TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Valeri Andrejev 164224IAPB

FEASIBILITY ANALYSIS OF MIGRATION FROM SPRING MVC TO SPRING WEBFLUX: A CASE STUDY

Bachelor's thesis

Supervisor: Vadim Kaparin
PhD

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Valeri Andrejev 164224IAPB

SPRING MVC-ST SPRING WEBFLUX-LE ÜLEMINEKU OTSTARBEKUSE ANALÜÜS: JUHTUMIUURING

Bakalaureusetöö

Juhendaja: Vadim Kaparin
PhD

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Valeri Andrejev

18.05.2021

Abstract

During recent years reactive programming is gaining popularity among web-application developers. A reactive application promises better performance and more stability under load. With release of Spring WebFlux it is highly advocated to try it out to see gain in performance of your team's Spring-based web-product. It is rarely possibly to start a new project when you are working in a corporation. Usually, a team is developing an application for years, adding new features and changing old ones. So, question arises:

"Is our application future-proof, should we migrate?"

Due to of the complexity of the existing application and the lack of publicly available complex migration examples, it is impossible to quickly give a definite answer to those question. To find the answer, this thesis analyses an existing production application by migrating its minimally necessary part and comparing the migrated version's performance to the old one.

This thesis is written in andEnglish is 40 pages long, including 7 chapters, 16 figures and 2 tables.

Annotatsioon

Spring MVC-st Spring WebFlux-le ülemineku otstarbekuse analüüs: juhtumiuuring

Viimastel aastatel on reaktiivne liikumine veebirakenduste arendajate seas muutunud populaarseks. Reaktiivne rakendus lubab paremat jõudlust ning suuremat stabiilsust koormuse all. Spring WebFluxi väljaandmisega soovitatakse seda katsetada, et saada tagasisidet oma meeskonna Springi-põhise veebitoote jõudlustest. Suurettevõttes töötades on harva võimalik uut projekti alustada. Tavaliselt töötab meeskond aastaid rakendust välja töötades, lisades uusi ja muutes vanu funktsionaalsusi. Seega tekib küsimus:

"Kas meie rakendus on tulevikukindel, kas on seda otstarbekas migreerida?"

Olemasoleva rakenduse keerukuse ja avalikult kättesaadavate keeruliste harva esinevate migreerimisnäidete tõttu, on võimatu nendele küsimustele kiiresti kindlaid vastuseid leida. Selles lõputöös analüüsitakse olemasolevat toodangu rakendust, et nendele küsimustele vastust leida, migreerides selle rakenduse minimaalse vajaliku osa ja võrreldes migreeritud versiooni jõudlust eelmisega.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 40 leheküljel, 7 peatükki, 16 joonist, 2 tabelit.

List of abbreviations and terms

MVC	*Model-View-Controller*—architecture for web applications.
API	*Application Programming Interface*
POJO	*Plain Old Java Object*
IoC	*Inversion of Control*
DI	*Dependency Injection*
DTO	*Data Transfer Object*
LDAP	*Lightweight Directory Access Protocol*
JPA	*Java Persistence API*
MsSQL	*Microsoft SQL* —SQL server developed by Microsoft.
JDBC	*Java Database Connectivity*
R2DBC	*Reactive Relational Database Connectivity*—reactive analogue of JDBC.
SOAP	*Simple Object Access Protocol*
JTL	*JMeter Text Logs*
VCS	*Version Control System*
IDE	*Integrated Development Environment*
CI/CD	*Continuous Integration/Continuous Delivery*
CPU	*Central Processing Unit*
RAM	*Random-Access Memory*

Table of contents

1 Introduction 10
2 Overview of Spring MVC and Spring WebFlux 12
2.1 Spring MVC vs Spring WebFlux 12
3 Tools used for analysis 14
3.1 Choice of tools for monitoring 14
3.2 Choice of testing tools 14
4 Overview of the analysed application 17
5 Migration to WebFlux 19
5.1 Presentation layer migration: 19
5.2 Service layer migration 20
5.3 Persistence layer migration 22
5.4 Infrastructure layer migration 23
6 Performance testing 24
6.1 Input from monitoring 24
6.2 Performance test 25
6.3 Performance testing results 27
6.3.1 Normal load 27
6.3.2 Higher load 27
6.3.3 Stress load 27
6.4 Impact on system resources 32
7 Summary 35

List of figures

Figure 1. Spring MVC and Spring WebFlux comparison [2] 10
Figure 2. Thread-based concurrent system [5] 12
Figure 3. Event-loop-based system [5] 13
Figure 4. Monitoring tools communication 14
Figure 5. Infrastructure and communication 18
Figure 7. Example of developed performance test. 25
Figure 8. Testing infrastructure and communication 26
Figure 9. Performance test response times under normal load (lower is better). 28
Figure 10. Performance test response times under higher load (lower is better). 29
Figure 11. Performance test response times under stress load (lower is better). 30
Figure 12. Performance test failed requests under stress load (lower is better). 31
Figure 13. JVM total memory 32
Figure 14. JVM Heap Memory 33
Figure 15. CPU usage 33
Figure 16. Garbage collections 34

List of tables

Table 1. Comparison of tools for testing application performance [13] 15
Table 2. Methods before (A) and after (B) migration 20

1 Introduction

The aim of this thesis is to analyse feasibility of migration of the existing Spring MVC-based production application (hereinafter referred as *the application*) to Spring WebFlux.

"Before Spring WebFlux came in version of Spring boot 5.0, the only option was the classic Spring MVC framework built on top of Servlet API. This framework is, to this day, the most popular choice for new Spring projects and it works like a charm, but reactive, non-blocking programming model is gaining traction in recent years. The idea behind it is to reduce time application spends in blocking state waiting for data to arrive (from a database, another service, message queue, etc.) which could make an application faster" [1].

Spring MVC and Spring WebFlux "work together to expand the range of available options. The two are designed for continuity and consistency with each other, they are available side by side, and feedback from each side benefits both sides." [2] (See Figure 1)

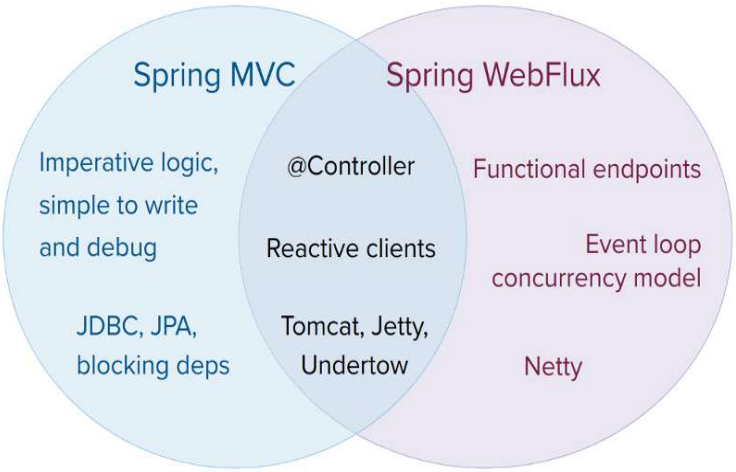


Figure 1. Spring MVC and Spring WebFlux comparison [2]

Motivation for migration to Spring WebFlux could be:

- Deprecation of existing blocking libraries (for example RestTemplate from Spring Web is in maintenance mode since Spring 5.0 [3])

- Current or expected increase of usage load [2]
- Live interest within the development team

Reasons for the analysis of migration of the application are:

- Expected double increase in usage load within a year.
- Demand for assessment within the development team and from the management.
- Complexity of the application (hard to access without deep analysis).

In the second chapter there is in more detail how Spring MVC and Spring WebFlux differ. The third chapter describes briefly the choice of tools needed for analysis. The fourth chapter gives an overview of the analysed application. The fifth chapter describes in details migration implementation. In the sixth chapter performance tests creation, setup and results are handled.

2 Overview of Spring MVC and Spring WebFlux

In this chapter Spring MVC and Spring WebFlux difference is explained.

2.1 Spring MVC vs Spring WebFlux

"Spring provides Model-View-Controller (MVC) architecture, and components that can be used to develop flexible and loosely coupled web applications. It uses the features of Spring core features like IoC and DI.

- The *Model* encapsulates the application data and in general they will consist of POJOs.
- The *View* is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- The *Controller* is responsible for processing user requests and building an appropriate model and passes it to the view for rendering." [4]

Spring MVC is uses a blocking model to handle requests. It means, that for each request there is a dedicated thread for getting data and producing response (See Figure 2).

