover, when a channel has been allocated to an event loop it does not ever move to another one, thus creating thread safety. To keep track of channels, the event loop uses a selector that recognizes events and alerts the event loop [10].

2.4.3. Backpressure

A part of project reactor and thereby reactive programming is backpressure. Backpressure is the ability for a consumer to signal to the producer that the rate of transmission is too high. The client can also request different levels of transmission; it is inherently demand-driven [7].

2.4.4. Mono and Flux

Mono and Flux are classes from Project Reactor, and they implement the Reactive Streams Publisher interface. The classes represent an asynchronous sequence. A Mono promises a result and that it is either one or zero items. A Flux either terminates with a successful completion signal or an error, and the result is a stream of zero to N items. Additionally, Flux and Mono are lazy and upon creation they wait indefinitely without consuming resources until a subscriber subscribes to them.

Because of their reactive inheritance, both classes provide a rich vocabulary of reactive operators to handle faults, chain multiple asynchronous steps, or combine multiple asynchronous responses. Unless forcefully blocked using block, they remain non-blocking. They also adhere to backpressure [11], [12].

In the context of this thesis Mono and Flux are the primary classes driving the asynchronous HTTP client implementation, allowing for declarative and efficient response handling.

2.5. CompletableFuture

CompletableFuture is a class in standard Java that represents the future result of an asynchronous computation. It implements the Future and CompletionStage interfaces. This allows for non-blocking, callback-based chaining. The result is always one item, which could be null, or an error [13].

This is the primary class that is used in the provided system by IKEA to make asynchronous HTTP requests.

2.6. Apache Tomcat Webserver

Apache Tomcat is a software implementation of many specifications from the Jakarta EE platform, for instance Jakarta Servlet and Jakarta Server Pages, having it function as a webserver and servlet container [14].

The Apache Tomcat server works on a principle of one thread per request. This means that for each HTTP request the server handles it dedicates one thread [15].

In the context of Spring, depending on the chosen stack, all Spring Boot web applications include an embedded web server. Spring MVC, the servlet stack has Tomcat as default embedded web server, however this can be changed to Jetty, Undertow or Reactor Netty [16]. This thesis utilizes Tomcat for the comparative testing of HTTP clients to align with IKEA's systems.

2.7. Application Programming Interface

Application Programming Interface (API) acts as a messenger or gateway, it is software with rules and contract that enables communication between two applications, usually as client and server. There are four different ways APIs work, and the most common and flexible ones today are REST APIs [17].

2.8. Representational State Transfer

Representational State Transfer (REST) is a software architecture that an API can implement to achieve seamless communication with the ability for easy modification. A REST architecture implies among other things, that server information is transferred in a standardized format and that all communication is stateless, meaning that every client request is completed independent from every other request [18].

REST client request

The client request contains a unique resource identifier to know exactly the resource to locate and is usually used together with a Uniform Resource Locator (URL) to specify the resource to the server. A HTTP method such as GET or POST is also specified for the server to know what needs to happen to that resource. In conjunction with a method, HTTP headers provide metadata, that is additional information such as the format of the request and response, the authentication method and information, and other data and parameters to guarantee successful operation.

The server response contains a three-digit status code to indicate general outcome, the response resource in XML or JSON format, and headers with metadata to add additional context about the response [18].

In this thesis, all server communication is based on these REST principles. Because of this, the ability to handle HTTP method, headers and JSON responses efficiently is the main way the different HTTP clients are compared.

2.9. REST Clients

REST Clients are software tools that help with and handle HTTP requests. They gather all information needed to make the request using organized fields and handle everything from structuring and sending the request to receiving and processing the response. Additionally, they can also manage exceptions and use underlying HTTP clients [19]. The Spring framework provides three different clients to choose from built for Spring.

This thesis utilizes two out of these three clients. RestTemplate is used as reference for synchronous testing and the IKEA provided Java HTTP Client is used as a second reference for asynchronous testing in evaluation of this paper's WebClient implementation.

RestTemplate

RestTemplate is the original Spring MVC(REST) client and it performs synchronous requests and is blocking. It utilizes a template method API, where one method is called at a time. Underlying HTTP client libraries consists of the JDK HttpURLConnection, Apache HttpComponents, among others. The client is deprecated in favor of the more modern RestClient for synchronous needs [20].

WebClient

WebClient is the client offered in Spring WebFlux to carry out asynchronous requests. Each method of operation on the WebClient returns a new immutable instance, enabling asynchronous logic without side effects. It is a reactive and asynchronous HTTP-client that can be utilized both as non-blocking or blocking and for streaming purposes. The client makes use of a fluent API based on Reactor that simplifies code and streamline non-blocking operations. And it handles complex threading by itself when chaining predefined methods. WebClient requires an underlying HTTP client library and has native support for Reactor Netty, JDK HttpClient, Jetty and Apache HttpComponents. This is the recommended client by Spring [21].

Java HttpClient

Java HttpClient is Javas native client and is built into the JDK, thereby eliminating need for external libraries. The client can send either synchronous or asynchronous requests. Java HttpClient is low-level, as in not part of Spring and its own underlying HTTP client [22]. Both RestTemplate and WebClient can use Java HttpClient as their underlying client.

2.10. Spring Boot System provided by IKEA

The Spring Boot system provided by IKEA consists of the parts needed for the system to function. These parts are a *Controller, Repository,* and HTTP client. Additional parts present in IKEA's production services have been bypassed or excluded. The remaining classes within the provided system consist of domain models, data transfer objects, and utility components that define the internal data structures across the different architectural layers.

The system is built upon the traditional Spring MVC framework.

Controller

The REST controller handles incoming HTTP requests by defining specific endpoints, one for each HTTP client, that the frontend or other services can connect to. The controller layer extracts path variables, query parameters, and HTTP headers from the incoming request. Then it parses the parameters and signals the underlying repository layer to retrieve the data. The controller methods return standard synchronous objects.

Repository

The repository handles external communication from the controller by packaging the input parameters into a standard request object, such as *DexfOriginRequest*. The repository layer extends an HTTP client base class to execute the network request. Regardless of whether the underlying client is synchronous or asynchronous, the repository uses a shared *getReponse* method to block the execution thread until all requests have returned. The response is then bundled and returned synchronously to the controller, allowing any HTTP client to function within the existing system.

2.11. Custom Clients provided by IKEA

In this section the clients and their implementation provided to the thesis workers by IKEA will be explained.

2.11.1. AsyncClient using Java 11 HTTP Client

IKEA's custom made HTTP client uses the built-in Java client introduced in Java 11 [22]. Since this built-in client only contains the methods to create a working client, engineers at IKEA have made a custom Async client using the Java client. The Java client supports both synchronous and asynchronous HTTP request sending but does not have the ability to manage several Clients and DNS connection updates.

Firstly, the public methods that are meant for using the client are an *executeRequest* method that takes two parameters, a request template and a returntype class. Secondly there is also a method for getting a response in the form of the class chosen before.

In order to have more than one client sending and receiving IKEA has used an ExecutorService with a fixed number of executors based upon the number of wanted http clients. These executors are then attached to an HTTP client and that client is placed in a list with all the clients.

Furthermore, IKEA needs the clients to be able to go idle and by that create new connections with DNS services when the address to such a service change. To accommodate client idling, IKEA created a basic pooling system with a specified number of pools that the clients are divided into. The active pool changes based upon a timeout and allowed the inactive pools' clients to idle.

Additionally, there are methods for choosing an available client for requests based on the active pool and client availability. When sending a request, the client creates a CompletableFuture object that promises to have a response at a later time but does not block the main thread, only the thread delegated to the request.

Finally, IKEA also have methods for validating and processing the response that is received from the sent requests.

2.11.2. RestTemplate

A blocking HTTP client in the form of RestTemplate is also in use at IKEA. Firstly, the RestTemplate client has the same public methods as the AsyncClient mentioned in 2.11.1. I.e. to execute requests and get the corresponding response.

On the other hand, this client does not create Java HTTP clients but RestTemplates instead, these are put in a list and handled like the async clients regarding pools.

The contrast against the Async client is that a RestTemplate blocks an entire thread after it has sent a request until it is received.

2.12. Testing Tools

This section explains different tools used for testing REST Clients and IKEA's provided system in Spring Boot.

2.12.1. Grafana k6

Grafana k6 is a load testing tool that is built to generate loads e.g. HTTP requests and collect measurements e.g. request duration. To apply a load, scripts typed in JavaScript are constructed that can create and mimic loads for different scenarios on a system. Throughout a test k6 collects measurements that compounds generating a report and can alternatively be exported raw. [23]

This thesis leverages k6 to carry out full system comparative load tests on IKEA's provided Java HttpClient and this thesis's WebClient implementation, incorporated (bundled) into IKEA's provided system.

2.12.2. WireMock

WireMock is an API mock testing tool that can run as library, standalone server or cloud-service. It supports advanced matching and dynamic responses with the use of stubs to test normal operation as well as complex error situations [24].

In this thesis WireMock is used as a standalone server to handle predefined request matching criteria and to simulate various API

behaviors, including success scenarios, delays, and error states, without the need for IKEA's actual services.

3. Related Work

During the last years, several topics related to this thesis have been made relevant. Reactive programming has, for example, been introduced in Spring; therefore, several relevant studies have been made and one of them, especially relevant to this thesis, have been summarized in this chapter.

3.1. Migrating from Developing Asynchronous Multi-Threading Programs to Reactive Programs in Java

Zbarcea and Tudose investigated the process and results from migrating to reactive programs from classic async programs [25].

Firstly, the authors focused on the process of switching to the reactive realm with a particular focal point on architecture, technologies, and design. However, results touched, besides architecture, technologies, and design, on response time and other performance metrics.

Additionally, Zbarcea and Tudose examined the current state of technologies and frameworks for the use of reactive programming. The authors point out that Spring Boot and WebFlux are a solid combination for developing reactive web applications and that several large organizations have transitioned or are transitioning to such web applications [25].

Moreover, the authors explore two architectures, one asynchronous and one reactive, both based on Java 21 and Spring Boot 3. One of the architectures was based on Javas HttpClient and asynchronous operations such as completableFuture. The other architecture was created with WebClient and operations such as Mono and Flux [25].

Finally, Zbarcea and Tudose discuss the results of the study and find that the reactive model had less memory and CPU usage and shorter response times for several scenarios of different concurrencies of simulated users over the async model.

4. Method and Analysis

In this chapter, the methods and analysis of them done by the thesis workers for the purpose of completing the thesis are presented. From the workflow to communication, studies, implementation, and testing.

4.1. Work stages

In this section the steps by the thesis workers to complete the thesis will be presented.

4.1.1. Workflow

The steps taken in this thesis to achieve the final conclusions were as follows.

1. Pre-studies of Spring, HTTP, HTTP clients, Reactive programming
2. Understanding of IKEA provided Spring boot system
3. Understanding of IKEA provided HTTP clients
4. Learning of IKEA GitHub praxis
5. Implementation/Development of WebClient
6. Implementation/Development of comparative and local performance tests for WebClient, AsyncClient and RestTemplate
7. Implementation/Development of K6 external endpoint tests for WebClient, AsyncClient and RestTemplate
8. Evaluation

4.1.2. Communication within the thesis work

For the duration of the thesis work, communication between the workers was mostly done through online meetings and collaborative online work tools. These tools, such as Discord, Microsoft Word and JetBrains's IntelliJ "code with me", where used to accommodate close collaboration with minimal communication loss.

Similarly, communication with IKEA's engineers was largely conducted through online interaction. Specifically, this interaction took place on Slack and Microsoft Teams. Furthermore, the thesis workers also made regular visits to the IKEA office to accommodate a more direct and less timely forced collaboration.

4.2. Pre-studies

This section covers the different studies that have been made by the thesis workers. The studies are of factual source types; this means that studies, for e.g. the Spring Framework, were based on factual and technical documentation.

4.2.1. Understanding Spring and Spring Boot

To prepare for and succeed with the thesis the underlying framework, Spring, had to be understood. That meant going over the Spring official documentation and looking at examples from the usage of the framework. Furthermore, the knowledge of Spring had to be understood in the context of the provided Spring Boot system from IKEA. For the IKEA system, conversations were held with IKEA and its engineers to explain and answer questions from the thesis workers. The interaction between the IKEA engineers and the thesis workers helped to better understand the Spring implementation from IKEA.

4.2.2. Overview of HTTP Clients

The thesis questions target HTTP clients and their characteristics; thus, general knowledge of HTTP clients had to be acquired by the thesis workers. This knowledge was acquired by conversations with IKEA's engineers.

4.2.3. Understanding of Reactive Programming

To understand WebClient and Reactor, knowledge about reactive programming had to be acquired by the thesis workers. This understanding was collected with the help of project reactors reference guide and several meetings with IKEA's engineers [7].

4.3. Understanding of IKEA Spring Boot system

For the purpose of this thesis IKEA provided a Spring Boot system that acted as a base for further implementation of HTTP clients and allowed for testing, see 2.10. To understand this system the thesis workers did a study on Spring and Spring boot, see 4.2.1, and used it as a base of knowledge before meetings with IKEA's engineers for more specific knowledge.

4.4. Understanding of IKEA HTTP clients

Prior to implementation of a new HTTP client the former had to be understood. IKEA provided a custom HTTP client and an implemented RestTemplate, see 2.11, for use in this matter.

Firstly, during several meetings with IKEA's engineers' knowledge and understanding was built for the thesis workers regarding the provided clients.

However, further understanding was acquired through analyzing the code and testing it with simple spring boot tests.

