Master thesis

Philip Dakowitz

Comparing reactive and conventional programming of Java based microservices in containerised environments

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Philip Dakowitz

Comparing reactive and conventional programming of Java based microservices in containerised environments

Master thesis eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuer: Prof. Dr. Stefan Sarstedt
Betreuer: Prof. Dr. Olaf Zukunft

Eingereicht am: 27. März 2018

Philip Dakowitz

Thema der Arbeit

Vergleich von reaktiver und konventioneller Programmierung Java-basierter Microservices in containerisierten Umgebungen

Stichworte

Reaktive Systeme, Reaktive Programmierung, Microservices, Softwarearchitektur, Java, Containervirtualisierung, Docker, Kubernetes, Softwareentwicklung, Lasttests

Kurzzusammenfassung

Dieses Dokument beschreibt den Vergleich von reaktiver und konventioneller Programmierung anhand eines in Java entwickelten Beispielprojekts mit dem Ziel herauszufinden, ob eine der beiden Varianten Vor- oder Nachteile gegenüber der anderen hat. Es wurden automatisierte Last- und Widerstandsfähigkeitstests auf beiden Systemen durchgeführt und zusammen mit systeminternen Metriken aufgezeichnet. Es hat sich gezeigt, dass reaktive Programmierung in diesem Szenario keine besseren Ergebnisse bei den Lasttests vorweisen konnte, jedoch Vorteile bei der Fehlerbehandlung hat und die Widerstandsfähigkeit einer Software erhöhen kann.

Philip Dakowitz

Title of the paper

Comparing reactive and conventional programming of Java based microservices in containerised environments

Keywords

reactive systems, reactive programming, microservices, software architecture, Java, containerisation, Docker, Kubernetes, software development, load testing

Abstract

This document describes the comparison of reactive and conventional programming on an exemplary Java based application with the goal to determine whether one of the approaches has advantages or disadvantages over the other. Automated load and resiliency tests have been performed on both systems and the results have been recorded together with application internal metrics. It is shown that in this scenario reactive programming has no advantages regarding performance under load. However, it can improve error handling and resiliency of a software.

Contents

1 Introduction

Nowadays, many applications evolve from monolithic architectures into mobile optimised and cloud based microservice systems [1]. This evolution results in altered application requirements which lead to an increased popularity of reactive programming patterns among large groups of computer scientists and in software libraries alike [2].

Using an example microservice application, this thesis examines the possible differences between reactive programming and conventional programming patterns and identifies whether one has certain advantages over the other.

Before diving into the details of the example application, the following sections will explain the essence of the buzzwords used in the title of this thesis.

1.1 Quick introduction to microservices

Similar to what the single responsibility principle [3, 4] is to software design, microservices are to software architecture. Fully-featured applications with large code bases violate the microservice pattern in the aspect that they manage a whole system in its entirety and when e.g. scaling the software up or down, the whole system is scaled. Microservices are about decoupling parts of the software and *tearing the monolith apart* (for example a microservice would be an application that only manages customer data but not bookings, offers or payments). This results in a number of loosely coupled standalone programs with the ability to be scaled up or down independently but requiring network communication to exchange data. Having a system split up into independent programs comes with the advantage of shorter release cycles, as services can be updated separately [5, 6].

While the code base of a software system created either as a monolith or as microservices might be of comparable size, the code base of a single microservice is a smaller fraction of the whole. This allows developers unfamiliar with the code to quickly grasp the gist and keep track of changes more easily [7].

Another key towards the intended autonomy of microservices is to make technology independent choices, especially regarding communication. If a service exposes a Java

RMI interface for communication, all services connected to this service need to be programmed for the JVM in order to be able to communicate. A popular approach for web based microservices is to use REST over HTTP, though a variety of other options, including message brokers or cross language RPC libraries, exist [8].

In an ideal environment, microservices can be programmed in any programming language the responsible team chooses to use, without having any effect on other services. Services could even be rewritten in an entirely different language as long as they stay compatible with the API of their preceding version [1, 8].

1.2 Reactive programming and the reactive manifesto

Reactive programming is often described by its four main characteristics: Responsiveness, resiliency, elasticity and an asynchronous, message driven design. These have been defined in the reactive manifesto [2] by a group around Jonas Bonér, a popular Swedish software developer and entrepreneur who is also the creator of the Akka [9] actor framework for the JVM [10, 11]. According to the manifesto and its authors, a system is responsive when it reacts in a reasonable time to its users. In case of failure, a system should still be available and react accordingly, which would define it as resilient. To be elastic, a system is required to react even under variable load conditions and finally, to fulfil asynchrony, a system needs to be message driven and loosely coupled [2].

Applied to microservices, these attributes mean, among other things, that services should not fail due to errors or downtimes of other services. This does not imply that errors should never occur in these applications. It rather demands applications to eliminate a single point of failure scenario by embracing errors and handling them in a way that has minimal impact on the users or other services in the system [12].

Besides the reactive manifesto, reactive programming is also often associated with streams. Typical examples are popular reactive libraries like ReactiveX [13] which implement the asynchronous, message driven design in a stream-like fashion. ReactiveX is a collection of open source reactive extension libraries for various programming languages. Publishers can push events into data streams while observers can subscribe to the values emitted. These reactive streams provide a vast amount of operators which can be used to transform the data stream [14]. An example that supports responsiveness is the timeout operator, which allows switching to an error handling procedure if a reactive stream (which is for instance providing a response from an API call) did not emit data within a specified time. Such error handling could then display an error message to the user or continue normally by fetching values from a cache, depending on the detailed situation.

A supporting indicator for the rise of reactive programming is the latest addition to the popular Java Spring Framework [15]. As of version 5 or Spring Boot 2, it adds and builds on its in-house reactive stream library Reactor [16].

1.3 Containerised environments

The aforementioned trend of building microservices in the cloud leads to the wish of scaling applications based on current loads and resource demands of those applications. Scaling applications dynamically allows to allocate resources like virtual machines when they are required and release those resources in times of lower loads. Thinking in an enterprise scale this can lead to lower expenses. Cloud service providers like Amazon Web Services or Google Cloud Platform even provide APIs or automations to dynamically allocate and delete virtual machines with the costs being calculated using an hourly usage rate [17, 18].