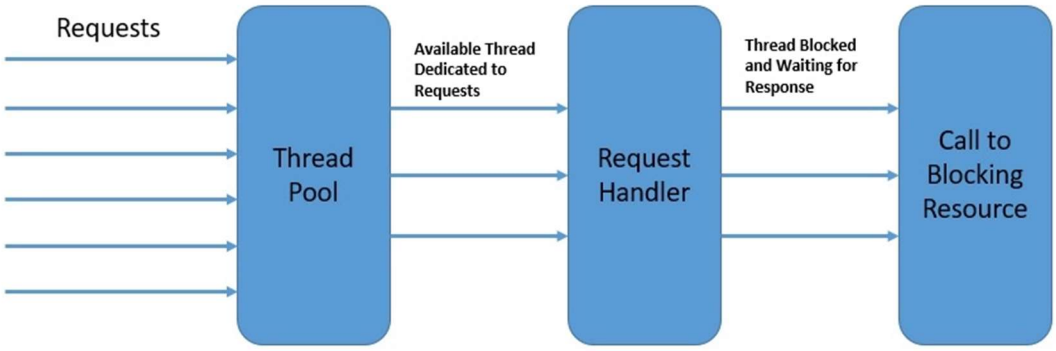


Figure 2. Thread-based concurrent system [5]

Spring WebFlux discards the thread-per-request blocking model to embrace a multi-event-loop, asynchronous, non-blocking model with back-pressure.

- “The event loop runs continuously in a single thread, although we can have as many event loops as the number of available cores
- The event loop process the events from an event queue sequentially and returns immediately after registering the callback with the platform
- The platform can trigger the completion of an operation like a database call or an external service invocation
- The event loop can trigger the callback on the operation completion notification and send back the result to the original caller.” [5] (See Figure 3)

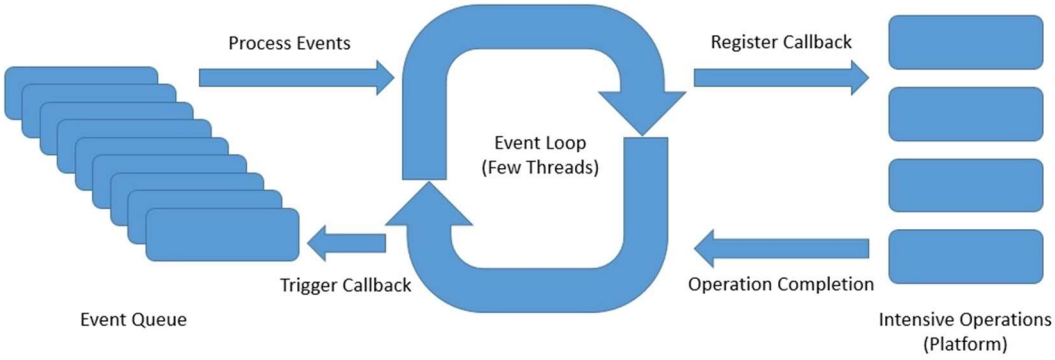


Figure 3. Event-loop-based system [5]

From developing perspective the most drastic change is the use of reactive classes: Mono [6] and Flux [7], that wrap requested object or objects respectively by data provider (Publisher [8]) and unwrapped by data consumer (Subscriber [9])

3 Tools used for analysis

In this chapter performance-testing and monitoring tools used for analysis are described.

3.1 Choice of tools for monitoring

Before the start of the analysis there was no monitoring for the application. For the purpose of primary statistics and for overseeing further performance testing monitoring was set up.

The combination of Micrometer + Prometheus + Grafana was selected for monitoring as they are easily integrable with Spring Boot and sufficient for analysis and further maintenance of the application [10]. Docker containers with Prometheus and Grafana were deployed to the same nodes as the application using Rancher 2 with Kubernetes as container orchestration framework. Configurations for both Prometheus [11] and Grafana [12] are done accordingly to existing tutorials and are, therefore, intentionally left out of the description. Communication within one node is shown in Figure 4.

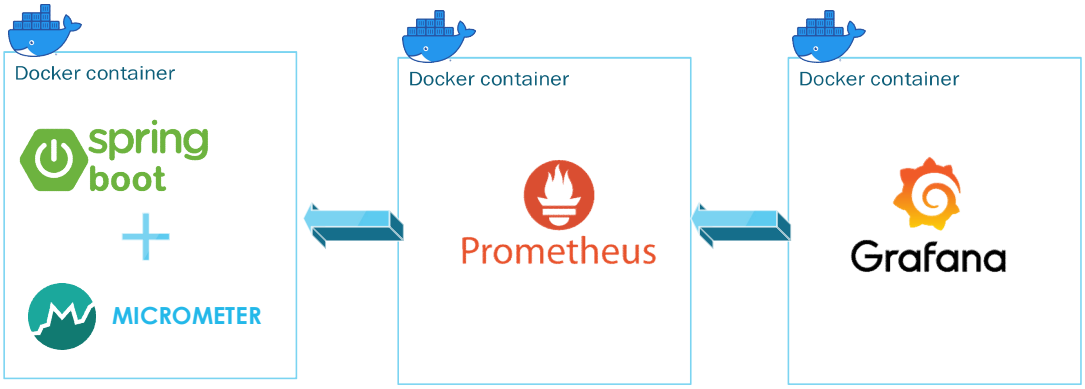


Figure 4. Monitoring tools communication

3.2 Choice of testing tools

Before the start of the analysis there were no active performance tests running for the application.

Previous performance test attempt was couple of years old and using Gatling for running the tests. Comparison data used for research can be seen in Table 1.

Table 1. Comparison of tools for testing application performance [13]

Tool	**Pros**	**Cons**
JMeter	GUI for non-programmers Popularity Protocols support Documentation Rich ecosystem	Slow test plan creation No VCS friendly format Not programmers friendly No simple CI/CD integration
Gatling	VCS friendly IDE friendly (auto complete and debug) Natural CI/CD integration Natural code modularization and reuse Less resources (CPU & RAM) usage All details of simple test plans at a glance	Scala knowledge and environment required Smaller set of protocols supported Less documentation & tooling
Taurus	VCS friendly Simple CI/CD integration Unified framework for running any type of test Built-in support for running tests at scale All details of simple test plans at a glance Simple way to do assertions on statistics	Both Java and Python environments required Not as simple to discover (IDE auto-complete or GUI) supported functionality Not complete support of JMeter capabilities (nor in the roadmap)
ruby-dsl	VCS friendly Simple CI/CD integration Unified framework for running any type of test Built-in support for running tests at scale	Both Java and Ruby environments required Not following same naming convention and structure as JMeter Not complete support of JMeter capabilities (nor in the roadmap) No integration for debugging JMeter code

Tool	Pros	Cons
	All details of simple test plans at a glance	
jmeter-java-dsl	VCS friendly IDE friendly (auto-complete and debug) Natural CI/CD integration Natural code modularization and reuse Existing JMeter documentation Easy to add support for JMeter supported protocols and new plugins Could easily interact with JMX files and take advantage of JMeter ecosystem All details of simple test plans at a glance Simple way to do assertions on statistics	Basic Java knowledge required Same resources (CPU & RAM) usage as JMeter

For purpose of analysis there was the need for easier setup and shorter learning curve, so it was decided to use jmeter-java-dsl tool from Abstracta team.

4 Overview of the application

The analysed application is an internal tool of a medium-sized international financial corporation. Its purpose is to aggregate data from external partners, store, map, process and provide it to internal users for various business use cases. Use cases of the application are left intentionally out of the scope of the thesis for confidentiality reasons.

Data acquiring process consists of daily delta files uploading by external partners to File Storing server, from where the application downloads them and processes. The delta files contain all the units which were changed during previous day. Depending on business need and internal configuration it is possible to query single data unit directly from external data repositories to get the freshest data. Infrastructure and communication can be seen in Figure 5.

Technological stack of the application is based on Spring Boot framework. Detailed technological stack is thoroughly described in following chapter.

The application has instances in three different environments: development, test and production. To achieve near-production experience we analyse only test environment instances.

The application is deployed using Kubernetes template to two separates nodes, which are located on two physically separated servers. It is possible to connect directly to individual node, but for analysis purpose testing is limited to connection through load balancer, as it allows to simulate production conditions.