4.5. Learning of IKEA's GitHub Praxis

Version control is of outmost importance to structure development of code. IKEA uses GitHub and therefore Git for this purpose. IKEA's way of using GitHub was at the start of development of the WebClient taught to the thesis workers. The IKEA praxis in using GitHub was then applied for the remainder of the thesis work.

The following steps are taken when a developer starts, and finishes, a new feature implementation in an existing GitHub repository.

1. Developer selects master branch and pulls all new changes.
2. Developer implements start of new feature.
3. Developer creates new branch from master with the additions from point 2.

4. Developer immediately requests to merge the new branch with master, thus creating a pull request on GitHub.
5. Developer writes description of the feature it is developing in the first commit message.
6. Developer pushes changes over time to the new branch and thus the pull request.
7. When developer is done with implementation it asks coworkers for review.
8. Coworkers either ask questions, demand correction or accept the pull request.
9. Developer can now “squash and merge” the new branch to master.

4.6. Implementation of WebClient

For the implementation of WebClient, a method of copying the style of IKEA’s other clients was first chosen. By looking at primarily the AsyncClient, WebClient was built using the same structure.

The focus on the implementation was to adapt the logic from IKEA’s provided asynchronous client but to turn it reactive to use modern library functions. The hope of improvement was that enough time had passed for modern techniques to show real benefits.

To begin with the implementation, the differences regarding the objects used in all the parts linked to the clients’ actions in the AsyncClient versus WebClient were identified with the help of studies for the respective clients. The method of studying the objects used before start of implementation was chosen to help identify where the differences were going to be.

Furthermore, the goal was to make the WebClient implementation agnostic, like the other clients, regarding using it. This means that the public methods and their parameters remain the same. Therefore, the methods were kept and only their internal logic changed.

Also, a configuration file was made for the WebClient. In the file, several settings were specified such as, maximum connections,

timeout and size of queue. These settings were chosen to mimic the AsyncClient's to achieve comparability and similar configurability.

Later, when confirmation that the client worked as intended was acquired, methods required to be implemented for the AsyncClient and that WebClient had built in support for were removed and replaced with WebClient specific functionality.

4.7. HTTP Client Testing

Testing of the clients was done with more than one method. Primarily Spring Boot tests were implemented and tested to verify the basic functionality of all the clients. These JUnit tests were made with HTTP request tests.

Furthermore, the primary objective of testing was to measure performance and for this, two methods were used.

4.7.1. Local Spring Tests

For the purpose of testing performance, a first attempt with local Spring Boot tests was chosen, this was because of guidance from engineers at IKEA who used a similar method to test another HTTP client.

Firstly, a mock server that returns HTTP requests as responses needed to be implemented, this mock server was provided by IKEA. Several methods were provided in the mock server to imitate different HTTP responses. Specifically, the methods provided were a ping, status, invalid, slow and, the most versatile, payload method. The payload method enabled, configurable size, delay and jitter of the response.

Along with a server mock, the actual tests were implemented using @SpringBootTest, a Spring test module. By using that module, the test has the ability to use Springs context and i.e. start an embedded TomCat server.

Furthermore, a base class was formed containing attributes affecting the performance of a HTTP client, such as threads, total requests, concurrency per thread, payload size and delay. Furthermore, the base class does the calculations for all the metrics

collected. Moreover, the base class starts and interrupts a metric collection thread that is dedicated to collect performance data, CPU usage and memory usage, while running the test. This method of creating a base class, with attributes that affect performance, was chosen to make comparative tests across the extended subclasses consistent.

Finally, together with the base class three separate test classes for each HTTP client were shaped. These test classes contain a warmup that sends a few requests, this was done to allow the JVM to optimize to improve test consistency. Further, the subclasses have logic for the respective tests, this logic is sent to the base class as a runnable for it to be used as the logic for the created executors in the base class.

In the following description of the three specific tests, AsyncClient's test will be described in detail, and the additional clients' descriptions will only point out differences to the AsyncClient's test. Therefore, it can be assumed that any stages of the test not mentioned in the WebClient and RestTemplate descriptions are the same as in the AsyncClient's test.

AsyncClient Performance Test

Performance testing the AsyncClient was done with the implementation of the AsyncClientPerfTest subclass. The subclass is used together with the base class to run the complete test, whose sequence can be seen in Fig. 2.

To start, the AsyncClientPerfTest sends its testing logic to PerformanceTestBase that starts the MetricsCollector and the execution of the testing logic for each executor as indicated by the vertical loop. The executers were chosen to represent and simulate a set of users using the AsyncClient simultaneously.

Each executor then asks the AbstractAsyncClientHttpService to execute a prepared request. This execute method is called within a nested loop, with the outer horizontal loop representing the round of requests to split all requests into the allowed concurrency and the inner loop sending all the execute calls. These loops were implemented to simulate the maximum concurrency per thread.

When the AsyncClient is asked to execute a request, it sends it immediately to its destination and returns a CompletableFuture containing null.

Next, the AsyncClient receives a response and sends it to its method validateResponse, this is a custom coded method that checks the response's status code. Finally, the test logic waits then joins all CompletableFutures, and the collected test data is sent to the PerformanceResultWriter.

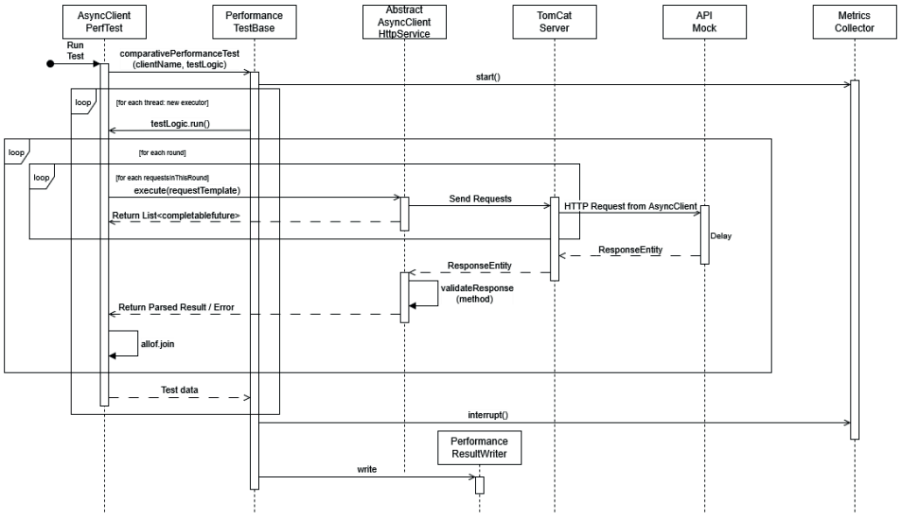


Fig. 2. AsyncClient local performance test

WebClient Performance Test

Performance testing the WebClient was done with the implementation of the WebClientPerfTest subclass. The subclass is used together with the base class to run the complete test, whose sequence can be seen in Fig. 3.

Firstly, when WebClient receives the call to execute a request, it does not send it immediately, therefore the exclusion of that stage in the inner loop in Fig. 2. WebClient, at that stage, only adds the eventual request to a list of Monos and sends it back to the test logic. Only when the list of Monos is subscribed to, in blockLast, does WebClient send the requests.

Moreover, when WebClient receives a response, it uses a built-in function onStatus to validate the response.

Finally, tests were completed with a changed webserver from Tomcat to Netty. Further, the Api mock was changed to be reactive i.e. return Mono as response. This change was completed to test a full reactive chain from client, webserver and API.

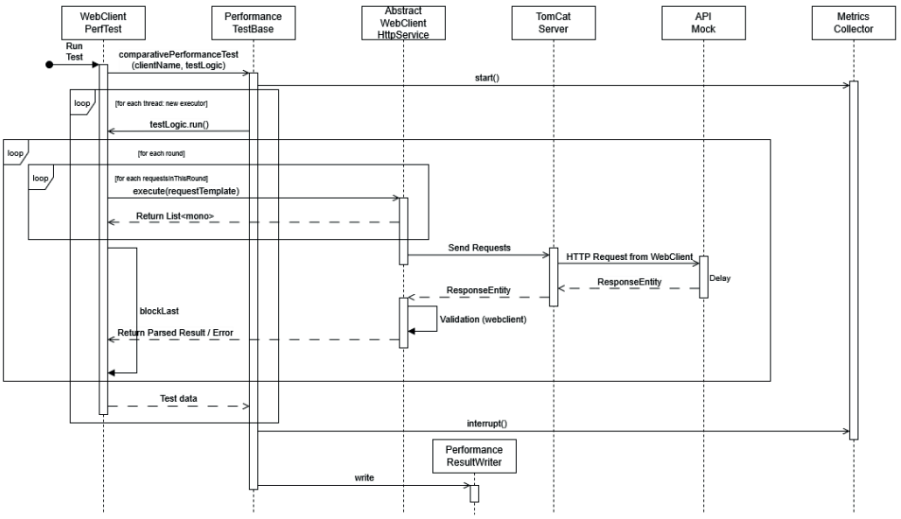


Fig. 3. WebClient local performance test

RestTemplate Performance Test

Performance testing the RestTemplate was done with the implementation of the RestTemplatePerfTest subclass. The subclass is used together with the base class to run the complete test, whose sequence can be seen in Fig. 4.

RestTemplate does not send requests asynchronously and therefore the test in Fig. 4 does not contain an inner loop. For every request RestTemplate is asked to send, it blocks and thereby waits until the response has been received and validated before sending another request.

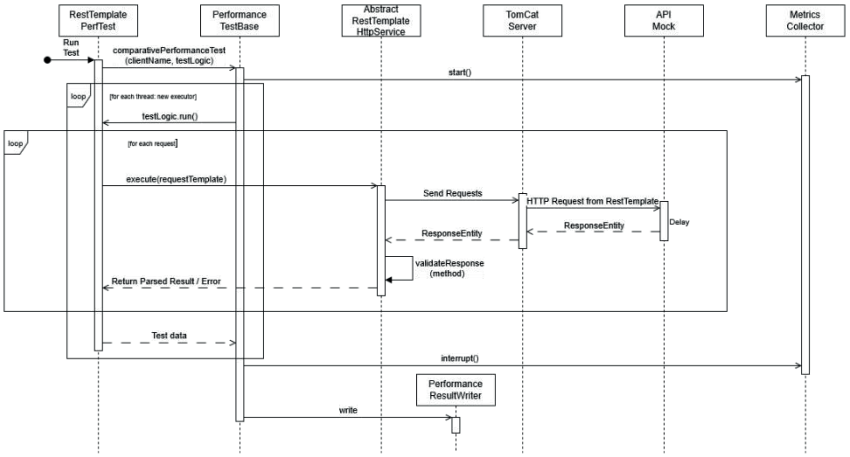


Fig. 4. RestTemplate local performance test

Testing variables

To run the local tests, variables had to be chosen to make the tests show the intended comparisons, see Table II in which the tested factor is marked with an X. For the tests regarding concurrency, a single thread was chosen to show RestTemplates synchronous behavior. On the other hand, the tests for threads were completed with concurrency set to one for a level playing field.

Furthermore, for testing the implications of differing processing delays a smaller number of requests were chosen to allow the clients to complete all before a timeout, thus making the tests focus on performance rather than error handling. For the processing delay tests a concurrency of one thousand and a large payload size was chosen to both allow the asynchronous clients to use their concurrency advantage and to enhance differences in the clients' ability to handle large ongoing requests.

Similarly, for the payload size tests, a thread count of ten and a concurrency of one hundred was chosen to both allow RestTemplate to handle large payloads simultaneously and to allow the asynchronous clients to use their concurrency.

Finally, for the stress test a similar approach for the payload size test was chosen as any number of threads less would make the

RestTemplate untestable regarding length of the tests. Furthermore, concurrency was decided to be one hundred as that was the highest number, together with ten threads, that would not cause mass errors.

TABLE II. TESTING VARIABLES

Chapters	Total requests	Threads	Concurrency	Processing delay	Payload size
Concurrency	10000	1	X	50	64
Threads	10000	X	1	50	64
Processing delay (ms)	1000	1	1000	X	640000
Payload size (kB)	10000	10	100	50	X
Stress test	X	10	100	50	64

Test environment

For running the local tests, a single computer, with specifications shown in Table III, was chosen to run all of them. This choice was made to eliminate system resources as a differentiating factor.

TABLE III. COMPUTER SPECIFICATIONS

Test computer	Asus Zephyrus G15 (2021)
Processor	AMD Ryzen 9 5900HS
RAM	16 GB DDR4 3200 MT/s
Graphical processor	AMD Radeon Vega (Cezanne)

4.7.2. K6 tests

To test the HTTP clients inside the system IKEA provided in whole, detached from their services and locally, a few things had to be done. Firstly, the task involved researching a tool to simulate requests and allow request load scenarios. Secondly, another tool to receive these requests from the system, map them and mock a relevant response had to be researched. This research amounted to searching online and asking IKEA employees that provided recommendations.

To align with the thesis goals, this pair of tools had to be advanced enough, that combined they could simulate close to real testing scenarios, with the ability to log them. Which was the reasoning for adding k6 together with WireMock to carry out these tests.

Implementation of these tools meant additional modification of the provided system. It also meant writing test cases to be able to utilize those tools (see Fig. 5 for example of a test). Information to do so was gathered from their respective documentation. Here different WireMock stubs were mapped to filters in the provided system controller.

Testing parameter numbers to accurately display relevance toward IKEA's real production environment, such as connection pools and timeouts, were either extracted directly from the system's properties files or established through discussions with IKEA engineers.