1.3.1 Containers

While microservices can be deployed directly to run on virtual machines (VMs) or dedicated servers, each service might require a different operating system with specific capabilities and tools installed to run smoothly. On top of this, virtual machines come with a full operating system that takes up space on the hard drives of the host machine and take seconds to start up [19]. A system consisting of hundreds or thousands of active microservice instances, where each service comes along with a full operating system, would waste a significant amount of storage resources just for the operating systems. Several services could be grouped into one virtual machine to save resources, but this would work against the principle of independent scaling and loose coupling.

In order to tackle this problem, developers can package their microservices into containers. Whereas hypervisors build an abstraction of the hardware for virtual machines, containers run on an abstraction of the hosts operating system, hence containers do not provide their own full operating system but can just contain the tools the packaged application requires to run. The result is a significantly smaller storage footprint of the individual services in comparison to deployments on top of VM images. Besides taking up less space, containers can start up rapidly, making them ideal candidates for fast and dynamic scaling scenarios [20].

Nowadays various container engines exist. However the most popular one supported by large enterprises such as Google, Amazon, IBM or Microsoft (to name but a few) is Docker [21]. Docker isolates containers completely from the underlying operating system. Container images are described in so called Dockerfiles and begin with a base image (every Docker image can be a base image for another image) followed by commands which either execute inside the container during its build process or e.g. copy local files like executable artifacts into the container. Each of the commands creates a new layer on top of the preceding commands. A container host, no matter how many copies of a specific container or similar images it runs, only needs to keep one copy of each layer in the file system to run a container [20, 22, 23]. An exemplary Dockerfile which can be used to deploy Java applications is shown in listing 1.1.

```
FROM openjdk:8-jdk-alpine
ADD ./inventory-service-1.0.0-SNAPSHOT.jar inventory-service.jar
ENV JAVA_OPTS=""
ENTRYPOINT exec java $JAVA_OPTS -jar /inventory-service.jar
```

Listing 1.1: A minimal example of a Dockerfile which starts with an image containing a Java development kit and copies a Java artifact into it.

1.3.2 Cluster management

Given a reasonable amount of container hosts, a microservice based system can be deployed completely using containers, allowing the precise and dynamic scaling of each individual service. However, in large scale applications the concept of containers arises new challenges:

- Microservices require a network to communicate.
- Applications need to be able to automatically find the services they are required to connect to.
- Requests to a specific application should be balanced between all containers inhibiting this application.
- Specific endpoints should be exposed to the internet.

- Not all containers should be accessible from outside of the internal application network.
- What happens to important files that might be written to the abstract file system inside those (potentially) short living containers?

To conveniently orchestrate thousands of containers, a cluster management software is mandatory. Docker provides a simple solution with Docker Swarm [24] and there are various other candidates like Rancher [25], which is a small and light weight orchestration software, but the most popular and most capable software for this purpose nowadays is Kubernetes [26, 27, 28].

Kubernetes is an open source cluster manager created by Google and is capable of scheduling containers across various servers (*nodes* in Kubernetes' terminology). Each container group consisting of one or more containers is called a *pod*. Pods can be grouped to so called *deployments*. Each microservice can have its own deployment and can be scaled by adjusting the number of replicas needed. Kubernetes schedules containers automatically across all available nodes in the cluster. To allow the applications to communicate, Kubernetes provides a *pod network*. This is a virtual network across all nodes where every pod gets its own IP address. By default it is not possible to connect to those pods from outside of the network. To make application discovery easier, deployments can be exposed via so called *services*. A service can act as a load balancer to all instances of a specific microservice. Services are identified by names and can be accessed using those names e.g. as part of a URL. Per configuration it is possible to expose a certain service to allow access from outside of the network. Furthermore Kubernetes can manage hardware resources such as storage. When a container in Kubernetes is destroyed (due to whichever good or bad reason), all files the container has created are removed, too. Containers are treated in a throwaway fashion to be replaceable at any time. Therefore, the application state needs to be stored externally [20, 26].

Eventually, there are scenarios where containers, such as databases or external caches, need to be able to write data to disk without losing it. In Kubernetes those containers can create a Persistent Volume Claim (PVC). PVCs are requests for storage which can be mounted into a specific path in the container. All files written to that path are persisted until the specific PVC is deleted. This way a container can be deleted and the PVC can be mounted into another container without losing data. There are several ways to provide the storage Kubernetes uses for the PVCs but this would exceed the scope of this quick introduction.

1.4 Hypotheses

The trend of dropping software monoliths in favour of containerised applications and microservices and the increasing popularity of reactive programming leads to the question, whether there is an overall benefit of using reactive programming in containerised microservice environments. This thesis will examine the features and drawbacks of reactive versus conventional programming using a containerised and microservice based online ticket shop as an example to prove or refute the following hypotheses:

- H1 Reactive programming has benefits over conventional programming patterns regarding performance (possibly under various loads).
- H2 Reactive programming changes (and possibly improves) the software development process.
- H3 Reactive programming improves the stability of a software.
- H4 Reactive programming improves error handling.

2 Reactive and conventional programming

Without actually performing any kind of testing, this chapter provides a theoretical analysis of reactive and conventional programming and identifies and explains, what conventional programming actually is. The four main pillars of reactive programming are responsiveness, resiliency, elasticity and asynchrony. These are not entirely new concepts and can partly be found even in applications not being advertised as reactive. They can be reactive to some extent simply because the application might be required to be fault tolerant or scalable and not because the developers were following the reactive manifesto. At least to some degree, all high availability systems are reactive. Though, they may not be designed with a "reactive first" approach. Taking this under consideration, being entirely anti-reactive would probably raise critical issues in the stability of a software system - especially in a system depending on the availability of other services.

So what exactly is the opposite of reactive programming? And what is conventional programming? Is conventional programming the opposite of reactive programming, or do they share some characteristics?

Beginning with the main features of reactive programming - responsiveness, resiliency, elasticity and asynchrony - the absolute opposite of reactive programming would be an unresponsive, rigid, inelastic and synchronous programming style. This would result in an application that responds at its own convenience, propagates errors of other services or even crashes as a result of those, and which is designed to handle a fixed maximum number of users and operates only synchronously and thus wastes resources by waiting for results to appear. This does not seem like a programming style any real world application with productive use can survive for a long time and it is probably safe to assume that conventional programming patterns are not exactly the obverse of reactive.

