Load Testing of

Vaadin Flow applications

UNIVERSITY OF TURKU
Department of Future Technologies
Master of Science in Technology Thesis
Communication Systems
March 2020
Anastasiia Smirnova

Supervisors:
Antti Hakkala
Seppo Virtanen

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service

UNIVERSITY OF TURKU
Department of Future Technologies

Anastasiia Smirnova: Load testing of Vaadin Flow applications

Master of Science in Technology Thesis, 80 p., 34 app. p.
Communication Systems
March 2020

All types of businesses, from small start-ups to big enterprises, have an online presence. Their web pages and applications can be used to acquire products ans services and are thus expected to be efficient. Yet, the web environment imposes additional requirements on software, such as the need for reliable security and adequate response times. To ensure these requirements are met and the product is of the expected quality, various types of testing are utilized during development.

This master's thesis evaluates a procedure for verifying a non-functional requirement of a web application – its performance. It focuses on load testing, which is used to analyze and assess an application's behavior with different user loads.
The scope of applications is limited to server-side applications that are developed with the latest long-term support version of the Vaadin framework. The effects of the performance arising from the server-side architecture of the framework and the Java ecosystem are reviewed. Furthermore, an overview of available improvement techniques, such as cache and load balancers, is given.
From a load testing perspective, the biggest challenges that arise from Vaadin's architecture are its unique features. These include node values of user interface components, synchronization and Cross-Site Request Forgery protection tokens. The defined universal regular expressions that capture these attributes can be used again later.

The main contribution of this thesis is formulating a ready-to-use method of load testing a Vaadin Flow application. Once established and analyzed, the method is then applied to a real-life situation to verify its applicability and usefulness. Two widely used load testing tools are utilized – JMeter and Gatling. Furthermore, a method to estimate a web application's session size is presented. Potential bottlenecks and other potential issues are identified by using a profiled to track the application's memory consumption during a test run. After the load test is finalized and completed, a session size estimation is conducted.

As a result of test execution, a potential bottleneck is identified and fixed in the application. Complete test plans for both JMeter and Gatling are defined and implemented. Alternatives and possible improvements to the proposed solution are reviewed. Based on the literature review, when deploying an application on multiple servers, the best solution is enabling the sticky sessions feature.

Keywords: Load Testing, Vaadin, Performance, JMeter, Gatling, Java

Acknowledgments

This Master's thesis is the biggest independent project I have ever completed, but I didn't do it alone. I would like to thank my supervisors at the University of Turku, Antti Hakkala and Seppo Virtanen, for reviewing my thesis and offering valuable advice and suggestions. Finding a topic and writing my thesis would not have been possible without the help of my supervisor at Vaadin, Johannes Tuikkala, who provided me with technical guidance and expertise.

I would like to express my gratitude to the Finnish Meteorological Institute and especially Mikko Parviainen for giving me access to the application and providing me with vital information. I also want to thank everyone at Vaadin who found the time to answer my questions or lend me advice. Special thanks are owed to Leif Åstrand and Denis Anisimov for all their technical assistance and Erik Lumme and Matti Hosio for helping me to find and set up the FMI project.

Achieving this milestone would not have been possible without the dedication and influence of my teachers over the years. I would like to thank every teacher whose course I enrolled in at Kovdor's school, Salla's gymnasium, Shanghai's Fudan University and the University of Turku. I have never received anything, but encouragement from all of you.

Most of all, I would like to thank two people who supported me the entire way, who had trust in me, when I had little in myself and whom I love the most - Ville and my mum, to whom this thesis is dedicated.

Table of Contents

1 Introduction 1

2 Load testing and application performance 3

2.1 Performance testing 3

2.1.1 Load testing 4

2.1.2 Scalability testing 4

2.2 Performance enhancements 5

2.2.1 Load balancers 5

2.2.2 Cache 7

2.3 Java 11

2.3.1 JVM memory management 11

2.3.2 Performance pitfalls and countermeasures 12

3 Vaadin Flow applications 15

3.1 Communication model 15

3.1.1 Deployment 15

3.1.2 Push 16

3.1.3 User session 17

3.2 Client-side 17

3.2.1 Shadow Domain Object Model 18

3.2.2 HTML Template 18

3.2.3 Custom Elements 18

3.2.4 Optimizations and IE11 support 21

3.3 Performance 21

4 Designing load tests for a Flow application 24

4.1 Flow features 24

4.1.1 An HTTP request 24

4.1.2 A csrfToken 25

4.1.3 v-uiId and node 25

4.1.4 Synchronization tokens 26

4.2 JMeter 26

4.2.1 Recording 27

4.2.2 Listeners 29

4.2.3 Regular Expression Extractor 29
4.3 Gatling 30
4.3.1 Recording 31
4.3.2 A request building blocks 31
4.3.3 Reports 32
5 Load Test Implementation and testing 33
5.1 Overview 33
5.2 Prerequisites 35
5.2.1 Versions 36
5.2.2 Browser proxy set-up 37
5.2.3 Regular expressions 37
5.2.4 Scenario 38
5.3 JMeter implementation 39
5.3.1 A CSRF token handling 40
5.3.2 Extracting an element's node value 41
5.3.3 Randomizing and supplying data 45
5.3.4 Output 49
5.4 Gatling implementation 52
5.4.1 Payloads 53
5.4.2 A CSRF token handling 54
5.4.3 Extracting clientId and syncId 54
5.4.4 Feeders and random data 55
5.4.5 Setting session variables 55
5.4.6 Debugging and Generating load 56
5.4.7 Output 59
5.4.8 Comparing outputs 61
5.5 Application memory consumption 62
5.6 Integrating tests into maven 65
6 Discussion 66
6.1 Assuring randomness and finding node id's in the test scenario 66
6.2 Debugging 67
6.3 JMeter 68
6.4 Gatling 69
6.4.1 Recording 69

6.4.2 Test execution ... 70
6.5 Push ... 70
6.6 Alternatives ... 71
6.7 Future work ... 71
7 Conclusion ... 73
References ... 75
Appendix A A custom element from an HTML Template ... A-1
A.1 CustomDiv.html ... A-1
A.2 MainView.java ... A-2
A.3 CustomDiv ... A-2
Appendix B A configured JMeter's Test Plan ... B-1
Appendix C Scala script created for a Gatling test ... C-1
C.1 Test script ... C-1
C.2 Bodies ... C-16
Appendix D Profile for integration tests ... D-1

Abbreviations and Acronyms

API	Application Programming Interface
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
CSV	Comma Separated Values
CDN	Content Delivery Network
DOM	Document Object Model
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifier
JS	JavaScript
JVM	Java Virtual Machine
OS	Operating System
Regex	Regular Expression
UDV	User Defined Variable
UI	User Interface
URL	Uniform Resource Locator

List of Figures

Figure 1. JVM Memory management 11
Figure 2. A request handling process 16
Figure 3. Push definition with transport and mode specified for an individual view 17
Figure 4. Adding elements to a VerticalLayout. 19
Figure 5. Content of a custom-div.js template 20
Figure 6. CustomDiv rendered in Google Chrome browser 20
Figure 7. A default Test Plan generated from Recording template 27
Figure 8. A JMeter’s configured User Defined Variables component. 28
Figure 9. Configured Regular Expression Extractor for a CRSF token. 30
Figure 10. A protocol definition and set-up for a scenario 32
Figure 11. A response payload to a request presented in Chapter 4.1.1 34
Figure 12. The method to load test a Flow application 36
Figure 13. Defined user scenario 38
Figure 14. The tested UI 39
Figure 15. Configuration in "Advanced" tab for a request 40
Figure 16. Recorded request without modifications 40
Figure 17. Replaced csrfToken, syncId and clientId tokens 41
Figure 18. A Snippet of the response with the "Producer" grid definition. 42
Figure 19. Grid position inside div 43
Figure 20. Configured Regular Expression extractor for syncId and clientId 45
Figure 21. BeanShell script for parsing and saving synchronization tokens into memory 45
Figure 22. Comma-separated values Data set configuration in JMeter 46
Figure 23. Generating two random parameters. 48
Figure 24. Request presented in Figure16 after alterations 49
Figure 25. JMeter reported results when running 50 users 51
Figure 26. Scenario definition via objects 52
Figure 27. A request payload passed as a RawFileBody 53
Figure 28. Extraction of a csrfToken 54
Figure 29. Definition of regular expressions for syncId and clientId tokens 55

Figure 30. Capturing synchronization tokens from a request 55
Figure 31. Setting default values for the lead times using session object 56
Figure 32. Printing information for debug purposes 57
Figure 33. LoginAndNavigate object 58
Figure 34. Statistics table generated by Gatling tool after test execution 60
Figure 35. Global information provided by Gatling after the execution of the second scenario is completed 61
Figure 36. Memory consumption over test execution 64
Figure 37. Structure of a src/test folder 65
Figure 38. Part of the JSON for an element with defined id 67
Figure 39. A response with an `Internal error` message 68
Figure 40. Parsing string to integer 69

List of Tables

Table 1. Regular expressions used to extract node ID of the "Producers" grid 44
Table 2. Regex expressions used to extract ID for the first available parameter id........48
Table 3. Summary of JMeter captured responses 50
Table 4. Common code snippets used in Gatling test 53
Table 5. Gatling run summary info 61
Table 6. Session's size measurement results..63

1 Introduction

Testing is an essential part of perfecting a software development process. A client and a provider who agree on the assets of a deliverable software product sign a Service-Level Agreement (SLA) to ensure that the requirements are met. Testing is therefore a tool that guarantees that these promises are upheld. There are various software testing types, techniques, and practices available for any aspect of a product. Load testing verifies that a system functions properly for expected simultaneous users.

Businesses that target a wide audience must also provide services online. However, building a client-oriented web application is not always an easy task. There are dozens of frameworks and libraries available on the market that facilitate and ease this development. Vaadin is one of the better known-known frameworks; it makes it possible to design and implement user interfaces entirely in Java.
To accurately load test an application, one must be aware of its distinct traits. The available tools and a scenario should be established and analyzed. Several resources that discuss load testing of a Vaadin application can be found online. Unfortunately, none of them provide a complete picture of how a test can be conducted, nor of any peculiarities of the test that one might need to beware of. Moreover, most of the available material is out of date and concerns the Vaadin Framework 8. The goal of this Master's thesis is to establish a way of load testing a Vaadin Flow application. To do this, it is important to discuss available performance improvement techniques. Three research questions are set to help achieve this goal:

- *How does one perform load testing for a Vaadin Flow application?*
- *What are the distinctive features of a Vaadin app from a load testing perspective?*
- *What are the performance improvement approaches utilized during the development and deployment of an application?*

This thesis is divided into seven chapters, which can be classified into three groups as follows: background (Chapters 1-3), implementation (Chapters 4-5), and retrospective (Chapters 6-7). A Flow-based application's state is stored on a server; therefore,

improvement techniques studied and presented for thesis are predominantly on the server-side. Chapter 2 presents load balancers and cache as available software and hardware options. A Java Virtual Machine (JVM) memory structure and leaks scenarios are reviewed. The software family used for performance testing is introduced and a definition for load testing is given.

The Vaadin framework's building blocks are outlined in Chapter 3. Both the server-side and client-side parts are presented. A simple component is built to demonstrate a connection between the client-side techniques and a Vaadin application. The chapter closes with a discussion of the performance of a flow application, which concludes the section on theory and background.

Chapter 4 introduces the JMeter and Gatling tools that are used to create and execute load tests. The prerequisites for building a test against a Flow application are established by defining a Flow application's unique features. Chapter 5 is the central chapter of this thesis, as it covers practical details of test configurations and implementation details. A procedure of load testing a Vaadin application is defined. The tests implemented have been conducted for the Finnish Meteorological Institute (FMI), who has kindly provided access to their application. This chapter answers the central thesis question, which is *how to perform load testing for a Vaadin Flow application.* Numerical values for session size and response times are presented as well. The chapter then concludes integrating tests into a maven build's verify phase.

Chapter 6 is an analysis of the research of the thesis. It contains a description of the difficulties faced during the assembling of tests and outlines some limitations of the tools. The parts of the implementation that could have been done differently are also discussed. Alternative solutions available on the market are also presented. An evaluation of the work is presented, and future steps are examined. The last chapter is a conclusion, which summarizes the results and implications of the work.

2 Load testing and application performance

Our world has changed dramatically over the last 20 years, with ever-evolving and expanding offerings presented on the Internet. With business continuously moving to the web, more and more services [1] are becoming available online.

In a competitive software business-environment, it is critical to ensure availability and responsiveness and to meet user expectations, thus enhancing revenue and achieving business targets. Based on research conducted on online retailers, a half-second difference in page loading time may make a 10 % difference in sales [2]. To verify that SLAs and Quality of Service (GoS) requirements are met, various types of tests are run against target software. Each of these tests is aimed to trial a different aspect of a product, from functionality to performance.

This chapter provides background on load testing, which is the cornerstone of the thesis. It also discusses methods and techniques available on the market to improve the performance of web applications as well as Java language and its memory management. By the end of this chapter, the thesis question ***What are the performance improvements approaches utilized during development and deployment of a web application?*** is answered.

2.1 Performance testing

Performance testing is a non-functional type of testing whose goal is to identify and, where possible, eliminate *stability*, *availability* and *scalability* issues of software [3]. It is an umbrella term that encompasses scalability, load, and stress testing. Although those words are sometimes used interchangeably in the literature, in this thesis they are differentiated and will be defined in subsequent sections.

Performance requirements for software can be divided into two parts: *service-oriented* such as availability and response time and *efficiency-oriented* such as throughput and utilization [4]. Availability requirements for applications vary significantly. Companies competing for customers aim for *high availability*, which is the 5th availability class, guaranteeing 99.999% (five nines) uptime of service. In other words, a permitted downtime for a service is only five minutes per year [5]. To achieve this level, an application often runs on multiple servers to distribute an incoming load.

Another goal of performance testing is to carry out a capacity estimation, which includes an evaluation of the Random-Access Memory (RAM) needed to function smoothly and a number of servers on which a new application is deployed. For this purpose, load testing is employed, among other methods for estimating minimum requirements.

2.1.1 Load testing

Load testing is a type of performance testing used to verify and study the behavior of a system under generated load. By simulating hundreds or thousands of simultaneous users, it helps to reveal a system's performance, investigating metrics such as *latency* and *Central Processing Unit (CPU) utilization vs. request amounts*.

To obtain valid and feasible results, the most critical and common flows should be tested. Checking the load and response time of a *Contact Us* page is less important than verifying that money transferring operations are consistent under different loads in an online banking app. Examining non-critical paths does not add any value; therefore, ideally, a test scenario should be obtained from a real user iteration or based on insights of a person who knows the system.

Many tools for running web application loading tests are available on the market. Among the best known are LoadRunner, JMeter, and Gatling. A free version of the latter two is available, including a recorder tool, which creates a test scenario/script from recording a user iteration with an application in a browser. JMeter and Gatling will be reviewed in greater detail in Chapter *4*.

2.1.2 Scalability testing

There seems to be no generally accepted definition of *scalability*, though it is used extensively in the literature [6]. For the purposes of this thesis, I will define scalability as the ability and property of a system to scale and continue to perform well as a user changes demands [7] [8].

Achieving scalable software is the responsibility of the whole development team at any stage, but the most impactful decisions are made during architectural planning. A system should be designed from the ground up to be scalable; making changes later will increase expenses significantly. Testing of a newly implemented feature or modification against requirements to verify no regressions are introduced to a codebase should be an automatic

routine. As part of this process, scalability testing checks the ability of an application to adjust itself to meet user needs by simulating changing load traffic. Modules that decrease performance should be identified and fixed. Any hardware limitations that are revealed should be addressed if user growth is expected. At this stage, there is no need to verify performance under a load exceeding a forecast; this is a goal of stress testing.

2.2 Performance enhancements

There are two ways to improve scalability of an application at a deployment stage: scaling up or scaling out. The latter is better known as horizontal scaling. As the name implies, new server replicas with a deployed application are added to a set-up. The vertical approach, on the other hand, improves scalability by enhancing a running server through increased hardware capacity and power (CPU, RAM). However, it may ultimately fail when there is no better hardware available, or the cost of maintenance becomes uneconomical. While horizontal scaling may require modifications to a codebase of software, it is the preferred of the two methods, promoting rapid and cost-effective growth of service [9].

2.2.1 Load balancers

The fastest way to handle a request is not to handle it at all. While sometimes this can be achieved by serving previously cached data, other times the request will be forwarded to a server for processing. An application is generally deployed and run on different servers to increase its availability and reduce response time. Load balancers are then employed to distribute the workload between replicas equally.

A load balancer is a hardware device or software program, whose primary goal is to distribute a workload (incoming requests) among servers behind it. Servers can be grouped in a cluster, where nodes are aware of one another and can share information, or in a farm, where servers operate independently. A load balancer can operate at the application or transport level of the OSI model. The seventh(application) layer load balancer makes routing decisions based on the content of a received message such as a cookie or URL. In contrast, the fourth level(transport) acts on individual Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) network packets [10].

A load balancer periodically checks which of the managed servers is still active and can respond to an incoming request. A server that ignores check connection attempts will be

excluded from a pool of available nodes [11] [12]. This ensures that work is distributed only among servers, which are capable of processing it.

For users, this process is transparent, meaning that they will not know a load balancer is being used at all. In case of a default server failure, an incoming request will be redirected to one node among a pool of active nodes. From a user perspective, a workflow can be described as follows:

1. User hits the Uniform Resource Locator (URL) in a browser.
2. A request reaches a load balancer.
3. Based on an underlying load balancing algorithm, the request is propagated to a chosen server.
4. The request is processed on the server.
5. A response is sent back to the client. (The fourth layer load balancer does not require that traffic be routed back through it, whereas the application balancer does.)

As mentioned in the third step, a load balancing algorithm decides which server will handle an incoming request. There are multiple algorithms available; the most commonly used are *Round Robin* and *Least connections* [13] .

A load balancer that uses a *Round Robin* algorithm distributes requests among servers in turn. For example, having three servers behind a load balancer, the first request is sent to the first one, the second to the 2nd server, whereas the 4th request is sent again to the first node. Although this algorithm is the easiest to implement, it does not consider the current workload of a server. Each request takes a different amount of time to be processed, resulting in some servers being overwhelmed [14]. To overcome this problem, *the Least-connection* algorithm can be used instead. A request will be sent to a server having the least connections open, balancing the overall workload. Both round-robin and least-connections algorithms have weighted versions, where hardware capabilities of servers are taken into considerations. Each server has a value assigned to it by an administrator, and the bigger the value it has, the more amount of work it can process. For example, assuming the first server weights four and the second is of weight 1, then a round-robin

configured balancer sends every fifth request to the second server. By contrast, the least-connection algorithm re-calculates the capacity of a server based both on weight and number of current connections to distribute requests.

In the case of a stateful application, requests from the same user must always be sent to the same server, which has created and stores the session [10]. A solution to this consistency issue is sticky sessions, also sometimes referred to as a session affinity. Enabling this feature makes load balancer route traffic to the same server instance for requests coming from one user.

There are various ways to define and later determine which server has a session object for a user and where an incoming request should be sent. Standard identifiers for the 4th layer load balancer are source's IP or VIP address and a port number specified in a TCP/UDP packet. For a load balancer operating at the 7th layer, request cookies are usually used to find a server. A request from a new user can be sent to any server, based on an underlying load balancer algorithm.

Load balancers improve scalability and load performance of an application significantly. Nevertheless, employing LB should not introduce a single point of failure for the system. Deploying a second load balancer is an example of a failover strategy, which might be utilized. A heartbeat mechanism can be used to verify a primary load balancer is active. If a primary load balancer crashes, the second node will become a new default.

2.2.2 Cache

Another technique that is widely used to improve latency and throughput of an application is caching [15]. A cache is a temporary data storage where already fetched information can be persisted for faster access in the future. The best candidates for caching are static resources, or those that change infrequently, such as pictures, JavaScript (JS) files, and Cascading Style Sheets (CSS) stylesheets. Another candidate for caching are operations that are expensive to calculate.

Between client and server, various caches are available, from a browser and application cache to Content Delivery Networks (CDNs) and reverse proxies in front of an application server. Caches can be divided into two groups based on an allowed access level: shared (available to multiple users) and private (single user has access) [16].

2.2.2.1 The Hypertext Transfer Protocol (HTTP) caching

To specify which resources are cacheable and for how long, developers define values for HTTP cache headers. A `cache-control` directive dictates how content can be cached. Available options for the rule are `public`, `private`, `no-cache`, and `no-store`. Without this header set, all other applied caching rules would not have any effect [17].

A header with `public` permission allows a resource to be stored at any caching level. Consequently, it is not a viable option for sensitive content, access to which should be limited. Based on this, either `private` value, which dictates that storage is allowed only in a private user's cache, or `no-store` value, which prohibits caching completely, should be used.

To specify the amount of time needed for a previously downloaded resource to be considered alive (not stale and applicable for re-use), the `max-age` is used. After a value has expired, a resource has to be fetched again.

A cache miss occurs trying to read data from a cache, which is not present there. In this case, the request is forwarded to a higher cache in a chain (closer or to an application server).

2.2.2.2 Reverse proxy server cache

A proxy server is an intermediate node between a client and a server, passing a user's request through and forwarding it to the server if needed. A forward proxy is commonly used in enterprise systems to restrict access to the Internet and enhance security. All requests are forwarded directly to a proxy, and if it has the requested data available, it will send it back immediately. The primary benefits of a forward proxy are savings in network bandwidth and faster client response time. A reverse proxy, by contrast, prevents clients from accessing an application server directly, as a DNS lookup for the site's IP- that user wants to reach- is resolved to the proxy's IP [18]. The user is not aware that content is provided from a reverse proxy server, assuming it communicates directly to the application server. Aside from faster content delivery and improved security, the number of requests needed to be processed by a server is reduced. A decreased number of requests for fetching static data allows allocating resources to serve requests for dynamic content

[19]. Various implementations of reverse proxies such as Varnish [20] and NGINX [21] are available.

2.2.2.3 Database cache

An enormous amount of data needs to be stored and processed in business applications nowadays. For many of them, a database becomes a common performance bottleneck. Queries to data repositories are usually identified as among the most substantial reasons for low scalability of software.

To improve throughput and responsiveness of a database, additional cache memory is needed. It can be integrated within the database itself, added as a local cache (discussed with the application cache), or located on a separate server instance.

Multiple caching strategies are available, and some of them are described below [22] :

- *Cache Aside*. This type of cache is separate from a database. It is positioned *aside* and does not communicate with a database directly. When a user requests data, the cache is always asked first. If no result is present, an application retrieves missing data from a database and writes it back to the cache. Data can be written in any form. The *cache aside* approach is great for read-heavy applications, yet the first request will take longer because data are not present in the cache yet. A drawback of this strategy is that an application itself is responsible for writing to and handling data between an underlying datastore and the cache, adding overhead and complexity to its codebase [23].

- *Read-through*. As with a *Cache aside* storage, a *read-through* strategy implementing cache is always asked first before issuing a request to a database. Data is also loaded lazily when first requested. Nevertheless, contrary to the above option, a *read-through* cache is aware of an underlying database and usually initiates read and write actions to it on a user's behalf. This also implies that information is stored in the same form as in the original data source.