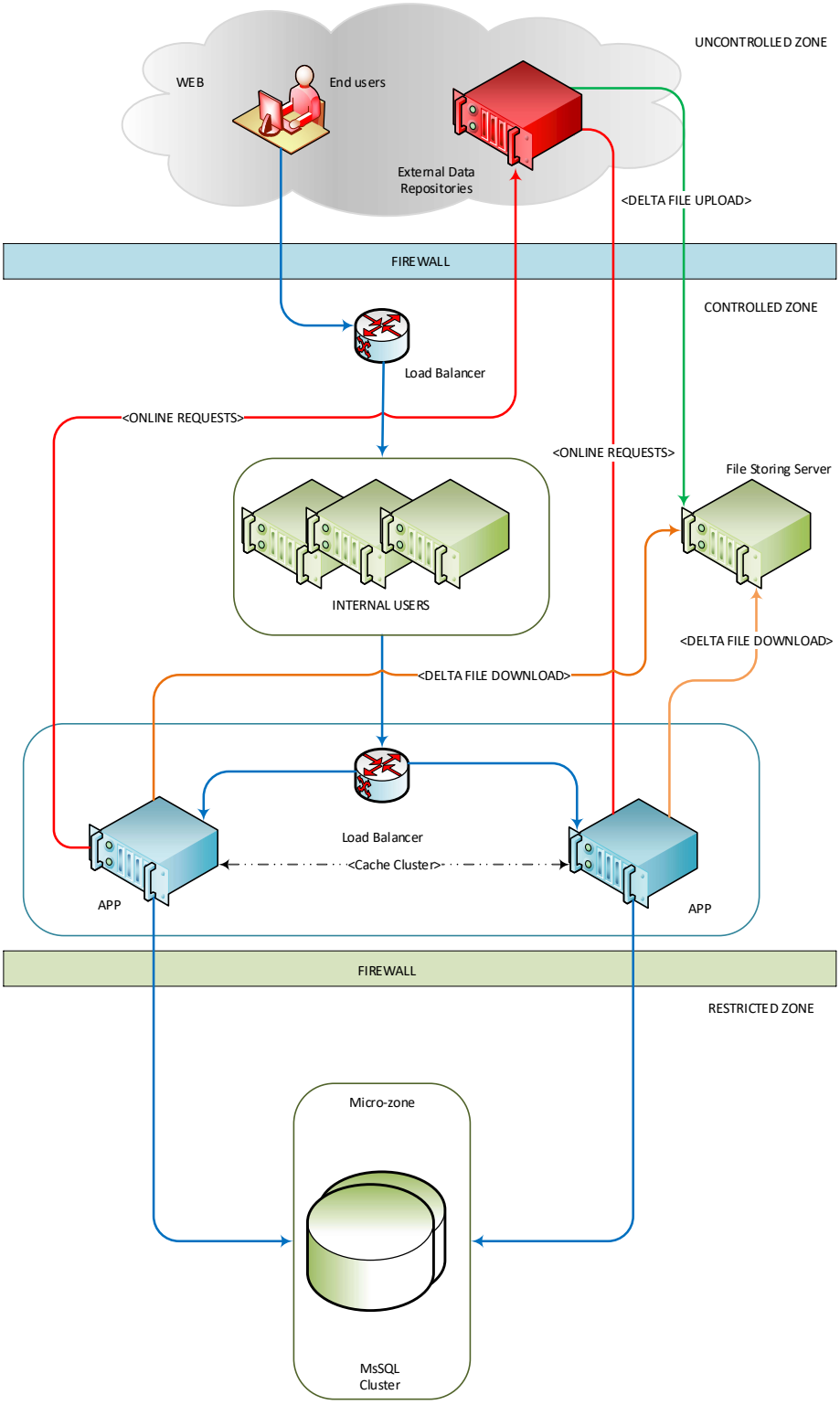


Figure 5. Infrastructure and communication

5 Migration to WebFlux

In this chapter migration to WebFlux is described.

Migration of the application consisted of static analysis and replacement of dependencies as well as of experimental one, where in order to compile some classes, it was needed to import required dependency. The application is using Gradle building tool for dependencies management. During migration minimal working set of endpoints was migrated.

5.1 Presentation layer migration

Presentation layer of the application consists mostly of REST endpoints to serve other internal applications. For internal purposes there is also a page for collecting various statistics and testing.

Migration started with replacing Spring Boot Web with Spring Boot WebFlux

```
~~implementation 'org.springframework.boot:spring-boot-starter-web'~~
implementation 'org.springframework.boot:spring-boot-starter-webflux'
```

This dependency also “pulls in all other required dependencies:

- spring-boot and spring-boot-starter for basic Spring Boot application setup
- spring-webflux framework
- reactor-core” [14]

To serve html for page the application uses Thymeleaf. Reactive Thymeleaf does not require dependency change. In the migrated version attributes for Thymeleaf page model are wrapped in a class named ReactiveDataDriverContextVariable, that increases code produced to serve content. The example from the migrated endpoint:

```
model.addAttribute("config",
        new ReactiveDataDriverContextVariable(getRegistryConfig()));
```

As alternative, it is possible to extend this class with a local wrapper to give this method more aesthetic look or use DTO with builder to hide new object creation for every model attribute.

For REST endpoints it is used the same @RestController annotation as in MVC version, but response is composed not from Object, but Mono of Flux wrapped around this object.

5.2 Service layer migration

No additional dependencies or dependency replacement was needed for service layer.

Most noticed difference in the produced code is that imperative paradigm is replaced with functional. Table 2 displays a part of the code migrated with corresponding changed code.

Table 2. Methods before (A) and after (B) migration

Case	*Code*

1A	`public void checkRequestAllowed(RequestSource request) throws RequestNotAllowedException {`
	`  checkServiceAllowed(request);`
	`}`

1B	`public Mono<Boolean> checkRequestAllowed(RequestSource request) {`
	`  return checkServiceAllowed(request);`
	`}`

2A	`private void checkServiceAllowed(RequestSource request) throws RequestNotAllowedException {`
	`  String service = request.getService();`
	`  if (!isRequesterAllowed(service)) {`
	`  throw new RequestNotAllowedException();`
	`  }`
	`}`

2B	private Mono<Boolean> checkServiceAllowed(RequestSource request) {
	 String service = request.getService();
	 return isRequesterAllowed(service)
	 .flatMap(isAllowed -> {
	 if (!isAllowed) {
	 return Mono
	 .error(new RequestNotAllowedException());
	 }
	 return Mono.just(true);
	 });
	}

3A	private boolean isRequesterAllowed(String requesterName) {
	 Map<String, Boolean> accessMap = getRegistryConfig();
	 if (accessMap.containsKey(requesterName)) {
	 return accessMap.get(requesterName);
	 }
	 log.warn("No status for requester '{}' in configuration property {}", requesterName, SERVICE_STATUSES.getName());
	 return false;
	}

3B	private Mono<Boolean> isRequesterAllowed(String requesterName) {
	 return getRegistryConfig()
	 .filter(value -> value.containsKey(requesterName))
	 .flatMap(value -> Mono.just(value.get(requesterName)))
	 .switchIfEmpty(Mono.fromSupplier(() -> {
	 log.warn("No status for requester '{}' in configuration property {}", requesterName, SERVICE_STATUSES.getName());
	 return false;
	 }));
	}

4A	private Map<String, Boolean> getRegistryConfig() {
	 return entityManager.getRegistryConfig(null, SERVICE_STATUSES);
	}

4B	private Mono<Map<String, Boolean>> getRegistryConfig() {
	 return entityManager.getRegistryConfig(null, SERVICE_STATUSES)
	.next();
	}

In some cases (e.g., case 1 and 2) there was the need to replace `void` with *Mono<>* to handle error within the existing code thrown by *`Mono.error(…)`*.

5.3 Persistence layer migration

Most of the changes made for dependencies are done for persistence layer. In this layer appears most difficulties too. The first thing to migrate, MsSQL driver, has fresh reactive version:

```
implementation 'io.r2dbc:r2dbc-mssql'
```

Although for requests handling this version is sufficient it has significant drawbacks for the usage in the application:

- No support for integrated security (the application uses LDAP with Kerberos) [15].
- Database version control framework (FlyWay) used in the application is not supporting this version.

Repository migration to reactive version was done with replacing JPA dependency with corresponding R2DBC analogue. Major replacements for this change include also Hikari pool replacement with R2dbc pool and drop of Hibernate framework.

```
~~implementation 'org.springframework.boot:spring-boot-starter-data-jpa'~~
implementation 'org.springframework.data:spring-boot-starter-data-r2dbc'
```

The application uses a lot of JPA annotations. R2dbc repositories are not supporting large portion of JPA annotations such as composite keys (*`@Embeddable`* and *`@EmbeddedId`*), generation strategy for primary keys *(`@GeneratedValue(strategy = GenerationType.AUTO`)*), etc. To support those, it is possible to use Hibernate Reactive Core [16], but it currently does not support MsSQL database.