What distinguishes reactive programming from conventional programming then? While thoughtful conventional programming surely should cover reactive aspects like in-depth error handling or putting the reliability of an application into focus, this may be seen more as a necessity to make the existing application logic more stable. Contrariwise reactive programming embraces those approaches and forces the application logic to be built around these reactive principles. Also the reactive streams mentioned earlier (which are asynchronous by default) have a strong influence on the way an application is developed. While it may be fairly possible to develop a reactive application without using reactive streams, the streams do help to keep the code clean. Asynchronous code without reactive streams is often referred to as *callback hell* or the *pyramid of doom* [16, 12] in the context of chained callbacks. Generally,* it can be assumed that the extra focus on error handling, asynchrony and responsiveness adds a new layer of complexity to the code and could therefore be interpreted as harder to read or understand.

Therefore, the answer to the above question whether conventional programming is the opposite of reactive programming is no, as they do share characteristics. Of course the influence of the programmer cannot be neglected, but generally speaking conventional applications contain some degree of reactiveness.

What is special about reactive programming then? Reactive programming absolutely focuses and enforces the programmers to make their applications conform to the four tenets of reactive programming. Especially in containerised environments with an application being split up into many parts, these tenets are useful to ensure application stability in most situations. As microservices in general are asynchronous in some way due to the increased amount of communication, reactive programming and reactive streams natively fit into this scenario and provide developers with almost ready-to-use solutions for many common problems in distributed systems.

What significance does Java have in this scenario? According to a survey made in early 2017 [29], Java is the most popular programming language after JavaScript and SQL. There is no doubt that many large scale applications and microservices are developed in Java. An example is the video streaming platform Netflix, which uses Java to create many of its highly scalable microservices in a containerised environment and which is also an active developer of a lot of popular open source libraries used in microservice environments and a contributor to the JavaScript ReactiveX library RxJS [30, 31, 32]. On the one hand, this means that Java is already a popular choice for microservices in containerised environments, and on the other hand, suggests that Java also provides the required support for reactive programming. The amount of popular reactive libraries for Java can be seen as proof of this.

Considering this, it can be stated that conventional programming patterns already offer a subset of reactive features, but reactive programming, at least from a theoretical perspective, has the potential to offer improvements in comparison to conventional programming. Java, being in the main focus due to its popularity and large community, can be used to research the effects of reactive programming.

3 Specifying a scenario

As mentioned in chapter 1.4, the scenario being used for the comparison of the programming styles involves an online ticket shop. Such a shop (in the following also called *Ticketsystem*, which is the working title of the example application) sells tickets for events like concerts and can offer those tickets in various price classes per event. Users can visit the website, browse events and add tickets to a virtual shopping cart. The contents of a user's shopping cart can eventually be purchased.

A shop system can suffer under heavy loads once the ticket sales for an excessively popular event begin. Worst case consequences include users being able to add tickets to their shopping cart but once they attempt to purchase the selected tickets the event (or that particular kind of tickets) is sold out, inconsistencies in the underlying database, the site crashing and losing user data or even a total failure and complete unavailability of the shop application (signalised by the HTTP status code 503[1] or refused connections).

Those consequences are almost certainly not wanted and may lead to frustration among the users, effectively causing them to switch to a different shop, leading to financial losses. Desired reactions of such a system to higher loads are possibly reasonable response times or, in the case of errors or unavailabilities, appropriate error messages to help a user understand a problem and, if possible, advise them on how to solve it. An ideal system should not show any different behaviour under higher loads than usual.

3.1 Describing the motivation

Actually, the description depicts a real world scenario many users of such services have experienced, with some specific cases even being reported by newspapers. Famous examples are problems the ticket shop of the 2017 opened *Elbphilharmonie* concert hall in Hamburg experienced [34, 35]. Even though these are not the only cases of such outages, they gained popularity due to the hugely anticipated opening of the concert

[1]The HTTP status code 503 means *Service Unavailable* and is generally returned when a server is temporarily not available due to maintenance or overload [33].

hall and the highly demanded tickets. While it was able to identify that the shop is based on ASP.NET[2], it is not possible to tell more about the underlying software architecture.

It is possible that the choice of software architecture or programming patterns has an effect on the reaction of such a shop under higher loads. Detecting whether this is the case could help developers of web shop systems to improve their code to make their software more stable and resilient. Although this thesis specifically focuses on Java based microservices, it might be possible, but not guaranteed, that obtained results can be applied to similar scenarios.

3.2 Requirements

In order to be able to make relevant and objective comparisons, the Ticketsystem has to be programmed both conventionally and reactively. The goal is to make the two systems appear identically from the outside but differ internally. To achieve this, the core libraries and potential external services like databases should support reactive approaches natively. Only then, differences are not a result of different libraries but the direct result of a different programming approach. In order to be able to determine differences in performance, metrics need to be measured in the code.

Furthermore, the Ticketsystem needs to be programmed as a system of several microservices. In that way it is possible to detect whether one of the programming approaches has advantages over the other one regarding network communication. Those microservices need to be stateless, or an application state needs to be stored externally, to be able to scale them easily. Moreover, the microservices need to be packaged in Docker containers to be able to deploy them using a cluster manager.

Once the services are available as container images, those images need to be deployed to a Kubernetes instance. To perform reliable load and performance tests on a Kubernetes cluster, the required cluster needs to be dedicated to the Ticketsystem in a way that no other containers can use the resources and possibly distort the metrics.

3.3 Objectives

Using the deployed software and a suitable test setup, the primary goals are the following:

[2]This has been identified by checking the HTTP response headers received from the server.

- Detecting whether reactive programming will improve a software to decimate service outages and improve performance under heavy loads, if such a system is designed as a containerised microservice application.
- Detecting if and how reactive programming can change or even improve the way an application is programmed and whether it has any changing effect on reliability or error handling.
- Detecting whether reactive programming changes the developer's approach to software development.

A first version of a sufficient setup has already been presented in [36] in preparation for this work. That setup provides a base for modifications and improvements in the following scope of this thesis.

3.4 Defining boundaries

After describing the goals of this thesis, this section serves to distance the work from certain topics, which will not be discussed in detail or which are not going to be an objective of this work.