- *Write-through*. Data are always written to a cache as it is updated to a database. Consequently, the most recent version is always present in the cache and is not fetched again so long as its expiration date is in the future. The downside is that rarely accessed modified information is also written to a cache.

- *Write-Back*. Data are written to a cache but propagated to a database after a specified delay. This is a good choice for applications with a write-heavy logic.

Selecting a cache strategy depends on the use case and available resources. No matter which one is chosen, one must always ensure that stale data is not served. Setting an expiration date or updating data manually should be considered potential countermeasures. For instance, Redis [24] is often used as a cache along with the primary storage. Another example is a ReportServerTempDB database, which is used internally for a Report Server Database [25].

2.2.2.4 Application cache

In cases where some operations and calculations take a long time to complete or are expensive to recalculate, application cache is a practical storage solution to persist results during an application run. Data created or requested by one user are stored locally or distributed on outside servers for later re-use. Popular choices for local storage are Guava's cache [26] solution and cache2k [27]. Memcached is a cache memory distributed on multiple servers [28]. It is important to note that such an application cache is neither a replacement for a database nor meant to store a lot of information. If such outcomes are needed, another optimization technique should be considered.

2.2.2.5 Content Delivery networks

A larger user base implies diversity of users' physical locations. While caches described above increase the throughput and availability of any deployed to a server service, none of them brings improvements to the network performance. CDNs are generally employed to fulfill a service's SLA and ensure transparent and efficient delivery of content to end-users from different locations. CDN is not a replacement for any of the techniques presented earlier, but an enhancement to further improve performance.

CDN is a group of servers distributed over different geographical regions, which uses caching and/or the origin server's replication to deliver content to end-users. A request to a web-service is redirected to the nearest edge server, which increases reliability, responsiveness, and performance of the system as its content is located closer to the user. Numerous CDN providers are available. Among the biggest market shareholders are Akamai and Amazon CloudFront [29].

2.3 Java

Java is a widely used, strongly-typed, and object-oriented language. First released in 1995, it has since evolved considerably. By default, it is a language used to develop a server-side part of Vaadin applications. In this subchapter, a brief introduction to language details are given, and its performance influence on applications is discussed.

The development process of any Java application can be described as follows: a developer writes code in a `.java` file, then the `javac` compiler compiles code into a `.class` bytecodes (JVM instructions) file. A virtual machine will process those class files during an application run.

A JVM is an abstract computing machine, multiple implementations of which are available, each fulfilling Java Language and Virtual Machine specifications. Efficiency and performance of an application depend greatly on how memory is managed and allocated; therefore, I now turn to an analysis of memory management inside a JVM.

2.3.1 JVM memory management

Java has an automatic built-in memory management system that is responsible for the lifecycle of objects. Consequently, developers have no way of discarding any object explicitly. Memory can be viewed as a heap or not heap-based. A heap is created at a JVM start-up and is common for all executing threads. All class instances, as well as arrays, are allocated from a heap. The heap is

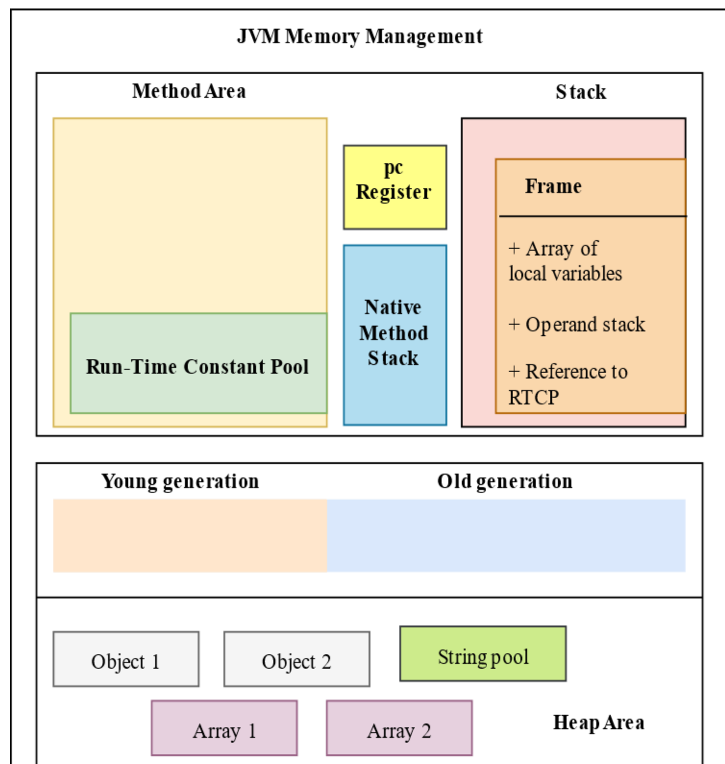

Figure 1. JVM Memory management

structurally divided into two areas: young and old generation, used in garbage collection mechanism. New objects are allocated from the young generation. Objects that have

survived a certain amount of garbage collection cycles are propagated to the old generation. Because garbage collection and storage management implementation details are JVM's vendor-specific (Oracle or OpenJDK, for instance), no exact type is assumed.

A method area also shared among running threads is logically a part of a heap area, but because as specification does not require the exact location it may vary from implementation to implementation. The method area contains per-class information, such as method code and data. It also contains the Run-Time constant pool, created per class or interface.

Java is a concurrent language, where each thread has its own stack. A stack is thread-private and not accessible for other running threads. Per every method execution, a new frame is created and allocated from the thread. The frame stores an array of local variables, an operand stack, and a reference to the method of class inside the run-time constant pool. There is always only one current frame and method being executed in a thread of control. The simplified diagram of the JVM memory structure is presented in Figure 1, adding such parts as native method stack and pc register, which were not mentioned earlier. Moreover, the method area is placed outside the heap [30].

2.3.2 Performance pitfalls and countermeasures

Except for an *Application cache,* all previously discussed improvements techniques are applied to a fully developed system, which might already be in production. This subchapter will try to bridge the gap by introducing programming pitfalls, avoidance of which can potentially improve the liveness, stability, and performance of any Java software.

2.3.2.1 Memory leaks

Garbage collection is a crucial process of memory management that deallocates unreached objects from memory. Objects created during a program runtime are always allocated from a heap. To be picked by a garbage collection process, an object should not have any strong references from the garbage collection (GC) root's objects [31]. An object reachable outside a heap, such as a local variable in a stack frame, an active thread, or a class loader, is called a GC root.

Under some circumstances, a GC may retain a no longer needed object, causing a memory leak. Thus, objects that are irrelevant to program execution are not reclaimed, and memory is not freed, causing heap size to grow over time. In the worst-case scenario, performance decreases, and a program crashes with the `java.lang.OutOfMemoryError` thrown. Often, such situations arise from incorrect handling of resources, bad coding practices, or use of third-party flaw software.

One of the most common scenarios that produces a leak is a resource left open after handling is complete. For example, after reading an input stream using `BufferReader`, `close()` must be called. The same applies to database connections and `ResultSet`. With Java 7, a new `try-with-resources` block was introduced to mitigate the risk of leaving resource implementing `AutoClosable` interface open, guaranteeing it will be closed at the end of statement execution [32].

Eliminating memory leaks improves the efficiency and stability of software.

2.3.2.2 Deadlocks

Parallelizing task executions can improve resource utilization and responsiveness. Java implements concurrency using threads. If processing capacity allows, new tasks can be started while existing ones are still running. Although performance can be enhanced significantly up to a certain point, one must make sure that excessive lock contention or deadlocks are not accidentally introduced.

A lock is an object held by a thread that allows for exclusive access to some resource. Deadlock is a situation where a thread waits to acquire a lock, which is held by a second thread, which in turn waits for another lock to be released by the first thread. For example, thread A has acquired X, and thread B holds Y a lock. Now, if both A and B try to acquire Y and X locks respectively, they are stuck in a deadlock and no further progress can be made. A Java program, compared to a database system, does not recover from a deadlock, leading at the very minimum to degraded performance and at maximum to a completely non-functional stale application [33].

The example described above represents a lock-ordering deadlock, which can be eliminated if locks that are used in conjunction, are always acquired in a fixed global

order. The same applies to a dynamic lock-ordering deadlock, which can be avoided, for instance, by acquiring the lock with the biggest hash first.

Starvation and livelock are two other problems to the liveness of an application caused by threads. Starvation occurs when a thread cannot obtain access to a needed resource such as CPU cycles. In liveness, in turn, the thread retries to accomplish operation, which always fails. While in starvation and livelock scenarios a thread is not blocked, no work can be accomplished either.

2.3.2.3 Connection pools

Creating and maintaining a new connection to a database is an expensive operation. Re-using of pre-created connections is a de-facto optimization technique used in software development called connection pooling. Instead of opening a new connection, an existing one is retrieved from a pool and reclaimed back when it is no longer needed. Several established implementation frameworks exist, including Apache Commons DBCP and HikariCP. The latter is the default pooling choice of the Spring Boot framework [34].

3 Vaadin Flow applications

Flow is an open-source Java web framework that is offered as a part of the Vaadin platform. It is the next generation of the well-known Vaadin 8 framework and has a completely re-written integration for a client-side part, which brought support for Web Components.

A web application developed with Flow runs on JVM; thus, any language belonging to the JVM family, such as Java, Kotlin, and Scala, can be used to write an app. Actively developed and maintained by Vaadin Ltd., the framework is licensed under the Apache 2.0 license, and the source code is available at the official GitHub page [35]. This chapter is an exploration of Flow. It outlines its essential components, which characterize any application built with the framework.

3.1 Communication model

Flow is a server-side framework that leverages automated bi-directional communication between a client (web-browser) and a server. A client-side Document Object Model (DOM) tree has a corresponding representation on a server-side with a User Interface (`UI`) element at a root. Each `Element` instance of a server-side DOM represents a real browser DOM element. The synchronization of changes that applied to an element happens automatically between a server and a browser. A communication process is transparent to a developer and happens on top of the HTTP or WebSocket protocols [36].

3.1.1 Deployment

When a request comes from a user to a server, the corresponding `VaadinServlet` (mapped to the specified path) receives it and delegates it to `VaadinService` for processing. Actual processing happens in the `VaadinServletService`, which implements VaadinService's abstract methods. The incoming request is further associated with `VaadinSession`, which determines the particular UI instance that it belongs to. `VaadinSession` contains relevant information needed for a Servlet to process a request, and it is usually stored inside an `HttpSession` object, although this is not required. `VaadinServlet` is an extension of an `HttpServlet` class, which implements a `Servlet` interface. A servlet is a Java program that runs inside a web server, which receives and responds to requests from a client. The diagram below shows

the relation between the classes described above; arrows follow the Unified Modeling Language (UML) convention.

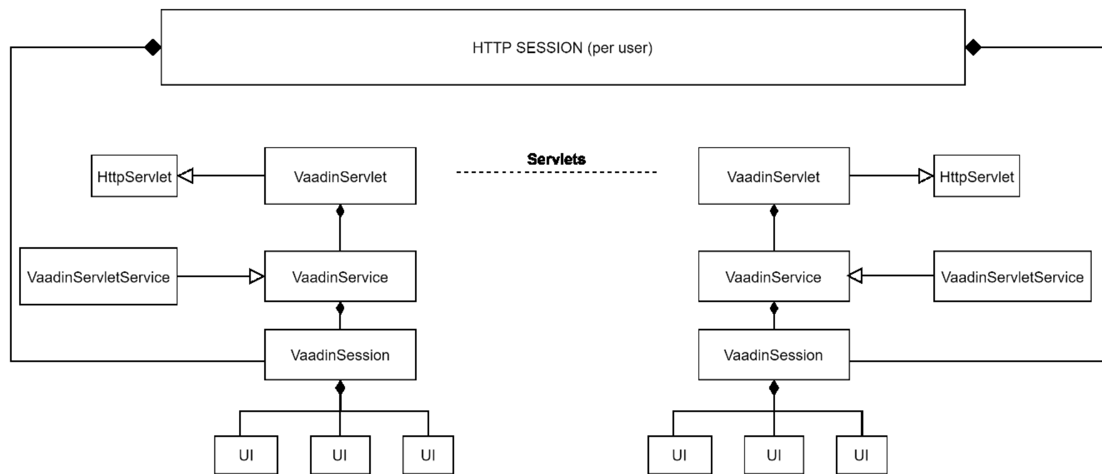

Figure 2. A request handling process

The Vaadin application is deployed to a Java application server, such as *WildFly* and *Tomcat*, implementing the Servlet Specification. The minimal required version to be supported by a servlet container provider is 3.1.

3.1.2 Push

In a client-server architecture (request-response workflow), communication is always initiated by a browser. To immediately receive updates to a UI, without waiting for the next browser request, a server push technology is equipped with the Vaadin framework. After a connection is established between a client and a server, updates can be sent asynchronously between the parties. By default, the transport protocol used by Vaadin Push is `WEBSOCKET_XHR`, which uses `WEBSOCKET` for a server to communicate with clients and `XHR` for clients to communicate with a server. As well as `WEBSOCKET` and `WebSocket+XHR`, `Long_Polling` is also available via the HTTP transport mode. Compared to other options, WebSocket enables a full-duplex communication channel over a single TCP connection.

A push method can be configured by setting a *transport* property. The framework adopted by Vaadin that provides push functionality is Atmosphere [37]. If a browser does not support the WebSocket, Atmosphere will revert to one of the supported transports.

There are three modes available for push management: *disabled* (push is not used), *automatic* and *manual*. As the name implies, the *manual* mode requires a developer to call push explicitly, whereas, the automatic mode does not require developer involvements, as changes are pushed after a session lock is released.

Push can be configured for an individual view or for the whole application. The same annotation, shown in Figure 3, is used in both cases. The only difference is the annotation's placement. For an application-wide setting, it is in the main layout, and for a view, it is in a view class itself. The default mode for push, when unspecified, is automatic.

```
@Push(transport = Transport.WEBSOCKET,value= PushMode.AUTOMATIC)
```

Figure 3. Push definition with transport and mode specified for an individual view

3.1.3 User session

The HTTP is a stateless protocol. To associate requests coming from one user, there are different session tracking mechanisms. One of the most used techniques on the web, also adopted by Flow, is cookies. Therefore, for a Vaadin application to function correctly, cookies should be enabled on a client's browser.

A new session is instantiated the first time a user visits an application's page. A server cookie, named `JSESSIONID` by default, is generated by a servlet container and sent with a response back to a client. Any subsequent request from the same user contains the cookie that is automatically added by a browser, which associates the request with a session on a server. A created `HttpSession` is scoped at an application level [38].

3.2 Client-side

An element's client-side part is built on top of the Web Components. Web Components is a specification and a set of Application Programming Interfaces (APIs), which defines a way to build new custom Hypertext Markup Language (HTML) tags to use on web pages. The specification is based on four smaller specifications: *Custom Elements*, *Shadow DOM*, *ES(ECMAScript) Modules,* and *HTML template* [39]. Together, they provide a way of building custom components with encapsulated functionality. Encapsulation ensures that an object is protected from unwanted modifications by restricting access to its internal representation or state.

ES Modules, also sometimes referred to as JavaScript Modules, is a part of the ECMAScript language specification, which JavaScript conforms to. JS has a standardized syntax, which defines how code that is structured in modules can be exported to and imported from other modules. As ES Modules is a feature of a JS language that is not directly related to a Flow application's use and development, it will be omitted from further discussion.

The current Long-Term Support (LTS) version of Vaadin uses the Polymer web component library to facilitate and ease the creation of custom components.

3.2.1 Shadow Domain Object Model

DOM represents the structure of an HTML page as a tree of nodes that have parent-child relationships. The original DOM API does not support encapsulation, making it hard to preserve the identity fields and stylistic rules. Consequently, HTML and CSS documents imported from elsewhere might overlap in both style names and IDs, causing bugs in the rendering of the pages.

To address this issue, a *Shadow DOM* API specification introduced a scoped DOM called a shadow tree. *Shadow DOM* is attached to an element of the tree and encapsulates its content from the other components presented in the document. Any node added or style applied to this shadow tree is local to a host element and is not accessible using the regular DOM API [40].

3.2.2 HTML Template

A template is a reusable fragment of an HTML content, which can be inserted in a document using JavaScript during runtime. It is not rendered nor executed by a browser until it is activated [41]. A template itself does not have child elements in DOM, and its content is located inside a `DocumentFragment` object [42]. Flow supplements bind with Polymer templates to simplify the creation of custom components. To use a template on a server-side, a Java counterpart class extending `PolymerTemplate` is created. A class can also implement an API to modify a template [43].

3.2.3 Custom Elements

Custom Elements is a JavaScript API, which specifies the way to build custom DOM elements. These elements can be divided into two groups, depending on the extended

element: autonomous and customized. A customized element inherits properties from an existing HTML element, simply adding functionality to it. An autonomous element, with rare exceptions, extends an `HTMLElement` interface.

For a custom element to be made available on a page, it should be registered to a `CustomElementRegistry`. Subsequently, an autonomous element can be added to a page using a tag name added to a registry [44] [45].

In this thesis, a simple component called `CustomDiv` was created to illustrate how the *Custom Elements*, *HTML template*, and *Shadow DOM* work within a Flow application. Complete code for it can be found in Appendix A; in this section, only the relevant parts are presented.

The content and styles of a new element are specified inside the `<template>` tag in a JS file, which represents an *HTML Template* technology. *Shadow DOM* encapsulates styles and content defined inside the tag. As part of the custom elements' specification, the element is added to the register. Figure 5 contains the defined element.

To use an element from a Java API, a counterpart class extending a `PolymerTemplate` should be created. For simplicity, a class for the custom element defined above is a `CustomDiv`. Along with a Vaadin button, this component has been added to a `VerticalLayout`. The rendered layout is presented in Figure 6.

```
Button button = new Button("Vaadin Button");
//A newly created component using HTML template
CustomDiv customDiv=new CustomDiv();
add(customDiv);
add(button);
```

Figure 4. Adding elements to a VerticalLayout.

```
static get template() {
    return html`
       <style>
            .displayColumn {
                display: flex;
                flex-direction: column;
                background-color: antiquewhite;
            }
       </style>
       <div id="customDivContainer" class="displayColumn">
            <paper-input id="inputId" label="Put your name here!">
               </paper-input>
            <button style="width:120px" on-click="handleClick">
             Alert the name!</button>
       </div>`;
}
    //Register a new custom component
    customElements.define(CustomDiv.is, CustomDiv);
```

Figure 5. Content of a custom-div.js template

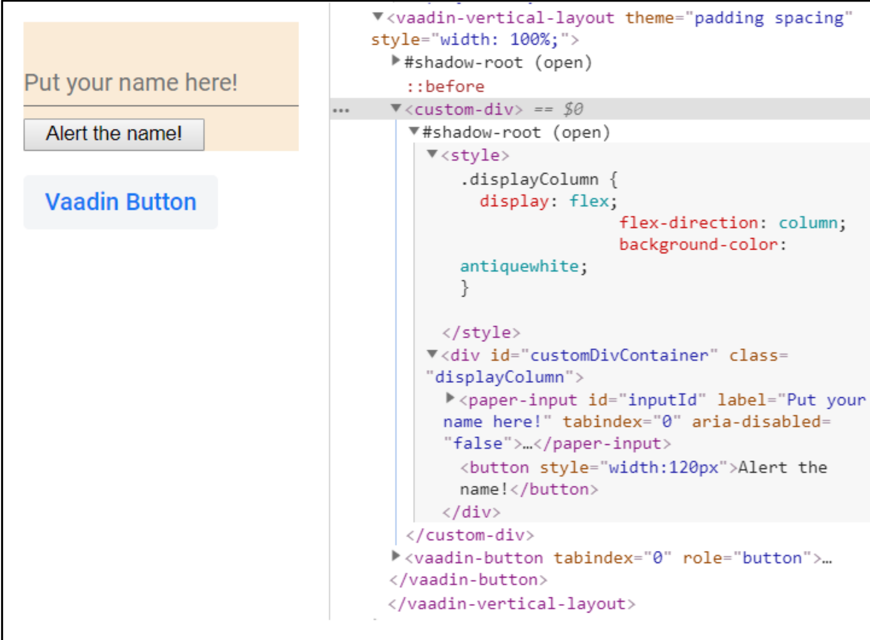

Figure 6. CustomDiv rendered in Google Chrome browser

By inspecting an attached element in Chrome DevTools, one notices that a reusable `custom-div` tag from a `CustomDiv` class was created. A Shadow DOM of this element contains encapsulated styles and input with a button field specified inside the `template` tag. Attempting to modify styles or call functions of inner elements from a page using regular HTML API has no effect. For example, attempting to get a value of paper-input element calling `document.getElementById("inputId").value` throws `NullPointerException`, as there is no globally available element with the `inputId` ID.

3.2.4 Optimizations and IE11 support

To bundle JavaScript files, Flow uses a webpack library. As well as bundling, webpack also performs transpilling, gzipping, and minification of JS files, which improve the client-side performance of an application. Better performance is achieved by a reduced amount of HTTP requests aimed at fetching frontend resources, as only one minimized bundle is downloaded. During minimization, unused code, including any comments and spaces, is removed. Variables are renamed to reduce the size and consequently load time. Those optimizations are run once an application is packaged in a production mode.

JavaScript transpilling is a conversion of a newer version of an ECMAScript syntax into the older one, supported by a required environment. It is needed for IE11, which does not have native support for a syntax introduced with an ES6 and used by Flow [46]. While transpilling transforms script so that it is understandable in a browser format, some of the features needed have not yet been implemented in IE11 and cannot be used. To emulate them, a polyfill, which is a JS piece of code used to provide a missing functionality on older browsers, is included [47]. As of March 2020, IE11 still lacks support for *Custom Elements* and *HTML Templates*. Performance in IE11 is considerably worse compared to the other browsers, due to the need for polyfills.

3.3 Performance

As mentioned above, a Vaadin application is stateful. A state is persisted on a server-side within a user session. It contains a list of UI components with their internal states, such as captions and style names, data displayed on a screen -for example, a combo box's content- and any other data referenced from UI objects [48]. Thus, performance of an application significantly depends on the amount of data stored, used, and shown for a user and can be configured by a developer.

The predominant performance issues of Vaadin applications usually do not arise due to the framework itself. These issues include retrieving and storing too much data from a backend, initializing views prior to their actual use, and complicated and un-optimized Structured Query Language (SQL) queries to a data source. Each issue has a significant negative impact. These examples are sources of server-side performance issues, which increase memory print and response time. On the client-side, rendering time can increase if a UI layout has a complicated structure or has too many components added to it.

A lazy loading technique is used to delay the fetching of resources until they are needed. For example, a Grid component, which is used to display tabular data, employs this mechanism to only fetch visible rows of information. Once one scrolls, the next section of the required data is loaded. This is an example of lazy loading between an application server and a client that is implemented in the framework by default. A similar method is utilized for a combo box component.

Data fetched from a server is usually first retrieved from backend storage, such as a database. When a significant amount of data needs to be fetched, an example of a recommended approach is to implement lazy loading for an interaction with a database using pagination.