For connection to external repositories the application uses class HttpClient from Apache and mostly SOAP dependencies to receive data in XML format:

```
implementation 'org.apache.httpcomponents:httpclient:4.5.3'
implementation 'javax.xml.ws:jaxws-api:2.3.1'
implementation 'com.sun.xml.messaging.saaj:saaj-impl:1.4.0'
```

Because of latter performance testing is unreliable if tested with external repositories, migration of mentioned implementation was left out of the scope of this thesis and was not implemented, but was investigated, nevertheless. Starting from version 5 of Apache client it provides reactive support [17]. For SOAP it is possible to wrap service with asynchronous handler ReactorAsyncHandler [18]. As alternative for both approaches it is possible to use WebClient – reactive http client included in Spring WebFlux [2], [19].

5.4 Infrastructure layer migration

Currently there is no reactive implementation of integrated security as LDAP with Kerberos, that is used in the application. As alternative it is possible to use class from Spring Security - ReactiveAuthentificationManagerAdapter, which "adapts an AuthenticationManager to the reactive APIs. This is somewhat necessary because many of the ways that credentials are stored (i.e. JDBC, LDAP, etc) do not have reactive implementations." [20]

The application uses Infinispan for clustered caching. It stores information on one node into cache so another node receives reference of that stored instance, which can be retrieved if needed. Although it is stated, that Infinispan is supporting non-blocking caching [21], there was not luck during analysis to make it working in clustered mode for both Mono and Flux units. Therefore, it was disabled during migration. For that reason, the special version of the application with disabled caching was made.

6 Performance testing

This chapter describes preparation, execution and results of performance testing.

6.1 Input from monitoring

Grafana and Prometheus allow us to query and visualize wide spectre of application statistical data. To setup performance tests firstly it is needed to determine normal load of the application. From Grafana it is possible to query statistics, that collects Prometheus, and draw the graph of queries per minute using following command:

```
sum(increase(http_server_requests_seconds_count[1m]))
```

The result of this query gives us combined load for both nodes of the application as seen in Figure 6.

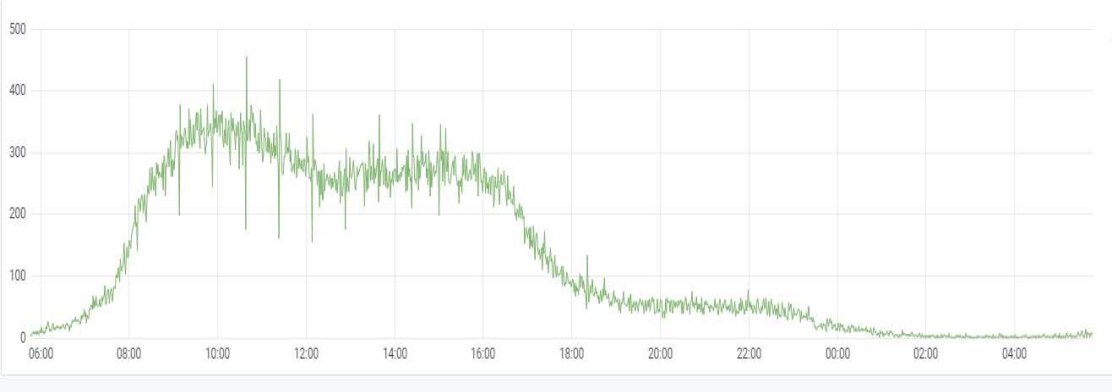


Figure 6. Requests per minute during 24h.

Request's load varying during working hours (8:00 to 17:00) on average between 200 requests per minute and 400 requests per minute with spikes reaching 450. In peak hours between 9:00 and 11:00 average load is 350.

6.2 Performance test

To examine the application performance tests were created. Using the determined average request during peak hours two scenarios were defined to mimic live load:

- normal load of 350 users increased gradually for 1 minute requesting once (see Figure 7).
- higher load of 350 users increased gradually for 1 minute making two requests each.

```
@Tag("normal")
@Test
void testPerformanceNormal() throws IOException {
  TestPlanStats stats = testPlan(
    threadGroup(350, 1,
                httpSampler(ENDPOINT, PARAMS)
                  .header("Authorization", CREDENTIALS)
                  .children(
                    responseAssertion()
                      .containsSubstrings("\"success\": true")
                  ))
    .rampUpPeriod(Duration.ofSeconds(60)),
      htmlReporter(getClass().getClassLoader()
                   .getResource("")
                   .getPath()
                   .split("build")[0] + "build/reports/jmeter"))
    .run();
  assertThat(stats.overall().elapsedTimePercentile99())
    .isLessThan(Duration.ofSeconds(10));
  assertThat(stats.overall().errorsCount())
    .isLessThan(10);
}
```

Figure 7. Example of developed performance test.

The third scenario for stress testing was determined experimentally to better show

performance of WebFlux version of the application in comparison to its non-migrated version: 10000 users increased gradually for 1 minute making four requests each.

Endpoint used is common URL for both application nodes behind load balancer.

The code in Figure 7 is the example of the jUnit test created for normal load performance testing using jmeter-java-dsl library. Tags before the test are used to be able to run it individually with Gradle in pipeline. Each test produce html report as well as JTL files. The tests also use AssertJ dependency for code readability.

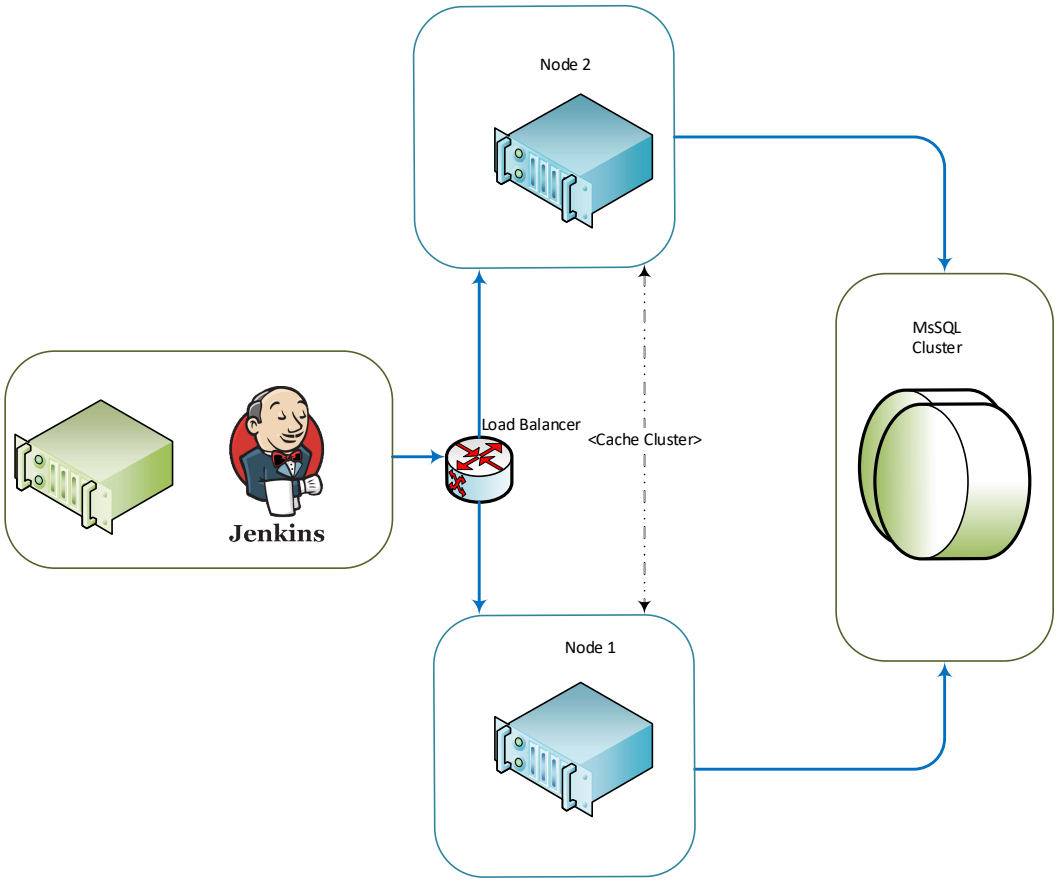


Figure 8. Testing infrastructure and communication

To analyse and test properly we need near-production conditions. For that purposes a special Jenkins pipeline was created to run performance tests automatically against test environment of the application as seen in Figure 8. As mentioned in the previous chapter, the special version of the application with cache disabled was created to be better comparable with the migrated version. Testing was consequently performed on the migrated version, on the version without cache and on the existing application.

6.3 Performance testing results

In this chapter we will analyse performance tests result.