3.4.1 CQRS

Reactive programming is often mentioned in combination with Command Query Responsibility Segregation (CQRS) [37] and vice versa. While CQRS works well together with reactive principles, this work and the Ticketsystem will not make approaches towards implementing, describing or comparing CQRS with other patterns. CQRS is a useful pattern when many participants require and store different domain models of the same data structure. This is not the case in the Ticketsystem and thus CQRS will not be considered.

3.4.2 Appliance of results

The reactive versus conventional programming comparison will only be made under the circumstances mentioned. It is not a goal to make general assumptions about reactive or conventional programming which are valid for entirely different application scenarios. Additionally, the test results and insights of this work might depend heavily on the conglomerate of the used programming languages, frameworks, external tools and other technological choices. It cannot be guaranteed that the results will apply to a different environment.

4 Ticketsystem: An exemplary application

Chapter 3 has already given a brief introduction into the functionality of the Ticketsystem. The system developed in the scope of this work is a booking system for event tickets. It is possible to add tickets for events and several price categories into a shopping cart. Tickets in the shopping cart will be reserved for a certain time and can be purchased until the reservation expires. During the checkout process, users are prompted to enter their billing data. As the system is only used for educational and research purposes, it is not connected to any payment providers. Instead, payments are simulated by stretching the purchase time artificially. After a successful purchase the customer sees a list of ticket identifiers.

Systems used in production offer a variety of further functions like user account management, real payments, invoices or shipping of tickets via mail.

4.1 The architecture of the system

The Ticketsystem is divided into four microservices, each covering a certain functionality. As there is the need to manage an inventory and orders these are already two categories worth of a separate microservice. Additionally, there is the need to simulate payments and finally the necessity to provide a frontend to the customers. Figure 4.1 shows the four microservices and how they are connected to each other.

4.1.1 Inventory

The inventory service is responsible for managing the inventory. In this example of a ticket shop, the domain of the inventory contains events, tickets (with their corresponding available stock) and reservations, which expire automatically after a certain time. The service stores its data in an external MongoDB [38] database.

To request or modify data in the inventory, a REST API is exposed. The API can be used to create and query events including price classes and the available tickets per

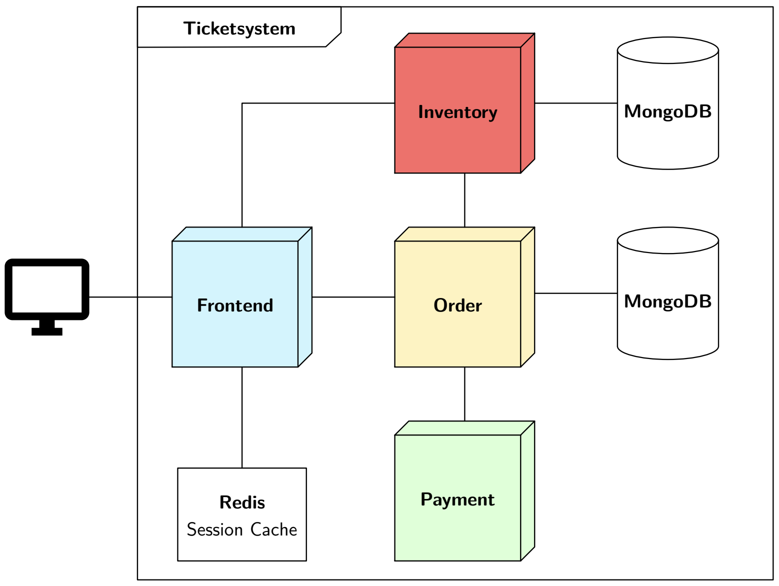


Figure 4.1: The Ticketsystem's architecture. It consists of four microservices. The inventory and the order service connect to a database and the frontend service connects to an external session cache.

price class. It also allows to create reservations. In that case, it removes the reserved tickets from the available stock and returns the reservation objects with information about the expiry date.

When a reservation expires, the reserved tickets are returned into the inventory and can be reserved or purchased again. While this service does not know of orders, it can mark tickets and reservations as sold, which allows to purchase the contents of a reservation. The inventory then flags the corresponding tickets as permanently sold. Expired reservations can not be purchased anymore and an error would be returned in such case. Sold tickets can also be invalidated, but this functionality is not currently used by the Ticketsystem.

It is not possible to reserve oder buy more tickets than there are available and the total availability count is calculated by subtracting the number of reserved and sold tickets from the total amount of tickets.

The inventory service does not depend on or connect to other services in the Ticketsystem, which makes it completely standalone.

4.1.2 Payment

As mentioned before, payments in the Ticketsystem are simulated to resemble an external payment system, like for instance EC, VISA or MasterCard. In order to do this, there is the payment service. It exposes a simple REST API which allows to make a payment. The API expects an object containing an amount to pay, customer related data and the type of card to perform the payment with. The card type has an influence on the duration of a payment request. Incoming requests are, if valid, delayed and return a confirmation message afterwards. There is no artificial error rate, so by design all payments succeed, if no unintentional problem occurs.

Similar to the inventory service this service does not make any requests to other services and additionally works without any external resources.

4.1.3 Order

The order service exists to manage shopping carts or create orders. It persists shopping carts based on a session ID which needs to be provided in requests. Whenever items are added into the shopping cart, the order service creates a reservation by making a request to the REST API of the inventory service. Afterwards, it stores the IDs of the reservations in the shopping cart and uses those to request further information from the inventory service.