Another approach widely used to improve the availability of a web application is session replication. While, in theory, standalone session replication is possible with Vaadin, its implementation is not trivial. *Sticky sessions*, discussed in the second chapter of this thesis, are a recommended approach to achieve results comparable to standalone session replication. Nevertheless, if high availability is a requirement, then a nearly identical outcome could be obtained utilizing both *sticky sessions* and *session replication*. In such a set-up, a session is serialized and replicated to another server (or saved to a database) after each request. However, requests coming from the same user are still processed by the original server [49]. In the case of failure, a user's request will be redirected to another server. If a replicated session is persisted into a database, it will be loaded into a server's memory, which becomes a new default choice for further communication.

While session replication is a powerful technique to improve availability, it can bring its own performance bottlenecks to the app, especially when the session size is big. Therefore, sometimes a trade-off must be made.

All performance techniques and improvements presented in the previous chapter, such as load balancing and caching, are applicable to Vaadin applications as well. If performance constraints are revealed during testing, the introduced methods should be considered. An application, which is expected to process more traffic over time, must introduce the discussed techniques to guarantee the availability of its services. The next chapter lays out a foundation for building load tests, which helps in exposing and detecting performance bottlenecks. Different profilers are used in conjunction with load testing

tools to locate and analyze problematic parts of the code. For instance, commercial *JProfiler* tool and bundled with a JDK *Java VisualVM* are widely adopted options for profiling a Java application.

4 Designing load tests for a Flow application

The previous chapter has discussed the theoretical background of performance testing and the internals of a Vaadin application. The answer to the thesis question "***How can a Flow application be load tested,"*** is given in the next chapter by introducing a procedure on an existing production-ready Flow application. This chapter, instead, discusses the vital parts of JMeter and Gatling tools, which are employed to build and execute a load test.

To facilitate and simplify testing, both tools provide utilities to record user interaction with an app. A test script, which is generated from captured requests, provides a base for further development. Subsequent subchapters will show, how script can be recorded and adjusted to accommodate unique Flow elements, which are defined and discussed next.

4.1 Flow features

To answer the second thesis question, "***What are the distinctive features of a Vaadin app from a load testing perspective?",*** the unique aspects of a Flow application must be considered. These traits dictate rules to follow during test creation and, therefore, it is essential to establish them. Failing to do so, results in the best-case scenario, in a failure on a test start-up, and, in the worst-case scenario, in incorrect and misleading results. The following subchapter will answer this question.

4.1.1 An HTTP request

An HTTP request is a base unit of a web application load testing. To provide additional security measures and synchronization, the framework adds special-use attributes to a request's body. For a simulation to be successful, one must consider these attributes.

To inspect a payload and the URL of a request, which is sent to a server, a simple starter application, downloaded from the Vaadin website, was deployed to a local jetty container. Its view consists of a vertical layout and a button. Clicking the button issues a request to the server with this URL: `http://localhost:8080/?v-r=uidl&`**`v-uiId`**`=1`.

The request payload is

{"**csrfToken**":"9f558d39-3f2d-4c94-b1cb-
d9711e2700e5","rpc":[{"type":"event","**node**":**4**,"event":"click","d
ata":{"event.shiftKey":false,"event.metaKey":false,"event.detail

```
":1,"event.ctrlKey":false,"event.clientX":49,"event.clientY":32,
"event.altKey":false,"event.button":0,"event.screenY":103,"event
.screenX":1585}}],"syncId":0,"clientId":0}
```


The `v-uiId` URL parameter with `syncId`, `clientId`, `csrfToken` and `node` attributes are of interest for this thesis, as they have an impact on a load test configuration. A brief introduction to each, with an emphasis on their functional purpose, is given below.

4.1.2 A csrfToken

A `csrfToken`, which is included in the payload, is a defence mechanism implemented by Vaadin against a Cross-Site Request Forgery attack (CSRF). It prevents undesirable actions being performed on a user's behalf by a malicious program on an authenticated site. Each user has their own unique token generated by the framework. A token is created and passed to a user with the first response from a server. With every subsequent request, it is sent back. The framework verifies both the validity and the existence of value for each user request [50].

Due to the unique nature of the token, replaying a previously recorded load scenario as is will not produce any meaningful output. Trying to do so results in an `InvalidUIDLSecurityKeyException` exception thrown on a server-side while processing a received request. This error indicates that a retrieved token does not have an expected value.

To simplify test execution, if possible, one can disable the CSRF protection during load testing. Otherwise, a generated value must be extracted from the first response and parameterized for all the following requests.

4.1.3 v-uiId and node

The `v-uiId` is an ID with a goal of associating the server-side UI with the corresponding client-side instance. A new UI is created every time a new tab or a browser window is opened or refreshed. UIs belonging to one `VaadinSession` are stored inside the <Integer,UI> map, where a key value is an ID of UI obtained from a request.

A `node` is a unique value of an element and its goal is to associate a client-side DOM element with a server-side component. It is used in Remote Procedure Call (RPC) requests to exclusively identify the component's listener that must be fired or to identify

where changes must be applied in the UI. The next value (incremented by one) is assigned to a `node` during an attach event. If the UI's hierarchy remains unchanged, this value is a constant for an element.

4.1.4 Synchronization tokens

To ensure messages are processed in order, the framework internally uses two synchronization tokens: `syncId` (server-to-client) and `clientId` (client-to-server). Every time a new request is sent, a `clientId` is incremented by one and added to a payload. If expected and received values do not match on a server, a re-synchronization attempt should be issued. Currently, a re-synchronization implementation for a case of `clientId` mismatch is being developed [51]. At the time of writing this thesis, if an unexpected value is received on a server, the exception below is logged in server logs: `java.lang.UnsupportedOperationException: Unexpected message id from the client. Expected sync id: X, got Y.`

The `syncId` token, in contrast to the `clientId`, is incremented by one with every response. A client always appends the last seen `syncId` value to a payload, which guarantees that requests are processed in order on a server. If a received value is bigger than expected, the operation is postponed until the missing messages arive. A `syncId`'s value verification can be disabled in an application to simplify the load and scalability tests run.

There is one more available attribute not presented in the payload above, which is `pushId.` If an application is push enabled, each request has a unique identifier, which associates it with the current session [52]. If this is the case, then an ID should be extracted in the same way as a `csfrToken`, as discussed in the next chapter.

4.2 JMeter

First released at the end of 1998, JMeter became a default choice for many companies that needed to measure performance of their software. In 2016, it was estimated that half of all companies used JMeter for load testing [53]. Written in Java language, it can be run on any operating system compatible with Java.

In this subchapter, the fundamental and crucial features of JMeter that are used throughout the thesis are presented. Therefore, the aim is not to offer a comprehensive guide to JMeter, but to outline the features later used to load test a Flow application.

A saved test is stored in a `.jmx` format, which is effectively a `.xml` file. Its root element is a *Test Plan* (represented by a `jmeterTestPlan` tag), which, when run. contains all configurations and components to be executed by JMeter. All the subsequent discussed elements are children of some order to it.

4.2.1 Recording

JMeter provides a built-in template named *Recording* to capture a user's navigation on a web page. *User Defined Variables, HTTP Request Defaults, HTTP Cookie Manager, Thread Group,* and *HTTP(s) Test Script Recorder* are components that are added initially to a generated test plan.

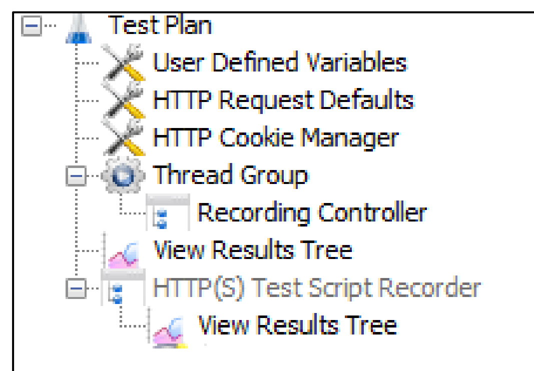

Figure 7. A default Test Plan generated from Recording template

The *HTTP(s) Test Script Recorder* element allows JMeter to capture and store requests that are caused by user actions while browsing a tested web application via a regular browser. By default, recorded HTTP requests are stored as samples under the *Recording Controller*, which is usually added under the *Thread Group*. Furthermore, intercepted requests that should not be added into a final script are specified here. In the *Request Filtering* tab of a recorder component, *URL patterns to exclude* are configured. Suggested elements to exclude, such as images and fonts, along with the custom patterns to ignore, such as accidental Firefox requests, are added to the rules.

4.2.1.1 Thread Group

At the heart of any *Test Plan* is a *Thread Group* element, to which samplers and logic controllers are added. The number of threads and the time needed to start them (the "ramp-up period") are also specified here. Every thread represents a single user, thus simultaneously running a few simulates concurrent access to a server. A test plan can contain any amount of thread groups, in which each serves as a separate scenario.

Samplers and *logic controllers* are two types of controllers provided by JMeter to manipulate the flow of test execution. *Samplers* handle dispatching requests to a server. For each of the supported request types, a sampler is available. For instance, JMeter's *HTTP* and *TCP requests* are samplers that provide the functionality of sending HTTP and TCP requests, respectively. *Logic controllers*, on the other hand, alter and control an execution order of requests. For example, the *If* and *Only once* controllers belong to this group [54].

4.2.1.2 An HTTP Cookie Manager

As previously discussed, cookies are widely used to implement session management. To simplify configuration, JMeter provides an *HTTP Cookie Storage Manager* element, which handles cookies during a load testing run. JMeter automatically extracts and stores cookies from an application response, which are then appended to all subsequent requests. As a result, each thread has its own session [54].

If an HTTP Cookie manager is not added to a test plan, the value must be extracted manually from the first response and passed to the following requests. The extraction process conducted on an example of a `csrfToken` is described below.

4.2.1.3 User Defined Variables and HTTP Request Defaults

To define constants and commonly referenced values such as a host URL address, *User Defined Variables (UDV)* element is added to a test plan. A value is stored to a variable and later referenced by its name using `${variableName}` syntax. UDVs are shared among all Thread Groups, and, once set, their value cannot be changed. In Figure 8, a configured UDV element, that uses a test discussed in the next chapter, is presented.

User Defined Variables

Name: User Defined Variables

Comments:

User Defined Variables

Name:	Value	Description
host	localhost	
scheme	https	
first_producer	${__Random(2,18)}	First producer value
second_producer	${__Random(2,18)}	Second producer value
third_producer	${__Random(2,18)}	Third producer value

Figure 8. A JMeter's configured User Defined Variables component.

The *HTTP Request Defaults* element contains default values that are shared between all *HTTP Request Controllers*. If no value is specified in a controller, the default will be the value inherited from the element. For example, a server name or an IP address should be specified in the *HTTP Request Defaults* element if most requests are targeting the same server.

4.2.2 Listeners

There are various types of listeners available that allow a test plan to view the results of the samplers' requests and responses. For example, a commonly used *View Results Tree* listener gathers details about all received responses, whereas the *Aggregate Report* contains statistics for each request sent in a test. Based on the listener type, results can be displayed in a tree, graph, or table mode [55].

4.2.3 Regular Expression Extractor

JMeter provides helper extractors to retrieve information from a response. There are multiple built-in options available, such as a *JavaScript Object Notation* (*JSON)* extractor and *CSS Selector* extractor. The choice of extractor is based on the type of data to be parsed to obtain results. Extractors can be found from a *Post Processor* menu available for an *HTTP Request Sampler* on a right-click of the mouse.

The *Regular Expression Extractor* is the only extractor used for this thesis. Below, using this extractor, this thesis shows how a `crsfToken` is extracted from a response. This is a routine procedure for any application in which a CSRF protection is not disabled for a test run.

Vaadin-Security-Key is the name of the attribute that contains a generated CSRF token, which can be extracted using a Universally Unique Identifier (UUID) pattern. By combining this, a regular expression `Vaadin-Security-Key":"([a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}){1}"` identifies the token. To verify regex works, one can navigate to the *View Results Tree* listener → *Response Data* → *Response body* and paste the value into a *find* input field by selecting the *Regular Exp* checkbox. If a pattern has matches, these will be highlighted in green. A configured extractor is presented in Figure 9 below.

Figure 9. Configured Regular Expression Extractor for a CRSF token.

One can see that in the picture above, two other parameters are set: *Templates* and *Match No*. The last one specifies which match by ordinal number should be stored into a variable if there is more than one option that fulfills the criteria. As will be shown in the next chapter, there are two values used during testing: *1* and *-1*. Value *1* guarantees that the first found occurrence will be persisted, while *-1* makes JMeter process all the variables saving them in the form `variableName_n`. Any negative number could also be used instead of *-1*.

Sometimes, there is a need to obtain multiple values from a request or response. *Match No.* defines which match's occurrence should be extracted. A *Template* pattern, in turn, defines which groups should be derived. For example, used extensively in regular expressions, the `$1$` template means that only the first group is stored.

4.3 Gatling

Gatling is a relatively new load and performance testing tool, which was first released in 2012. Built upon the Netty and Akka frameworks, it exploits asynchronous architecture that facilitates simulating users via messages rather than dedicated threads. Thus, compared to other load testing solutions, it is less resource-intensive, and more simultaneous users can be simulated in the same set-up.

Gatling emphasizes the responsibility of a developer. The recording tool is minimalistic, yet it provides basic functionality to capture requests and generate a simulation stub from them. Further modifications are accomplished using Scala language in conjunction with the Gatling's domain-specific language. As a result of empowering the creation of tests as code, Gatling has found significant support in the developers' community. Th general settings available for a recorder are described in the following subchapters.

4.3.1 Recording

The recorder is started by running *recorder.bat* (or *.sh*), which is executable from an installation's folder *bin* directory. Once the Graphical User Interface (GUI) is displayed, it is possible to configure basic network configurations such as port and proxy, simulation rules for redirects and static recourses, and filtering policies.

Gatling uses `WhiteList` and `BlackList` filters to define the requests that should be captured or ignored by the recorder. Once a recording is finished, one can remove the applied filters from a generated script. The recommended approach for fetching static resources rather than capturing and re-playing requests is to let Gatling retrieve them by selecting an *Infer html resources* checkbox. This way, Gatling will simulate a browser's behavior and fetch embedded files asynchronously.

4.3.2 A request building blocks

The fundamental unit of web application load testing is an HTTP request. In Gatling, it is defined using the `http("Name of an action")` syntax, which is followed by an HTTP method. Only two methods are used in the test created and described in the next chapter: `get` and `post`. The latter one is used in most cases, despite navigating to the `/login` and `/logout` pages.

An HTTP protocol utilized by scenarios is declared using an `http` object. A `baseUrl` method takes the root URL of an application as a parameter. Any path that is later specified in an HTTP request's method will be relative to this parameter. The defined protocol is passed to a scenario object via its `.protocols(httpProtocol)` method.

There are numerous helper methods provided by the tool to simplify test creation. Since introducing all of them is out of the scope of this thesis, only the crucial ones are described, which are *exec*, *check*, and *body*.

As its name implies, an *exec* method is used to execute an action during a simulation, which, in this case, is an HTTP request. The payload information is passed via a request's *body* method. There are multiple overloaded versions of the method available, which take different input file formats depending on the use case. After a server has answered to a request, there is usually a need to capture data from a received response. This is the use

case for a *check* method, which either verifies that the response is of an expected form or extracts data of interest.

Gatling provides a `saveAs(key)` method to store retrieved results. A value is saved into the user session and can be referenced later using what is familiar from a JMeter syntax: `${key}` [56].

```
val httpProtocol = http
 .baseUrl("http://localhost:8080")
 .inferHtmlResources()
 .acceptHeader("*/*")
 .acceptEncodingHeader("gzip, deflate")
 .acceptLangua-geHeader("en-US,en;q=0.5")
 .userAgentHeader("Mozilla/5.0 (Windows NT 10.0; Win64; x64;
rv:66.0) Gecko/20100101 Firefox/66.0")
```

```
// User injection for the defined user scenario
setUp(userScenario.inject(
 rampUsers(150) during (15 seconds)))
.protocols(httpProtocol)
```

Figure 10. A protocol definition and set-up for a scenario

4.3.3 Reports

Gatling does not require any additional configuration steps to get a visual representation of results after a simulation run. An HTML file containing statistics is created for the overall run and for each request separately. Gathered data is presented in both tabulated and chart views. One can find *min*, *max,* and *mean* response times from the generated report. Furthermore, *active users over time* and *response time distribution* charts provide an in-depth overview of how an application behaves under a generated load. A developer does not have to be heavily involved in the process, except for giving a unique name for each request if she wants to ensure that statistics for both are not accumulated into one.

5 Load Test Implementation and testing

To achieve a meaningful outcome and to identify problematic modules from running a load test, a test plan should be appropriately specified and documented. Metrics such as the *expected* and the *peak number of users* must be defined. Ideally, tests should be run in an environment equal or similar to a production one. Unfortunately, this happens quite rarely. Typical justifications for this are the privacy of production data, which should be depersonalized, and the inability to spin-up the same amount of running servers. Nevertheless, it is vital to preserve the proportions of a production system and introduce components such as cache servers or load balancers, if a system has them. However, this does not happen often either, which generally leads to scalability and load performance issues that are only revealed in production.

From the perspective of this thesis, it is not necessary to meet hardware requirements and mimic a production build, but it is essential to design and plan a load test carefully. Yet, anyone designing and executing load tests for real-life use case must also pay attention to the physical set-up.

Based on the discussions in the previous chapters, the goal of this chapter is to answer the first question of this thesis: ***"How does one perform load testing for a Vaadin Flow application?"***. This is done by designing and creating a test against a Flow-based application. To do this, the first step of this chapter will be to outline the elements that set Flow load testing apart from other web applications.

5.1 Overview

A load test against any web application that is built using a server-side framework, such as Django (Python language) or Laravel (PHP), demands, at a minimum, the proper handling of cookies, headers, cache, and embedded resources. Typically, load testing tools retrieve and pass cookies automatically to the following requests. Gatling has this feature enabled by default, and JMeter provides a configurator element that can be added if needed. This simplifies testing of any cookie-enabled web application. Since both JMeter and Gatling intercept HTTP traffic, the framework used to build a web app does not bear any relevance to the tools. However, from a tester perspective, this is essential since it defines the format of the request's and response's payloads.

A Flow application, as discussed above, has its own unique traits that are not conventional to other apps. While any web application might have CSRF protection, there is no way for a loading tool to know beforehand how, exactly, this is implemented, as this may vary from one technology to another. Thus, a testing engineer can not necessarily re-use an existing solution, such as regular expression, and must conform to a current application under the test. Moreover, `clientId`, `syncId,` and `pushId` are Vaadin-specific implementation details and this should be acknowledged during test creation. Extraction of `clientId` and `syncId` is not handled automatically by tools; thus, it is a tester's responsibility to take care of the synchronization tokens. Lastly, AJAX techniques are used in Flow to update a changed part of a page without its complete reload; therefore, only the required data is transmitted.

In Chapter 4.1.1 a payload data that was sent on a button click was presented; a response to it similar to the one in Figure 11:


```
for(;;);[{"syncId":1,"clientId":1,"changes":[{"node":5,"type":"splic
e","feat":2,"index":1,"addNodes":[7]},{"node":6,"type":"attach"},{"n
ode":6,"type":"put","key":"text","feat":7,"value":"Button
clicked!"},{"node":7,"type":"attach"},{"node":7,"type":"put","key":"
tag","feat":0,"value":"label"},{"node":7,"type":"clear","feat":2},{"
node":7,"type":"splice","feat":2,"index":0,"addNodes":[6]}],"timings
":[1070,0]}]
```


Figure 11. A response payload to a request presented in Chapter 4.1.1

From here, one can notice that `syncId` and `clientId` values are incremented by one and that a new *label* element with a caption *"Button clicked!"* is attached.

Compared to other web applications, in the Flow app, a tester must ensure and verify that

- A `csrfToken` is extracted from the first server's response and passed to all the subsequent requests.
    - If a user authentication mechanism is implemented, there might be three CSRF tokens generated for a single browser session: before login, after a user is authenticated, and after logging-out. Each must be extracted and passed forward for a test to be functionable.
- Each consecutive request has synchronization token values incremented by one.

- Each response is in the form of `for(;;);[response JSON]`.
- An extraction post-processor is applied to the correct response.
    - Information that might help reveal the element behind a *node* value is sent only once the component is attached to a UI. Afterwards, only an integer value is used in communication between a client and server to determine an element.
- The correct syntax is used for extracting values from a response's payload.

To elaborate on the last point, let's assume that one wants to find an ID of a newly attached label from the snippet in Figure 11 for a later re-use. If only one label is available in the whole UI, then this is a straightforward task. We only need to find that element's node value, which has a *"value":"label"* in its JSON. Unfortunately, it is rarely that simple; therefore, we must acknowledge and take advantage of available information. A caption *"Button clicked!"* is the only hint available and it is, thus, rational to start from this. By inspecting the response, one will find the ID of a text node (*6*), but not the label's ID. Reviewing this further, we can see that a text node (*6*) is directly added to a label element (*7*), which is what we are looking for. Knowing that any element can be attached only to one parent at time, we can extract a parent's ID, which is a label `node` value we were seeking. Overall, the process goes as follows:

1. Extract a `node` value of a text element that contains a caption.
2. Find an element node's value that contains the previous `node` inside the `addNodes` array.
3. Extract that `node` value as an ID of an element.

This procedure will be described in greater detail for both JMeter and Gatling with concrete regular expressions, which can be reused later by substituting one's own values.

5.2 Prerequisites

There are no strict rules on how load testing against a Flow application must be accomplished. In Figure 12, a possible procedure is defined. The order of some actions can be changed without a negative impact on a test output. Nevertheless, the prerequisite

is that no action should be started before the previous one has succeeded. The method is utilized in the next subchapters to show its applicability and the concrete steps for its implementation.

- Define a user scenario with a customer.
  - Specify pick and average number of users.
- Configure proxy settings for Firefox browser.
- Record defined scenario using a load testing's recording utility.
- **Mandatory**: Extract `csrfToken`.
  - In case application has push enabled, then `pushId` must also be extracted.
- *Optional*: Extract `clientId` and `syncId`.
- *Optional*: Extract and replace with session's variables *node* values of the vital elements.
- *Optional*: Supply log-in credentials from an external file.
- Inject decided number of users over defined period .
- Run the test.
- Analyze results.

Figure 12. The method to load test a Flow application

5.2.1 Versions

By the time of writing this thesis, there are updated minor versions available for some of the tools used during test creation. The latest stable versions should be used, since they contain the most recent bug fixes, new features, and performance improvements. The tests created and discussed below should be backwards compatible with newer versions of Java, Gatling, and JMeter.

Gatling 3.0.3 officially supports OpenJDK 8 and 11, whereas JMeter 5.1.1 works with 8 or 9. Therefore, JDK version 1.8 is the only compatible option at the time of writing this thesis and is used during development. The IDE of choice is IntelliJ IDEA, due to author's familiarity with the tool and its support for Scala development.

5.2.2 Browser proxy set-up

For a recorder to capture navigation, a proxy server must be set up to intercept the traffic between a browser and a requested server. The port defined in the proxy settings of a loading tool should match the one specified in the browser proxy setting. Firefox is generally used to interact with an application during recording. Its settings for proxy can be found under *Options*→*Network Settings*. After a developer has configured a browser and a load testing tool, she can start a test recording.