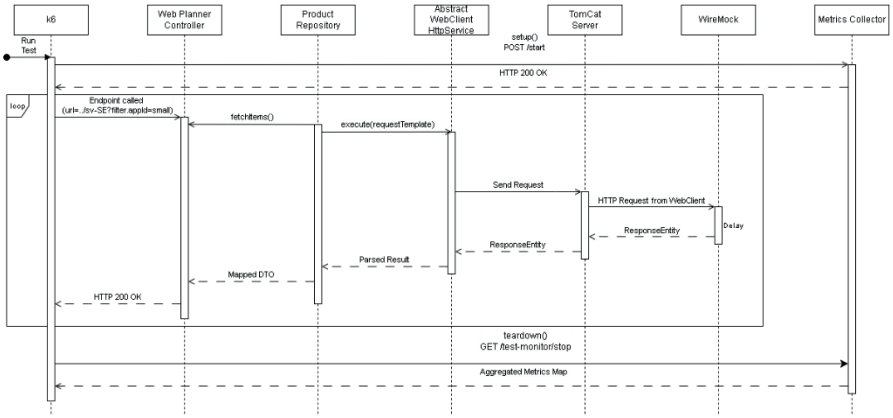


Fig. 5. WebClient K6 performance test

Testing variables

To conduct a fair and comprehensive evaluation of the three HTTP clients, specific testing variables were defined to expose potential bottlenecks within the complete transaction chain. The tests were designed to measure the system's capacity to handle increasing numbers of virtual users (VUs), limited thread availability, and network irregularities. The primary variables are summarized in Table IV.

TABLE IV. TESTING VARIABLES

Test Scenario	Total Duration	Virtual Users (Vus)	Tomcat Threads	Error Rate
Load/Scaling	3 min	50-4000	100	0%
Thread Starvation	3 min	400	10, 15, 20, 100	0%
Error Handling	3 min	400	100	10%

The simulation of requests and responses with varying number of VUs was performed using k6 and WireMock. The tests were 3 minutes long including a 30-s ramp-up period and a 30-s ramp-down

period for each scenario. The focus was on monitoring and analyzing the performance of the complete system using different Http clients, including an increase and decrease of virtual users.

For the thread limited scenario, a fourth client (WebClient Reactive) is introduced. The implementation is still the identical WebClient using a simulated reactive architecture within the provided Spring MVC system environment. By returning the reactive type Mono across the entire execution chain, from the WebClient through the repository and up to the controller, it creates a partially reactive flow inside the synchronous system.

For the error handling scenario, a 10% fault rate was artificially induced using WireMock. This fault rate cycled between three problematic responses: a delayed response (2 seconds), an HTTP 503 "Service Unavailable" status, and a malformed JSON payload.

Data Collection

The metrics were collected using two reports. A k6 report was generated and provided all HTTP related metrics such as Request Duration, Requests Per Second and success related metrics. A report using Micrometer [26] was also generated and provided additional metrics such as Memory consumption and CPU usage, as well as thread related metrics. These two reports combined formed a comprehensive variety of relevant metrics, offering insight into the performance of the provided system's complete transaction chain, utilizing different Http clients. Additionally, they display the system's ability to handle increasing number of Virtual Users and Threads.

Test Environment

To run the external load tests, a single dedicated machine with specifications detailed in Table V was chosen. This ensured that the hardware resources available for the k6 load tests and the WireMock server remained consistent across all comparative tests.

TABLE V. COMPUTER SPECIFICATIONS

Test computer	Microsoft Surface Laptop 7 (2024)
Processor	Snapdragon® X 10-core X1P64100
RAM	16 GB DDR5 8448 MT/s
Graphical processor	Qualcoom® Adreno™ X1-85 GPU

4.8. Methodological Limitations

Initially, testing with a fully reactive embedded web server like Netty was considered, as prior research [25] suggests this optimally leverages WebClient's capabilities. However, for the K6 testing, this was deemed unfeasible since the provided IKEA system is built on Spring Web MVC, which relies on the synchronous Java Servlet API. Because Netty does not implement this API, integrating it would have required a fundamental rewrite of the baseline application. Therefore, all K6 tests were conducted using the standard Tomcat server, ensuring the performance analysis reflects the actual constraints of the existing real environment. However, all non-K6 local tests were performed with a full reactive chain including Netty as a webserver.

4.9. Criticism of References

Here in this section the references, that are, mainly, used to explain relevant information to this thesis in the technical background, are reviewed and evaluated. The references originate from different sources that could be websites, articles or books. Direct sources can be found in Chapter 9.

Sources [1], [4], [5], [7], [8], [9], [11], [12], [16], [20], [21] and [26] were from the official documentation of Spring or related to Spring by the maintaining company Broadcom Inc. They are a large company designing and manufacturing infrastructure software and semiconductors, and have a long history of innovation, setting global standards within Wi-Fi 6 and Bluetooth. These sources are considered

highly credible, also as they are primary information and documentation written by the developers of the framework itself.

Source [10] is a book by Marvin Wolfthal and Norman Mauer, two knowledgeable people about software. Maurer being a senior software engineer at Apple and a core developer of Netty, combined with the book having to be reviewed and published, makes it a credible source.

Source [17] and [18] is articles about general explanations of APIs and RESTful APIs respectively. They are provided by Amazon Web Services, who are currently the world's leading provider of cloud services. Handling critical infrastructure for banks and the CIA and have guides that are the gold standard for how modern software should communicate securely. Consequently, this source is considered highly authoritative and reliable.

Source [19] is an article that provides an overview of REST API clients. The article is by BrowserStack Limited, which has the leading platform for testing software on real devices. They are trusted by companies like Microsoft and Spotify, which ensure their trustworthiness.

Source [23] is a blog post by Théo Crevon and published by Grafana Labs, that explains the k6 load testing tool. They are one of the largest tech companies creating tools for visualizing metrics. While it is an official post, it might not be as credible compared to the documentation. It is great for understanding the tool and since most projects by them are open source, thus they value transparency, it is deemed credible enough.

Source [6], [13] and [22] is from the official Oracle documentation. Because Oracle owns the Java brand and platform, they provide an undeniable authority on the subject matter.

Source [14] and [15] is from the Apache Software Foundation that oversees hundreds of open-source projects, one of them being the Apache Tomcat, an HTTP server software. This, combined with them being a nonprofit organization, ensures the integrity and trustworthiness of the provided documentation.

Source [24] is from the official documentation of WireMock. This source can be deemed credible, as it is open-source, community-

driven and constantly peer-reviewed by its users. WireMock from WireMock inc, is a tool for API simulation.

Source [3] and [2] are two books written by Craig Walls and published by Manning Publications. That is a well-regarded independent publisher of mainly computer books for software developers. The books being published by this publisher go through editorial and review processes, hence these can be considered credible sources. The first book is a comprehensive guide through the Spring framework and the second focuses specifically on Spring Boot, a tool that simplifies the process of setting up and running Spring based applications.

Source [25] is a scientific paper published by MDPI in the journal Applied Sciences. MDPI is a publisher of open access scientific journals, and the fact that the paper has been peer reviewed by independent academic experts makes it a highly credible source.

Source [27] is from Elsevier and describes the Scopus AI tools. Elsevier is a widely recognized publisher of scientific literature; the source can therefore be considered credible.

4.10. Usage of AI

Throughout the thesis some use of AI (LLMs) was performed. The areas of AI usage were restricted to finding sources of information, limited spell and formulation checking and code inspiration.

Since the usage of AI was only done with the purpose of finding sources for information and inspiration for code, the thesis workers can fully stand behind the thesis work.

4.10.1. In Research

To find information regarding parts of the Spring Framework, Reactor and Tomcat, AI was used as a springboard for finding resources that were relevant to the thesis. Here the sources were preferred to be academic, therefore Scopus AI was the preferred tool

[27]. In case Scopus AI could not find relevant sources, Google Gemini was used in the same manner.

As some of the documentation was found to be lacking in explanations AI was used again to find sources that explained the previously found documentation.

4.10.2. In Writing

Writing the thesis was all done by the thesis workers with no direct involvement by AI. Although AI was used as a sparring partner when formulating a small part of the sentences.

4.10.3. In Code

By request from IKEA some code completion was used. Although the thesis workers decided to minimize the usage of generated code. Only inspiration from AI was used, no actual classes were written entirely by AI.

5. Results

In this section the complete results from the thesis are presented. Firstly, results from implementation and at the end from testing.

5.1. Implementation

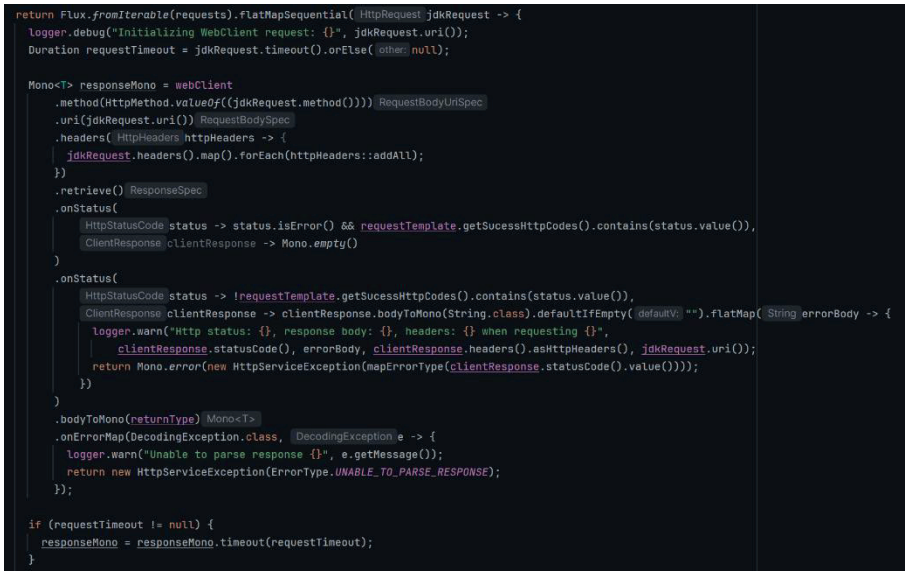
Results from implementation of code are inherently context based as they can look different for each application. The following results are from the perspective of IKEA's existing HTTP clients provided to the thesis workers.

5.1.1. WebClient Integration and Novel Features

To modernize the HTTP communication, several new components and architectural patterns from Spring WebFlux and Project Reactor were fitted into the implementation.

A configuration file for the WebClient instance was the core addition that was developed. The configuration file exposed relevant options to modify the instance. Options like those to match configurations the provided client had and to improve, replace or simplify them.

The most significant architectural change introduced by the WebClient implementation was the transition from Javas CompletableFuture to Project Reactors Mono and Flux. The system's return types and processing logic were rebuilt with reactive streams using these two classes. As shown in Fig. 6. The new method flatMapSequential subscribes to all the requests at the same time and ensures that they are executed asynchronously. It also guarantees that the response is in the same order it was executed in.



```
return Flux.fromIterable(requests).flatMapSequential( HttpRequest jdkRequest -> {
  logger.debug("Initializing WebClient request: {}", jdkRequest.uri());
  Duration requestTimeout = jdkRequest.timeout().orElse( other: null);

  Mono<T> responseMono = webClient
      .method(HttpMethod.valueOf((jdkRequest.method()))) RequestBodyUriSpec
      .uri(jdkRequest.uri()) RequestBodySpec
      .headers( HttpHeaders httpHeaders -> {
        jdkRequest.headers().map().forEach(httpHeaders::addAll);
      })
      .retrieve() ResponseSpec
      .onStatus(
          HttpStatusCode status -> status.isError() && requestTemplate.getSucessHttpCodes().contains(status.value()),
          ClientResponse clientResponse -> Mono.empty()
      )
      .onStatus(
          HttpStatusCode status -> !requestTemplate.getSucessHttpCodes().contains(status.value()),
          ClientResponse clientResponse -> clientResponse.bodyToMono(String.class).defaultIfEmpty( defaultV: "").flatMap( String errorBody -> {
            logger.warn("Http status: {}, response body: {}, headers: {} when requesting {}",
                clientResponse.statusCode(), errorBody, clientResponse.headers().asHttpHeaders(), jdkRequest.uri());
            return Mono.error(new HttpServiceException(mapErrorType(clientResponse.statusCode().value())));
          })
      )
      .bodyToMono(returnType) Mono<T>
      .onErrorMap(DecodingException.class, DecodingException e -> {
        logger.warn("Unable to parse response {}", e.getMessage());
        return new HttpServiceException(ErrorType.UNABLE_TO_PARSE_RESPONSE);
      });

  if (requestTimeout != null) {
    responseMono = responseMono.timeout(requestTimeout);
  }
```

Fig. 6. Code Snippet from the execute method

Additionally, to perform requests, built in methods in WebClient were added and its fluent API was used to chain them together. The methods onStatus and onErrorMap, in Fig. 6, was mainly added to handle errors declaratively. The first method was used before parsing through the HTTP codes, to allow for custom success codes and catch error codes, and the second method was used after parsing to catch JSON errors. To automatically parse JSON and map it to a defined type the bodyToMono method was used. Moreover, fault tolerance with onErrorResume was added to handle single faults in the stream to retain the other requests in that batch.

Finally, to manage timeouts from the HttpRequest the native timeout handling for Mono was added and chained to the response. This allows the reactive framework to drop the connection dynamically.