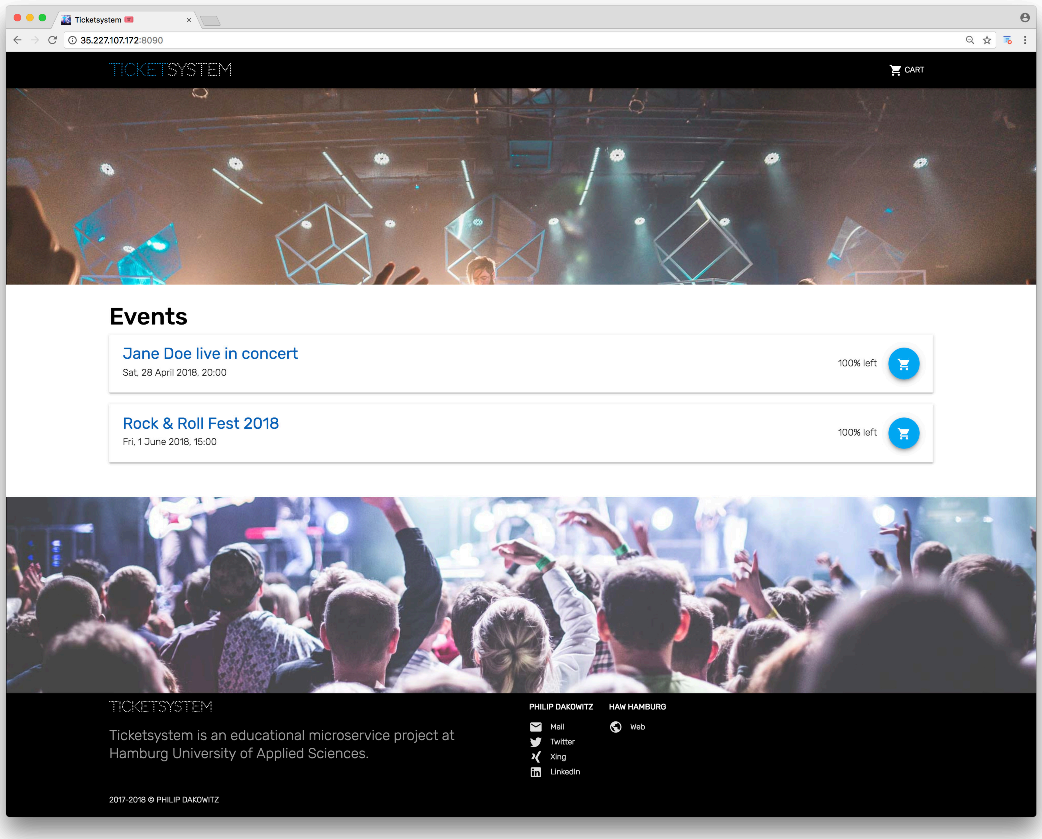
When purchasing the items in a shopping cart, the order service expects invoice and payment details. As soon as an order is placed, the order service makes a payment request to the payment service. When that request returns successfully, the order service notifies the inventory service of the purchase and instructs it to mark the reservation and corresponding tickets as sold. The inventory returns the generated ticket numbers to the order service and the order service wraps them in an order confirmation object and returns this to the caller. Whenever the payment request fails, the order stops processing and returns an error instead.

The order service uses an external MongoDB to store customer, order and shopping cart data and exposes a REST API to the other services. To successfully fulfil its function in the system it depends on the availability of the inventory and the payment service.

4.1.4 Frontend

End users visit a website to interact with the Ticketsystem. This website is hosted by the frontend service, which acts as a gateway into the Ticketsystem. The service also provides a backend which performs requests to the other services in the system.

The landing page of the frontend can be seen in figure 4.2. The user is confronted with a list of events with short information about their availability and the dates. A detailed page for each event allows the user to add tickets to their shopping cart.



Ticketsystem is an educational microservice project at Hamburg University of Applied Sciences.

Figure 4.2: The landing page of the Ticketsystem. It shows a list of events and provides brief information about the availability of the tickets for those events.

After adding items to the shopping cart, users are redirected to a detailed shopping cart page which shows an overview of all added items. From this page they are able to proceed to a checkout page which asks for invoice and payment data. After filling out and submitting the data, a list of ticket numbers is displayed as a confirmation of the purchase.

There is no database connected to the frontend service as it does not need to store any persistent data. All persistence is realised through the inventory and order services. The frontend communicates with those services, depending on the requests. Reading requests regarding event information are made directly via the inventory service's API, while writing requests, such as adding items to the shopping cart, are performed via the order service. To be able to share sessions between multiple instances of the frontend, the service connects to an external Redis cache, which is used to store the volatile sessions externally[39, 40].

The data flow through the system of a user visiting the website to order tickets is visualised in figure 4.3. It starts with the user opening the landing page of the Ticketsystem, which invokes requests from the frontend to the inventory and the order service to fetch the events and shopping cart data. Next, the user opens the detailed page of an event, which results in a request for event details to the inventory service. When tickets are added to the shopping cart, a request is sent to the order service, which then attempts to create a reservation in the inventory and, on success, returns the reservation. Finally, when placing an order, the frontend sends a request to the order service, which first contacts the payment service and secondly continues to mark the reservation as sold in the inventory, when the payment succeeded. In the end, a list of ticket numbers is returned to the frontend and displayed to the user.

4.2 Utilised technology

The technology stack used for the Ticketsystem builds on the Java ecosystem. This allowed to use the Spring [15] framework as a core dependency to build the microservices. Spring comes with many features likes dependency injection, built in web servers or support for extension libraries, which, for instance, create connections to databases and work like object relational mappers, providing repositories with CRUD like methods to save the developer from writing queries manually. A particular reason why Spring was chosen for this project was the recent focus on reactive programming in Spring 5 / Spring Boot 2. This allowed to program the services conventionally and reactively using the same library in different versions.