5.2.3 Regular expressions

Regular expressions(regexes) are extensively used throughout the tests as a body for a *Regular Expression Extractor*s. Regex is a searching pattern that is applied to a text to match a character sequence. The regexes most used during the testing are listed and described below.

- A dot **.** matches any character.
- A UUID pattern **[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}** [57] is used to extract a `crsfToken`. The number in curly brackets **{X}** specifies how many characters should be matched. The information in square brackets **[]** define a range for characters to be in. In this case, each character can be any letter or digit. Another regex to extract a `csrfToken` is **[^"] +**, which matches anything until `"` is encountered in non-greedy (stops right after the first occurrence) mode.
- Respectively, a **[0-9]+** expression matches any sequence of digits of length longer than one. It is used extensively through the tests to extract the values of IDs' such as `node`, `clientId`, and `serverId`.

The examples above are usually part of a bigger regex. The part which should be extracted and saved is placed inside the parentheses **()**.

5.2.4 Scenario

An application to be tested is provided by the FMI. It is used by meteorologists to assess the accuracy of past weather predictions. Since it is designed for internal use only, the maximum expected number of simultaneous users is 10. As the amount of expected traffic is low, the application is deployed to a single server. A relatively simple deployment scheme gives an opportunity to reliably reproduce the behavior of a system under test on the author's working machine.

There are three groups of users: *verified user*, *meteorologist,* and *admin,* which each have their own access to privileges. Due to the space limitation length of the thesis, only the tests covering the *meteorologist* role are presented and described. There are multiple locales available for the app. Scenarios were recorded and further modified with the assumption that the UI's language is the default English. The defined test scenario for the application is presented in Figure 13. A meteorologist has performed all the actions.

An application must be started in production mode, which ensures that all built-in performance improvements take place.

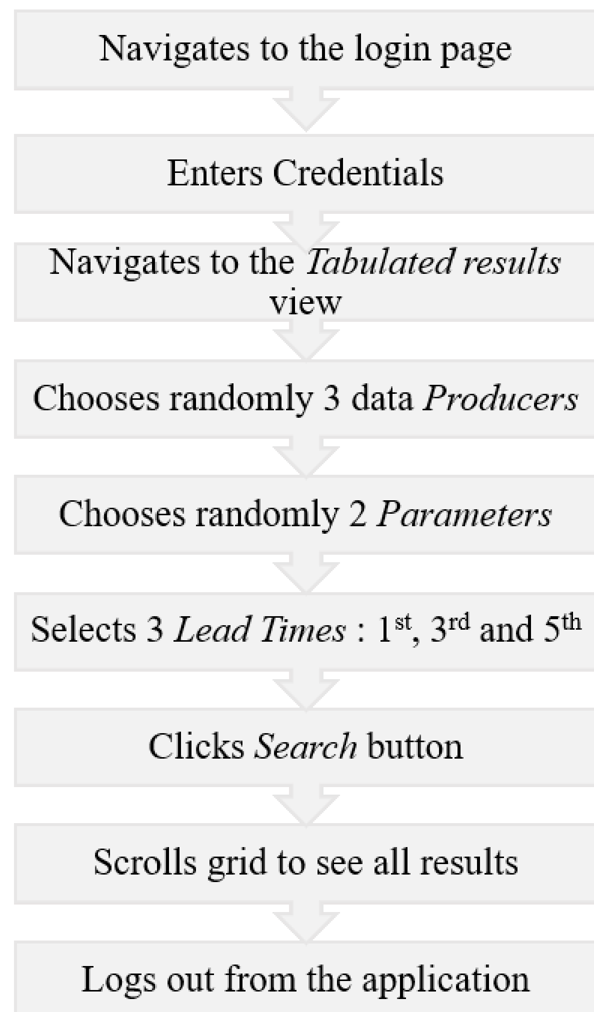

Figure 13. Defined user scenario

A screenshot of the *Tabulated results* view is shown in Figure 14.

Figure 14. The tested UI

In the following sections, JMeter and Gatling implementations for the scenario defined above are described.

5.3 JMeter implementation

The creation of a test starts by recording the user's interaction with an application. A browser is configured to listen to the port specified in the JMeter's Recorder, which acts as a proxy. During navigation on a web page, requests are saved under the *Recording controller* component. After navigation in the UI is finished and the recorder is stopped, a generated stub file for further development and modification is available. Development happens in the same GUI.

All controllers of the generated requests should be visited and altered by selecting a "*Retrieve all Embedded resources*" checkbox in the *Advanced* tab. This selection acknowledges to JMeter that all static downloadable files must be retrieved within a corresponding request during a test run. Initially, these requests were excluded from a test plan by the recorder filter policy. Fetching static resources is a required step if one wants to mimic user behavior as closely as possible.

HTTP Request
Name: Select 1.st parameter
Comments:
Basic Advanced
Client implementation
Implementation:
Timeouts (mill
Connect:
Embedded Resources from HTML Files
Retrieve All Embedded Resources
Parallel downloads. Number: 6
URLs must match:

Figure 15. Configuration in "Advanced" tab for a request

A JMeter captured request's payload is in a form similar to the one shown in Figure 16. By inspecting the request, one can observe that a user has clicked an item with `node` value *340* (`item-click` event has been fired), pressed the *ctrl*, and released the *shift* keys. This is the only way to determine exactly which element has fired the event. As will be explained later, it usually makes sense to extract IDs of the frequently used elements.


```
{"csrfToken":"626843e3-5a31-4c24-a119-
30ed4009f5d9","rpc":[{"type":"publishedEventHandler","node":3
40,"templateEventMethodName":"setDetailsVisible","templateEve
ntMethodArgs":["16"]},{"type":"event","node":340,"event":"ite
m-
click","data":{"event.detail.screenY":451,"event.detail.metaK
ey":false,"event.detail.button":0,"event.detail.shiftKey":fal
se,"event.detail.screenX":4719,"event.detail.itemKey":"16","e
vent.detail.altKey":false,"event.detail.clientX":1776,"event.
detail.detail":1,"event.detail.clientY":322,"event.detail.ctr
lKey":true}}],"syncId":4,"clientId":4}
```


Figure 16. Recorded request without modifications

5.3.1 A CSRF token handling

As discussed previously, Vaadin has built-in CSRF protection. There are two ways to handle this during application testing: either by disabling the protection or by extracting

the received csrfToken and substituting it dynamically for all the subsequent requests. Since disabling the protection is a straightforward step, the second way- extraction- will be used for this thesis. The token is derived by following the same steps as described in the *Regular Expression Extractor* subchapter.

The token is generated on a server and sent back to a client within the first response. Therefore, the extractor should be placed under the first request to a server. If a value is saved into a variable called secKey, it can be referenced in subsequent requests using the ${secKey}syntax. After that, all captured requests should be altered by replacing a recorded token value with a variable reference. The modified request of navigating to a *Tabulated Results* view is shown in Figure 17.

```
{"csrfToken":"${secKey}","rpc":[{"type":"navigation",
"location":"tabulatedresults","link":1}],"syncId":${syncId},
"clientId":${clientId}}
```

Figure 17. Replaced csrfToken, syncId and clientId tokens

This is the only step that is required to execute the test successfully, though it will not provide an in-depth insight into how an application works in production. Randomization of data is needed to get a better overview. Otherwise, one can end up verifying that cache is working correctly.

An application that employs authentication and authorization might invalidate the first generated csrfToken and issue a new one, once a user has logged in. In this case, a token should be obtained again by adding the same extractor under the log-in request.

5.3.2 Extracting an element's node value

Before demonstrating how randomization can be accomplished, it is necessary to introduce a method of obtaining an element's node ID.

There are multiple ways to find and retrieve data from a response in JMeter. A *Regular Expression Extractor* is one of the most convenient and powerful utilities to use when processing Vaadin responses, as has been discussed in the previous chapter. Here, instead, a logic for finding a correct regex is discussed.

A node attribute contains an ID, which uniquely identifies an element on a page. It is used to keep track of changes between a server-side and a client-side state of the

component. The attribute is sent to a client when the connected component is attached to a UI. If an application's UI remains unchanged after recording a test and the number of simulated users is low, the captured `node` values will stay the same. If a developer is confident that no modification will ever be introduced, she can skip the further configuration presented below. Nevertheless, this makes a test vulnerable to any changes. Adding even one component to the application will likely cause it to not work. Also, it was observed that with a higher number of simulated users, a `node` value could change between test runs. Therefore, it is suggested that an ID of an element is derived into a variable to ensure a test's stability and continuity. Later, the variable can be substituted into requests acting upon the element.

5.3.2.1 Filter grids

The right side of the *Tabulated Results* view contains a group of filters. Selected values are propagated to a query, which is used to fetch results from a backend. Each employed filter (*Producers*, *Parameters*, *Analysis hours,* and *Lead Time*) is implemented using a grid element. Selecting anything in one grid affects the available options in the subsequent ones. Therefore, to pick an existing option in a filter, it is crucial to detect what new choices are available after the preceding one is updated.

None of the grid elements have a unique value or identifier, which would help in distinguishing each one and extracting its *node* id. Figure 18 shows a snippet of the response, which contains a segment of the definition for the *Producers* grid. As one can notice, there are no unique characteristics sent; thus, there is no reliable way to identify a filter without additional data. In this case, available additional information is the name of the grid rendered in a separate label.

```
{"node":340,"type":"attach"},{"node":340,"type":"put","key":"
tag","feat":0,"value":"vaadin-grid"}
```

Figure 18. A Snippet of the response with the "Producer" grid definition.

Each grid is added to a `div`, which also, has another child, namely - a label. This label contains a text node, which is the filter's name that is visible to a user. The described hierarchy for the *Producer* grid is presented in Figure 19. All the filter grids are structured in the same way.

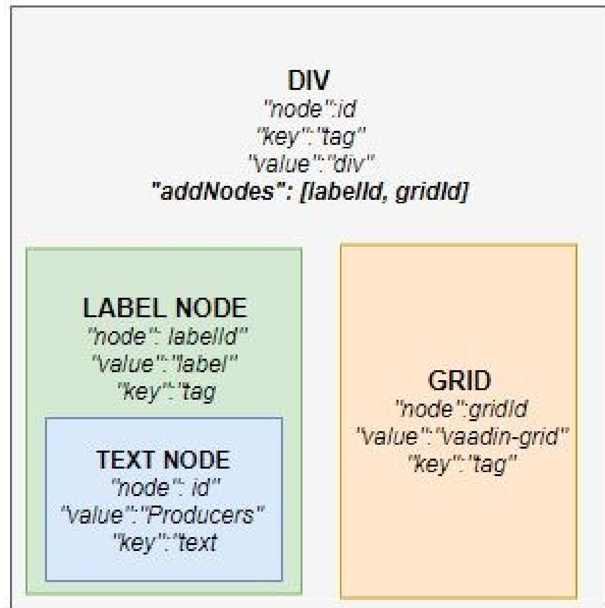

Figure 19. Grid position inside div

The process of extracting a grid's *node* value starts by identifying a text node, which has a pair of *"value":"Producers*" in its attribute list. Based on this, the label element containing the text node is identified. A component can be added only once to a UI. It is also known that `div` contains both the label and the grid. An `addNodes` array contains the IDs of the `div`'s child elements. Therefore, the *gridId* can be retrieved from the second position of the array, if the value in the first position matches the *labelId*. The assumption made in this case is that the label is always positioned above the grid. Thus, it is added first to the `addNodes` array.

There are three extractors needed for each grid to find its ID. Every next regex includes a value extracted from a previous one, thus it must be applied in the correct order. Table 1 contains the names of created variables and regular expressions used to extract them from the response.

Name of created variable	Regular expression
parameters_text	`"node":([0-9]+),"type":"put","key":"text","feat":[0-9]+,"value":"Parameters"`
parameters_label	`"node":([0-9]+),"type":"splice","feat":[0-9]+,"index":[0-9]+,"addNodes":\[${parameters_text}\]`
parameters_grid	`"addNodes":\[${parameters_label},([0-9]+)\]`

Table 1. Regular expressions used to extract node ID of the "Producers" grid

5.3.2.2 syncId and clientId tokens

As was discussed in the previous chapter, Flow has a built-in synchronization mechanism that relies on `syncId` and `clientId` values that are sent and received with every request and response. There is usually no need to modify these tokens. Nonetheless, to improve stability when the amount of generated requests is high, it is recommended that one extract value from a previous response and to pass it to the next request.

JMeter *UDVs* are final; therefore, they cannot be used to store dynamic token values, but variables created and modified using *BeanShell PostProcessor* can be changed. For this reason, two *PostProcessor* elements were used to retrieve and pass tokens: *Regular Expression Extractor* and *BeanShell*. The first one extracts the `syncId` value from a response and turns it into the `synchronizers_1` variable and turns the `clientId` into `synchronizers_2`. The script that saves extracted variables into `syncId` and `clientId` is presented in Figure 21, and regex configuration is presented in Figure 20.

Saving the synchronization tokens into `syncId` and `clientId` variables is an optional step. `Synchronizers_g1` and `synchronizers_g2` generated variables can be used directly instead, thus skipping the *BeanShell* script. However, the current solution gives descriptive names to variables and simplifies the perception of a transformed request's payload. A more natural and straightforward way would be to add a separate extractor for each token.

Figure 20. Configured Regular Expression extractor for syncId and clientId

```
syncId= vars.get("synchronizers_g1");
clientId=vars.get("synchronizers_g2");
vars.put("syncId",syncId);
vars.put("clientId",clientId);
```

Figure 21. BeanShell script for parsing and saving synchronization tokens into memory

All requests, except for the first one, were altered with variables' references for `syncId` and `clientId` values. A default value of 0 is left for the first request since a client always opens the communication.

5.3.3 Randomizing and supplying data

An application under the test is dynamic - any user's choice affects what information is applicable and should be available. This means that previously fetched data is not necessarily valid anymore.

Any time a user selects a new value in a field, a state of the application is changed. The new state with the updated options for other fields is included in the response, which should be parsed to extract them to be used later.

Contrasting this, constant values such as credentials for logging-in are known beforehand and can be supplied to the test before executing it. Since both approaches of acquiring and passing data are used, a possible solution to each is presented.

5.3.3.1 Comma-separated values Data Set Configuration

One of the most popular ways to supply static data into a test is by consuming a Comma-Separated Values (CSV) data source. Each row represents an entity in which a comma separates its attributes. JMeter provides a convenient configuration of the element to access CSV data called *CSV Data Set Config*. In this thesis, this is employed to provide a username and password during log-in.

Credentials were stored in the *credentials.csv* file, in which each row contains authorization details in a `username, password` form. A configuration component for the file is presented in Figure 22. Variables defined in the *Variable Names* field can be later referenced using the familiar `${username}` syntax.

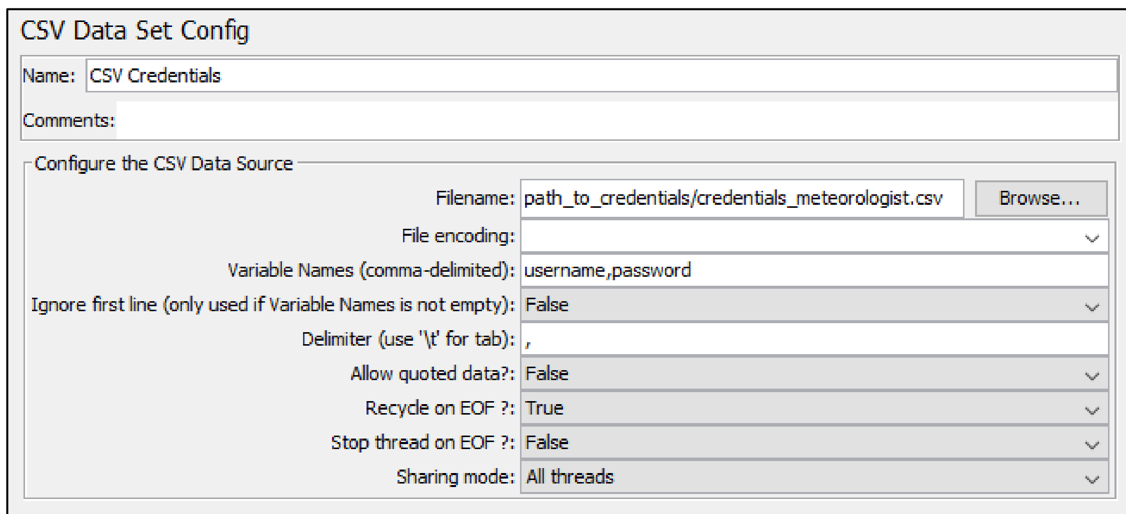

Figure 22. Comma-separated values Data set configuration in JMeter

5.3.3.2 Dynamic data

A way of randomizing data is by using JMeter's `${__Random(x,y}` random number generator function. The returned value is inside the specified $[x, y]$ range. This function is used to randomly choose three producers. There are also multiple complementary functions available such as `RandomDate` and `RandomString`.

All producers in the grid are known beforehand: it is a list of 18 authorities, which provide input data for a system. An `event.detail.itemKey` attribute identifies each

selection passed to a server on a click action. Its value is equal to a physical position of an option. Therefore, exact names are irrelevant from a load test perspective. It is enough to substitute any value in $[1,18]$ to simulate a random choice.

Initially, the first option is always selected. To preserve this behavior, a range between $[2,18]$ is passed to a random generator. Default selection also ensures that there are some choices always available in the next filter. Due to the enormous data source size and its privacy, the local set-up only has part of the data available; thus, some parameters will not produce any output. Furthermore, the same random number can be generated twice. Consequently, the final configuration might have only one option selected, since clicking an item twice deselects it.

After the third producer has been chosen, all values available for the *Parameters* grid are sent in the response, and three random parameters can be chosen. A request payload is similar to the one in the *Producers* grid. The `event.detail.itemKey` for each parameter ranges between the first parameter key and the sum of this key and the total amount of parameters. In a mathematical form, this can be expressed as $[x, x + \sum y]$, where x- is the value of the first key and y- is the number of available parameters.

After these values are known, Java's `Random` object's `nextInt` method is used to generate two parameters. The regular expressions used to extract the first key are presented in Table 2.

Name of variable	Regular Expression
availableParameters	`"node":([0-9]+),"type":"put","key":"text","feat":[0-9]+,"value":"[^"]*(temperature|precipation|wind|sea){1}[^"]*"`
first_parameter_span	`"node":([0-9]+),"type":"splice","feat":[0-9]+,"index":[0-9]+,"addNodes":\[${availableParameters}`
first_key_parameter	`"key":"([0-9]+)",("selected":true,)?"_renderer_[0-9]+":${first_parameter_span}`
amountAvailableParameters	`"node":([0-9]+),"type":"put","key":"text","feat":[0-9]+,"value":"[^"]*(temperature| precipitation |wind|sea){1}[^"]*"`

Table 2. Regex expressions used to extract ID for the first available parameter id

All available parameters contain one of the keywords: *temperature*, *precipitation*, *wind,* or *sea*. The `availableParameters` variable stores the first occurrence of any text node that matches the requirement. Each is later appended to a *span*. An ID of the span of the `availableParameters` value is stored in the `first_parameter_span` variable. Then, the key for the first parameter is found based on this and is stored into the `first_key_parameter` variable.

A regex is used again to catch the first available parameter and to find the total amount of parameters, with the only difference being that *Match No.* is set to -1. The *BeanShell Postprocessor* script used to generate two random parameters is presented in Figure 23.

```
int firstKe =
Integer.parseInt(vars.get("first_key_parameter"));

int availableAmout=
Integer.parseInt(vars.get("amountAvailableParameters_matchNr"
));

Random rand = new Random();

int n = rand.nextInt(availableAmout-1) + firstKe;

int m = rand.nextInt(availableAmout-1) + firstKe;

vars.put("first_parameter",String.valueOf(n));

vars.put("second_parameter",String.valueOf(m));
```

Figure 23. Generating two random parameters.

The `first_parameter` and `second_parameter` variables keep the generated values.

Unfortunately, parameters are not always sent in the third response. If the second and third producers have the same parameters available, then there is no update to the *Parameters* grid, and values are sent only once within the second producer. To ensure that this does not occur, the default parameters that were captured during the initial recording are used, assuming that there are no parameters sent in the third response.

5.3.4 Output

Adding a complete JMeter test would take approximately 60 pages; therefore, the *.jmx* file has been left out of the thesis. The full code can be found at the author's Github repository under *anasmi/LoadTestingVaadin14* folder [58]. Furthermore, Appendix B contains a screenshot of a configured *Test plan* and *Thread Group* element of the test.

After the transformations described in the previous subchapters are applied to all the requests, a developer can execute the test. For example, the request presented at the beginning of section 5.3 in Figure 16 is changed to the form shown in Figure 24. Differences are outlined in bold.

```
{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandle
r","node":${producers_grid},"templateEventMethodName":"setDet
ailsVisible","templateEventMethodArgs":["${first_producer}"]}
,{"type":"event","node":${producers_grid},"event":"item-
click","data":{"event.detail.screenY":451,"event.detail.metaK
ey":false,"event.detail.button":0,"event.detail.shiftKey":fal
se,"event.detail.screenX":4719,"event.detail.itemKey":"${firs
t_producer}","event.detail.altKey":false,"event.detail.client
X":1776,"event.detail.detail":1,"event.detail.clientY":322,"e
vent.detail.ctrlKey":true}}],"syncId":${syncId},"clientId":${
clientId}}
```

Figure 24. Request presented in Figure16 after alterations

The test was run three times with different number of users: 15, 50, and 300. Since the maximum expected number of simultaneous users is 10, it is sufficient to verify that the application can respond promptly to 15 users. Nevertheless, a more sophisticated case with a ramp-up period for 50 and 300 users is also set-up. Simulating users at ramp-up time implies that at any given moment the number of concurrent users can range from no

users to the total number of users defined. Such a scenario mimics how an application is used in the real-world.

It would be beneficial to increase the number of users and this should be done once the test is executed in a proper testing environment. Unfortunately, due to limited capacity and the CPU resources of the author's computer, the number of users is limited. However, it is still possible to analyze the application's behavior under these conditions.

Before beginning the measurements, the test was run with 10 simultaneous users, to warm-up the server. Among other things, this involves populating cache, which is empty on start-up. The time this took was not measured, as it does not provide valuable insights into the application's performance.

The test script is started from a command line using the

```
jmeter -n -t fullPath\FMITabulatedResultsView_meteorologist.jmx -
l fullPath\output.jtl
```

command.

The *-n* option instructs JMeter to be started in non-GUI mode, *-t* is followed by a full path to test location, and *-l* is a path where sample results are stored to.

The table below summarizes the results of executing the defined scenario for three target groups. In all cases, the longest response time was for the *Login* request and the fastest was for the *Logout*.

Number of users	**Started during (seconds)**	**Minimum response time**	**Maximum response time**	**Error rate**
15	1	1 (logout)	1773 (Login)	0%
50	10	0(logout)	2512 (Login)	0%
300	200	0 (logout, picking lead times)	1920(Login)	0%

Table 3. Summary of JMeter captured responses

After completing the execution, the generated *output.jtl* file was imported into JMeter to analyze the results. The importing is done by using the *Listeners* component via the path *Test plan* → *Add* → *Listeners* →*Summary Report* → *Read from file.*

A JMeter generated report for 50 users is presented in the figure below.