6.3.1 Normal load

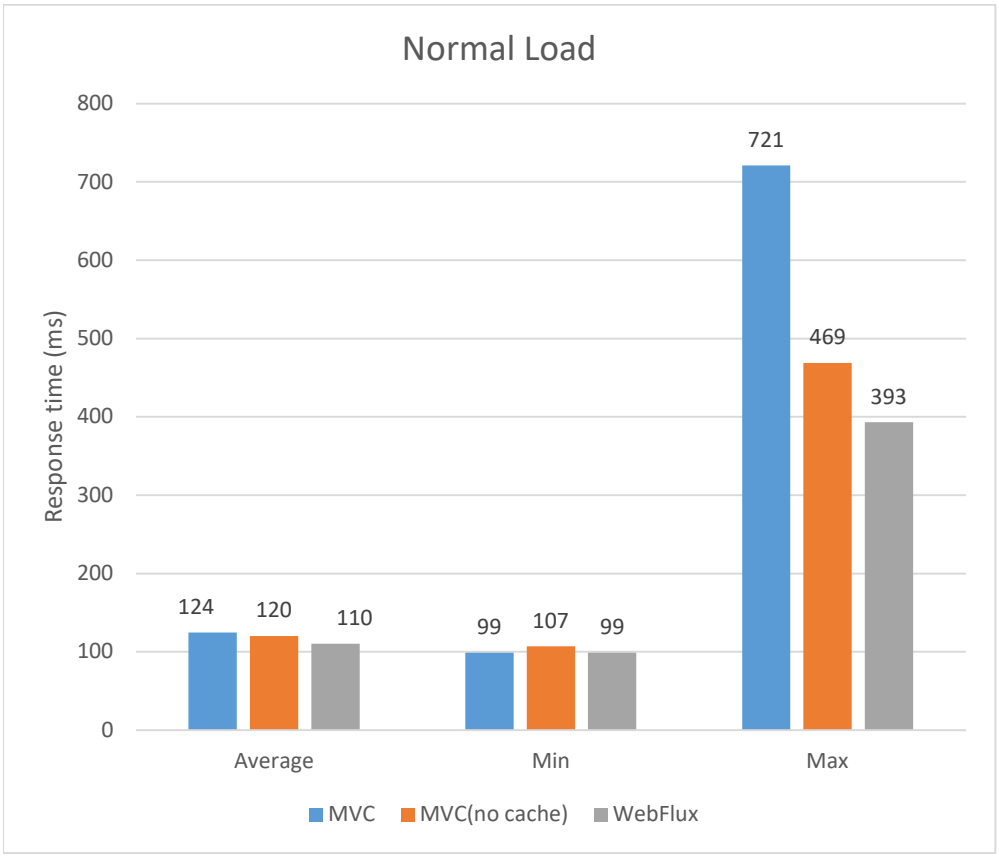
Results of the tests under normal load are seen in Figure 9. Average response times and minimal response times for the migrated version were the smallest, but with none to small difference with other versions. The longest response time for the migrated version was almost half compared with the original version. The version of the application with no cache was better, than cache version in average and maximal response time, but in minimal slower than two others. Longer response times for the original can be explained with clustered caching- extra request between caching layer is done every time, when request is landing on the version, where a cached object has only its reference from the other node. Interestingly, in 99th percentile of responses the version with disabled cache showed the lowest response time of three versions.

6.3.2 Higher load

Results of the tests under higher load are seen in Figure 10. They are very similar to the previous results. Slightly higher response times on average and in minimal. Maximum response time gap decreased between the migrated and the non-cache version but increased in comparison with the original version. Percentile statistics show, that the migrated version was more consistent showing slight increase in 90th and 95th and slight decrease in 99th percentile.

6.3.3 Stress load

Results of response time under stress load are seen in Figure 11. Comparing response times, it could be misread as the migrated version being the slowest of three versions, but if to consider the number of failed responses in Figure 12 it makes the migrated version most successful to produce response under heavy load. On the other hand, 10 % of responses in the migrated version are nearly three minutes long and longer, making them also too long for a requesting agent and probably droppable by a requestor.



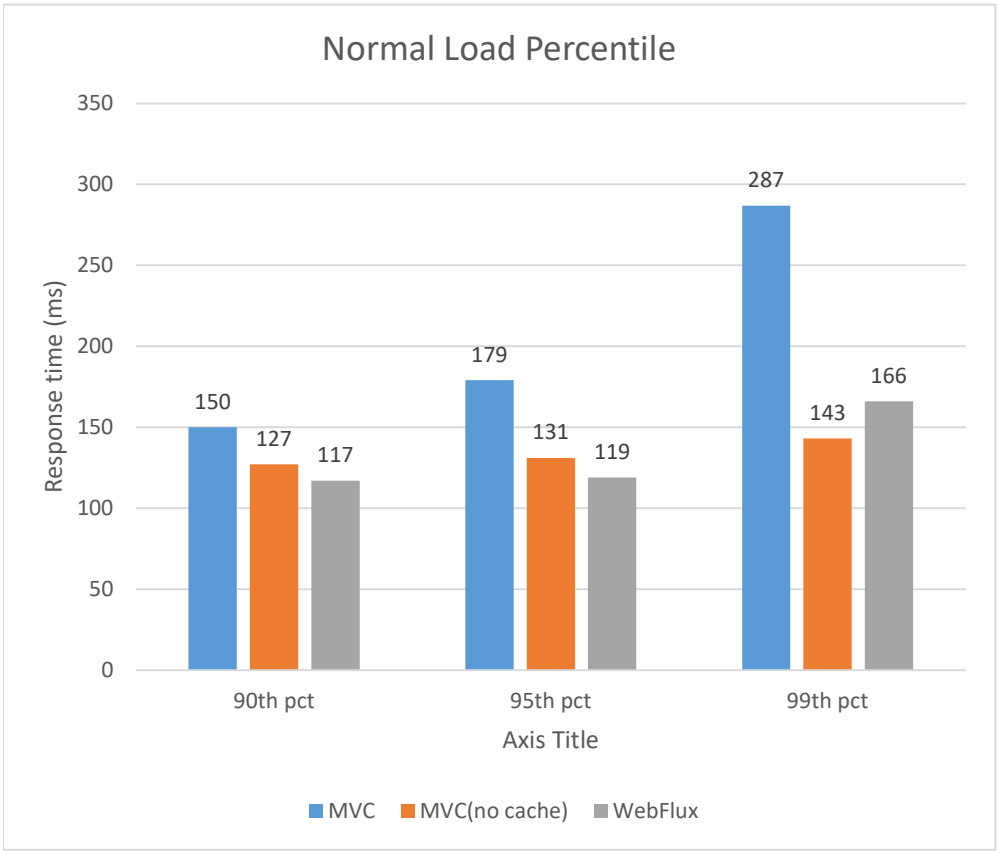
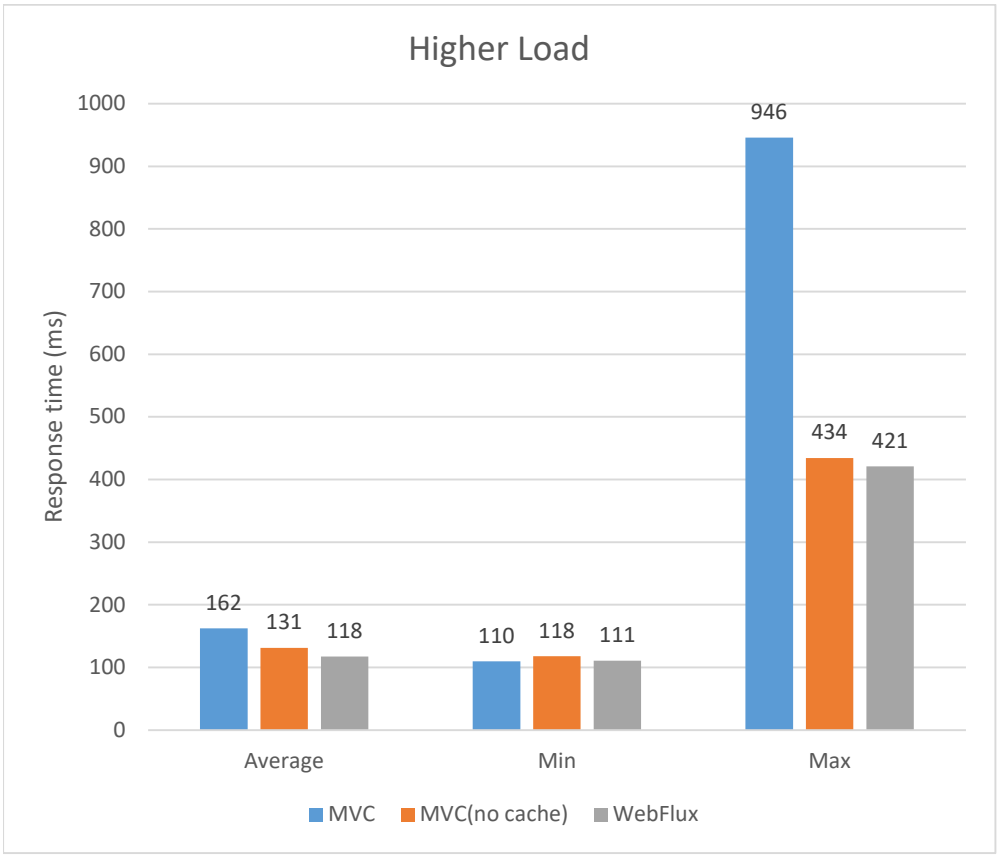


Figure 9. Performance test response times under normal load (lower is better).