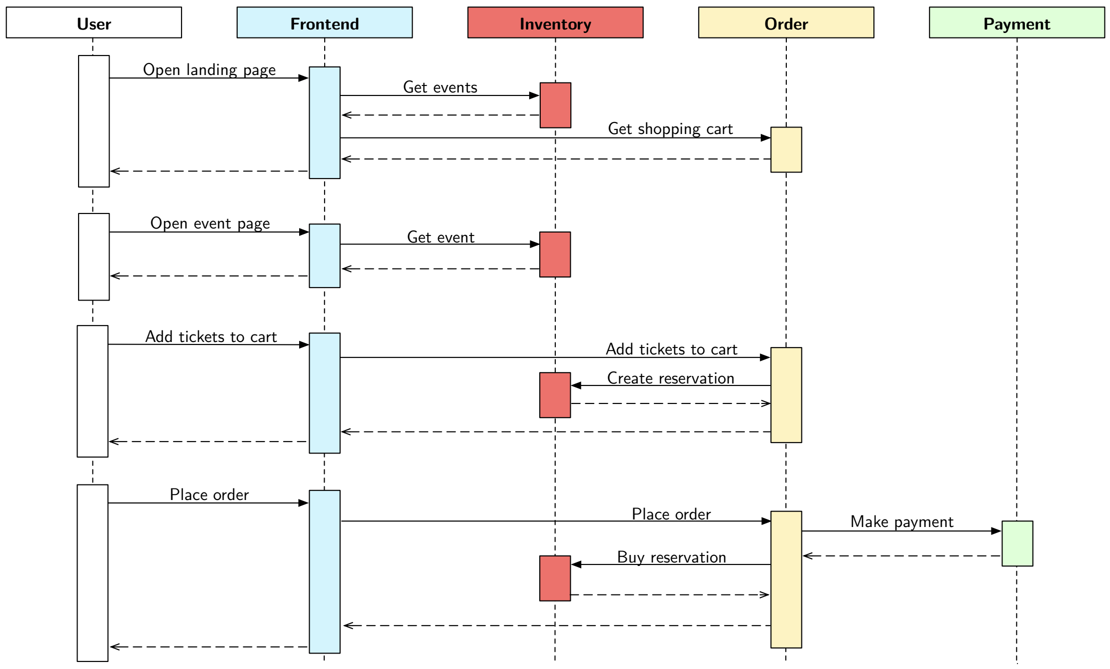


Figure 4.3: Sequence diagram of the requests in the Ticketsystem during a full order process. Some intermediate requests, like fetching the shopping cart again, after adding items to it, are not displayed in favour of readability.

All microservices in the Ticketsystem make use of the Spring Web module to create a REST API. Communication between the services happens only via those APIs and is built entirely with Spring-provided libraries and tools.

With the built-in web server, the frontend service can host the website's content naturally by including the files in the resources of the application. The web application itself is built with Angular [41] using TypeScript, SCSS and HTML and fetches data from the backend through AJAX requests to the frontend's REST API. To allow multiple instances of the frontend to share the users' sessions, Spring can connect to an external Redis installation, which is used as a session cache.

As mentioned before, the inventory and the order service use an external MongoDB to store data. This choice was made because there is, in addition to a regular driver, a reactive MongoDB driver for the JVM [42] and Spring supports MongoDB through the Spring Data MongoDB modules. Furthermore, MongoDB can be easily configured to run in a cluster/replica set, which makes it a good candidate for containerised deployments.

To realise deployments to a Kubernetes cluster, all microservices have been wrapped inside Docker containers.

4.3 The conventional way

While the utilised frameworks are similar in the two versions of the software, some key parts are handled and implemented differently.

In order to make requests to other services, a *feign client* is used in the conventionally programmed version. In Spring this is simply an annotated Java interface specifying methods with the desired parameters and return types. On application startup, Spring instantiates this interface and translates the method calls into HTTP requests. A short example can be seen in listing 4.1.

```java
@FeignClient(url = "http://inventory:8080")
public interface InventoryClient {
  @GetMapping("/api/events")
  ReponseEntity<List<Event>> getEvents();
}
```

Listing 4.1: An example of the Spring feign client.

When `getEvents()` is called (an example can be seen in listing 4.2), Spring performs a GET request to the URL `http://inventory:8080/api/events`. The result is automatically converted into a list of objects with the type `Event` and returned to the caller.

In general, the code style of the conventionally programmed variant of the microservices resembles typical object oriented Java code with, for instance, synchronous methods returning objects or exception handling using try-catch clauses.

```java
public List<Event> getEvents() {
  try {
    ResponseEntity<List<Event>> response = client.getEvents();
    if (response.statusCode().isError()) {
      throw new TicketsystemException();
    }
    return response.getBody();
  } catch(IOException e) {
    handleIOException(e);
  }
}
```

Listing 4.2: A method to fetch events programmed conventionally and object oriented using a try-catch clause to handle exceptions.

4.4 The reactive way

Contrariwise, the code of the reactive Ticketsystem focuses on the usage of reactive streams. That code may be harder to read, as it requires to re-think the control flow as a pipeline that is fed with data and processes it at various stages. A comparison can be made between listings 4.2 and 4.3. Both show a simplified method fetching events from a client with basic error handling. Comparing the code, the conventional way may look more familiar. While the logic in these examples is fairly simple, it is already possible to find various differences. The first distinction can be seen in the return type of the method. Where the conventional approach returns a `List<Event>`, the reactive method returns a `Flux<Event>`. Flux is the data type of reactive streams in Spring, originating in the Reactor [16] library. Reactor differentiates between a reactive stream that can contain multiple elements, like in this example, and a stream, which can contain only one element. A method like `getEventById(Long id)`, which only returns one event, would rather have the return type `Mono<Event>`.

```java
public Flux<Event> getEvents() {
  return client.getEvents()
    .doOnNext(response -> {
      if (response.statusCode().isError()) {
        throw new TicketsystemException();
      }
    ).flatMapMany(response -> response.bodyToFlux(Event.class))
    .doOnError(IOException.class, e -> handleIOException(e));
}
```

Listing 4.3: A method to fetch events using a reactive stream with built in error handling operators.

```java
getEvents()
  .filter(event -> event.hasTicketsAvailable())
  .subscribe(event -> {
    // process each event individually
  });
```

Listing 4.4: Subscribing to a reactive stream in order to process the values.

While the return statement in the conventional approach is made at the end of the try block, the reactive method starts with the return statement. That is, because the reactive stream operators do not execute immediately at the point the `getEvents()` method is called. Instead, the code is only executed once, at some point, the reactive stream is subscribed to. Beginning with reactive programming, especially this could be a common pitfall. An example showing how a subscription to a reactive stream looks like is shown in listing 4.4. It can be seen that each event is processed individually by the handler.

Another difference in the reactive Ticketsystem code is the way HTTP calls to other services are made. The reactive version of Spring comes with a web client library, which allows to make HTTP requests and get the results natively as reactive streams. This allows to easily develop client classes with custom error handling and configurations, hence providing an essential feature for reactive programming. Listing 4.5 shows the `getEvents()` method analogous to the method presented in listing 4.1, but with reactive approaches.