Figure 25. JMeter reported results when running 50 users

5.4 Gatling implementation

The fastest way to set up a base for a Gatling test is to record a user's activity in a web app. This will, at a minimum, ensure that the correct headers for each request are captured and added. This approach was also utilized here. Yet, compared to JMeter, after recording an initial stub, there is no way to modify a generated *.scala* file using the recording's GUI. Any IDE, preferably supporting Scala, could be used to further alter the script. All required steps, which were presented in the previous subchapter, such as token extractions, should also be applied to a Gatling's test.

The final test script is composed of Scala's `object` classes. Based on the scenario, six objects are defined, and each describes a singular independent action. The actions are: `LoginAndNavigate`, `ChooseProducers`, `ChooseParameters`, `ChooseLeadTime`, `FetchResultsGrid`, and `LogOut`.

```
val userScenario =
scenario("Meterologist").exec(LoginAndNavigate.loginAndNaviga
te,
  ChooseProducers.chooseProducers,
  ChooseParameters.chooseParameters,
  ChooseLeadTime.chooseLeadTime,
  FetchResultsGrid.fetchResultsGrid,
  LogOut.logOut)
```

Figure 26. Scenario definition via objects

Arranging a test as a chain of independent decoupled tasks makes it conform to a Selenium's page object pattern. While this is not obligatory, it significantly improves its maintenance and readability.

Before analyzing the implementation details, it is useful to first introduce the common structures used throughout the test. These are summarized, with a brief description of the purpose, in the table below.

Structure	Purpose
`s"$parametersURL"`	One of Scala's string interpolation methods. References a variable defined previously.

`var first_key_leadTime="328"`
`(raw"""key":"([0-9]+)",("selected":true":$first_span""".r).findFirstMatchIn(response)`
`match {`
`  case Some(m) =>`
`    first_key_leadTime = m.group(1)`
`  case None =>`
`    println("An id is not found for the first key lead. Using default 328")`
`}`
	1. String interpolation using `raw` guarantees that characters are not escaped [59].
	2. Regex is defined via `.r` and `findFirstMatchIn` takes a string as input in which to look for occurrences [60].

`session("producersResponse").as[String]`	Obtaining a value called *"producersReponse"* from a session and casting it to String.

Table 4. Common code snippets used in Gatling test

5.4.1 Payloads

In a recorded simulation, a request's payload is saved in a file, which is passed to a request via its `body` method using `RawFileBody(payload_file)` as an argument. To be able to substitute dynamic values into a request's body, its content should be saved and feed as an `ElFileBody` or passed as a `StringBody` instead. Bodies of these types can be parsed by a Gatling Expression Language (EL) engine [61]. As the name implies, `StringBody` takes a string as an argument, whereas `ELFIleBody` takes a file. The latter one will be mostly used for passing payload to a request, for readability and clarity reasons. In Figure 27, a snippet of the original code is presented, saved by a recorder where `RawFileBody` should be replaced with an `ELFileBody`.

```
.exec(http("request_2")
      .post("/?v-r=uidl&v-uiId=0")
      .headers(headers_2)
.body(RawFileBody(
       "FMITabulatedViewMeteorologist_0002_request.txt")))
```

Figure 27. A request payload passed as a RawFileBody

5.4.2 A CSRF token handling

After the files' format passed to a `.body` method has been changed, the text stored in these files can be altered to use variables. A `check` method presented in the figure below is called upon a *Login* request to retrieve `csrfToken`.

The same regex, as in the JMeter script, is used to extract a token value. After the value has been stored to a variable `secKey`, it can be referenced in a request's body using a familiar `${secKey}` syntax. As discussed before, this is the only required step to get a load test simulation running.

```
.exec(http("Login")
 .post("/login")
 .headers(headers_1)
 .formParam("username", "${username}")
 .formParam("password", "${password}")
 .check(regex("""Vaadin-Security-Key":"([a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})""").saveAs("secKey"))
)
```

Figure 28. Extraction of a csrfToken

As may be expected, by the end of this chapter, after all the adjustments are applied, a request payload in both Gatling and JMeter will look the same.

5.4.3 Extracting clientId and syncId

The parameterization of synchronization tokens is not required, yet it improves the stability and reliability of a simulation; thus, it is preferable to undertake this step. Similarly to `csrfToken` extraction, obtaining `syncId` and `clientId` can be achieved using regular expressions.

A `regex` function takes as a parameter a regular expression that matches a synchronization token. An extracted value can be reused later if it is persisted in a session using a `saveAs` method. For synchronization tokens, a chain of extracting and persisting a value is stored into two immutable Scala variables: `syncIdRegex` and `clientIdRegex` for each token, respectively. Each is passed as a parameter to a `check` method, which is called on an HTTP request. Figures 29 and 30 exemplify the regular expression definition and usage described above.

```
val syncIdRegex= regex("""syncId":([0-
9]+)""").saveAs("syncId")
val clientIdRegex= regex("""clientId":([0-
9]+)""").saveAs("clientId")
```

Figure 29. Definition of regular expressions for syncId and clientId tokens

```
.exec(http("request_3")
 .post(s"$parametersURL")
 .headers(headers_2)

.body(ElFileBody("FMITabulatedViewMeteorologist_0003_request.t
xt"))
 .check(syncIdRegex)
 .check(clientIdRegex)
)
```

Figure 30. Capturing synchronization tokens from a request

5.4.4 Feeders and random data

Since the generated script is a Scala file, it is possible to randomize data using the Scala programming language directly. For instance, producers' values can be assigned using familiar syntax from Java `2+ Random.nextInt(17).`

There are multiple ways to supply data to a request. Feeder is one utility used to provide credentials for a login request. The login request presented in Figure 28 takes its form values from a *credentials.csv* file, which is supplied to the scenario using the `feed(csv("credentials.csv").circular)` method. Contrary to JMeter, the supplied file has a header row, which defines the variables' names `username`, `password` that will be referenced later using a `formParam` method. A `circular` method instructs Gatling to iterate over the file again when it reaches the end; otherwise, the test will exit abruptly.

5.4.5 Setting session variables

Under some circumstances, a response may not contain a value, which a regex extractor component is trying to extract. This generally happens if there are no updates to the client caused by a request. In the test scenario, lead times options become available only after choosing parameters, which are also randomly appointed. Therefore, it is not possible to

know the final permutations of values in advance. Moreover, some of the possible matches do not yield any changes.

If no new lead times options become available after a parameter selection, the test will exit abruptly. To avoid a failing test and to continue its execution, lead time variables with pre-defined values are added to a session's object. These default values are the ones initially captured during recording. There is no guarantee that all of them are available in the current run; however, if they are not, a server will log a warning and continue accepting new requests.

Figure 31 contains a snippet that is responsible for assigning values if none were defined previously. Like Java's `String` class object, a session is an immutable object. Similarly, either all applied changes must be chained, and the modified object returned, or, after each modification, the session must be stored into an intermediate object, which is finally returned. Otherwise, all the changes disappear.

```
//If no lead times were added when choosing parameters, use
default values
.doIf(session=>session("first_key_leadTime").asOption[String]
.isEmpty){
    exec(session=>
      session.setAll(
        "first_key_leadTime" ->328,
        "second_key_leadTime"->330,
        "fifth_key_leadTime"-> 332
      )
    )
```

Figure 31. Setting default values for the lead times using session object

5.4.6 Debugging and Generating load

Debugging is a process that can be used when trying to identify the root cause of a failure during development. In Gatling, the easiest way to verify if a variable has the expected value is to log it using an `exec` method. It has an overloaded version that takes the `Expression` function as a parameter. For instance, to print a session's variable value, the code below can be used.

```
.exec{
  session=>
   println(session("variable").as[String])
   session
 }
```
Figure 32. Printing information for debug purposes

Once the test is finalized, it can be executed under different usage scenarios. Gatling provides a fluent API to specify the number of users to be generated. Ramp-up time, delay, and simultaneous number of users is just some of the information offered by the tool. For example, one of the injections used by Gatling script is `setUp(scn.inject(atOnceUsers(1))).protocols(httpProtocol).`

Most of the requests presented above belong to a `LoginAndNavigate` action. A snippet from the final object with all modifications applied is presented in the Figure below. The full created test and its response payloads can be found in Appendix C.

```
//Login and navigate to a "Tabulated Results View" page
object LoginAndNavigate{
  val loginAndNavigate=exec(http("Get /")
    .get("/")
    .headers(headers_0)
  )
    .exec(initProducersId)
    .pause(5)
    .feed(csvFeederCredentials)
    .exec(http("Login")
      .post("/login")
      .headers(headers_1)
      //Set credentials from a csv file
      .formParam("username", "${username}")
      .formParam("password", "${password}")
      .check(regex("""Vaadin-Security-Key":"([a-fA-F0-9]{8}-
[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-
9]{12})""").saveAs("secKey"))
    )
    .pause(1)
    .exec(http("request_2")
      .post(s"$parametersURL")
      .headers(headers_2)

.body(ElFileBody("FMITabulatedViewMeteorologist_0002_request.
txt"))
      .check(syncIdRegex)
      .check(clientIdRegex)
    )
    .pause(0,3)
    .exec(http("request_3")
      .post(s"$parametersURL")
      .headers(headers_2)

.body(ElFileBody("FMITabulatedViewMeteorologist_0003_request.
txt"))
      .check(syncIdRegex)
      .check(clientIdRegex)
    )
}
```

Figure 33. LoginAndNavigate object

5.4.7 Output

By default, after test execution, results are automatically assembled into an *index.hml* file. The generated page contains the *Statistics* table and various charts that visualize the user, response, and request characteristics, such as time distribution. A developer can open the page in any software capable of parsing HTML files. A regular web browser is the obvious choice.

As shown in the test script, no separate requests were specified for fetching embedded resources relying on `inferHtmlResources` functionality. Nevertheless, for a user's convenience, Gatling has added them to the *Statics* table. Due to its large size, only one is presented below. From there, a tester can find the maximum and minimum response times for any request with its overall success rate.

From the gathered results, it is self-evident that, except for downloading static files, the requests that on average take the most time to complete are *Choosing producers* and *Tabulated results view*. Since navigation to a new page presumes fetching a significant amount of information to re-draw a view and selecting a new producer implies database querying, this is an expected outcome.

All values presented in this thesis are relative, as running the test again might report a mismatched result. If the reported values are of the same range, this is an acceptable fluctuation.

STATISTICS | Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	570	570	0	0%	16.286	0	5	52	229	522	655	48	98
Get /	15	15	0	0%	0.429	2	4	8	20	29	31	7	7
Get / Redirect 1	15	15	0	0%	0.429	1	2	3	5	7	7	2	2
signin.css	30	30	0	0%	0.857	9	31	45	65	70	71	32	19
bootstrap.min.css	15	15	0	0%	0.429	55	73	86	150	173	179	83	33
Login	15	15	0	0%	0.429	6	57	147	370	456	477	108	131
Login Redirect 1	15	15	0	0%	0.429	131	224	282	581	621	631	276	146
webcompo...oader.js	15	15	0	0%	0.429	8	70	145	295	371	390	104	107
index.js	15	15	0	0%	0.429	34	134	364	526	566	576	216	180
index.es5.js	15	15	0	0%	0.429	37	174	370	595	643	655	255	196
client-5...cache.js	15	15	0	0%	0.429	14	148	337	532	619	641	223	179
request_2	15	15	0	0%	0.429	2	3	4	6	7	7	3	1
request_3	15	15	0	0%	0.429	1	2	2	6	9	10	3	2
Navigate...ts view'	15	15	0	0%	0.429	41	69	89	110	119	121	71	25
request_5	15	15	0	0%	0.429	2	3	3	6	8	9	3	2
Choose 1...producer	15	15	0	0%	0.429	16	34	53	80	89	91	40	23
Componen...producer	15	15	0	0%	0.429	3	5	8	35	55	60	10	14
Choose 2...producer	15	15	0	0%	0.429	22	51	60	88	90	91	52	19
Componen...producer	15	15	0	0%	0.429	2	4	8	11	12	12	6	3
request_10	15	15	0	0%	0.429	1	2	3	7	7	7	3	2
Choose 3...producer	15	15	0	0%	0.429	32	84	104	161	167	169	87	39
request_12	15	15	0	0%	0.429	1	4	9	36	42	44	9	12
Choose f...arameter	15	15	0	0%	0.429	18	56	70	90	104	108	57	24
request_14	15	15	0	0%	0.429	1	2	4	6	7	7	3	2
Choose s...arameter	15	15	0	0%	0.429	19	62	99	121	123	124	73	32
request_16	15	15	0	0%	0.429	1	5	7	19	25	27	6	7
Select 1...ead time	15	15	0	0%	0.429	1	3	4	4	4	4	3	1
Select 2...ead time	15	15	0	0%	0.429	1	3	4	5	7	8	3	2
Select 5...ead time	15	15	0	0%	0.429	1	4	5	8	11	12	4	3
Click Search button	15	15	0	0%	0.429	12	26	42	45	47	47	30	11
request_21	15	15	0	0%	0.429	1	2	2	5	7	7	2	1
Scroll Grid	15	15	0	0%	0.429	2	3	4	5	6	6	3	1
Update c...led Grid	15	15	0	0%	0.429	2	3	3	4	5	5	3	1
Click co...u button	15	15	0	0%	0.429	2	4	5	5	5	5	4	1
Open log...ext-menu	15	15	0	0%	0.429	1	3	3	10	22	25	4	6
Click 'L...ext-menu	15	15	0	0%	0.429	1	3	4	4	4	4	3	1
/logout	15	15	0	0%	0.429	0	2	3	3	3	3	2	1
/logout Redirect 1	15	15	0	0%	0.429	1	2	2	3	3	3	2	1

Figure 34. Statistics table generated by Gatling tool after test execution

Along with the visual reports, Gatling also prints global information into a console during and after test execution. The screenshot of the output for the scenario with 50 simulated users run is shown in Figure 35. Given that 50 users start in an interval of 10 secs, 100% of requests are answered in less than 800ms. As seen in the generated *Statistics* table, fetching static resources is the most consuming action. However, this is not reflected in

the printed summary, as download happens asynchronously and does not prevent user interaction.

```
Simulation FMITabulatedViewIT completed in 42 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================
---- Global Information --------------------------------------------------------
> request count                                       1900 (OK=1900   KO=0     )
> min response time                                      0 (OK=0      KO=-     )
> max response time                                    718 (OK=718    KO=-     )
> mean response time                                    16 (OK=16     KO=-     )
> std deviation                                         31 (OK=31     KO=-     )
> response time 50th percentile                          3 (OK=3      KO=-     )
> response time 75th percentile                         21 (OK=21     KO=-     )
> response time 95th percentile                         69 (OK=69     KO=-     )
> response time 99th percentile                        106 (OK=106    KO=-     )
> mean requests/sec                                 44.186 (OK=44.186 KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                          1900 (100%)
> 800 ms < t < 1200 ms                                   0 (  0%)
> t > 1200 ms                                            0 (  0%)
> failed                                                 0 (  0%)
================================================================================
```

Figure 35. Global information provided by Gatling after the execution of the second scenario is completed

The results for all three runs are summarized in Table 5.

Number of users	Started during (seconds)	Minimum response time	Maximum response time	Error rate
15	1	1(Login)	778(Login redirect)	0%
50	10	0(Logout redirect)	1042(Fetching static resources)	0%
300	200	0(Logout redirect)	2368 (Fetching static resources)	0%

Table 5. Gatling run summary info

The static resources mentioned in the table are *index.js, bootstrap.min.css,* and *signin.css*.

5.4.8 Comparing outputs

JMeter and Gatling report different results for a *maximum response time* metric. Gatling measures each request's response time separately, even for a static resource. JMeter, on the other hand, does not split the time of fetching a static resource from a request causing

it. This meant that the *Login* request was reported as being the one that took the longest time to respond to.

Therefore, while it seems that the tools are reporting conflicting results, this is not the case. Each of them is just treating the requests differently, but for both the *Login* phase takes the longest time.

5.5 Application memory consumption

Above, the response times for three different user load scenarios were measured and discussed response times. Both the reported results from Gatling and JMeter were presented. The next step is to evaluate the memory consumption.

There are two essential questions to consider when investigating memory requirements imposed by an application. The first one is estimating how much memory an application requires in its idle state (when all variables are initialized, but there is no load). The second question is what the approximate size of a user session is. To acquire these values, the Gatling test with a simple modification described below is run twice for 20 and 55 users. Overall, computation happens as follows:

1. Deploy an application to a server and simulate, for example, 20 users to initialize variables and warm-up a JVM.
2. Trigger Garbage Collector multiple times and after it has run, write down a memory consumption size. The captured value is a minimum memory size that the application requires.
3. Disable a log-out part from a test. This will ensure sessions remain in memory after test is exited.
4. Simulate 55 users over 10 seconds. *
5. After execution is finished, trigger GC multiple times.
6. Write down the current memory consumption.
7. Calculate an approximate application's session size as $\frac{final-idle}{users}$, where *final* is a memory utilization value captured at step 6, *idle* is the memory required in the application's idle state, and *users* is the number of generated users at step 4 (55).

* For more accurate results, one should increase the number of users and decrease the ramp-up period. Unfortunately, due to limitations of CPU and the memory on the author's computer, the results presented here are approximate.

Disabling a log-out part is a strict requirement; otherwise, a session is invalidated, and the garbage is collected by a JVM. Since ramp-up time is short, total time execution is less than a session timeout; thus, all 55 sessions remain in memory after a test has finished.

After performing the steps outlined above, a measured session size is approximately 16,4 MB, which is large. Usually, this size range between 200-800 KB. The following calculation, $\frac{0.98\ GB - 0.08\ GB}{55} \approx 0.0164\ GB = 16{,}4\ MB$ was done, in which 0.08 GB is the memory consumption by the application in the idle state, and 0.98 GB is the captured size with 55 sessions in memory.

To find a root cause for this large size, one user was simulated to find what classes remain in memory and the number of their instances. Analyzation has been performed using the JProfiler tool. All JVM classes such as `String` and `byte[]` were filtered out, as they do not provide any in-depth insights. By analyzing the remaining classes, one could see that , per one user, there were 921 instances of a class called `ListSelectMenu`, which contains multiple grids and calls itself numerous times. After refactoring it to two separate classes, session size dropped to 545 KB, which is a tremendous difference. This change has also been included in a newer version of the software.

Final session size estimation is done utilizing both JMeter and Gatling. The results are reported in the table below.

	JMeter	**Gatling**
Memory consumption in idle state	0.08 GB	0.08 GB
Memory consumption after 55 users are simulated	0.11 GB	0.11 GB
Session size	0.545 MB	0.545 MB

Table 6. Session's size measurement results

A screenshot of a *Memory* tab of the JProfiler tool, where spikes and memory consumption dynamics are illustrated, is presented in Figure 36. Vertical lines 1, 2, and 3 indicate three states:

1. Before test execution.
2. Right after the test has finished.
3. After GC has run a couple of times and 55 sessions are remaining in memory.

The difference between the third and second state is caused by removing unreached objects from the memory. Since none of the users have logged out yet, their related sessions remain in memory. This makes it possible to estimate a session size, as described above.

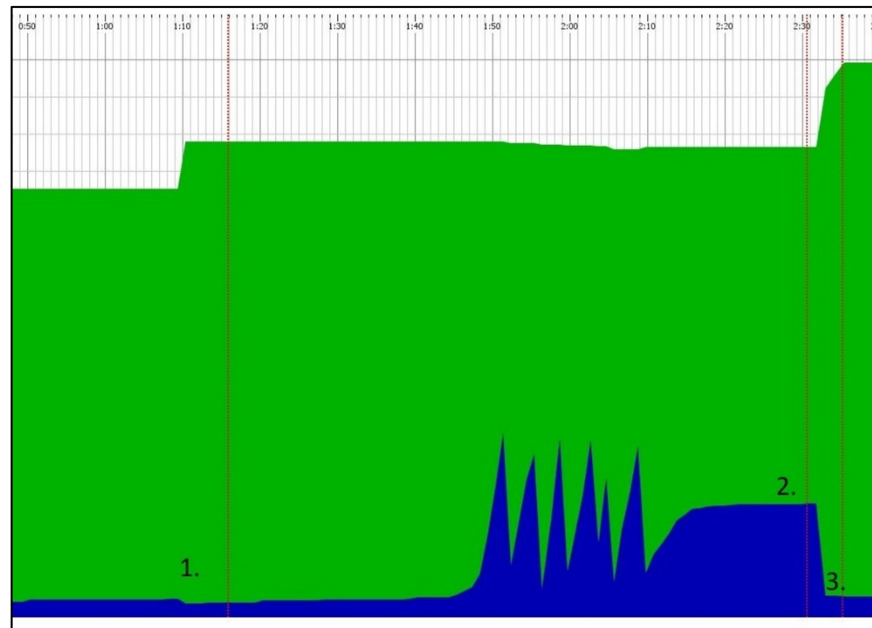

Figure 36. Memory consumption over test execution

Another way to access a session's size is to run the same test without the log-out part for a small number of users – for example five- and write down the memory consumption. Then, this process should be repeated for a considerably increased number - for example, 1,000 users- and the size should be captured. Calculation happens by subtracting the first value from the second and dividing the obtained result by the number of simulated users [62].

5.6 Integrating tests into maven

It is suggested that the testing process should be automated to receive feedback on an application performance early and proactively react to it. Automation can be accomplished by configuring maven to execute integration tests as a part of a build process. The recommended approach is to create a maven profile for an `integration-test` and `verify` phases. The configured profile can be enabled in multiple ways. Here, a profiler's ID, which is passed as an argument, is used. The syntax for the maven command looks like this: `mvn verify -PprofileID`.

Integration tests in a maven project are executed by the *failsafe* plugin, which, among other things, allows for the configuration of names and location of the tests. Running integration tests against an application presumes that the application is previously deployed to a web server. As the application being tests is a spring-boot application, a *spring-boot-maven-plugin* plugin is used to start and stop an application server. Otherwise, one can use a *jetty-maven* plugin. In the plugin's `<execution>` tag, during the `pre-integration-test` phase, a server is started. The server is stopped by a goal defined inside the `post-integration-test` phase. Both JMeter and Gatling provide maven plugins that simplify test configurations.

Tests are located under the *gatling* and *jmeter* directories inside a project's *src/test* folder. The CSV file used for a feeder in the Gatling test should be located under the *src/test/resources* directory [63]; otherwise, the file is not found by Gatling, and the test execution is aborted. A *resources* directory under the *gatling* folder contains the requests' payloads. Each payload is stored into a separate *.txt* file. A hierarchy of the test folder is shown in Figure 37.