5.1.2. Deprecated Components and Code Reduction

The use of a modern, reactive and Spring native HTTP client meant that many lines of code that are present in the provided HTTP client were replaced by reactive methods. The Java native HttpClient has less connection to Spring and is comparatively more bare-bones,

needing to handle more logic manually. Most of that logic the reactive framework handles internally.

The provided Java HttpClient, the AsyncClient, has a dedicated ForkJoinPool and an ExecutorService to handle asynchronous requests and JSON parsing. The reactive WebClient replaced these services by utilizing Reactor Netty's LoopResources internally in the background, which manages workers dynamically based on available processor cores.

Another benefit of having Reactor Netty as the underlying HTTP client is that the manual rolling pool algorithm, present in the AsyncClient, to manage DNS updates and connection idling, is also not necessary. That is now handled natively via the ConnectionProvider in the configuration file for WebClient, see Fig. 7.

Fig. 7. Code snippet from Reactor Netty's ConnectionProvider

Additionally, several helper methods could be entirely removed as WebClient absorbs their functionality:

- getBodyStream: Manual GZIP decompression is no longer needed, as WebClient handles this internally.
- getBody: Translating byte arrays to Strings during error handling is now managed directly through the onStatus method.
- Response processing & validation: The standalone methods for these tasks were eliminated, as WebClient handles them declaratively within the main execution chain.

5.1.3. Retained System Architecture and Interfaces

There were some code that stayed identical to maintain compatibility with IKEA's system or make it more straightforward for their engineers to understand the WebClient implementation.

The method to execute HTTP requests still accept two parameters, RequestTemplate and Class<T> returnType that are IKEA specific and based on a template specification. Additionally, the function to translate normal HTTP status-codes into domain specific once was also kept alike.

Lastly, there was one more method that was preserved and translated over to the WebClient implementation, getResponse. To utilize the reactive framework efficiently the application ought to maintain reactiveness from the HTTP client and all the way up to the controller and return a Mono or Flux. The preserved method getResponse is called at the end and uses block to wait for response and handle faults. The reactive chain breaks here to retain compatibility as the logic higher up return to be synchronous.

5.2. HTTP Client Test Results

In this section, results from tests run locally and with K6 are disclosed in two separate chapters.

5.2.1. Local tests

Local tests were conducted according to 4.7.1. Results shown in chapters are divided by a common factor such as concurrency. Testing variables for the different factors are shown in Table II in 4.7.1.

Firstly, the initial figure (Fig. 8, 13, 17, 22, 27) in each chapter shows number of requests per second (RPS) for each factor's different levels. For the second figure (Fig. 9, 14, 18, 23, 28) in each chapter the client's response time in the 90^{th} percentile is shown. Further, for chapters *Concurrency, Processing delay* and *Payload size* the third figure (Fig. 10, 19, 24) shows total time for the completed tests for each client. Moreover, all chapters except *Stress test* have one figure (Fig. 11, 15, 20, 25) showing the maximum measured

memory used during each test and one figure (Fig. 12, 16, 21, 26) showing a calculated value for requests per second per megabyte of memory used.

Finally, since RestTemplate is, except in the *Threads* tests, at a disadvantage against the asynchronous clients and by that is not relevant for comparison against them. Because of this irrelevance results from RestTemplate are not described as thoroughly.

Concurrency

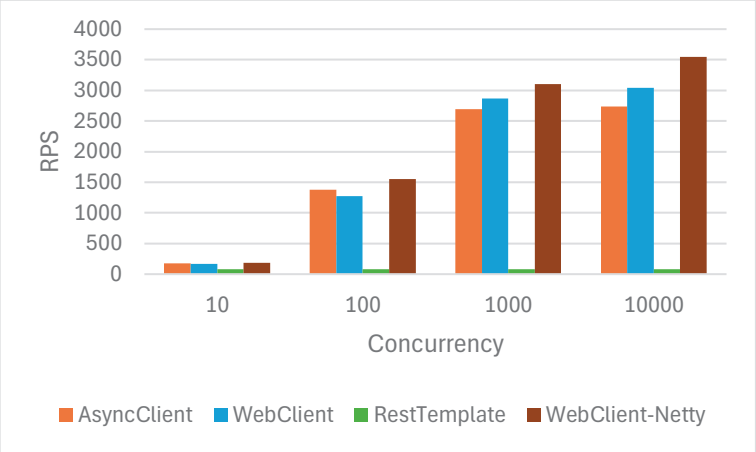


Fig. 8. Chart of RPS by concurrency

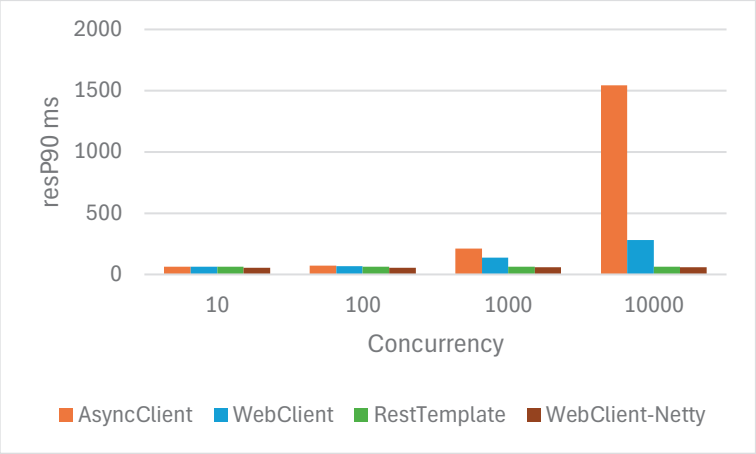


Fig. 9. Chart of response time in the 90th percentile by concurrency

Fig. 10. Chart of total time by concurrency

Fig. 8 shows that both WebClient with tomcat and Netty were almost exclusively faster than the async client, the Netty version was up to 17% faster than the WebClient without and up to 30% faster than the async client. With only high concurrency set, RestTemplate is much slower as seen in Fig. 10.

Regarding response times as seen in Fig. 9, the AsyncClient showed a substantial increase with concurrency set at ten thousand. Moreover, WebClient with Netty had consistently lower response times than without Netty.

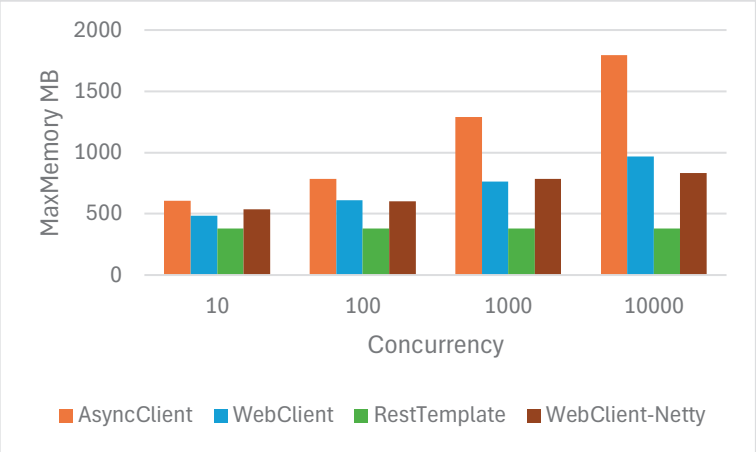


Fig. 11. Chart of maximum used memory by concurrency

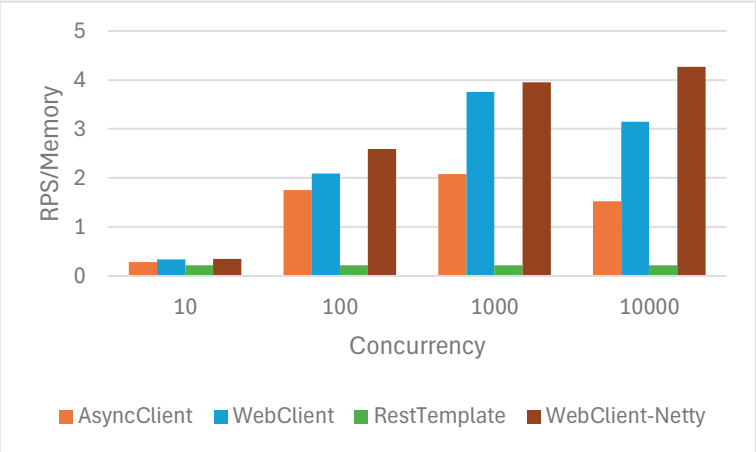


Fig. 12. Chart of RPS per MB of used memory

Memory and the efficiency of its use are shown in Fig. 11 and 12. Here, the WebClient with Netty is up to 36% more efficient than without Netty and up to 180% more efficient than the AsyncClient.

Threads

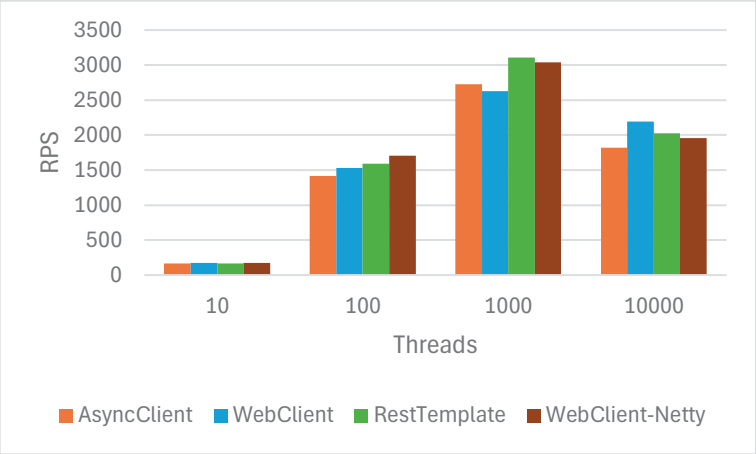


Fig. 13. Chart of RPS by number of threads

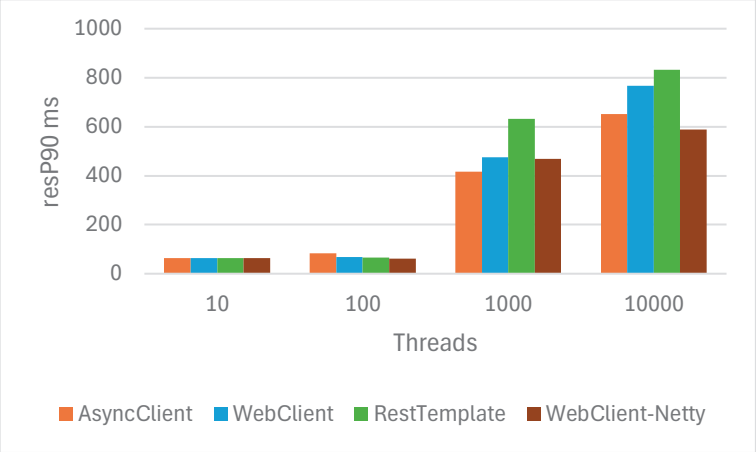


Fig. 14. Chart of response time in the 90th percentile by number of threads

Fig. 13 reveals that with a high number of threads all clients are almost equal regarding requests per second. However, the AsyncClient is slowest in all but one scenario.

Concerning the response times, Fig. 14 displays a consistently higher response time for RestTemplate and similarly consistent lower response time for WebClient with Netty.

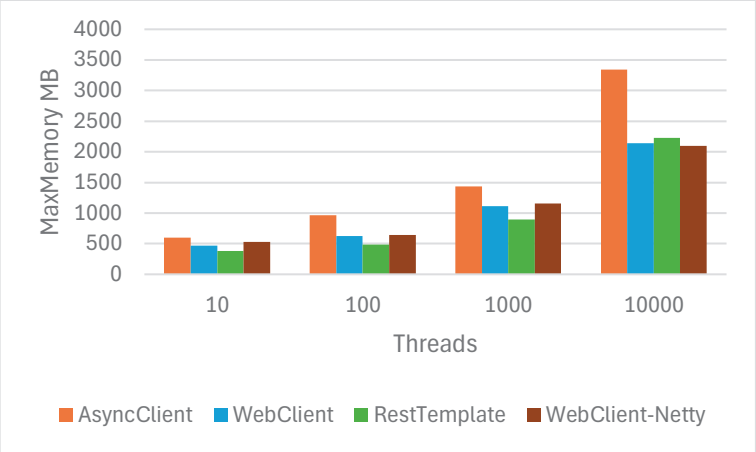


Fig. 15. Chart of maximum used memory by number of threads

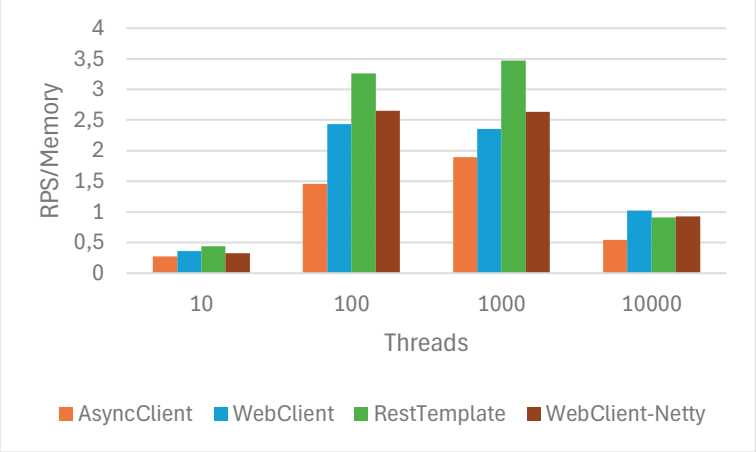


Fig. 16. Chart of RPS per MB of used memory

Finally in Fig. 15 it is revealed that the AsyncClient uses more memory than the other clients, up to 59% more than WebClient with Netty. Further, since the AsyncClient also had lower overall RPS it resulted in a lower memory efficiency seen in Fig. 16.