In the example, the client is initialised with a base URL, similarly to the annotation of the feign client. Usually this happens in the constructor of a custom client class. Requests made by the client return a `Mono<ClientResponse>` and the developer has to

```
 1 WebClient client = WebClient.create("http://inventory:8080");
 2
 3 public Mono<ClientResponse> getEvents() {
 4   return RxJava2Adapter.singleToMono(
 5     Single.fromPublisher(client.get().uri("/api/events")
 6       .accept(MediaType.APPLICATION_JSON).exchange()
 7       .timeout(Duration.ofSeconds(TIMEOUT_SECONDS))
 8       .onErrorMap(/* transform some exceptions */)
 9       .doOnNext(/* check response status; throw exception if 503 */))
10     .lift(CircuitBreakerOperator.of(circuitBreaker)));
11 }
```

Listing 4.5: An example usage of the Spring WebClient making use of timeouts and a circuit breaker. The initialisation of the client usually takes place in the constructor of a custom client class.

take care of further type conversions using methods like `bodyToFlux()` (see line 7 in listing 4.3). In this example, the Reactor stream needed to be converted to a RxJava2 stream and back at the end. This was required to be able to use the Resilience4j library [43], which offers a circuit breaker operator for reactive streams[1]. If a certain error rate occurs at requests to the inventory service, the circuit breaker will *open the circuit* and let subsequent requests fail immediately. After waiting for a specific time, the circuit breaker allows some calls through to the inventory, to check whether the error rate is decimated and closes the circuit if the error rate has improved [44]. This way, a defective inventory is not flooded with requests and can conceivably recover better from its faulty state.

To stay responsive in a reasonable time, calls to external services are guarded by using a timeout operator (see line 7 in listing 4.5). If a HTTP request does not return a result after the specified amount of time, a timeout exception will be thrown. As the circuit breaker is engaged at a later point in the stream, timeouts also affect the state of the circuit.

The design of reactive streams allows the circuit breaker in an open state to let the stream fail without doing any previous calculations. Upon subscription to the stream, all operators receive a notification. If the circuit is open at this point, the circuit breaker will fail immediately and no other processing needs to take place in the stream.

[1]The Resilience4j library currently only supports RxJava2 streams. Reactor is fully compatible with RxJava2 streams but needs to be converted using the provided RxJava2Adapter.

4.5 Measuring metrics

In order to be able to determine differences in performance between the two implementations, the durations of method calls and network requests needs to be recorded and stored. As suggested in [36], the Dropwizard Metrics library can be used to perform the desired measurements in Java based applications [45]. It allows to create timers, counters, gauges and other monitoring objects which can be used to capture the state of the application. All metrics are stored in a registry, which can be extended with reporter objects periodically processing the metrics and for instance transferring them into a message queue, writing the metrics to a file or printing them out to the console for further analysis. The Dropwizard Metrics library is widely used by many popular projects like Spring, Akka or many open source Apache projects [46]. Listings 4.6 and 4.7 show how the library can be used to create timers.

```java
// a
Context ctx = timer.time();
expensiveMethod();
ctx.stop();

// b
try(Context ctx = timer.time()) {
  expensiveMethod();
}
```

Listing 4.6: Using timers in the conventional Ticketsystem.

```java
// a
Mono.fromCallable(timer::time)
  .map(timer -> expensiveStream()
    .doOnTerminate(timer::stop))

// b
Mono.fromCallable(timer::time)
  .zipWith(expensiveStream())
  .doOnSuccess(tuple -> {
    tuple.getT1().stop();
  });

// c (don't do this)
Context ctx = timer.time();
expensiveStream();
ctx.stop();
```

Listing 4.7: Using timers in the reactive Ticketsystem.

In the conventional Ticketsystem the timers can be used by calling the `time()` method of the timer, then making the calls to be measured and afterwards calling the `stop()` method of the context that was returned by `time()` (example *a*). An even more convenient and less error-prone approach is to call `time()` in a try-with-resources statement (example *b*). The timer implements the `AutoClosable` interface and therefore stops the measurement automatically once the try block finished executing.

This technique will not work in an environment with reactive streams though, because the composition of streams does not call the actual stream chain. Timers like example *c* in listing 4.7 will measure the time it took to create the stream instead of the time the stream needed to execute. It is necessary to convey the timer's context through the reactive stream itself and e.g. either zip it together with other streams (example *b*) or call `stop()` from inside of the `doOnComplete()` method of a stream (example *a*).

Example *b* will only take times on successful executions of the stream. Errors will not be considered in the calculations, as the timer is only stopped in the success callback.

In the Ticketsystem timers are the most prominently used instruments of the metrics library. They are used to measure the execution time of calls to the databases, to other services and to record the time of some internal method calls.

5 Test architecture

The Ticketsystem, albeit being as far production ready as it needs to be, requires an extended architecture to fulfil its purpose as a test project. So far, all directly required services like MongoDB or Redis have been introduced in chapter 4.2, as these are critical for the general functionality of the Ticketsystem. To monitor the whole system and collect the metrics from each running instance, a setup to process and store these metrics is required. The desired architecture consists of a database, a tool for collecting the metrics and a visualisation tool which allows to analyse the metrics.

5.1 Challenges

In a containerised environment, the metrics need to be collected from all instances of each service. There are many approaches on how to do this. One approach is to write metrics to a file and let another service constantly collect the files to append them into a database. Another approach might be to create a REST API, which exposes the metrics, periodically poll it and store the results. Finally, an approach might be to not let an external system take care of collecting the metrics but instead let the services themselves take care of delivering the metrics. The latter idea is probably the most viable in a containerised setup, because the Ticketsystem services will only need to know the one service to send the metrics to. In contrast, the other two methods would require a service to keep track of all currently running Ticketsystem services and to periodically connect to the individual containers to collect the metrics.

The collected metrics need to be visualised (or at least processed) to allow further analysis. A desired result would be an overview of various graphs indicating the mean value of all taken metrics across all running instances of a service in close to real time.

Finally the metrics architecture requires a database which is capable of storing and processing the data and can be connected to the visualisation tool and to the Ticketsystem services.

5.2 Collecting the metrics

To deliver the metrics from each running instance of a service to the database, the Ticketsystem services connect to a RabbitMQ message broker, which then processes and forwards the metrics [47]. This way the services only need to dispatch the metrics and do not have to take care of the actual process of connecting to a database or storing the values there appropriately. Furthermore this allows to pre-process the metric reports. The metrics are sent to RabbitMQ as JSON objects. In order to serialise all metrics periodically, a custom reporter (see chapter 4.5) had to be implemented, as the metrics library itself does not provide a simple JSON reporter. There are community projects with reporters dispatching metrics directly to RabbitMQ. However, these are poorly maintained, thus the custom solution turned out to be more feasible. The connection to RabbitMQ is handled manually with a Spring extension. In order to be able to distinguish single instances of the services, the metric reports are enriched with custom values such as an instance identifier. In a Kubernetes environment this would be the pod's ID.