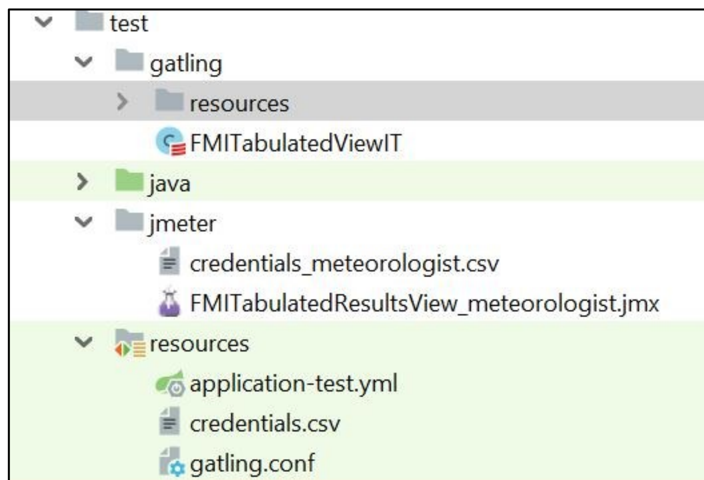

Figure 37. Structure of a src/test folder

The complete XML snippet for a `<profile>` set-up from a *pom.xml* can be found in Appendix D.

6 Discussion

As the results have shown, the defined test scenario was overly sophisticated as it tried to take too much into account. A better way to complete the test scenario would be to create multiple smaller tests. In this chapter, some of the findings and a discussion of the implemented and executed scenario are presented. Furthermore, future paths and alternatives to the proposed solution are discussed, and the decisions made for this thesis are considered.

The findings presented here are universal, as they are issues that most professionals would have to tackle as they perform initial performance testing for a flow application.

6.1 Assuring randomness and finding node id's in the test scenario

Guaranteeing randomness required the manipulation of many parameters. It would be enough to randomly choose only one *Producer* and one *Lead Time* attribute to simulate similar behavior. Contradictorily, in the current implementation, various permutations of the available options produced different scenarios each time, which resulted in variable responses' payloads. For example, if choosing the third parameter did not update the available options of the *Lead Times* grid, then its response would not contain any values that could be extracted. In this scenario, JMeter ignores lacking values and continues execution. A developer, in turn, can substitute default values if none are found. Gatling, on the other hand, fails if there are no matches found during extraction. A fail-fast approach necessitates, first, verification that there are enough parameters returned before processing them and, second, execution of the same block of code for all three requests. This complicates code significantly.

To uniquely identify an element on a page the framework uses `node` attributes. Since it depends on a UI hierarchy, the value is vulnerable to its changes. If a tester has access to the code, a better approach would be to assign an ID to the element using a `setId` method provided by the framework. Once set, an additional JSON item will be sent during the attach event of the element in a response's payload. The snippet is similar to that in Figure 38:

```
{
     "node": 13,
      "type": "put",
      "key": "id",
      "feat": 3,
      "value": "idDefined"
  }
```

Figure 38. Part of the JSON for an element with defined id

The same process for extracting the ID using Regular Expressions can be utilized. The major benefit of this when compared to the current approach is the simplicity in identifying an element without the need to look for additional helpers, such as its caption.

6.2 Debugging

Another inevitable part of creating a test is debugging. Examples of debugging include trying to identify a cause for an incorrect response or a missing value. Debugging in JMeter is straightforward, as response or request payloads can be viewed directly in the GUI under the *View Results Tree* component. For instance, if a referenced variable is not available in the session, JMeter will parse a reference statement as a `String` instead. Thus, one will know right away that there is a missing key-value pair. With Gatling, the process is a bit more complicated. First, a user must enable logging and set it either to the `DEBUG` or `TRACE` level. Once enabled, all statements are written directly into a console, from where the test is started. As quite a lot of information is written, this also implies that output needs to be stored in a separate file for processing.

On the contrary, verifying that script works as intended is easier in Gatling. A test execution fails if an expected value is not found from a response body. For example, when an internal error happens on a server-side due to an issued request, a response does not contain a `clientId` value. Therefore, a request fails during a `.check(clientIdRegex)` extraction, which indicates that something went wrong, since each response should include one. JMeter, on the other hand, requires an additional response assertion component. Figure 39 shows a response indicating an internal error.

```
for(;;);[{"changes":{},"resources":{},"locales":{},"meta":{"appError":{"caption":"Internal error","url":null,"message":"Please notify the administrator.<br>Take note of any unsaved data, and <u>click here</u> or press ESC to continue.","details":null}},"syncId":-1}]
```

Figure 39. A response with an `Internal error` message

Another one of Gatling's advantages is a flexible API for injecting users, which includes methods like `atOnceUsers`, `nothingFor`, and `rampUsers`. These methods make it easy to mimic nearly any real-user iteration scenario. JMeter, on the other hand, allows developers only to set a ramp-up period and the number of users to simulate. In this case, an interval between users launching is always the same.

Sometimes developers working on the same task cannot reproduce an error reliably on another machine. If this occurs, the first thing to check is usually the version compatibility of installed software. For example, during the testing phase, Firefox was upgraded to version 67. Starting from Firefox 67, one has to set the

`network.proxy.allow_hijacking_localhost`

to `true` to incept traffic, if a tested app is running on the localhost [64]. Otherwise, specifying the exact IP address solves the issue.

Overall, it is easier to start with JMeter rather than Gatling, since nearly everything can be configured in its GUI. There are a lot of online tutorials for this, as JMeter has been functioning for 20 years. However, is a more complicated action needs to be simulated, the configuration becomes harder and less evident. Gatling is more convenient for a sophisticated use case, since some settings can be configured using Scala directly in a script. Below, the most prominent errors and limitations of these tools, which were also encountered during the test build, are analyzed.

6.3 JMeter

Creating a test with JMeter is a straightforward task, as a lot of examples are available online. For a developer unfamiliar with Scala, most of the issues faced are due to the lack of knowledge of the language needed to perform some tasks.

For example, trying to use an `intSum` function such as `${__intSum(${anyIntVariable}, 10)}` inside a regex extractor throws a `NumberFormatException`, since `anyIntVariable` variable is stored as string, but an integer value is expected. Consequently, a BeanShell script must be added to firstly parse value and only then store it back into the variable. Therefore, instead of using the `${__intSum({availableLeadTimes_1},1)}` method directly for finding the first lead time, a *BeanShell Postprocessor* element has to be used instead. Its code is presented in Figure 40.

```
int firstKey=
Integer.parseInt(vars.get("availableLeadTimes_1")) + 1;
vars.put("availableLeadTimes_1","" +firstKe);
```

Figure 40. Parsing string to integer

JMeter does not provide any separate statistics on fetched static resources, summing retrieval time with a request, which caused its load. While this might not be a big issue, it would be beneficial to see what additional requests were issued, and for which resources.

6.4 Gatling

For this thesis, Gatling has been mostly utilized to perform additional measurements. It was used, for example, to simulate users during the phase of estimating a session size. Using Gatling at this phase was more convenient than JMeter, since specifying the number of injected users and the ramp-up period is very flexible. Also, starting a script from a console is convenient as the immediate statistics of passed or failed request were available to a tester. Nevertheless, the initial configuration and start-up were more demanding than with JMeter.

6.4.1 Recording

During recording, many requests were stored and passed as resources to other requests. Gatling's official documentation states that the `resources` method is used to simulate browser behavior, where static resources are fetched in parallel. No explanation or clarification can be found for why some of the Flow requests are captured and treated as resources. One possible reason is that a request is sent nearly simultaneously with the previous one.

Unfortunately, this does not correspond to the nature of a Flow application, where each request is a separate unit. The most vital consequence of the incorrect recording is desynchronization of `syncId` and `clientId` tokens as each request must have a value from a previous response. All misinterpreted requests were re-implemented with an `exec` block, which required even more adjustments than a generated test.

6.4.2 Test execution

Integrating the created Gatling test into a maven-based project to run during `integration-test` phase on a Windows machine is more complex than a developer might anticipate. In the case of the tests run as a part of this thesis, the difficulties were caused by a verbose error:

```
Could not exec java: Cannot run program "C:\Program
Files\Java\jdk1.8.0_91\jre\bin\java.exe": CreateProcess
error=206, The filename or extension is too long.
```

The error occurs right after the inclusion of Gatling dependencies into the profile. After spending multiple days trying to solve the problem by reading GitHub tickets, StackOverflow questions, and moving the application to another directory structure, it turned out that the error was caused by adding *gatling* plugin as a dependency. Removing it made the test executable.

6.5 Push

The application used for testing does not use push to send data to clients asynchronously; thus, a client always initiates communication. Only one-way communication has simplified testing. In case an application is push enabled, there are multiple ways to handle asynchronous communication. However, nearly each one of them requires some adjustments or compromises when building a test:

- If *polling* is used, there is no need to perform additional steps when recording a scenario, since only *XHR* requests are sent to check if there are any updates available. Polling works with both Gatling and JMeter automatically.
- *Long-polling* is not supported by Gatling nor by JMeter, but it can be implemented in the latter using the *Parallel Controller* extension. However, it is recommended

that during testing, one switch to another protocol, as there is no way to simulate behavior reliably.

- *WebSockets* connection cannot be recorded by Gatling currently [65], but it can be simulated using the `ws` element directly in a `.scala` file. JMeter does not support *WebSockets* at all, although there are multiple extensions available on the market to mimic connection. Therefore, it is recommended that Gatling is used for load testing, if an application has *WebSockets* based push enabled.

- If *WebSockets* + *XHR* based push is adopted, a request is sent via the XHR channel and a response via *WebSockets*.

6.6 Alternatives

As was established in the previous chapter, to make a test as stable as possible, many `node` IDs should be extracted. Parameterization of IDs ensures that even if a structure of a UI layout change - for example, if a new component is added - a test is still functional. A SmartMeter.io tool, which is a build on top of JMeter that enhances its functionality by adding some missing features, claims that this process can be automated by letting a *Recorder* named tool take care of that [66]. Unfortunately, this tool is proprietary and requires an expensive license. Another option that extends JMeter and brings additional enhancements and features to it is BlazeMeter. This does not provide any additional support for a Flow application specifically but might improve the development experience.

In the Vaadin directory, there is an add-on called LoadTestDriver, developed by Johannes Tuikkala that makes it possible to create a scalability test from a TestBench test [67]. TestBench is a UI testing library built by Vaadin Ltd., which, among other things, implements helpful test wrappers for the Vaadin elements. The add-on's functionality is built on top of Selenium and Gatling, and a developer can declare settings for a scalability test via the Java API. This is an excellent alternative for any developer who wants to get started quickly without familiarizing herself with load testing tools and Vaadin internals.

6.7 Future work

This thesis emphasized the server-side performance of web applications. Performance was examined by generating load on a server, where the web application is deployed to.

Simulating users' activity helps, for example, in identifying bottlenecks in database queries and revealing the requests that take the longest to respond to. Furthermore, with the help of a profiler, a root cause can be determined. This work could be extended by considering the client-side performance, measuring rendering time of UI elements. The topic has not been considered in this thesis as it should be considered separately and Flow is a server-side framework; nevertheless, it would be the next natural step.

The created tests themselves could be improved even further by replacing all `node` IDs with variables. This would ensure that - if elements presented during recording and their state, such as a caption value, are immune - tests can be reused repeatedly, even with a modified hierarchy of a UI. Extracting a `node` attribute is often not a trivial task since some elements do not have any caption or identifier to rely on, and thus workarounds would most likely be introduced.

Another step would be to create tests for a push-enabled application. Limitations and obstacles that one might face for planning and implementation were discussed earlier. However, it would be beneficial to present a working set-up. Gatling's official documentation contains a comprehensive example with a WebSocket protocol [68].

Mimicking a hardware environment as close as possible is the most important next step. Reproducing a set-up includes setting up databases and an application's server on independent servers. Generating a load from a separate computer is necessary in this case. Unfortunately, in this research, it was not possible to do so due to the privacy of data utilized by the application and the limited scope of the thesis.

7 Conclusion

In this thesis, the performance of a server-side web application created with the Vaadin Flow Framework was analyzed. The main contribution of the thesis is a defined procedure for load testing an application built with the framework. The method gives a utility to ensure that performance requirements, which were set for an application, are fulfilled. To prove its applicability, an executable test following the method was implemented using two widely utilized load testing tools: Gatling and JMeter. An application provided by FMI was employed as a target system during a test creation and execution. Since it has real use and is already in production, it was a perfect choice for this thesis.

The same use case scenario was simulated for 15, 50, and 300 users, to investigate and leverage the knowledge of the system performance. The reported results of JMeter and Gatling were different, but, on a closer inspection, it became obvious that this was due to reporting capabilities of the tools. Overall, it was found that the *Login* phase requests took the longest to process. Also, it was shown that higher utilization of the application implies longer response times on the facilities. Therefore, one should carefully plan deployment set-up based on the number of expected users.

Various hardware and software performance improvement techniques were discussed to support the planning. Among them, a load balancer and additional cache memory were reviewed as mechanisms for enhancing responsiveness. To improve the availability of applications, they are usually deployed on multiple servers. Based on the literature review, it was shown that a sticky sessions feature of load balancers is the best alternative when a Vaadin application is served from multiple places. Yet, while additional tools might improve the overall performance of a system, one must remember the responsibility of developers. For instance, memory leaks as a result of incorrect resource handling are a consequence of a bug introduced by a developer.

As well as creating and running a test, the session size and memory consumption were estimated. Due to a strangely large result, further investigations were conducted. Based on the analysis, a simple fix was implemented to cut down a session size. The result was that the memory requirements imposed by a session were 10 times smaller. The application's memory utilization was followed in the JProfiler tool. It was unexpected

that I would reveal or fix any issue, yet the outcome once again proves the value of software testing.

Quite often, scalability and load testing terms are used interchangeably. While they have been treated separately in the thesis, one could still benefit from presented information performing a scalability assessment. For example, defined regular expressions for unique payload traits of the framework can be used again later. The documented attributes include `nodeId`, which associates an element's state between a server and a client, synchronization tokens `syncId` and `clientId` , which coordinate between server and client, and `csrfToken`, which is implemented as a defence mechanism against CSRF attacks.

There are many ways to continue from here, some of which were discussed in Chapter 6. The thesis has mostly covered the server-side performance issues and enhancement since Vaadin is a server-side framework. Therefore, a closer look at the client-side influence on performance is needed.

The proposed method implies the manual assembling of a test. It might be a difficult task for a developer, who has never had testing experience. To bridge this gap, alternative solutions such as LoadTestDriver add-on in Vaadin directory and SmartMeter.io might be utilized.

References

[1] Netcraft, "January 2019 Web Server Survey," 24 01 2019. [Online]. Available: https://news.netcraft.com/archives/2019/01/24/january-2019-web-server-survey.html. [Accessed 21 03 2019].

[2] M. Wall, "How long will you wait for a shopping website to load?," 19 08 2016. [Online]. Available: https://www.bbc.com/news/business-37100091. [Accessed 21 03 2019].

[3] M. L. Abbott and M. T. Fisher, The Art of Scalabilty. Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Crawfordsville: Pearson Education,Inc., 2015.

[4] I. Molyneaux, The Art of Application Performance Testing: Help for Programmers and Quality Assurance, O'Reilly Media, 2009.

[5] J. Gray and D. P.Siewiorek Gray, "High-availability computer systems," *Computer,* vol. 9, no. 24, pp. 39-48, September 1991.

[6] M. D. Hill, "What is Scalability?," *ACM SIGARCH Computer Architecture News,* vol. 18, no. 4, pp. 18-21, December 1990.

[7] M. Rouse, "Scalability," April 2006. [Online]. Available: https://searchdatacenter.techtarget.com/definition/scalability. [Accessed 24 04 2019].

[8] W. Kenton, "Scalability," 9 April 2019. [Online]. Available: https://www.investopedia.com/terms/s/scalability.asp. [Accessed 24 04 2019].

[9] M. L. Abbott and M. T. Fisher, Scalability rules. Principles for scaling web sites, Rawfordsville: Pearson Education, Inc., 2017.

[10] HAProxy, "Starter Guide.Quick introduction to load balancing and load balancers," 11 02 2019. [Online]. Available: http://cbonte.github.io/haproxy-dconv/1.8/intro.html. [Accessed 24 04 2019].

[11] M. Anderson, "What is Load Balancing?," 14 02 2017. [Online]. Available: https://www.digitalocean.com/community/tutorials/what-is-load-balancing#how-does-the-load-balancer-choose-the-backend-server. [Accessed 30 03 2019].

[12] T. Willy, "Making applications scalable with Load Balancing," 19 11 2006. [Online]. Available: http://wtarreau.blogspot.com/2006/11/making-applications-scalable-with-load.html. [Accessed 29 05 2019].

[13] NGINX, "What Is Load Balancing?," [Online]. Available: https://www.nginx.com/resources/glossary/load-balancing/. [Accessed 30 03 2019].

[14] NGINX, "What Is Round-Robin Load Balancing?," [Online]. Available: https://www.nginx.com/resources/glossary/round-robin-load-balancing/. [Accessed 22 04 2019].

[15] IBM Knowledge center, "Caching strategies," [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SS73R8_9.4.0/ com.ibm.help.wms.perf.doc/c_FND_PM_CachingStrategies.html. [Accessed 12 04 2019].

[16] Mozilla, "HTTP Caching," 21 03 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching. [Accessed 12 04 2019].

[17] Heroku Inc., "Increasing Application Performance with HTTP Cache Headers," 20 02 2019. [Online]. Available: https://devcenter.heroku.com/articles/ increasing-application-performance-with-http-cache-headers. [Accessed 12 04 2019].

[18] M. Hofmann and L. Beaumont, in *Content Networking*, Elsevier Inc, 2005, pp. 53-79.

[19] K. Chandra, "Chapter 7: Load Balancing Caches," in *Load Balancing Servers, Firewalls, and Caches*, John Wiley & Sons, Inc., 2002, pp. 99-107.

[20] Varnish, "Varnish Documentation 6.2.0," [Online]. Available: https://varnish-cache.org/docs/6.2/index.html. [Accessed 24 04 2019].

[21] NGINX, "NGINX official page," [Online]. Available: https://www.nginx.com/. [Accessed 24 04 2019].

[22] Codeahoy, "Caching Strategies and How to Choose the Right One," 11 August 2017. [Online]. Available: https://codeahoy.com/2017/08/11 /caching-strategies-and-how-to-choose-the-right-one/. [Accessed 25 04 2019].

[23] Amazon Web Services, Inc, "Database Caching Strategies Using Redis," May 2017. [Online]. Available: https://d0.awsstatic.com/whitepapers/Database/ database-caching-strategies-using-redis.pdf. [Accessed 25 04 2019].

[24] Redis, "Introduction to Redis," [Online]. Available: https://redis.io/topics/introduction. [Accessed 25 01 2020].

[25] Microsoft, "Administer a Report Server Database (SSRS Native Mode)," 14 03 2017. [Online]. Available: https://docs.microsoft.com/en-us/sql/reporting-

services/report-server/administer-a-report-server-database-ssrs-native-mode?view=sql-server-ver15. [Accessed 25 01 2020].

[26] Google Guava, "Guava CachesExplained," [Online]. Available: https://github.com/google/guava/wiki/CachesExplained. [Accessed 25 04 2019].

[27] cache2k, "cache2k – High Performance Java Caching," [Online]. Available: https://cache2k.org/. [Accessed 25 04 2019].

[28] Memcached, "Memcached overview," [Online]. Available: https://github.com/memcached/memcached/wiki/Overview. [Accessed 25 04 2019].

[29] B. Rajkumar, P. Mukaddim and V. Athena, "Content Delivery Networks," in *Content Delivery Networks: State of the Art, Insights,and Imperatives*, Berlin, Springer, 2008, pp. 3-33.

[30] Oracle, "Chapter 2. The Structure of the Java Virtual Machine," 21 08 2018. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-2.html#jvms-2.5.2. [Accessed 23 04 2019].

[31] Sun Microsystems, "Memory Management in the Java HotSpot™ Virtual Machine," 2006.

[32] Oracle, "The try-with-resources Statement," [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions /tryResourceClose.html. [Accessed 23 04 2019].

[33] B. Goetz, "Part III. Liveness, Performance, and Testing," in *Java Concurrency in Practice*, Westford, Pearson Education, Inc., 206, pp. 203-274.

[34] Pivotal, "Spring Boot Features. Configure a Datasource," [Online]. Available: https://docs.spring.io/spring-boot/docs/2.2.0.M2/reference/html/spring-boot-features.html#boot-features-configure-datasource. [Accessed 27 05 2019].

[35] Vaadin Ltd., "Vaadin Flow," 2016. [Online]. Available: https://github.com/vaadin/flow. [Accessed 29 04 2019].

[36] Vaadin Ltd., "Introduction to Vaadin Flow," 2019. [Online]. Available: https://vaadin.com/docs/v13/flow/introduction/introduction-overview.html. [Accessed 26 04 2019].

[37] Atmosphere, "Atmosphere Github page," [Online]. Available: https://github.com/Atmosphere/atmosphere. [Accessed 26 04 2019].

[38] R. Mordani, "Java™ Servlet Specification," Sun microsystems, December 2009. [Online]. Available: https://download.oracle.com/otn-pub/jcp/servlet-3.0-fr-eval-oth-JSpec/servlet-3_0-final-

spec.pdf?AuthParam=1556528312_b9d9deee2fcecbf992a76b67fc8ff87e. [Accessed 29 04 2019].

[39] WEBCOMPONENTS.ORG, "Web Components Specifications," [Online]. Available: https://www.webcomponents.org/specs. [Accessed 28 04 2019].

[40] E. Bidelman, "Shadow DOM v1: Self-Contained Web Components," [Online]. Available: https://developers.google.com/web/fundamentals/web-components/shadowdom. [Accessed 28 04 2019].

[41] E. Bidelman, "HTML's New Template Tag," 26 February 2013. [Online]. Available: https://www.html5rocks.com/en/tutorials/webcomponents/template/. [Accessed 28 04 2019].

[42] whatwg.org, "HTML Living Standart," 15 April 2019. [Online]. Available: https://html.spec.whatwg.org/multipage/scripting.html#the-template-element/. [Accessed 28 04 2019].

[43] Vaadin Ltd., "Creating a Simple Component Using the Template API," [Online]. Available: https://vaadin.com/docs/v13/flow/polymer-templates /tutorial-template-basic.html. [Accessed 03 05 2019].

[44] MDN Web Docs, "Using Custom Elements," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements. [Accessed 21 05 2019].

[45] WHATWG Community, "HTML Living Standart.Custom Elements," [Online]. Available: https://html.spec.whatwg.org/multipage/custom-elements.html #custom-elements. [Accessed 24 05 2019].

[46] A. Deveria, "Can I use ES6?," [Online]. Available: https://caniuse.com/#search=ES6. [Accessed 10 10 2019].

[47] M. Contributors, "Polyfill," 30 08 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Polyfill. [Accessed 10 10 2019].

[48] J. Jansson, "Do Vaadin apps scale?," 27 07 2018. [Online]. Available: https://medium.com/jens-jansson/do-vaadin-apps-scale-a0ce4dfefec6. [Accessed 21 05 2019].

[49] L. Åstrand, "Session Replication in the World of Vaadin," February 2019. [Online]. Available: https://vaadin.com/blog/session-replication-in-the-world-of-vaadin. [Accessed 29 04 2019].

[50] OWASP, "Cross-Site Request Forgery prevention cheat sheet," [Online]. Available: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets

/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.md. [Accessed 29 04 2019].