Processing delay

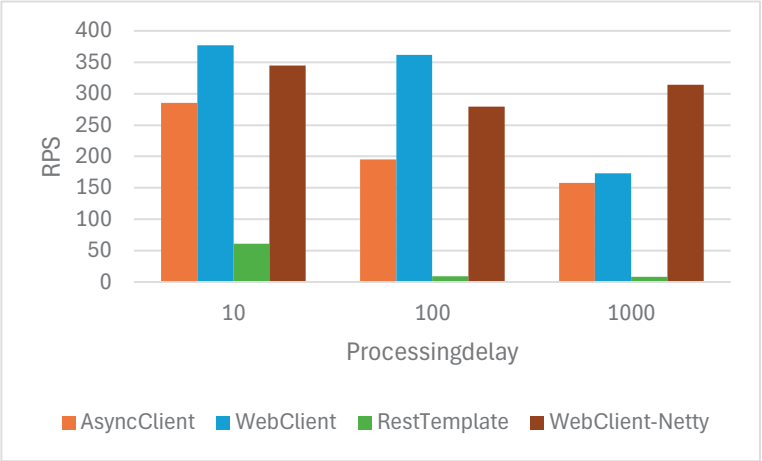


Fig. 17. Chart of RPS by length of processing delay in ms

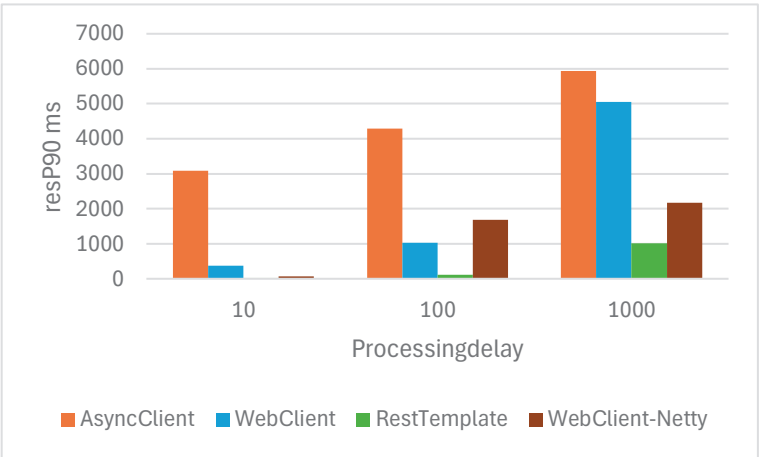


Fig. 18. Chart of response time in the 90th percentile by length of processing delay in ms

Fig. 19. Chart of total time by length of processing delay in ms

For the tests with differing processing delays, Fig. 17 shows a throughput advantage for WebClient without and with Netty, for the highest delay. This advantage is up to 85% over the AsyncClient. With only high concurrency set, RestTemplate is much slower as seen in Fig. 19.

A similar story with WebClient is told by Fig. 18 regarding response times. Especially WebClient with Netty has an advantage at higher delays.

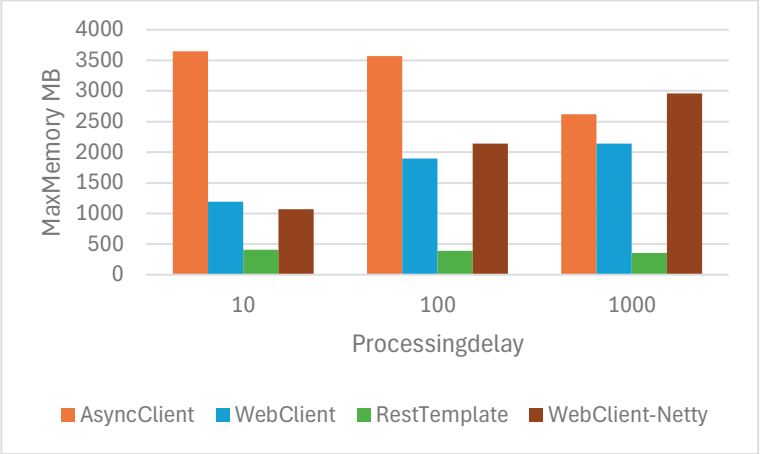


Fig. 20. Chart of maximum used memory by length of processing delay in ms

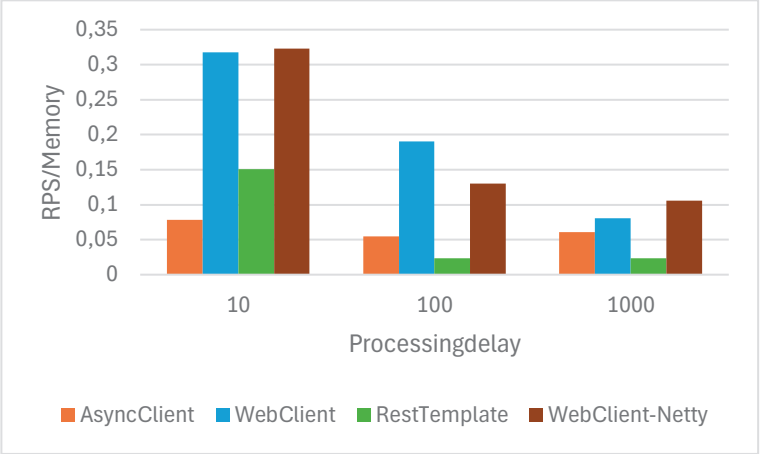


Fig. 21. Chart of RPS per MB of used memory

Lastly, Fig. 20 displays that the AsyncClient uses more memory when the delay is lower. However, the disadvantage disappears when the delay is increased. Furthermore, Fig. 21 shows a similar phenomenon with the results, evening out with higher delays.

Payload size

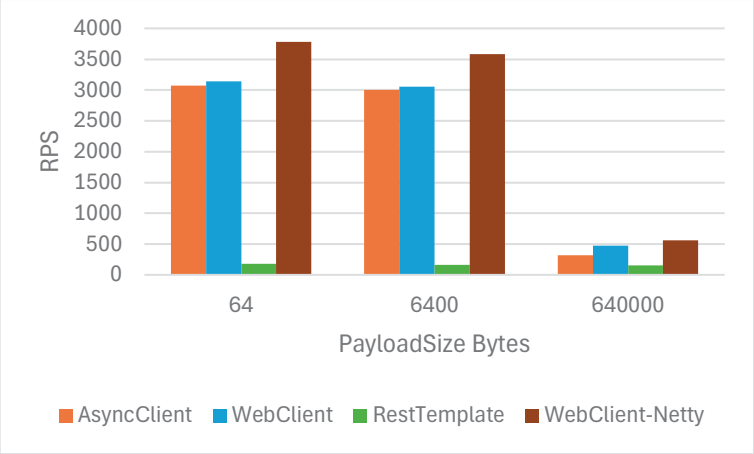


Fig. 22. Chart of RPS by payload size in bytes

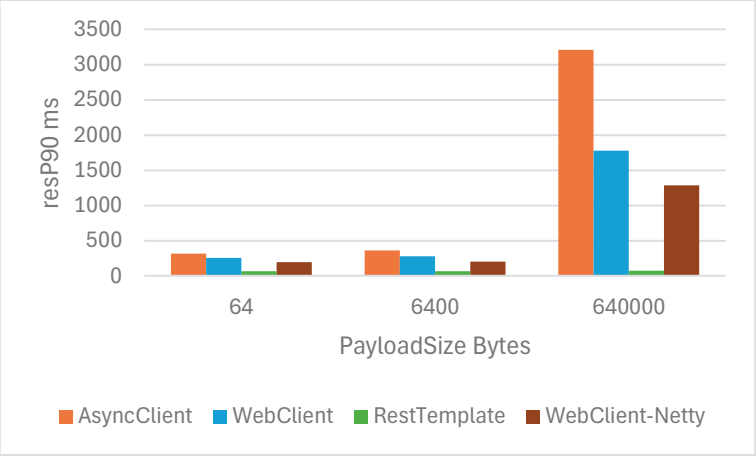


Fig. 23. Chart of response time in the 90th percentile by payload size in bytes

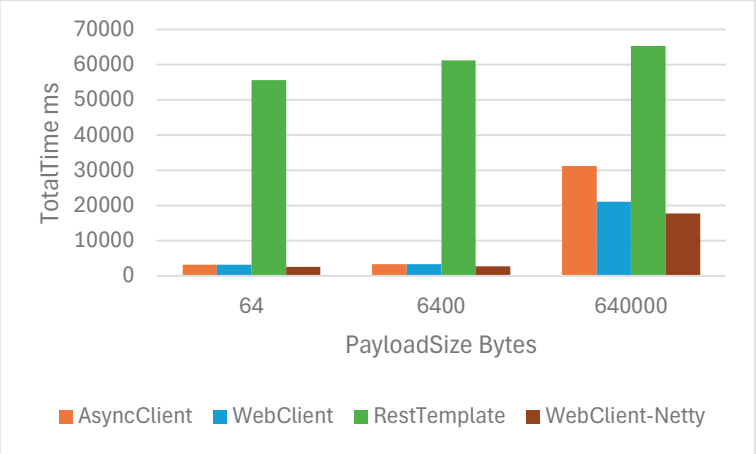


Fig. 24. Chart of total time by payload size in bytes

In Fig. 22 it clearly shows that WebClient with Netty have a higher RPS than the other asynchronous clients. However, both the AsyncClient and WebClient without Netty are equal. With only ten threads and high concurrency set, RestTemplate is much slower as seen in Fig. 24.

Regarding response times, WebClient with Netty is up to 185% and 38% lower than the AsyncClient and WebClient respectively, as seen in Fig. 23.

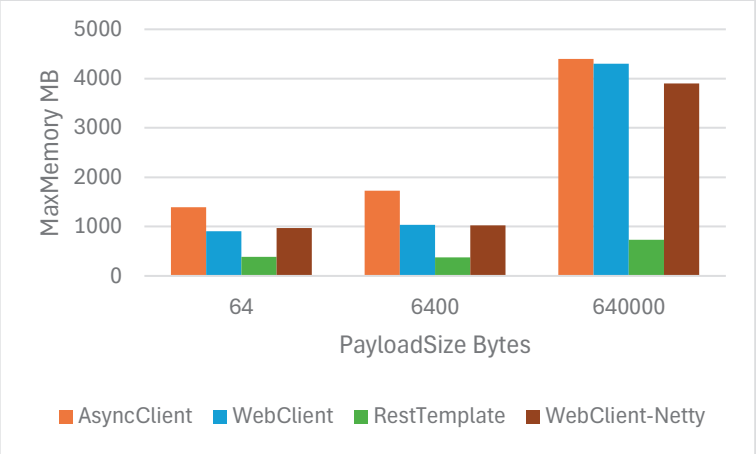


Fig. 25. Chart of maximum used memory by payload size in bytes

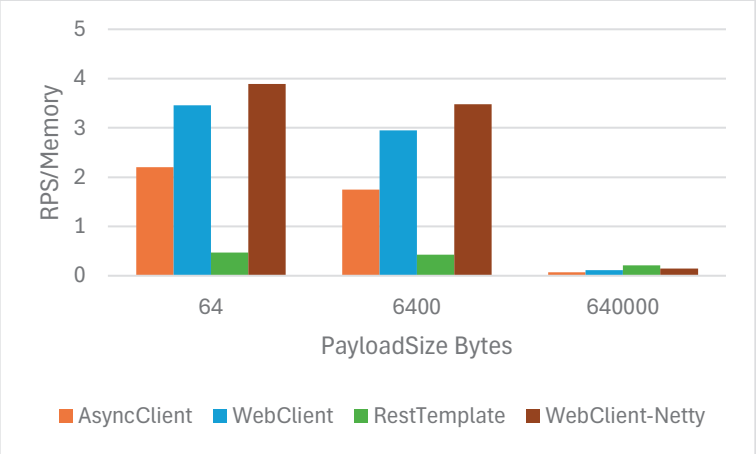


Fig. 26. Chart of RPS per MB of used memory

Finally, when looking at memory usage in Fig. 25 and 26 it reveals higher usage for the AsyncClient within the asynchronous clients.

Stress test

In this test with an increasing number of requests sent to the clients, RestTemplate was not tested above 100000 requests.

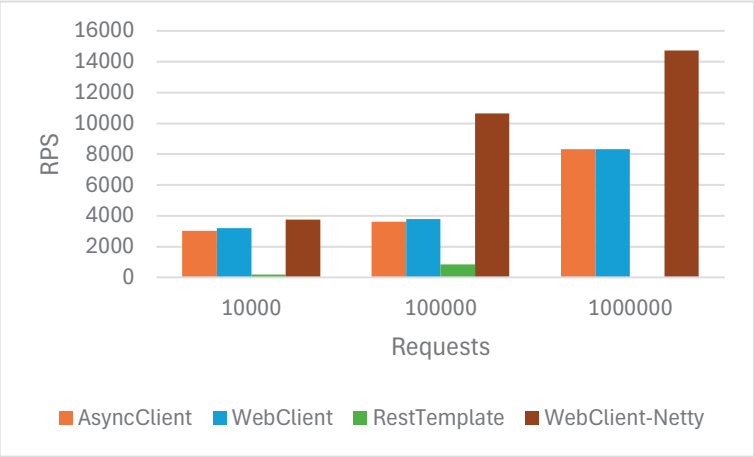


Fig. 27. Chart of RPS by number of requests sent

Fig. 28. Chart of response time in the 90th percentile by number of requests sent

Test (Fig. 27) showed a maximum of ~8333 RPS for both the AsyncClient and WebClient without Netty as a webserver. Furthermore, the response times were measured to be lower for the WebClient with Netty.