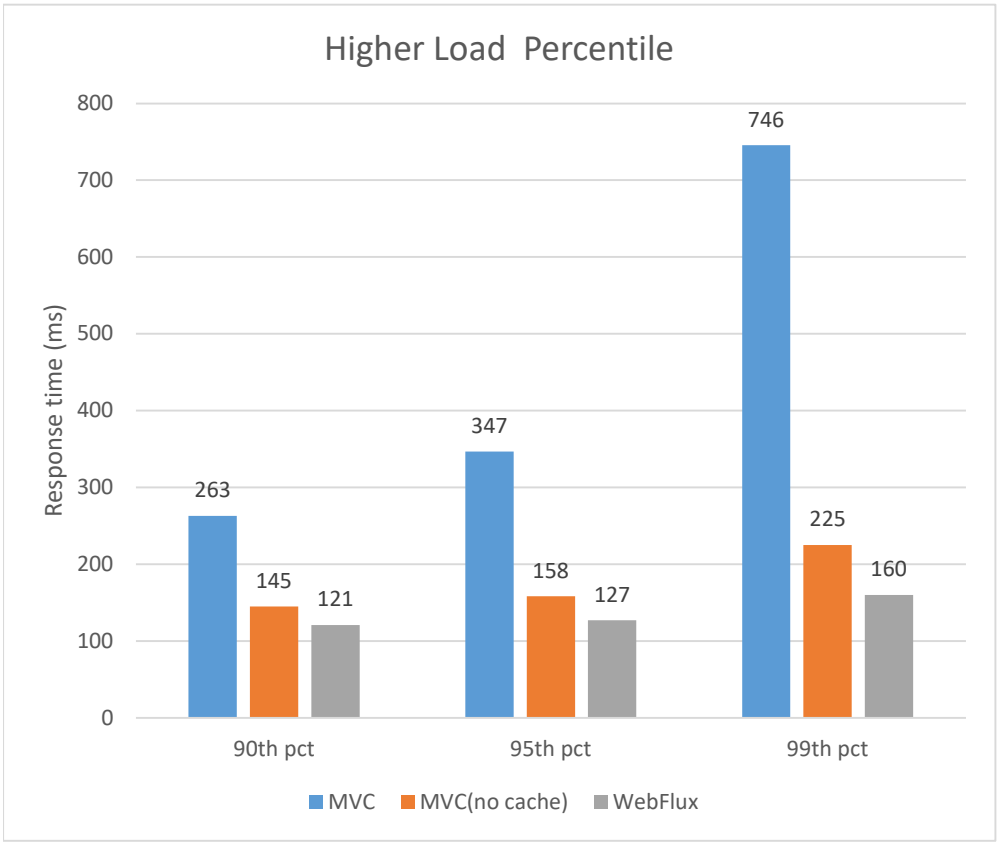


Figure 10. Performance test response times under higher load (lower is better).

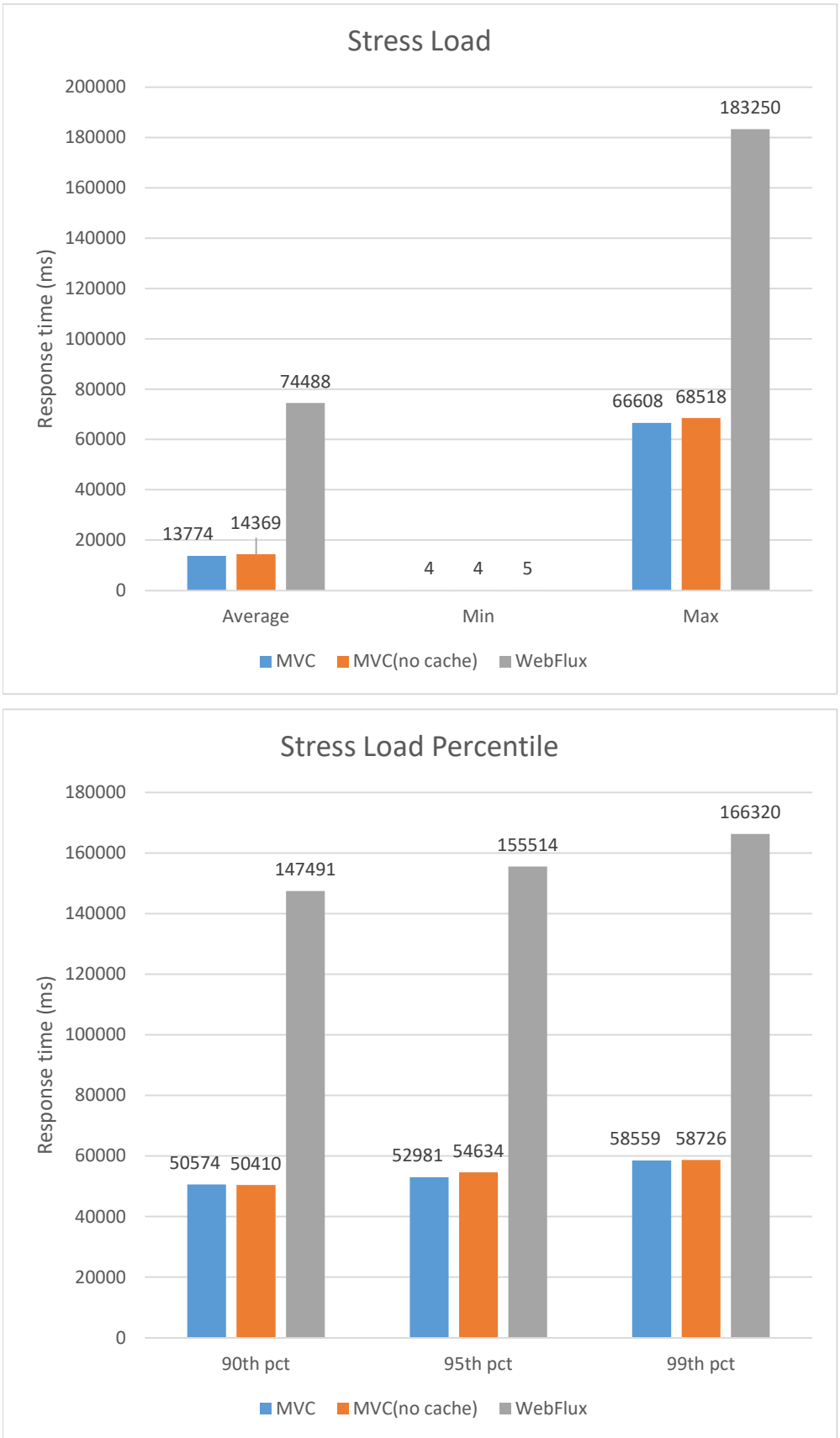
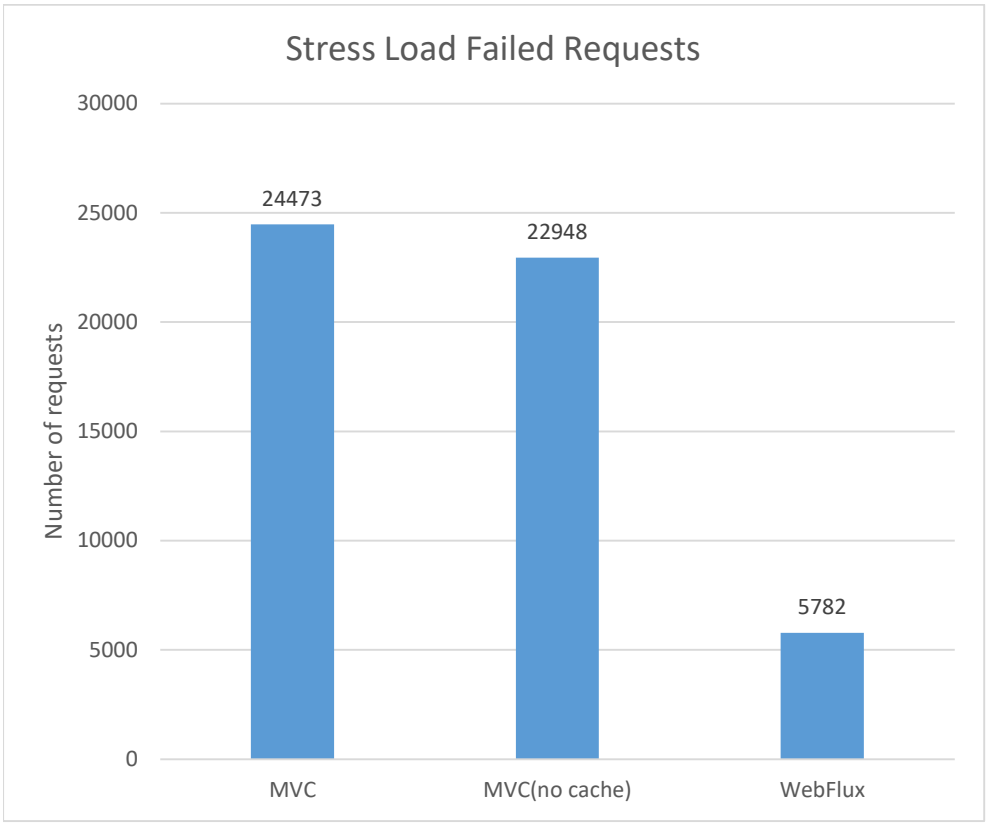