[51] Vaadin Ltd., "ServerRpcHandler.java," [Online]. Available: https://github.com/vaadin/flow/blob/master/flow-server/src/main/java/com/vaadin/flow/server/communication/ServerRpcHandler.java#L309. [Accessed 06 05 2019].

[52] L. Åstrand, "CSRF token passed in GET request," 28 02 2017. [Online]. Available: https://github.com/vaadin/framework/issues/8700#issuecomment-282957085. [Accessed 29 09 2019].

[53] H. Edwin, "What are the best Performance Testing Tools?," 01 08 2018. [Online]. Available: https://techcommunity.microsoft.com/t5/TestingSpot-Blog/What-are-the-best-Performance-Testing-Tools/ba-p/367774. [Accessed 28 05 2019].

[54] Apache JMeter, "User Manual. Component Reference," [Online]. Available: http://jmeter.apache.org/usermanual/component_reference.html#HTTP_Cookie_Manager. [Accessed 10 05 2019].

[55] Apache JMeter, "Elements of a Test Plan," [Online]. Available: https://jmeter.apache.org/usermanual/test_plan.html. [Accessed 13 05 2019].

[56] Gatling Corp, "Checks. Saving," [Online]. Available: https://gatling.io/docs/current/http/http_check#saving. [Accessed 23 05 2019].

[57] A. Scheller, "UUID regex," 04 02 2015. [Online]. Available: https://adamscheller.com/regular-expressions/uuid-regex/. [Accessed 24 05 2019].

[58] A. Smirnova, "JMeter script file," 13 02 2020. [Online]. Available: https://github.com/anasmi/LoadTestingVaadin14/blob/master/FMITabulatedResultsView_meteorologist.jmx. [Accessed 13 02 2020].

[59] J. Suereth, "String interpolation," [Online]. Available: https://docs.scala-lang.org/overviews/core/string-interpolation.html. [Accessed 20 11 2019].

[60] R. E. PATTERNS. [Online]. Available: https://docs.scala-lang.org/tour/regular-expression-patterns.html. [Accessed 20 11 2019].

[61] Gatling Corp, "HTTP Request Body," [Online]. Available: https://gatling.io/docs/current/http/http_request/#http-request-body-elfile. [Accessed 23 05 2019].

[62] J. Lehtinen, "Session Size With WebApplicationContext > 10MBs," 22 11 2010. [Online]. Available: https://vaadin.com/forum/thread/255090/session-size-with-webapplicationcontext-10mbs. [Accessed 14 11 2019].

[63] Gatling Corp, "File Based Feeders," [Online]. Available: https://gatling.io/docs/current/session/feeder/#file-based-feeders. [Accessed 12 10 2019].

[64] "Proxy settings for localhost not obeyed unless network.proxy.allow_hijacking_localhost is set," 03 2019. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1535581. [Accessed 15 08 2019].

[65] S. Landelle, "Recorder: Record WebSocket Upgrade requests as ws.connect," 31 10 2018. [Online]. Available: https://github.com/gatling/gatling/issues/3595. [Accessed 21 11 2019].

[66] M. Krutak, "Testing performance of Vaadin apps: step by step tutorial," 10 08 2017. [Online]. Available: https://www.smartmeter.io/blog/testing-performance-of-vaadin-apps-tutorial. [Accessed 10 18 2019].

[67] J. Tuikkala, "LoadTestDriver add-on," 17 10 2019. [Online]. Available: https://vaadin.com/directory/component/loadtestdriver-add-on/overview. [Accessed 18 10 2019].

[68] Gatling Corp, "WEBSOCKET," [Online]. Available: https://gatling.io/docs/3.0/http/websocket/. [Accessed 27 10 2019].

[69] J. Barr, "New Elastic Load Balancing Feature: Sticky Sessions," 7 April 2010. [Online]. Available: https://aws.amazon.com/blogs/aws/new-elastic-load-balancing-feature-sticky-sessions/. [Accessed 24 04 2019].

[70] Oracle, "JDK 11 Release Notes," [Online]. Available: https://www.oracle.com/technetwork/java/javase/11-relnote-issues-5012449.html. [Accessed 25 04 2019].

[71] AdoptOpenJDK, "Prebuilt OpenJDK Binaries for Free!," [Online]. Available: https://adoptopenjdk.net/index.html. [Accessed 27 04 2019].

[72] Vaadin Ltd, "Server Push Configuration," [Online]. Available: https://vaadin.com/docs/v13/flow/advanced/tutorial-push-configuration.html. [Accessed 26 04 2019].

Appendix A A custom element from an HTML Template

A.1 CustomDiv.html

```
import {PolymerElement,html} from '@polymer/polymer/polymer-
element.js';
import '@polymer/paper-input/paper-input.js';

class CustomDiv extends PolymerElement {

    static get template() {
        return html`
        <style>
            .displayColumn {
                display: flex;
                flex-direction: column;
                background-color: antiquewhite;
            }
        </style>

        <div id="customDivContainer" class="displayColumn">
            <paper-input id="inputId" label="Put your name here!">
            </paper-input>
            <button style="width:120px" on-click="handleClick">
            Alert the name!</button>
        </div>`;
    }

    static get is() {
        return 'custom-div';
    }
    handleClick(){
        alert(this.$.inputId.value)
    }
}
customElements.define(CustomDiv.is, CustomDiv);
```

A.2 MainView.java

```java
import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.html.Label;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.router.Route;

@Route("")
public class MainView extends VerticalLayout {

    public MainView() {
        Button button = new Button("Vaadin Button",
                event -> add(new Label("Clicked")));
        CustomDiv customDiv=new CustomDiv();
        add(customDiv);
        add(button);
    }
}
```

A.3 CustomDiv

```java
import com.vaadin.flow.component.Tag;
import com.vaadin.flow.component.dependency.HtmlImport;
import com.vaadin.flow.component.polymertemplate.PolymerTemplate;
import com.vaadin.flow.templatemodel.TemplateModel;

@Tag("custom-div")
@HtmlImport("src/CustomDiv.html")
public class CustomDiv extends
          PolymerTemplate<CustomDiv.CustomDivModel> {

   public interface CustomDivModel extends TemplateModel {
    }
}
```

Appendix B A configured JMeter's Test Plan

Appendix C Scala script created for a Gatling test

C.1 Test script

```
import scala.concurrent.duration._

import io.gatling.core.Predef._
import io.gatling.http.Predef._
import io.gatling.jdbc.Predef._
import scala.util.Random
import scala.collection.mutable.ListBuffer

class FMITabulatedViewIT extends Simulation {
 val baseURL="http://localhost:8080"
 val parametersURL="?v-r=uidl&v-uiId=0"
 //s"$postURL"

 val httpProtocol = http
  .baseUrl(s"$baseURL")
   .inferHtmlResources()
   .disableAutoReferer
  .acceptHeader("*/*")
  .acceptEncodingHeader("gzip, deflate")
  .acceptLanguageHeader("en-US,en;q=0.5")
  .doNotTrackHeader("1")
  .userAgentHeader("Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:66.0)
Gecko/20100101 Firefox/66.0")

 val headers_0 = Map(
  "Accept" ->
"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
  "Upgrade-Insecure-Requests" -> "1")

 val headers_1 = Map(
  "Accept" ->
"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
  "Referer" -> s"$baseURL/login",
  "Upgrade-Insecure-Requests" -> "1")

 val headers_2 = Map(
```

```
  "Content-type" -> "application/json; charset=UTF-8",
  "Referer" -> s"$baseURL")

 val headers_4 = Map(
  "Content-type" -> "application/json; charset=UTF-8",
  "Referer" -> s"$baseURL/tabulatedresults")

 val headers_25 = Map(
  "Accept" ->
"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
  "Referer" -> s"$baseURL/tabulatedresults",
  "Upgrade-Insecure-Requests" -> "1")

 val initProducersId = exec((session) => {
  session.setAll(
   "first_producer" -> (2+ Random.nextInt(17)),
   "second_producer" -> (2+ Random.nextInt(17)),
   "third_producer"-> (2+ Random.nextInt(17))
  )
 })
 val csvFeederCredentials=csv("credentials.csv").circular
 //Used in .check to retrive syncId and clientId to propagate  to the
next request
 val syncIdRegex= regex(""""syncId":([0-9]+)""").saveAs("syncId")
 val clientIdRegex= regex(""""clientId":([0-9]+)""").saveAs("clientId")
  //"Target group" combobox. Default value 312
  val targetCB=regex(""""node":([0-
9]+),"type":"put","key":"label","feat":1,"value":"Target group"""")
.saveAs("targetGroupCB")
 //Parameters grid:
 val parameterTextNode=regex(""""node":([0-
9]+),"type":"put","key":"text","feat":[0-
9]+,"value":"Parameters"""").saveAs("parameters_text")
 val parameters_label=regex(""""node":([0-9]+),"type":"splice",
"feat":[0-9]+,"index":[0-9]+,"addNodes":\[${parameters_text}\]""")
.saveAs("parameters_label")
 val parameters_grid=regex(""""addNodes":\[${parameters_label},([0-
9]+)\]""").saveAs("parameters_grid")
 //Producers grid:
```

```
 val producerTextNode=regex("""node":([0-9]+),"type":"put",
"key":"text","feat":[0-9]+,"value":"Producers"""")
.saveAs("producers_text")
 val producers_label=regex("""node":([0-
9]+),"type":"splice","feat":[0-9]+,"index":[0-
9]+,"addNodes":\[${producers_text}\]""")   .saveAs("producers_label")
 val producers_grid=regex(""""addNodes":\[${producers_label},([0-
9]+)\]""").saveAs("producers_grid")
 //Analysis grid:
 val analysisTextNode=regex("""node":([0-9]+),"type":"put",
"key":"text","feat":[0-9]+,"value":"Analysis hours"""")
.saveAs("analysis_text")
 val analysis_label=regex("""node":([0-
9]+),"type":"splice","feat":[0-9]+,"index":[0-
9]+,"addNodes":\[${analysis_text}\]""") .saveAs("analysis_label")
 val analysis_grid=regex(""""addNodes":\[${analysis_label},([0-
9]+)\]""") .saveAs("analysis_grid")
 //Lead Times grid:
 val leadTimesTextNode=regex("""node":([0-
9]+),"type":"put","key":"text","feat":[0-9]+,"value":"Lead
times"""").saveAs("leadTimes_text")
 val leadTimes_label=regex("""node":([0-
9]+),"type":"splice","feat":[0-9]+,"index":[0-
9]+,"addNodes":\[${leadTimes_text}\]""") .saveAs("leadTimes_label")
 val leadTimes_grid=regex(""""addNodes":\[${leadTimes_label},([0-
9]+)\]""").saveAs("leadTimes_grid")  //Value in recording was 325

 val searchTextNode=regex("""node":([0-9]+),"type":"put",
"key":"text","feat":[0-9]+,"value":"Search"""").saveAs("search_text")
 val search_button=regex("""node":([0-9]+),"type":"splice","feat":[0-
9]+,"index":0,"addNodes":\[${search_text}]""")
 //Available parameters
 val availableParameters=regex("""node":([0-
9]+),"type":"put","key":"text","feat":[0-9]+,"value":"[^"]*
(temperature|precipitation|wind|sea){1}[^"]*"""")
.saveAs("availableParameters")
 val first_parameter_span=regex("""node":([0-9]+),"type":"splice",
"feat":[0-9]+,"index":[0-9]+,"addNodes":\[${availableParameters}""")
.saveAs("first_parameter_span")
```

```
 val first_key_parameter=regex("""""key":"([0-9]+)",("selected":true,)
?"_renderer_[0-9]+":${first_parameter_span}""")
.saveAs("first_key_parameter")

 //Log-out button id inside a context-menu
 val logout_contextmenu_item=regex(""""node":([0-9]+),"type":"put",
"key":"tag","feat":[0-9]+,"value":"vaadin-context-menu-item"""")
.saveAs("logout_contextmenu_item")
 val contextmenu_logout=regex(""""node":([0-
9]+),"type":"put","key":"opened-
changed"""").saveAs("contextmenu_logout")
 val totalAmountParametersS=""""node":([0-
9]+),"type":"put","key":"text", "feat":[0-
9]+,"value":"[^"]*(temperature|precipation|wind|sea){1}[^"]* """".r
  val allAvailableLeadTimes=""""node":([0-
9]+),"type":"put","key":"text", "feat":[0-9]+,"value":"([0-
9]{1,3})"""".r

 //Select 3 lead times: First
 val availableLeadTimes=regex(""""node":([0-
9]+),"type":"put","key":"text","feat":[0-9]+,"value":"([0-
9]{1,3})"""").findAll.saveAs("availableLeadTimes")
  val availableLeadTimesCountIn=regex(""""node":([0-
9]+),"type":"put","key":"text","feat":[0-9]+,"value":"([0-
9]{1,3})"""").count.in(1,150)
 val availableLeadTimes_first=regex(""""node":([0-9]+),"type":"put",
"key":"text","feat":[0-9]+,"value":"([0-9]{1,3})"""").find(1)
.saveAs("availableLeadTimes_1")
 val availableLeadTimes_second=regex(""""node":([0-9]+),"type":"put",
"key":"text","feat":[0-9]+,"value":"([0-9]{1,3})"""").find(2)
.saveAs("availableLeadTimes_2")
 val availableLeadTimes_fifth=regex(""""node":([0-9]+), "type":"put",
"key":"text","feat":[0-9]+,"value":"([0-9]{1,3})"""").find(5)
.saveAs("availableLeadTimes_5")

 val availableLeadTimes_first_span =regex(""""node":([0-9]+),
"type":"splice","feat":[0-9]+,"index":[0-9]+,"addNodes":
\[${availableLeadTimes_1}\]""").saveAs("availableLeadTimes_1_span")
 val availableLeadTimes_second_span=regex(""""node":([0-9]+),
```

```
"type":"splice","feat":[0-9]+,"index":[0-9]+, "addNodes":
\[${availableLeadTimes_2}\]""").saveAs("availableLeadTimes_2_span")
 val availableLeadTimes_fifth_span=regex(""""node":([0-9]+),
"type":"splice","feat":[0-9]+,"index":[0-9]+, "addNodes":
\[${availableLeadTimes_5}\]""").saveAs("availableLeadTimes_5_span")
 //Instead of extracting a key for a lead time separately, one can
increment a span's value by one. Though this is a more reliable way
 val first_key_leadTime=regex(""""key":"([0-9]+)",
("selected":true,)?"_renderer_[0-9]+"
:${availableLeadTimes_1_span}""")
  .saveAs("first_key_leadTime")//328
 val second_key_leadTime=regex(""""key":"([0-9]+)",("selected":true,)
?"_renderer_[0-9]+":${availableLeadTimes_2_span}""")
  .saveAs("second_key_leadTime")//330
 val fifth_key_leadTime=regex(""""key":"([0-9]+)",("selected":true,)
?"_renderer_[0-9]+":${availableLeadTimes_5_span}""")
  .saveAs("fifth_key_leadTime")//332

  //Login and navigate to the "Tabulated Results View" page
  object LoginAndNavigate{
    val loginAndNavigate=exec(http("Get /")
      .get("/")
      .headers(headers_0))
      .exec(initProducersId)
      .pause(5)
      .feed(csvFeederCredentials)
      .exec(http("Login")
        .post("/login")
        .headers(headers_1)
        //Set credentials from a csv file
        .formParam("username", "${username}")
        .formParam("password", "${password}")
        .check(regex("""Vaadin-Security-Key":"([a-fA-F0-9]{8}-[a-fA-
F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})""")
.saveAs("secKey")))
      .pause(1)
      .exec(http("request_2")
        .post(s"$parametersURL")
        .headers(headers_2)
```

```
.body(ElFileBody("FMITabulatedViewMeteorologist_0002_request.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex)
      )
      .pause(0,3)
      .exec(http("request_3")
        .post(s"$parametersURL")
        .headers(headers_2)

.body(ElFileBody("FMITabulatedViewMeteorologist_0003_request.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex)
      )
      .pause(2)
      .exec(http("Navigate to 'Tabulated results view'")
        .post(s"$parametersURL")
        .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0004_request.txt"))
        //Extract node id's for the most relevant components
        // that are used in the script
        .check(syncIdRegex)
        .check(clientIdRegex)
        .check(targetCB,
          parameterTextNode,
          parameters_label,
          parameters_grid,
          producerTextNode,
          producers_label,
          producers_grid,
          analysisTextNode,
          analysis_label,
          analysis_grid,
          leadTimesTextNode,
          leadTimes_label,
          leadTimes_grid,
          searchTextNode,
          search_button)
```

```
        )
        .exec(http("request_5")
          .post(s"$parametersURL")
          .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0005_request.txt"))
          .check(syncIdRegex)
          .check(clientIdRegex)
        )
        .pause(4)
    }
//Choose three random data producers and extract 2 random parameters
    object ChooseProducers{
      val chooseProducers=exec(http("Choose 1.st producer")
        .post(s"$parametersURL")
        .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0006_request.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex))
        .exec(http("Components update after choosing 1.st producer")
          .post(s"$parametersURL")
          .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0007_request.txt"))
          .check(syncIdRegex)
          .check(clientIdRegex))
        .pause(1)
        .exec(http("Choose 2.nd producer")
          .post(s"$parametersURL")
          .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0008_request.txt"))
          .check(syncIdRegex)
          .check(clientIdRegex))
        .exec(http("Components update after choosing 2.st producer")
          .post(s"$parametersURL")
          .headers(headers_4)
```

```
.body(ElFileBody("FMITabulatedViewMeteorologist_0009_request.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex))
      .exec(http("request_10")
        .post(s"$parametersURL")
        .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0010_request.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex))
      .pause(1)
      .exec(http("Choose 3.rd producer")
        .post(s"$parametersURL")
        .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0011_request.txt"))
        .check(availableParameters)
        .check(first_parameter_span)
        .check(first_key_parameter)
        //Saving the whole response to a string called
        //"producersResponse"
        .check(bodyString.saveAs("producersResponse"))
        .check(syncIdRegex)
        .check(clientIdRegex))
      //Creating two random parameters
      .exec{
        session=>{
            //Find total amount of received parameters
            //in the last response
            val amount = totalAmountParametersS.
              findAllIn(session("producersResponse").as[String])
              .length
            //Find a key value for the first meet parameter
             val firstKeyParamete r= session("first_key_parameter")
                      .as[Int];
            //Parameters are in range from
            //[firstKeyParameter, firstKeyParameter+amount];
            //Selectining two random ones from this range.
```

```
                //Random function returns value in range [0,amount)
                val firstParameter= Random.nextInt(amount) +
                  firstKeyParameter
                val secondParameter= Random.nextInt(amount) +
                  firstKeyParameter
                session.set("firstParameter",firstParameter)
                  .set("secondParameter",secondParameter)
        }}
        .exec(http("request_12")
          .post("/?v-r=uidl&v-uiId=0")
          .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0012_request.txt"))
          .check(syncIdRegex)
          .check(clientIdRegex))
        .pause(2)
  }
  //Choose two random parameters defined in previous step and define 3
//lead times
  object ChooseParameters{
    val chooseParameters=
    //Choose first parameter&confirmUpdate
        exec(http("Choose first parameter")
          .post(s"$parametersURL")
          .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0013_request.txt"))
          .check(bodyString.saveAs("parametersResponseFirst"))
          .check(syncIdRegex)
          .check(clientIdRegex))
        .exec(http("request_14")
          .post(s"$parametersURL")
          .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0014_request.txt"))
          .check(syncIdRegex)
          .check(clientIdRegex))
        .pause(1)
        //Choose second parameter & confirm update
```

```
        .exec(http("Choose second parameter")
          .post(s"$parametersURL")
          .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0015_request.txt"))
          .check(syncIdRegex)
          .check(clientIdRegex)
          .check(bodyString.saveAs("parametersResponse"))
          .check(availableLeadTimes)
        //Amount of returned available lead times can be also 0 or 1.
        //In this case skip actions and use values defined in previous
        //step.
        .check(checkIf("${availableLeadTimes(2).exists()}")
                  (availableLeadTimes_first),
          checkIf("${availableLeadTimes(2).exists()}")
                  (availableLeadTimes_second),
          checkIf("${availableLeadTimes(2).exists()}")
                  (availableLeadTimes_fifth),
          checkIf("${availableLeadTimes(2).exists()}")
                  (availableLeadTimes_first_span),
          checkIf("${availableLeadTimes(2).exists()}")
                  (availableLeadTimes_second_span),
          checkIf("${availableLeadTimes(2).exists()}")
                  (availableLeadTimes_fifth_span),
          checkIf("${availableLeadTimes(2).exists()}")
                  (first_key_leadTime),
          checkIf("${availableLeadTimes(2).exists()}")
                  (second_key_leadTime),
          checkIf("${availableLeadTimes(2).exists()}")
                  (fifth_key_leadTime))
      //If there are 3 available lead times in response to the selection
      //of second parameter, then use those
      //Otherwise, parse the first response of parameter selection and
      //pick the values from there
      //If there are no values in either of responses, then select
      //default values captured during recording.
          // Might not actually do anything, but makes test stable
      ).doIf(session=>
allAvailableLeadTimes.findAllIn(session("parametersResponseFirst")
```

```
.as[String]).length >5
      && session("first_key_leadTime").asOption[String].isEmpty){
        //If  there are more than 5 lead times available in the first
//response and none were found in the second one, then:
      exec(session => {
        //Get saved response as a string to use in following regex
        val response=session("parametersResponseFirst").as[String]
        val availLeadTimes=new ListBuffer[String]()
        //Since extracted values are saved as first group, below
//modifications are needed
        allAvailableLeadTimes.findAllIn(response).matchData foreach{
          m=> availLeadTimes+=m.group(1)
        }
        val firstLead= availLeadTimes(0)
        val secondLead=availLeadTimes(2)
        val fifthLead=availLeadTimes(5)
        //Defing spans and keys; the value assigned prior is the one
//captured during recording
        //Defining first key
        var first_span = "1303"
        ((raw"""node":([0-9]+),"type":"splice","feat":[0-9]+,
               "index":[0-9]+,"addNodes":\[$firstLead\]""").r)
               .findFirstMatchIn(response) match{
          case Some(m)=>
            first_span=m.group(1)
          case None =>
            println("A span element for the first Lead Time
                      label is not found")
        }
        var first_key_leadTime="328"
        (raw"""key":"([0-9]+)",("selected":true,)?"_
         renderer_[0-9]+":$first_span""".r)
        .findFirstMatchIn(response) match {
          case Some(m) =>
            first_key_leadTime = m.group(1)
          case None =>
            println("An id is not found for the first key lead.
               Using default 328")
        }
```

```
//Defining second key
var second_span="1309"
((raw""""node":([0-9]+),"type":"splice","feat":[0-9]+,
"index":[0-9]+,"addNodes":\[$secondLead\]""").r)
.findFirstMatchIn(response) match {
  case Some(m) =>
    second_span = m.group(1)
  case None =>
    println("A span element for the second Lead Time
        label is not found")
}
var second_key_leadTime="330"
(raw""""key":"([0-9]+)",("selected":true,)?"_
 renderer_[0-9]+":$second_span""".r)
.findFirstMatchIn(response) match {
  case Some(m) =>
    second_key_leadTime = m.group(1)
  case None =>
    println("An id is not found for the second key lead.
        Using default 330")
}
//Defining fifth key
var fifth_span= "1319"
((raw""""node":([0-9]+),"type":"splice","feat":[0-9]+,
"index":[0-9]+,"addNodes":\[$fifthLead\]""").r)
.findFirstMatchIn(response) match {
  case Some(m) =>
    fifth_span = m.group(1)
  case None =>
    println("A span element for the fifth Lead Time label
        is not found")
}
var fifth_key_leadTime="332"
(raw""""key":"([0-9]+)",("selected":true,)?"_
renderer_[0-9]+":$fifth_span""".r)
.findFirstMatchIn(response) match {
  case Some(m) =>
    fifth_key_leadTime = m.group(1)
```

```
          case None =>
            println("An id is not found for the first key lead.
           Using a default value 332")
        }
        session.setAll(
          "first_key_leadTime" ->first_key_leadTime,
          "second_key_leadTime"->second_key_leadTime,
          "fifth_key_leadTime"-> fifth_key_leadTime
        )
      })
    }
      //If no lead times were added when choosing parameters,
        use default values

.doIf(session=>session("first_key_leadTime").asOption[String].isEmpty)
{
      exec(session=>
        session.setAll(
          "first_key_leadTime" ->328,
          "second_key_leadTime"->330,
          "fifth_key_leadTime"-> 332
        )
      )
    }
      .exec(http("request_16")
        .post(s"$parametersURL")
        .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0016_request.txt"))
        .check()
        .check(syncIdRegex)
        .check(clientIdRegex))
      .pause(3)
  }

//Picking 3 lead times defined in previous step
  object ChooseLeadTime{
    val chooseLeadTime=
      //Select first lead-time
```

```
      exec(http("Select 1.st lead time")
        .post(s"$parametersURL")
        .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0017_request.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex))
      //Select second lead-time
      .exec(http("Select 2.nd lead time")
        .post(s"$parametersURL")
        .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0018_request.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex))
      //Select 5.th lead-time
      .exec(http("Select 5.th lead time")
        .post(s"$parametersURL")
        .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0019_request.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex))
      .pause(2)
  }
//Fetching data into final Grid after filters are set
object FetchResultsGrid{
    val fetchResultsGrid=
      exec(http("Click Search button")
        .post(s"$parametersURL")
        .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0020_request.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex))
      .exec(http("request_21")
        .post(s"$parametersURL")
        .headers(headers_4)
```

```
.body(ElFileBody("FMITabulatedViewMeteorologist_0021_request.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex))
      .pause(2,5)
      .exec(http("Scroll Grid")
        .post(s"$parametersURL")
        .headers(headers_4)
.body(ElFileBody("FMITabulatedViewMeteorologist_scrollGrid_request.txt
"))
        .check(syncIdRegex)
        .check(clientIdRegex))
      .exec(http("Update confirmed in scrolled Grid")
        .post(s"$parametersURL")
        .headers(headers_4)
.body(ElFileBody("FMITabulatedViewMeteorologist_scrollGridUpdate_reque
st.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex))
      .pause(0,4)
  }
 //A log-out action
object LogOut{
    val logOut=
    exec(http("Click context-menu button ${username}")
      .post(s"$parametersURL")
      .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0022_request.txt"))
      .check(logout_contextmenu_item)
      .check(contextmenu_logout)
      .check(syncIdRegex)
      .check(clientIdRegex))
      .exec(http("Open log-out context-menu")
        .post(s"$parametersURL")
        .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0023_request.txt"))
        .check(syncIdRegex)
        .check(clientIdRegex)
```

```
        )
        .exec(http("Click 'Log-out button' in Context-menu")
          .post(s"$parametersURL")
          .headers(headers_4)

.body(ElFileBody("FMITabulatedViewMeteorologist_0024_request.txt"))
          .check(syncIdRegex)
          .check(clientIdRegex)
        )
        //Navigate to log-out page
        .exec(http("/logout")
        .get("/logout")
        .headers(headers_25))
    }

//Scenario definition
val userScenario=
scenario("Meterologist").exec(LoginAndNavigate.loginAndNavigate,
  ChooseProducers.chooseProducers,
  ChooseParameters.chooseParameters,
  ChooseLeadTime.chooseLeadTime,
  FetchResultsGrid.fetchResultsGrid,
  LogOut.logOut)

 // User injection for the defined user scenario
 setUp(userScenario.inject(
 // atOnceUsers(4),
  rampUsers(300) during (200 seconds))
 ).protocols(httpProtocol)
}
```