5.2.2. K6 tests

The external load tests utilizing k6 were conducted according to the methodology outlined in Section 4.7.2. The results are divided into four main categories, *Request Duration*, *Memory Consumption and CPU Usage*, *Thread Utilization*, and *Error Handling* to evaluate different performance aspects of the complete system. Throughout these tests, RestTemplate, AsyncClient and WebClient were evaluated. In the thread constraint tests and error handling tests, a simulated reactive implementation of WebClient (WebClient (Reactive)) within a traditional Spring MVC environment, was also introduced as a baseline for comparison.

Request Duration

The K6 tests analyzed the request duration under varying loads to evaluate the performance of the different HTTP clients. The tests measured the average response time, the 90th percentile (p90) response time and the systems throughput in requests per second (RPS).

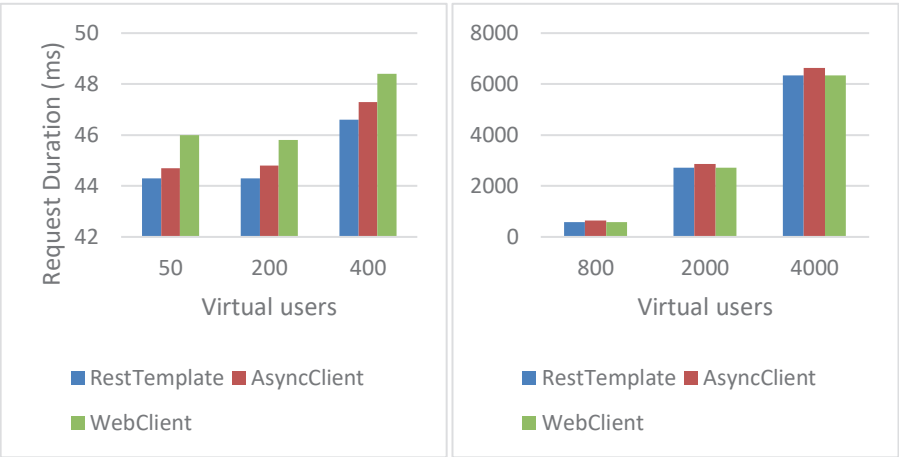


Fig. 29. Chart of average request duration across varying numbers of virtual users

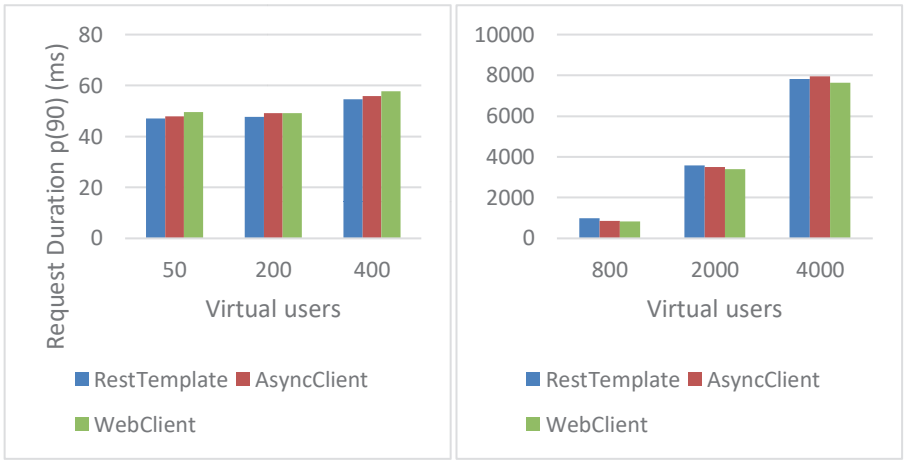


Fig. 30. Chart of the 90th percentile request duration across varying numbers of virtual users

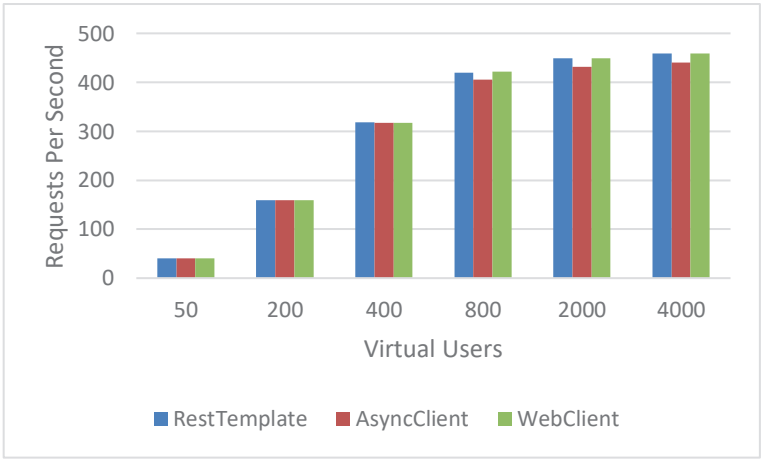


Fig. 31. Chart of average RPS across varying numbers of virtual users

As illustrated in Fig. 29, the average request duration from 50 to 400 virtual users (VUs) showed similar results across all clients, maintaining a low and stable duration of approximately 45 milliseconds. However, as the load increased from 400 to 800 VUs, a significant spike in request duration occurred across all clients. This degrading trend continued up to 4000 VUs. While the performance degradation was uniform, AsyncClient exhibited a slightly higher

average request duration at peak loads, which corresponds to its marginally lower RPS from 800 to 4000 Vus, as seen in Fig. 31.

The 90th percentile results shown in Fig. 30 closely mirror the average request durations, following the exact same trend across all clients from 50 to 4000 Vus without any major deviations.

Lastly, Fig. 31 demonstrates that throughput (RPS) remained similar for all clients up to 400 Vus. As the request duration increased after 800 Vus, the throughput stagnated, flattening out at approximately 450-460 RPS for all clients, with AsyncClient performing slightly slower than the others.

Memory Consumption and CPU Usage

To determine the resource efficiency of the different HTTP clients, memory consumption and CPU usage were analyzed under varying loads. The metrics gathered represent the maximum total memory consumed and the average CPU usage of the Java Virtual Machine (JVM).

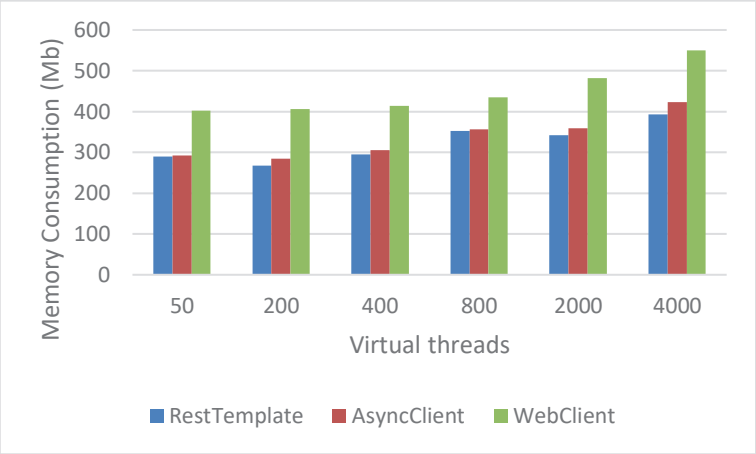


Fig. 32. Chart of memory consumption across varying numbers of virtual users

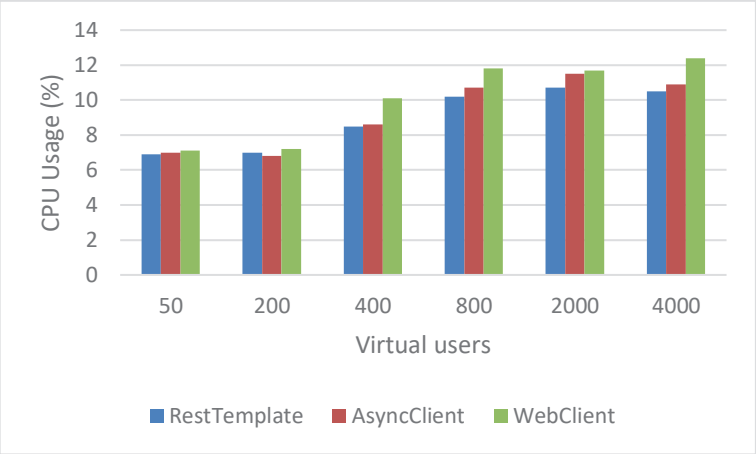


Fig. 33. Chart of CPU usage across varying numbers of virtual users

Fig. 32 highlights a distinct difference in memory utilization. The memory consumption of WebClient was constantly higher, using approximately 120 Mb more memory than both RestTemplate and AsyncClient across all tested VU configurations.

Regarding CPU utilization, Fig. 33 shows that WebClient required slightly more processing power across all tested VU configurations. The CPU usage stagnates across all clients over 800

virtual users which aligns with the stagnating RPS at the same load in Fig. 31.

Threads

To determine how the HTTP clients performed under thread starvation, tests were conducted to analyze request duration, Tomcat thread utilization, memory consumption, and CPU usage under severely constrained thread limits. For these specific scenarios, a fourth client was introduced, the WebClient (Reactive).

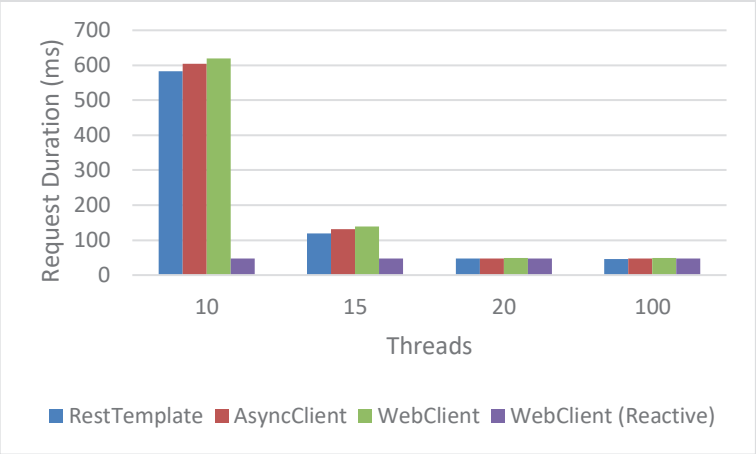


Fig. 34. Chart of request duration under constrained thread conditions

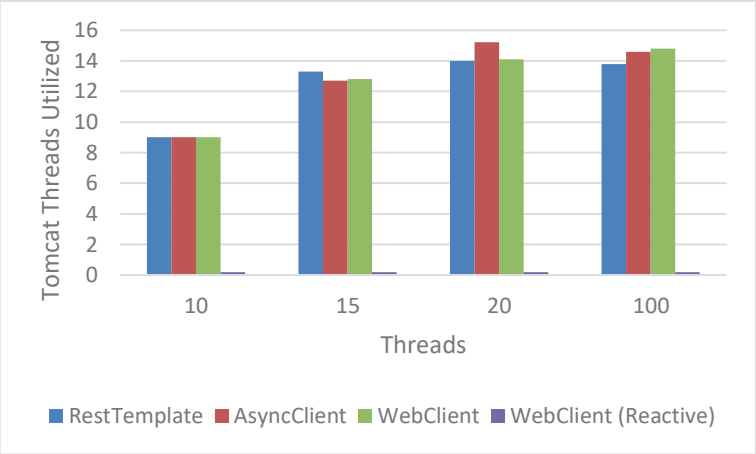


Fig. 35. Chart of Tomcat thread utilization under constrained thread conditions

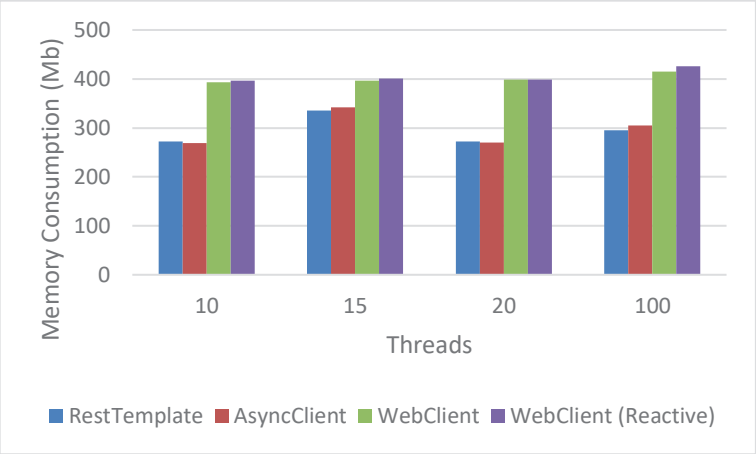


Fig. 36. Chart of memory consumption under constrained thread conditions

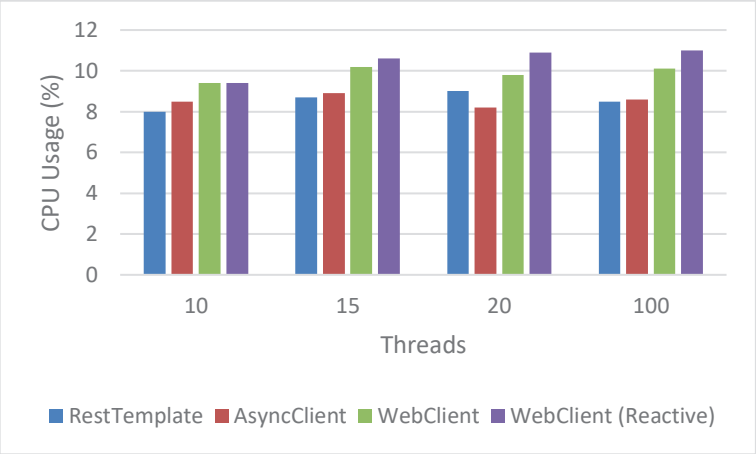


Fig. 37. Chart of CPU usage under constrained thread conditions

As seen in Fig. 34, the request duration for the fully reactive WebClient (WebClient (Reactive)) remained completely stable at around 45 milliseconds across all thread limited tests. The standard clients also maintained a 45-millisecond duration at the 20 and 100 thread limited scenarios. However, when the thread pool was restricted to 15 and 10 threads, all non-reactive clients experienced severe performance degradation. Request durations spiked to approximately 130 milliseconds at 15 threads, and up to 600 milliseconds at 10 threads.