Figure 11. Performance test response times under stress load (lower is better).



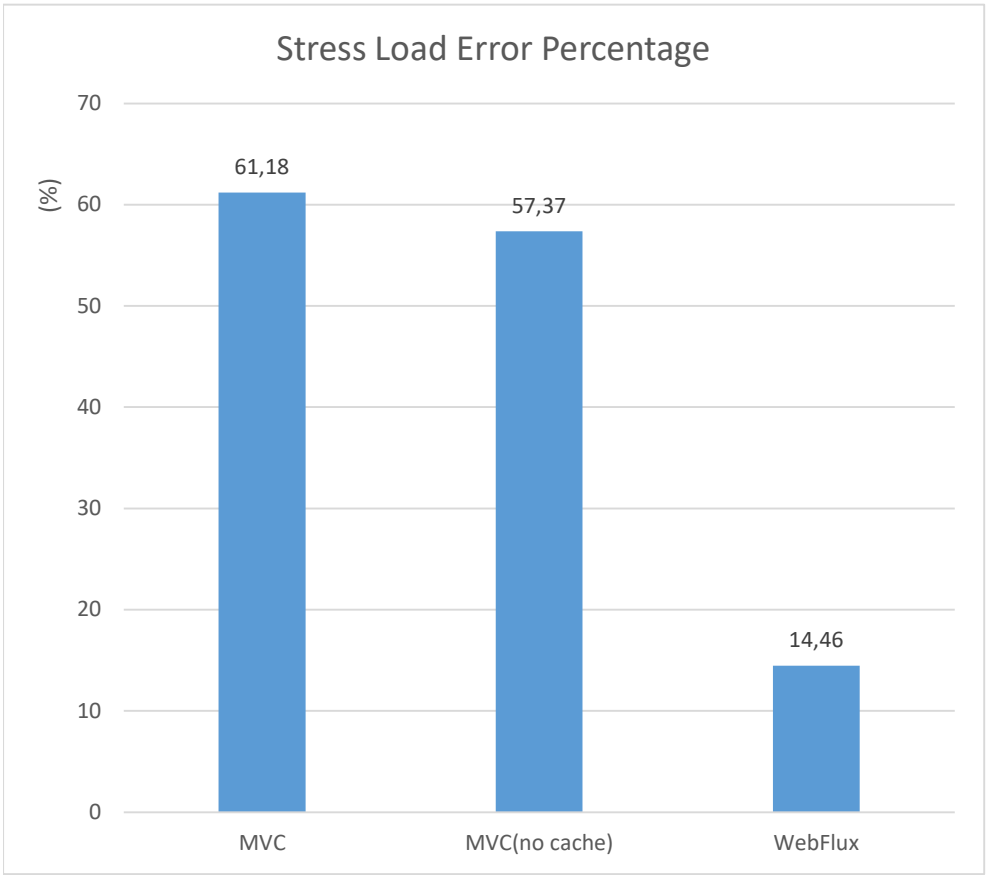


Figure 12. Performance test failed requests under stress load (lower is better).

6.4 Impact on system resources

Although supposed in multiple articles about WebFlux comparison to MVC (e.g., [22] [23]), definite advantage in memory and CPU usage was not seen during the tests. The root cause of it is probably the same number of initial threads in a thread pool of the application and the same memory allocation size in the application settings. During the performance tests there was also seen drawbacks of Prometheus monitoring. With peak numbers of request during stress tests Prometheus failed to connect to the application to read statistics, so there are minor gaps in some readings from the application, which, however, did not change overall trends.

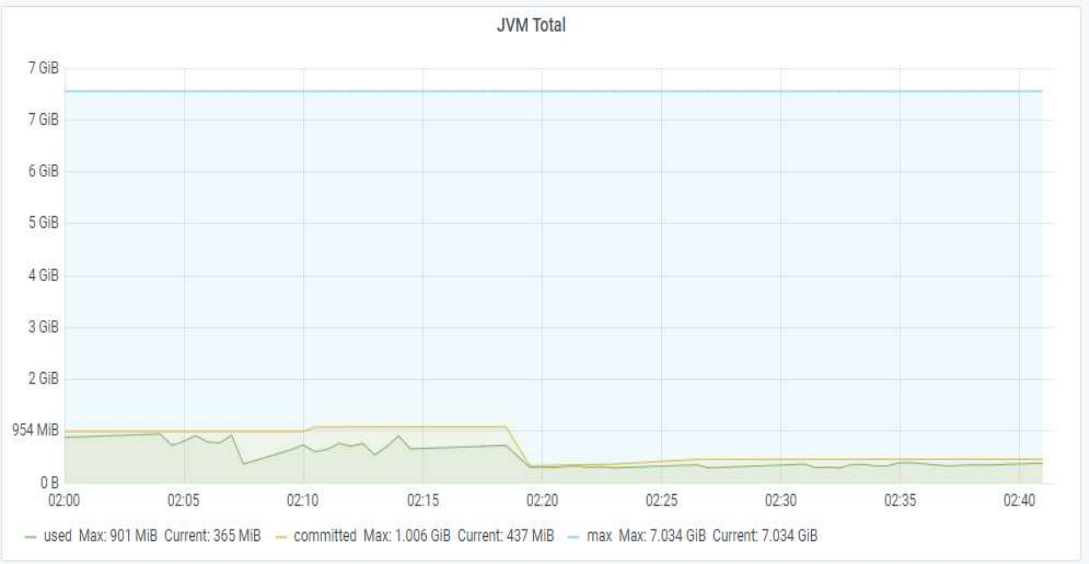


Figure 13. JVM total memory

As seen in Figures 13 through 17, the performance tests executed with normal, higher and stress load accordingly as following: from 2:03 to 2:19 – WebFlux version was tested, from 2:20 to 2:27 - the version with cache disabled was tested and from 2:32 to 2:38 – the original version.

Figure 14. JVM Heap Memory

As seen in Figures 13 and 14, the usage of memory is even higher in the WebFlux version.