C.2 Bodies

File name	Request body
FMITabulatedViewMeteorologist0002_request.tx	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":87,"event":"value-changed","data":{}},{"type":"event","node":34,"event":"value-changed","data":{}},{"type":"event","node":33,"event"

	:"value- changed","data":{}},{"type":"event","node":32,"event" :"value- changed","data":{}},{"type":"event","node":31,"event" :"value- changed","data":{}},{"type":"event","node":14,"event" :"value- changed","data":{}},{"type":"event","node":13,"event" :"value- changed","data":{}},{"type":"event","node":12,"event" :"value- changed","data":{}},{"type":"event","node":11,"event" :"value- changed","data":{}},{"type":"mSync","node":87,"featur e":1,"property":"value","value":""},{"type":"publishe dEventHandler","node":16,"templateEventMethodName":"s etDetailsVisible","templateEventMethodArgs":[null]},{ "type":"publishedEventHandler","node":16,"templateEve ntMethodName":"sortersChanged","templateEventMethodAr gs":[[]]},{"type":"publishedEventHandler","node":45," templateEventMethodName":"setDetailsVisible","templat eEventMethodArgs":[null]},{"type":"publishedEventHand ler","node":45,"templateEventMethodName":"sortersChan ged","templateEventMethodArgs":[[]]},{"type":"publish edEventHandler","node":56,"templateEventMethodName":" setDetailsVisible","templateEventMethodArgs":[null]}, {"type":"publishedEventHandler","node":56,"templateEv entMethodName":"sortersChanged","templateEventMethodA rgs":[[]]},{"type":"publishedEventHandler","node":67, "templateEventMethodName":"setDetailsVisible","templa teEventMethodArgs":[null]},{"type":"publishedEventHan dler","node":67,"templateEventMethodName":"sortersCha nged","templateEventMethodArgs":[[]]},{"type":"publis hedEventHandler","node":78,"templateEventMethodName": "setDetailsVisible","templateEventMethodArgs":[null]} ,{"type":"publishedEventHandler","node":78,"templateE ventMethodName":"sortersChanged","templateEventMethod Args":[[]]},{"type":"publishedEventHandler","node":16 ,"templateEventMethodName":"confirmUpdate","templateE

	ventMethodArgs":[0]},{"type":"publishedEventHandler","node":45,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[0]},{"type":"publishedEventHandler","node":56,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[0]},{"type":"publishedEventHandler","node":67,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[0]},{"type":"publishedEventHandler","node":78,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[0]},{"type":"mSync","node":34,"feature":1,"property":"filter","value":""},{"type":"mSync","node":34,"feature":1,"property":"invalid","value":false},{"type":"mSync","node":34,"feature":1,"property":"opened","value":false},{"type":"mSync","node":33,"feature":1,"property":"filter","value":""},{"type":"mSync","node":33,"feature":1,"property":"invalid","value":false},{"type":"mSync","node":33,"feature":1,"property":"opened","value":false},{"type":"mSync","node":32,"feature":1,"property":"filter","value":""},{"type":"mSync","node":32,"feature":1,"property":"invalid","value":false},{"type":"mSync","node":32,"feature":1,"property":"opened","value":false},{"type":"mSync","node":31,"feature":1,"property":"value","value":""},{"type":"mSync","node":14,"feature":1,"property":"filter","value":""},{"type":"mSync","node":14,"feature":1,"property":"invalid","value":false},{"type":"mSync","node":14,"feature":1,"property":"opened","value":false},{"type":"mSync","node":13,"feature":1,"property":"invalid","value":false},{"type":"mSync","node":13,"feature":1,"property":"opened","value":null},{"type":"mSync","node":12,"feature":1,"property":"invalid","value":false},{"type":"mSync","node":12,"feature":1,"property":"opened","value":null},{"type":"mSync","node":11,"feature":1,"property":"value","value":""}],"syncId":0,"clientId":0}
FMITabulated ViewMeteorolo	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":87,"event":"value-changed","data":{}},{"type":"event","node":31,"event"

gist_0003_request.txt	:"value-changed","data":{}},{"type":"event","node":11,"event":"value-changed","data":{}}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0004_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"navigation","location":"tabulatedresults","link":1}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0005_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":369,"event":"value-changed","data":{}},{"type":"event","node":323,"event":"value-changed","data":{}},{"type":"event","node":${targetGroupCB},"event":"value-changed","data":{}},{"type":"mSync","node":369,"feature":1,"property":"invalid","value":false},{"type":"publishedEventHandler","node":314,"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":[null]},{"type":"publishedEventHandler","node":314,"templateEventMethodName":"sortersChanged","templateEventMethodArgs":[[]]},{"type":"publishedEventHandler","node":319,"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":[null]},{"type":"publishedEventHandler","node":319,"templateEventMethodName":"sortersChanged","templateEventMethodArgs":[[]]},{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":[null]},{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"sortersChanged","templateEventMethodArgs":[[]]},{"type":"publishedEventHandler","node":330,"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":[null]},{"type":"publishedEventHandler","node":330,"templateEventMethodName":"sortersChanged","templateEventMethodArgs":[[]]},{"type":"publishedEventHandler","node":${parameters_grid},"templateEventMethodN

ame":"setDetailsVisible","templateEventMethodArgs":[n
ull]},{"type":"publishedEventHandler","node":${parame
ters_grid},"templateEventMethodName":"sortersChanged"
,"templateEventMethodArgs":[[]]},{"type":"publishedEv
entHandler","node":${producers_grid},"templateEventMe
thodName":"setDetailsVisible","templateEventMethodArg
s":[null]},{"type":"publishedEventHandler","node":${p
roducers_grid},"templateEventMethodName":"sortersChan
ged","templateEventMethodArgs":[[]]},{"type":"publish
edEventHandler","node":389,"templateEventMethodName":
"setDetailsVisible","templateEventMethodArgs":[null]}
,{"type":"publishedEventHandler","node":389,"template
EventMethodName":"sortersChanged","templateEventMetho
dArgs":[[]]},{"type":"publishedEventHandler","node":3
14,"templateEventMethodName":"confirmUpdate","templat
eEventMethodArgs":[0]},{"type":"publishedEventHandler
","node":319,"templateEventMethodName":"confirmUpdate
","templateEventMethodArgs":[0]},{"type":"publishedEv
entHandler","node":325,"templateEventMethodName":"con
firmUpdate","templateEventMethodArgs":[0]},{"type":"p
ublishedEventHandler","node":330,"templateEventMethod
Name":"confirmUpdate","templateEventMethodArgs":[0]},
{"type":"publishedEventHandler","node":${parameters_g
rid},"templateEventMethodName":"confirmUpdate","templ
ateEventMethodArgs":[0]},{"type":"publishedEventHandl
er","node":340,"templateEventMethodName":"confirmUpda
te","templateEventMethodArgs":[0]},{"type":"published
EventHandler","node":389,"templateEventMethodName":"c
onfirmUpdate","templateEventMethodArgs":[0]},{"type":
"mSync","node":368,"feature":1,"property":"checked","
value":false},{"type":"mSync","node":365,"feature":1,
"property":"checked","value":true},{"type":"mSync","n
ode":362,"feature":1,"property":"checked","value":fal
se},{"type":"mSync","node":323,"feature":1,"property"
:"filter","value":""},{"type":"mSync","node":323,"fea
ture":1,"property":"invalid","value":false},{"type":"
mSync","node":323,"feature":1,"property":"opened","va
lue":false},{"type":"mSync","node":${targetGroupCB},"
feature":1,"property":"filter","value":""},{"type":"m

	Sync","node":${targetGroupCB},"feature":1,"property":"invalid","value":false},{"type":"mSync","node":${targetGroupCB},"feature":1,"property":"opened","value":false}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0006_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":${producers_grid},"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":["${first_producer}"]},{"type":"event","node":${producers_grid},"event":"item-click","data":{"event.detail.screenY":568,"event.detail.metaKey":false,"event.detail.button":0,"event.detail.shiftKey":false,"event.detail.screenX":3688,"event.detail.itemKey":"${first_producer}","event.detail.altKey":false,"event.detail.clientX":1005,"event.detail.detail":1,"event.detail.clientY":326,"event.detail.ctrlKey":true}}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0007_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":323,"event":"value-changed","data":{}},{"type":"event","node":${targetGroupCB},"event":"value-changed","data":{}},{"type":"publishedEventHandler","node":${parameters_grid},"templateEventMethodName":"setRequestedRange","templateEventMethodArgs":[0,50]},{"type":"publishedEventHandler","node":${parameters_grid},"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[1]},{"type":"publishedEventHandler","node":330,"templateEventMethodName":"setRequestedRange","templateEventMethodArgs":[0,50]},{"type":"publishedEventHandler","node":330,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[1]},{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"setRequestedRange","templateEventMethodArgs":[0,50]},{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[1]},

	{"type":"publishedEventHandler","node":314,"templateEventMethodName":"setRequestedRange","templateEventMethodArgs":[0,50]},{"type":"publishedEventHandler","node":314,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[1]}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0008_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":${producers_grid},"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":["${second_producer}"]},{"type":"event","node":${producers_grid},"event":"item-click","data":{"event.detail.screenY":706,"event.detail.metaKey":false,"event.detail.button":0,"event.detail.shiftKey":false,"event.detail.screenX":3673,"event.detail.itemKey":"${second_producer}","event.detail.altKey":false,"event.detail.clientX":990,"event.detail.detail":1,"event.detail.clientY":464,"event.detail.ctrlKey":true}}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0009_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":323,"event":"value-changed","data":{}},{"type":"event","node":${targetGroupCB},"event":"value-changed","data":{}},{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"setRequestedRange","templateEventMethodArgs":[0,100]},{"type":"publishedEventHandler","node":${parameters_grid},"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[2]},{"type":"publishedEventHandler","node":330,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[2]},{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[2]},{"type":"publishedEventHandler","node":314,"templateEventMethodName":"confirmUpdate","templateEventMe

	thodArgs":[2]}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0010_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[3]}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0011_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":${producers_grid},"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":["${third_producer}"]},{"type":"event","node":${producers_grid},"event":"item-click","data":{"event.detail.screenY":639,"event.detail.metaKey":false,"event.detail.button":0,"event.detail.shiftKey":false,"event.detail.screenX":3693,"event.detail.itemKey":"${third_producer}","event.detail.altKey":false,"event.detail.clientX":1010,"event.detail.detail":1,"event.detail.clientY":397,"event.detail.ctrlKey":true}}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0012_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":323,"event":"value-changed","data":{}},{"type":"event","node":${targetGroupCB},"event":"value-changed","data":{}},{"type":"publishedEventHandler","node":${parameters_grid},"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[3]},{"type":"publishedEventHandler","node":330,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[3]},{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[4]},{"type":"publishedEventHandler","node":314,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[3]}],"syncId":${syncId},"clientId":${clientId}}

FMITabulatedViewMeteorologist_0013_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":${parameters_grid},"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":["${firstParameter}"]},{"type":"event","node":${parameters_grid},"event":"item-click","data":{"event.detail.screenY":534,"event.detail.metaKey":false,"event.detail.button":0,"event.detail.shiftKey":false,"event.detail.screenX":3774,"event.detail.itemKey":"${firstParameter}","event.detail.altKey":false,"event.detail.clientX":1091,"event.detail.detail":1,"event.detail.clientY":292,"event.detail.ctrlKey":true}}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0014_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":323,"event":"value-changed","data":{}},{"type":"event","node":${targetGroupCB},"event":"value-changed","data":{}},{"type":"publishedEventHandler","node":330,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[4]},{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[5]},{"type":"publishedEventHandler","node":314,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[4]}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0015_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":${parameters_grid},"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":["${secondParameter}"]},{"type":"event","node":${parameters_grid},"event":"item-click","data":{"event.detail.screenY":703,"event.detail.metaKey":false,"event.detail.button":0,"event.detail.shiftKey":false,"event.detail.screenX":3758,"event.detail.itemKey":"${secondParameter}","event.detail.altKey":false,"event.detail.clientX":1075,"event.detail.detail":1,"event.detail.clientY":461,"event.detail.

	ctrlKey":true}}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0016_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":323,"event":"value-changed","data":{}},{"type":"event","node":${targetGroupCB},"event":"value-changed","data":{}},{"type":"publishedEventHandler","node":330,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[5]},{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[6]},{"type":"publishedEventHandler","node":314,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[5]}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0017_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":["${first_key_leadTime}"]},{"type":"event","node":${leadTimes_grid},"event":"item-click","data":{"event.detail.screenY":582,"event.detail.metaKey":false,"event.detail.button":0,"event.detail.shiftKey":false,"event.detail.screenX":3702,"event.detail.itemKey":"${first_key_leadTime}","event.detail.altKey":false,"event.detail.clientX":1019,"event.detail.detail":1,"event.detail.clientY":340,"event.detail.ctrlKey":true}}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0018_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":["${second_key_leadTime}"]},{"type":"event","node":${leadTimes_grid},"event":"item-click","data":{"event.detail.screenY":659,"event.detail.metaKey":false,"event.detail.button":0,"event.detail.shiftKey":false,"event.detail.screenX":3660,"event.detail.itemKey":"${second_key_leadTime}","event.deta

	il.altKey":false,"event.detail.clientX":977,"event.detail.detail":1,"event.detail.clientY":417,"event.detail.ctrlKey":true}}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0019_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":${leadTimes_grid},"templateEventMethodName":"setDetailsVisible","templateEventMethodArgs":["${fifth_key_leadTime}"]},{"type":"event","node":${leadTimes_grid},"event":"item-click","data":{"event.detail.screenY":737,"event.detail.metaKey":false,"event.detail.button":0,"event.detail.shiftKey":false,"event.detail.screenX":3659,"event.detail.itemKey":"${fifth_key_leadTime}","event.detail.altKey":false,"event.detail.clientX":976,"event.detail.detail":1,"event.detail.clientY":495,"event.detail.ctrlKey":true}}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0020_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":305,"event":"click","data":{"event.shiftKey":false,"event.metaKey":false,"event.detail":1,"event.ctrlKey":false,"event.clientX":1108,"event.clientY":592,"event.altKey":false,"event.button":0,"event.screenY":834,"event.screenX":3791}}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0021_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":389,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[1]}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_scrollGrid_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":389,"templateEventMethodName":"setRequestedRange","templateEventMethodArgs":[0,100]}],"syncId":${syncId},"clientId":${clientId}}

FMITabulatedViewMeteorologist_scrollGridUpdate_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"publishedEventHandler","node":389,"templateEventMethodName":"confirmUpdate","templateEventMethodArgs":[2]}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0022_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":120,"event":"click","data":{"event.shiftKey":false,"event.metaKey":false,"event.detail":1,"event.ctrlKey":false,"event.clientX":1220,"event.clientY":15,"event.altKey":false,"event.button":0,"event.screenY":257,"event.screenX":3903}},{"type":"event","node":120,"event":"vaadin-context-menu-before-open"}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0023_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":${contextmenu_logout},"event":"opened-changed"},{"type":"event","node":${contextmenu_logout},"event":"opened-changed"},{"type":"mSync","node":${contextmenu_logout},"feature":1,"property":"opened","value":true}],"syncId":${syncId},"clientId":${clientId}}
FMITabulatedViewMeteorologist_0024_request.txt	{"csrfToken":"${secKey}","rpc":[{"type":"event","node":${logout_contextmenu_item},"event":"click","data":{"event.shiftKey":false,"event.metaKey":false,"event.detail":1,"event.ctrlKey":false,"event.clientX":1200,"event.clientY":47,"event.altKey":false,"event.button":0,"event.screenY":289,"event.screenX":3883}},{"type":"event","node":${contextmenu_logout},"event":"opened-changed"},{"type":"mSync","node":${contextmenu_logout},"feature":1,"property":"opened","value":false}],"syncId":${syncId},"clientId":${clientId}}

Appendix D Profile for integration tests

```
<!--RUNNING JMeter integration tests-->
<profile>
    <id>integrationTest</id>
    <dependencies>
        <!--Gatling Dependencies-->
        <dependency>
            <groupId>io.gatling</groupId>
            <artifactId>gatling-core</artifactId>
            <version>${gatling.version}</version>
        </dependency>
        <dependency>
            <groupId>io.gatling</groupId>
            <artifactId>gatling-app</artifactId>
            <version>${gatling.version}</version>
        </dependency>
         <!—Needed to generate reports -->
        <dependency>
            <groupId>io.gatling.highcharts</groupId>
            <artifactId>gatling-charts-highcharts</artifactId>
            <version>${gatling.version}</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <!--To run integration tests -->
            <plugin>
                <artifactId>maven-failsafe-plugin</artifactId>
                <version>2.22.0</version>
                <executions>
                    <execution>
                        <goals>
                           <!--Runs Integration tests-->
                            <goal>integration-test</goal>
<!--running integration tests, you should invoke Maven with the `mvn
verify` rather than trying to invoke the integration-test phase
```

```
directly, as otherwise the `post-integration-test` phase will not be
executed.-->
                            <goal>verify</goal>
                        </goals>
                        <configuration>
                            <includes>
                                <include>**/*IT.java</include>
                                <include>**/*It.java</include>
<include>**/FMITabulatedResultsView_meteorologist.jmx</include>
                            </includes>
                            <excludes>
                                <exclude>**/*UT.java</exclude>
                                <exclude>**/*UT.java</exclude>
                            </excludes>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <wait>100000</wait>
                    <maxAttempts>18</maxAttempts>
                </configuration>
                <executions>
                    <!--Start-up -->
                    <execution>
                        <id>start-spring-boot</id>
                        <phase>pre-integration-test</phase>
                        <goals>
                            <goal>start</goal>
                        </goals>
                    </execution>
                    <!--Tear down-->
                    <execution>
                        <id>stop-spring-boot</id>
                        <phase>post-integration-test</phase>
                        <goals>
```

```
                        <goal>stop</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <!--Add JMeter plugin-->
        <plugin>
            <groupId>com.lazerycode.jmeter</groupId>
            <artifactId>jmeter-maven-plugin</artifactId>
            <version>2.1.0</version>
            <executions>
                <execution>
                    <id>jmeter-tests</id>
                    <goals>
                        <goal>jmeter</goal>
                    </goals>
                    <phase>integration-test</phase>
                </execution>
            </executions>
        </plugin>
<!--GATLING https://gatling.io/docs/2.3/extensions/maven_plugin Maven
plugin-->
     <plugin>
         <groupId>io.gatling</groupId>
         <artifactId>gatling-maven-plugin</artifactId>
         <version>${gatling.version}</version>
         <configuration>

<resourcesFolder>src/test/gatling/resources</resourcesFolder>
           <simulationsFolder>src/test/gatling</simulationsFolder>
        </configuration>
          <!-- Runs within maven verify phase-->
          <executions>
                 <execution>
                     <goals>
                         <goal>test</goal>
                     </goals>
                 </execution>
           </executions>
```

```xml
                </plugin>
            </plugins>
        </build>
    </profile>
```