Fig. 35 illustrates the Tomcat thread utilization during these scenarios. The fully reactive implementation utilized an average of only 0.2 threads regardless of the imposed limit. In contrast, the non-reactive clients utilized nearly all available resources, maxing out at an average of 9 threads (when limited to 10), 13 threads (when limited to 15), and approximately 14.5 threads (when given 20 or 100 available threads).

In terms of memory, Fig 36 shows that both WebClient and WebClient (Reactive) consumed roughly 110 Mb more memory than RestTemplate and AsyncClient across most constrained scenarios, though this gap narrowed to 60 Mb at the 15 thread mark. Furthermore, memory consumption saw a uniform increase of 20-30 Mb across all clients when the thread limit was raised to 100 threads.

Finally, Fig. 37 indicates that both WebClient implementations had a marginally higher CPU usage (by approximately 1pp) across all thread limited scenarios compared to RestTemplate and AsyncClient, with the reactive WebClient constantly having the highest CPU usage overall.

Error Handling

The final k6 test scenario focused on error handling resilience by analyzing RPS, request duration, resource utilization, and failure rates under an induced fault scenario.

TABLE VI. METRICS ACROSS CLIENTS WITH AN INDUCED 10% RATE OF COMBINED SLOW AND FAILED REQUESTS

Metric	Rest Template	Async Client	Web Client	WebClient (Reactive)
RPS	158.2	150.5	156.8	156.6
Request Duration (ms)	1090	1210	1120	1120
Request Duration p(90) (ms)	2700	2860	2770	2780
CPU Usage (%)	8.1	8.5	9.1	9.4
Memory Consumption (MB)	288.5	290.1	431.5	488.4
Total Requests	28757	27246	28388	28348
Failed Requests	1687	1578	1669	1662

As detailed in Table VI, the introduction of these simulated faults did not yield any distinct deviation in overall system performance between all clients. Apart from the increase in average request duration, compared to the baseline successful requests across all clients.

6. Discussion

In this section, the thesis results, methodology, limitations and ethical aspects are discussed. The results discussion is divided into a discussion about implementation and a discussion about performance.

6.1. Implementation interpretation

This chapter will present the thesis workers interpretation of the implementation and its results presented in 5.1.

6.1.1. Ease of implementation

The WebClient implementation proved to be successful. All tests passed, and an analysis across all tested scenarios revealed no significant aberrations that would indicate a subpar or faulty implementation. The error handling scenario demonstrated that, even with induced faults, the implementation performs consistently with its baseline results and shows similar characteristics to the other HTTP clients.

Integrating the reactive WebClient into the existing Spring MVC architecture proved to be straightforward in terms of configuration, with the help of Java's HttpClient as reference. Thanks to the already implemented asynchronous client setting up the WebClient config file to match its functionality was uncomplicated using Spring WebFlux.

However, it was challenging to mix and adapt IKEA's long-standing synchronous system with Java's standard library functions, such as HttpRequest, into the WebClient implementation.

A notable advantage of the WebClient implementation was the reduction of boilerplate code. An important factor was to preserve functionality and manual control. Despite this, plenty of manual handling described in chapter 5.1.2 could be removed and offloaded to Netty, that operates as engine beneath Spring WebFlux.

Furthermore, the fluent declarative API of WebClient greatly simplified error handling and response validation. The chaining of methods helped with tracking the request and response through the system. It made it quick to add in or exclude features. At a late stage

of implementation, functions such as onStatus and doOnError, could be inserted exactly where they needed to be in the chain without further modifications, with only a few lines of code.

However, the most significant obstacle during implementation was understanding how to shift from Java's CompletableFuture to Project Reactor's Mono and Flux. It was challenging to understand reactive streams. The learning curve was steeper than that of sequential programming, to learn how to find, differentiate and use suitable functions such as flatMapSequential to retain response ordering instead of using flatMap or concatMap.

Finally, the current WebClient implementation functions only as a GET client, as seen in Fig. 5, and does not process the body of incoming HttpRequest objects. This design choice was intentional, and the decision was taken following guidance from the supervisor at IKEA. It was deemed unnecessary for the scope of the target system. It is also worth noting that the provided RestTemplate implementation similarly excluded body processing.

6.2. Performance interpretation

This chapter will present the thesis workers interpretation of the performance tests presented in 5.2.

6.2.1. Local tests

This sub-chapter is split into separate sections corresponding to the layout of chapter 5.2.1.

Concurrency

Firstly, it's clear that WebClients reactive asynchronous behavior has a higher throughput and uses less memory than the custom-built AsyncClient. This can likely be attributed to WebClient's reactive architecture, which requires significantly fewer active threads to perform the same amount of work by using event loops. Consequently, the host running the client does not need to handle the same amount of running threads thereby it uses less memory and has more allocated time for other than thread context switching [10].

Furthermore, it can be observed that the AsyncClient's response time increases substantially when increasing concurrency. This increase strongly supports the reasoning that a higher number of threads leaves less available processing and memory for handling requests. This result mirrors what Zbarcea and Tudose [25] learned in their test, "Memory Consumption vs Concurrency", when they observed a higher memory usage and response time for the built-in Java client over all concurrencies.

Moreover, the same reactive benefit can be observed using Netty as the underlying webserver. Since Netty is a reactive server and is non-blocking, it naturally augments the benefits of less threads.

Finally, the increasing lead for the reactive client and server further strengthens the argument of less threads is more effective and scalable.

Threads

The tests with a differing number of threads firstly show that the number of requests per second (RPS) for the asynchronous clients are consistent with the concurrency test up to one thousand threads. However, at higher thread counts, the RPS is notably lower than the corresponding number of concurrency. This observation indicates that the higher number of threads uses more processing power for context switching [10] thread handling.

Moreover, the AsyncClient has less throughput than with the corresponding concurrency which can likely be attributed to an even greater number of threads in the threads test.

Furthermore, the results from the natively synchronous client, RestTemplate, implies that when forcing synchronous behavior on the asynchronous clients their overhead, for the concurrency handling, creates a larger use of memory and therefore less efficiency for RPS per memory unit.

Moreover, the response times present a contradiction as the RestTemplate has the highest response times even though it has the highest throughput. However, this observation confirms that asynchronous behavior is being performed by the non-blocking

clients in that they release threads to be used for other purposes when not needed. On the other hand, RestTemplate occupies its threads for the entire request and response process. Thus, the observation is not a contradiction but a confirmation that more active threads, which means more processing contention and context switching, leaves less room for processing requests and responses.

Finally, these findings strongly suggest that reactive architectures are not universally superior in high thread and low concurrency scenarios.

Processing delay

To start, the reactive WebClient is shown to outperform the AsyncClient thus implying an advantage for its reactive thread request handling. The lower throughput of the AsyncClient can likely be attributed to its underlying lack of backpressure mechanisms. The lack of the feature mentioned is also likely a contributing factor for its higher response times and memory usage. This can be explained by the continuous sending of requests that the client cannot handle and thereby uses up all the available memory.

Moreover, Netty's apparent advantage as webserver is at its greatest in this test with the longest delay. This suggests that Netty's event-loop [10] architecture allows it to park pending requests asynchronously in memory and continue to handle other requests. On the other hand, the tomcat server does not utilize an event-loop architecture and thus binds each request to a dedicated thread.

Payload size

Firstly, it is observed that through all levels of payload, the WebClient with Netty had the highest throughput. This result suggests that payload size does not in a significant way affect relative RPS as the clients relative RPS are similar to the concurrency test.

Furthermore, the AsyncClient's memory usage is higher although the difference to the WebClient dwindles for larger payloads. The decreasing difference in memory usage suggests that the higher initial

difference is connected to its higher number of threads, i.e. overhead and the lower difference hides this by the payload using a larger relative part of the available memory.

Stress test

Initially for a low number of requests, the divergence for the asynchronous clients is small. Although, for a greater number of requests the divergence soars. Both clients using tomcat hit a maximum of approximately 8300 RPS while WebClient with Netty reaches over 14000 RPS. The result strongly implies that the bottleneck is the tomcat server and how it requires a dedicated thread for each request.

Moreover, the WebClient with Netty appears to experience a paradox with decreasing response times. However, this decrease could have a cause in optimization by the compiler as it has a longer time to do optimizations.

6.2.2. K6 tests

This sub-chapter is split into separate sections corresponding to the layout of chapter 5.2.2.

Request Duration

The significant increase in request duration observed across all clients from 800 VUs, combined with the stagnating throughput that occurs in tandem, strongly suggests a system bottleneck. To support this hypothesis, the clients show no significant performance deviation between them, apart from a slight decrease in throughput for Java's AsyncClient. This bottleneck likely originates from the induced delay at the WireMock server, where an overwhelming volume of requests ultimately forms a queue, resulting in a spike in request duration across the board. Further testing with different variables is warranted to determine the exact cause of the bottleneck in this specific scenario with higher certainty.

However, the results up to 400 VUs remain viable for comparison. They show RestTemplate having consistently the lowest request duration, followed by AsyncClient, with WebClient in last place. Considering that the tests ran under low to medium load

without fully stressing the system, it is reasonable to assume these results reflect client side overhead. This difference indicates that WebClient introduces more latency than RestTemplate. To support this, according to Spring, WebFlux is designed for scalability rather than individual request speed [4]. The overhead observed is likely a combination of the internal processing required to build reactive streams [7], and the cost of bridging the asynchronous WebClient with the synchronous Tomcat thread pool. This also suggests that incorporating WebClient into a synchronous environment does not add any measurable overhead.

Memory consumption and CPU usage

Interestingly, the total JVM memory consumption for WebClient was consistently higher than that of the other clients. This contradicts the local tests in 5.2.1and the findings of Zbarcea and Tudose [25], who discovered that a fully reactive system utilizing WebClient consumed less memory than an asynchronous system using Java's HttpClient.

The discrepancy might lie in how Tomcat was utilized across the environment. In the local tests, Tomcat acted merely as an isolated mock API receiver, allowing WebClient to operate without resource competition. In the k6 tests, however, Tomcat served as the active entry point for all incoming requests, forcing Tomcat's synchronous threads and Netty's reactive event loops to interact directly within the same execution path.

Based on these observations, we hypothesize that incorporating a reactive client into a traditional Spring MVC architecture creates an architectural mismatch. Running Netty's independent event loop [10] alongside Tomcat results in dual active thread pools. When combined with Netty's tendency to aggressively pre-allocate memory buffers to handle high concurrency [9], this framework interaction significantly elevates memory consumption. However, due to time constraints, further investigation to confirm this hypothesis was not possible within the scope of this thesis.

Additionally, CPU usage remained higher for WebClient. This also diverges from the results presented by Zbarcea and Tudose [25], which showed a decrease in CPU usage for a strictly reactive

architecture. This elevated CPU usage could also be caused by the dual execution environments. Further research to fully understand this behavior is warranted.

Threads

The average request duration increases drastically when limited to a low number of available Tomcat threads. However, this also depends on the number of virtual users or more precisely on the RPS. The data from this test scenario reveals that the number of threads needed to make this variable not affect the results is roughly 20 threads. Raising the available threads further for this scenario does not increase performance.

At a reduced thread limit of 15 and 10 threads the number of utilized threads is at or close to that limit for all tested clients except for WebClient with the reactive request chain, which had a utilization of 0.2 threads. This causes it to sustain its low average request duration while the other clients, the normal WebClient included, show a drastic increase as the thread limit falls from 20 down to 10 threads. This suggests that integrating WebClient into a synchronous environment does not automatically yield the theoretical advantages described by Spring [4]. To utilize fewer threads, allocate less memory and reduce CPU overhead caused by constant context switching [10], the entire system needs to be reactive. This aligns with the findings of Zbarcea and Tudose [25], who demonstrated these exact resource efficiencies when evaluating a fully reactive architecture.

Lastly, focusing on memory consumption and CPU usage between the scenarios with limited threads, the results show that the reactive WebClient has similar or marginally higher resource utilization than the normal WebClient. This shows that the simulated system reactivity of the reactive WebClient does not avoid the elevated memory consumption and CPU usage that was previously observed with WebClient. This further points to Tomcat with Netty being the main reason for the higher resource consumption.