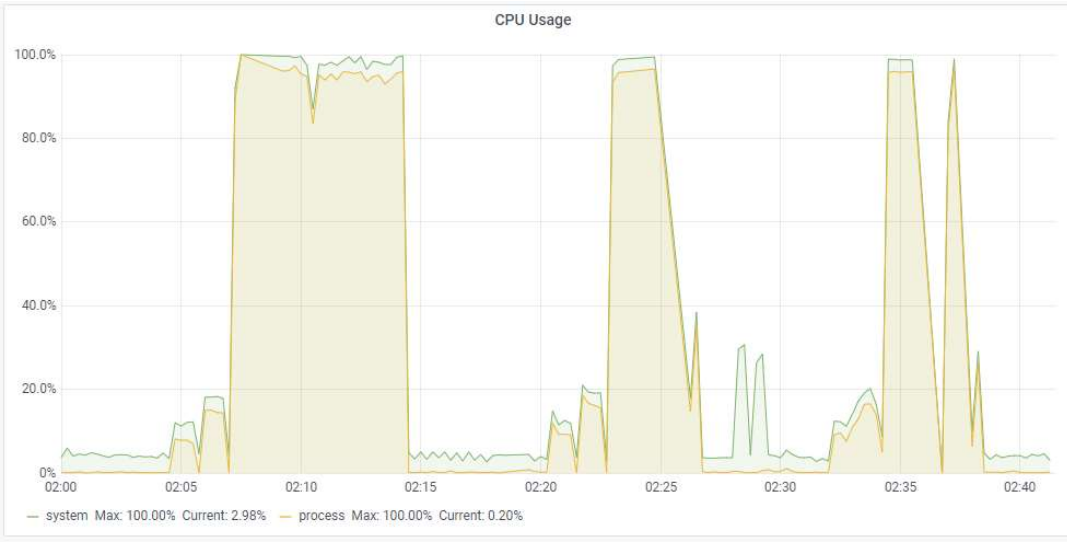


Figure 15. CPU usage

CPU level and garbage collection are also higher and longer in time in the WebFlux version (see Figures 15 and 16) due to higher number of successful responses during the stress test.

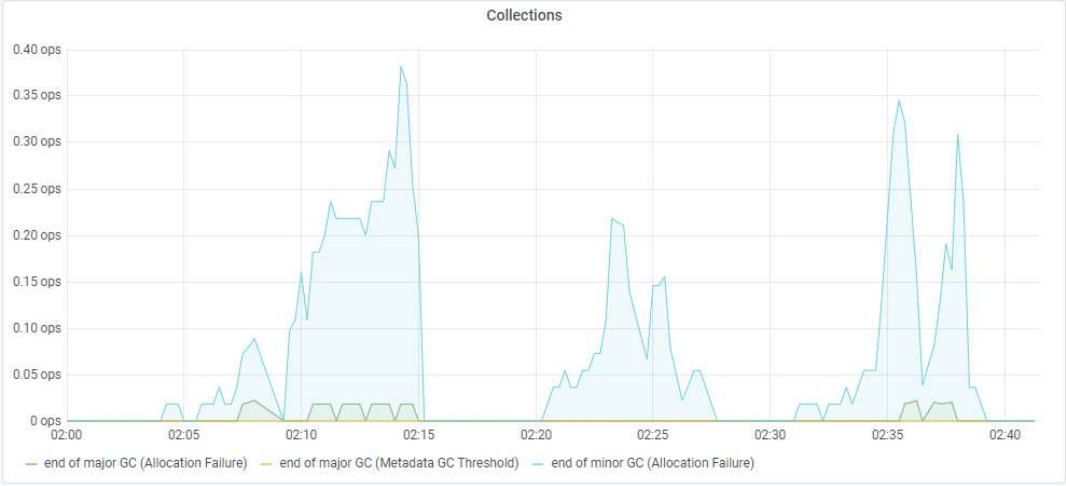


Figure 16. Garbage collections

To sum up, resource monitoring did not demonstrate definite improvement in system resource usage for the migrated application during the performance tests.

7 Summary

In this thesis migration to Spring WebFlux from Spring MVC was analysed and assessed.

Migration of the existing application is always harder, than making something from scratch. This is true also for migration done in the framework of this thesis. Code-wise migrated application did not have all the features as the original application, lacking in support of entities annotations, database versioning, caching and security features, that local developers and company's systems are using. Further investigation is needed for adapting some workarounds and alternate solutions for fully migrating some features or adapting code to existing solutions and paradigms.

The choice of monitoring and testing tools was sufficient, proving enough input for gathering performance data. During the performance tests it was found minor drawback of monitoring tool Prometheus, previously not encountered and hardly mentioned in external sources. Further investigation is needed for overcoming that.

The performance tests results show, that WebFlux has advantage in speed under normal and higher loads and has higher rate of successful responses than the original application under stress load. No system resources benefits were spotted for the existing application.

The analysis also pointed out some bottlenecks of the existing application. Current cashing solution for distributed cache is making response time longer due to exchange of cache between different nodes when same data is queried multiple times. This statement could be also applicable for having configuration for different kinds of endpoints cached only on one node, making each request to other node longer.

To sum up, migration of the existing application is not feasible right now. Although the application could benefit from migration in performance, important features, needed for the application is not fully supported yet. Furthermore, it could require more time for other team members for adaptation of functional paradigm and event-driven development. As suggested in many different sources (e.g., [2]), sometimes it is wise to

adopt only some features of reactive stack and continue using wide-featured solutions from blocking stack.

References

[1] S. Vidak, “Spring WebFlux Introduction” [Online]. Available: https://mister11.github.io/posts/spring_webflux/. [Accessed 18.05.2021].

[2] “Web on Reactive Stack” [Online]. Available: https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html. [Accessed 18.05.2021].

[3] “Class RestTemplate” [Online]. Available: https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html. [Accessed 18.05.2021].

[4] O. Elgabry, “Spring: A Head Start 🔥— Spring MVC (Part 5)” [Online]. Available: https://medium.com/omarelgabrys-blog/spring-a-head-start-spring-mvc-part-5-db6b7b195e51. [Accessed 18.05.2021].

[5] K. Chandrakant, “Concurrency in Spring WebFlux” [Online]. Available: https://www.baeldung.com/spring-webflux-concurrency. [Accessed 18.05.2021].

[6] “Class Mono<T>” [Online]. Available: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html. [Accessed 18.05.2021].

[7] “Class Flux<T>” [Online]. Available: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html. [Accessed 18.05.2021].

[8] “Interface Publisher<T>” [Online]. Available: https://www.reactive-streams.org/reactive-streams-1.0.1-javadoc/org/reactivestreams/Publisher.html. [Accessed 18.05.2021].

[9] “Interface Subscriber<T>” [Online]. Available: https://www.reactive-streams.org/reactive-streams-1.0.1-javadoc/org/reactivestreams/Subscriber.html. [Accessed 18.05.2021].

[10] “Comparisson to Alternatives” [Online]. Available: https://prometheus.io/docs/introduction/comparison/. [Accessed 18.05.2021].

[11] “Getting Started” [Online]. Available: https://prometheus.io/docs/prometheus/latest/getting_started/. [Accessed 18.05.2021].

[12] “Grafana Documentation” [Online]. Available: https://grafana.com/docs/grafana/latest/. [Accessed 18.05.2021].

[13] “jmeter-java-dsl” [Online]. Available: https://github.com/abstracta/jmeter-java-dsl. [Accessed 18.05.2021].

[14] “Guide to Spring 5 WebFlux” [Online]. Available: https://www.baeldung.com/spring-webflux. [Accessed 18.05.2021].

[15] [Online]. Available: https://github.com/r2dbc/r2dbc-mssql/issues/101. [Accessed 18.05.2021].

[16] “Hibernate Reactive” [Online]. Available: https://github.com/hibernate/hibernate-reactive. [Accessed 18.05.2021].

[17] “HttpClient Overview” [Online]. Available: http://hc.apache.org/httpcomponents-client-5.1.x/index.html. [Accessed 18.05.2021].

[18] B. Garvelink, “Reactive Web Service Client with JAX-WS” [Online]. Available: https://godatadriven.com/blog/reactive-web-service-client-with-jax-ws. [Accessed 18.05.2021].

[19] “Make asynchronous SOAP call in Spring WebFlux” [Online]. Available: https://stackoverflow.com/questions/60324772/make-asynchronous-soap-call-in-spring-webflux. [Accessed 18.05.2021].

[20] “Class ReactiveAuthenticationManagerAdapter” [Online]. Available: https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/authentication/ReactiveAuthenticationManagerAdapter.html. [Accessed 18.05.2021].

[21] “Infinispan 10.1.0.Final” [Online]. Available: https://infinispan.org/blog/2019/12/23/infinispan-10/. [Accessed 18.05.2021].

[22] A. Filichkin, “Spring Boot performance battle: blocking vs non-blocking vs reactive” [Online]. Available: https://filia-aleks.medium.com/microservice-performance-battle-spring-mvc-vs-webflux-80d39fd81bf0. [Accessed 18.05.2021].

[23] P. Minkowski, “Performance Comparison Between Spring MVC vs Spring WebFlux with Elasticsearch” [Online]. Available: https://piotrminkowski.com/2019/10/30/performance-comparison-between-spring-mvc-and-spring-webflux-with-elasticsearch/. [Accessed 18.05.2021].

.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Valeri Andrejev

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Feasibility Analysis of Migration from Spring MVC to Spring WebFlux: A Case Study”, supervised by Vadim Kaparin
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

18.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive licence shall not be valid for the period.