Error Handling

The results from the error handling scenario indicate that all tested clients manage induced faults with similar resilience. The lack of significant performance deviation between the clients is a positive outcome for all WebClient implementations. As it suggests that declarative error management handles errors just as efficiently as the provided clients. Moreover, this suggests that integrating WebClient does not introduce any unexpected vulnerabilities or overhead when the system handles errors.

6.3. Method reflection

In this chapter, a reflection on the thesis's methodology is made. The chapter is split into three sections corresponding to the work, implementation and testing.

6.3.1. Reflection on the thesis work

During the entire thesis an order of work was followed (see 4.1.1.) that allowed the thesis workers to focus on the correct work without distractions. This way of working made the work move forward and kept IKEA informed of the current and future steps.

Furthermore, during the implementation IKEA's approach of using git was applied to all code related work. This practice was instrumental in integrating with the IKEA engineering team and thus getting their feedback.

6.3.2. Reflection on Implementation

Firstly, the initial implementation of WebClient may have focused too heavily on matching the AsyncClient's structure for compatibility. As a result, better reactive design patterns were initially overlooked. However, this was later corrected and adjusted.

Similarly, the strong focus on compatibility and simplifying the understanding of both the provided system and reactive programming, due to an initial unfamiliarity with these two areas, resulted in IKEA-specific logic initially not being translated into

reactive components. However, this was also reworked and translated at a later stage.

6.3.3. Reflection on testing

To start, all testing was completed on seperate computers and not via an external network connection. Zbarcea and Tudose [25] used two machines connected over LAN which might have given more realistic results and a chance to vary the network conditions.

Moreover, the choice of values for the local tests are not as extensive as they could be for an even greater insight into performance. However, considering time for completion of the tests and length of the results, the values chosen were a sound compromise.

6.4. Ethical aspects of the thesis

Firstly, the ethical implications of this thesis are somewhat hidden as the goal was an evaluation, not a delivery of any code or program. However, certain aspects are still of outmost interest for an ethical evaluation.

One of the ethical aspects is from an environmental view, where the thesis's possible inspiration for the implementation of a faster and less resource intensive HTTP client results in a possible reduction in energy usage.

Otherwise, any ethical aspects are those who arise from the potential individuals using this thesis's results to implement any form of system.

6.5. Limits of the thesis

To start, tests were run completely on two local machines and were restricted by those machines computing power. Furthermore, the operating system could not be altered and might have impacted some results related to thread handling.

7. Conclusions

In this chapter, the thesis workers' conclusions will be presented. Starting with a summary of the thesis and its results followed by the problem questions and answers, uses of work and future work.

7.1. Thesis Summary

This thesis evaluated and researched the use of a new asynchronous HTTP client, WebClient, in the place of IKEA's custom made AsyncClient and RestTemplate. The areas focused on were implementation and performance comparisons. Results from the focused upon parts were presented and discussed with conclusions presented in this chapter.

7.2. Problem questions answers

In this section an answer or explanation for each of the thesis's problem questions will be given. The problem statement 4 refers to the fourth problem statement in Chapter 1 and problem statement 7.2.4.

7.2.1. How do the current HTTP clients work in the mock-up backend?

The clients used by IKEA, AsyncClient and RestTemplate, where used in a Spring Boot system to send HTTP requests from a service repository.

7.2.2. What modifications are needed in the mock-up backend for the WebClient HTTP client to be integrated?

During the thesis work it became clear that no modifications in the mock-up backend were needed for the WebClient to be integrated. However, as mentioned in 4.8 the backend would have to be modified to handle reactive requests and responses, i.e. Mono and a reactive webserver, to achieve the full benefit of the reactive WebClient.

7.2.3. Which modifications with the new HTTP client are needed for compatibility with IKEA's provided system?

To start with, no actual modifications needed to be made for the new client to work with the Spring Boot system provided. However, as discussed in 6.2.2 full utilization of the reactiveness of WebClient could not be achieved without the potential modification to a fully reactive chain in all system parts. These modifications would include changing HTTP responses to Mono or Flux and changing the webserver to Netty.

7.2.4. What measurement data should be collected to be able to compare the performance of HTTP clients?

Firstly, IKEA made it clear that it wanted to understand the implications of implementing WebClient. This was measured in differences for code regarding reactive programming, less or more code and different methods used.

Furthermore, what measurements of performance were to be collected were discussed and agreed with the engineers at IKEA. These measurements were; Requests per second, response time and system resource usage.

Finally, the thesis workers decided to, for the improvement of the results, collect further measurements such as total time and number of errors in some circumstances.

7.2.5. What is the performance of the current clients according to problem statement 4?

Performance for the AsyncClient and RestTemplate was thoroughly tested using two different methods. These methods were the local tests and the K6 tests (see chapter 4.7 and 5.2 for methodology and results respectively).

Moreover, it was found that the AsyncClient worked asynchronously, as intended, and RestTemplate struggled to match the AsyncClient in all but the thread test.

Furthermore, one can conclude that performance from RestTemplate was in all but one local test, threads, irrelevant for comparison as no concurrency could be achieved with the synchronous client.

Finally, it can be concluded that differences between the local tests and the K6 tests were the result of number of TomCat threads, test-computer and the involvement of the entire Spring Boot system for K6. The local tests can therefore be considered a rawer performance test that is relevant outside of the provided Spring Boot system.

7.2.6. What is the performance of the alternative client/s according to problem statement 4?

For WebClient performance was measured in the same manner as the AsyncClient and RestTemplate to achieve comparability. The results showed that WebClient was generally able to maintain a higher throughput and use less memory when doing so than the AsyncClient.

Furthermore, it was revealed that pairing WebClient with a reactive webserver, Netty, and making responses reactive with Mono WebClient performed better in almost every respect (see chapter 5.2.1 for detailed results).

Finally, the differences between the two test methods can be resounded to abide by the same reasons as described in 7.2.5.

7.3. Uses of the results in back-end development and beyond

The uses in back-end development from the results consist of both integrational and further development and research. The implementation of WebClient that has been built could from the test results be used as is or be developed on further by the engineering team at IKEA to fit their needs. The test results gathered are available to their team as a starting reference point for where to utilize the developed client but also where to use their current clients, and for further research.

As for academic research, this could be used for further research into how to transition from traditional synchronous systems into reactive ones. Where the implementation built in this study and the results obtained from testing could stand as the foundation for further research. Furthermore, the results can be used for further research into reactive programming in relation to HTTP clients.

7.4. Future work

There are multiple areas that could be explored with future studies. One of these could compare different HTTP clients with a fully reactive system. This would bring knowledge concerning a more developed solution and not only an adaptation to other clients.

Another area that could be explored with future studies could compare different HTTP clients in a synchronous environment utilizing virtual threads. Virtual threads are lightweight and do not demand the system to be rebuilt. They are straightforward to implement and would solve the issue of decreasing response times at lower numbers of available threads that was seen in this study.

Furthermore, an area that could be explored with future studies could explore how the newer RestClient, which is the replacement for RestTemplate, from Spring would perform compared to the other HTTP clients in this study. Especially compared to RestTemplate, since asynchronous clients might not be the needed fit for all scenarios.

Finally, exploring different webservers and underlying HTTP clients would give valuable insight into more granular differences than the client implementations. Changing the default underlying HTTP client in WebClient would address the uncertainty around the unexpected high increase in memory consumption seen in the k6 results.

8. Terminology

In the following list, acronyms and names used in this thesis will be explained.

AsyncClient = IKEA's custom-made HTTP client based on Java's built in HTTP client

RPS = Requests per second is a measurement that shows number of completed HTTP requests per second.

VUs = Virtual users are concurrent, independent execution loops that mimic the behavior of real human users to simulate traffic.

DEXF = Team at IKEA that thesis workers collaborated with.

9. References

[1] Broadcom Inc, "Spring Framework Overview," [Online]. Available: https://docs.spring.io/spring-framework/reference/overview.html#overview-philosophy. [Accessed 08 03 2026].

[2] C. Walls, Spring in Action, Shelter Island: Manning Publications Co, 2019.

[3] C. Walls, Spring Boot in Action, Shelter Island: Manning Publications Co, 2016.

[4] Broadcom Inc, "Spring WebFlux," [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webflux.html. [Accessed 13 04 2026].

[5] Broadcom Inc, "Web MVC framework," [Online]. Available: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html. [Accessed 13 04 2026].

[6] Oracle Corporation, "Interface ExecutorService," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html.

[7] S. Maldini and S. Baslé, "Reactor 3 Reference Guide," 2024. [Online]. Available: https://projectreactor.io/docs/core/release/reference/aboutDoc.html. [Accessed 22 03 2026].

[8] Bradcom Inc, "Project Reactor," [Online]. Available:

https://projectreactor.io/.

[9] S. Maldini and V. Georgieva, "Reactor Netty Reference Guide," 2024. [Online]. Available: https://projectreactor.io/docs/netty/1.3.5-SNAPSHOT/reference/about-doc.html. [Accessed 22 03 2026].

[10] N. Maurer and M. Allen Wolfthal, Netty in Action, Shelter Island: Manning Publications Co, 2016.

[11] S. M. D. K. S. B. I. K. Sebastien Deleuze, "Class Mono<T>," [Online]. Available: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html. [Accessed 18 04 2026].

[12] S. M. D. K. S. B. I. K. Sebastien Deleuze, "Class Flux<T>," [Online]. Available: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html. [Accessed 27 04 2026].

[13] Oracle, "Class CompletableFuture<T>," [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/CompletableFuture.html. [Accessed 19 04 2026].

[14] The Apache Software Foundation, "Apache Tomcat," [Online]. Available: https://tomcat.apache.org/index.html. [Accessed 12 04 2026].

[15] The Apache Software Foundation, "Apache Tomcat 8 Configuration Reference-The HTTP Connector," [Online]. Available: https://tomcat.apache.org/tomcat-8.5-doc/config/http.html. [Accessed 8 5 2026].

[16] Broadcom Inc, "Embedded Web Servers," [Online]. Available: https://docs.spring.io/spring-boot/how-to/webserver.html.

[Accessed 12 04 2026].

[17] Amazon Web Services, "What is an API (Application Programming Interface)?," AWS, n.d. [Online]. Available: https://aws.amazon.com/what-is/api/. [Accessed 16 03 2026].

[18] Amazon Web Services, "What is a RESTful API?," AWS, n.d. [Online]. Available: https://aws.amazon.com/what-is/restful-api/#what-is-restful-api--1hfzuqn. [Accessed 16 03 2026].

[19] BrowserStack Limited, "What Is a REST API Client: Definition, Use Cases & Tools," BrowserStack, n.d. [Online]. Available: https://www.browserstack.com/guide/what-is-rest-api-client. [Accessed 29 03 2026].

[20] Broadcom Inc, "Class RestTemplate," n.d. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html. [Accessed 30 03 2026].

[21] Broadcom Inc, “WebClient,” [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webflux-webclient.html. [Accessed 08 03 2026].

[22] Oracle Corporation, "Class HttpClient," [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.net.http/java/net/http/HttpClient.html. [Accessed 11 04 2026].

[23] T. Crevon, "Understanding Grafana k6: A simple guide to the load testing tool," Raintank Inc., 11 08 2023. [Online]. Available: https://grafana.com/blog/understanding-grafana-k6-a-simple-guide-to-the-load-testing-tool/. [Accessed 10 04 2026].

[24] WireMock Inc, "WireMock Java - API Mocking for Java and JVM," [Online]. Available: https://wiremock.org/docs/.

[Accessed 12 04 2026].

[25] A. Zbarcea and C. Tudose, "Migrating from Developing Asynchronous Multi-Threading Programs in Java," *applied sciences,* vol. 14, no. 24, p. 12062, 2024.

[26] Broadcom Inc., [Online]. Available: https://micrometer.io/. [Accessed 30 04 2026].

[27] Elsevier, "AI Discovery (formerly Scopus AI)," [Online]. Available: https://www.elsevier.com/products/scopus/scopus-ai#0-about-ai-discovery.

[28] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi and M. Mezini, "On the Positive Effect of Reactive Programming," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 43, no. 12, pp. 1125-1143, 2017.

Appendix:

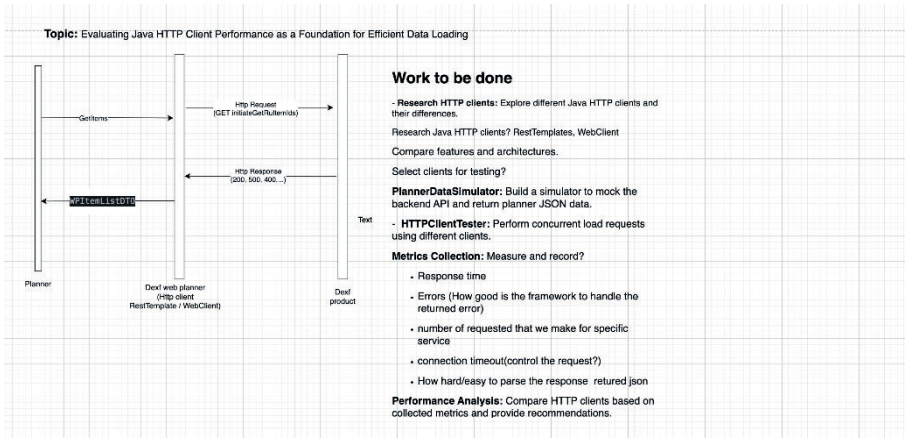


Fig. 38. Initial plan for the thesis work

Printed by Tryckeriet i E-huset, Lund 2026

LUND
UNIVERSITY

Series of Bachelor´s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2026-1139
http://www.eit.lth.se