From the RabbitMQ the metrics are collected by a Logstash instance [48]. Logstash is a processing pipeline tool that collects the reports, applies some transformations in preparation for the visualisation step and stores the data in Elasticsearch, a search database built on top of Apache Lucene [49, 50].

5.3 Visualisation and plotting

To achieve the desired visualisation of the measured data, a Kibana [51] instance was set up to connect to Elasticsearch. Logstash, Elasticsearch and Kibana, all made by the same manufacturer, are designed to work together as the so called *ELK stack* and can be used to visualise various data ranging from log files to metrics.

Kibana allows to create visualisations based on the data stored in Elasticsearch. To monitor the Ticketsystem, a variety of visualisations has been created, such as graphs showing the mean time of database calls, the mean response time of requests to other services or the mean internal processing time of users' calls to the frontend. To monitor multiple visualisations at the same time, Kibana allows to create dashboards consisting of a number of visualisations. Besides the possibility to visualise the data itself, Kibana allows to download .csv files of the visualisations' data.

Figure 5.1 shows the complete setup required to collect, process, store and visualise the metrics. Additionally, the graphic shows a service called Curator. In a Kubernetes setup, Curator can be started as a daily cron job to delete Elasticsearch indices older than a specified amount of time (with even more complex rules possible). This can be used to prevent excessive storage usage by Elasticsearch.

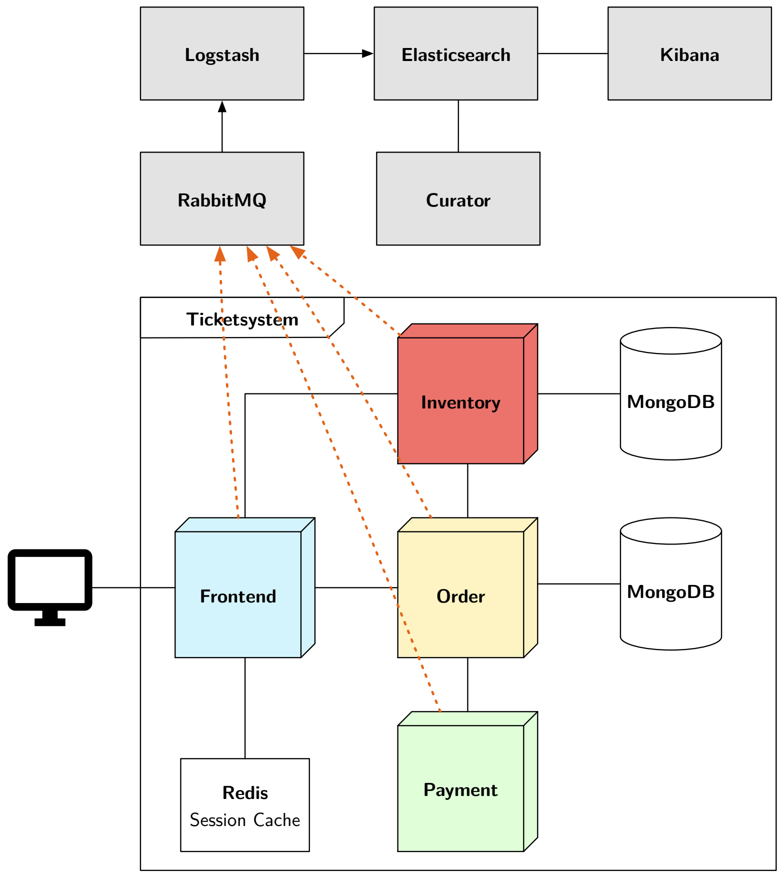


Figure 5.1: The full architecture of the Ticketsystem, including the external services required to collect, store and visualise the metrics.

Anecdote: While implementing the Ticketsystem and verifying the whole test setup, a bug in the metrics library continually led to broken indices in Elasticsearch. Whenever, for a specific time, no update to a specific metric was recorded, the exponentially decaying weight of past values led to a division by zero when calculating the current mean value of a timer. Instead of resulting in an exception, this led to the string "NaN" being serialised in the metrics report as the mean value for that timer, which led to Elasticsearch preventing this value to be treated as a number and thus, forbade Kibana to visualise it. As this constantly forced maintenance of the Elasticsearch indices, I chose to contribute to the metrics library and created a pull request on GitHub[1] with a fix for this bug which, fortunately, was merged and released a few days later.

[1]Pull request #1230: `https://github.com/dropwizard/metrics/pull/1230`

6 Shaping the tests

In [36] sample tests have already been made to prove the feasibility of the test setup. Those tests were performed using an early version of the conventional Ticketsystem in a simple setup, with each service running in one instance. The tests simulate users' actions. Specifically the tests are designed to imitate the users' activity when visiting the ticket shop to buy tickets for an event. As part of this work the tests have been modified and extended to make a viable comparison of the Ticketsystem's versions. The following sections introduce the used tools, test cases and provisions made to ensure verifiable and reproducible results.

6.1 Tools

Apart from the test setup presented in chapter 5, there was the need for a tool that simulates users' interactions with the website. For this, Gatling was used [52]. Gatling is a load testing tool which executes tests written in Scala and generates charts for each test run. It measures the time each HTTP call takes alongside several meta information like returned status codes or redirects that are made. In the test cases can be specified how many users to simulate and whether parts of the users should be simulated immediately on start or to defer some users over time and thus ramp up the load on the software.

6.2 Test scenarios

The Ticketsystem was subject to two separate test scenarios: The first scenario was used to find differences in the behaviour of the software under various load conditions in changing scaling setups. The second scenario analysed the software's behaviour in the case of an outage of one of the services to see whether one variant of the software handles such a situation better than the other. As reactive programming promises to show better behaviour in such cases, an expectation was that the reactive software shows better results.

6.2.1 Load tests

In order to determine differences in the behaviour of the software and to find the roots of potential differences or bottlenecks, a test scenario which the system underwent in various setups was designed. Table 6.1 gives an overview of all specified test cases in this scenario.

	Case 1	**Case 2**	**Case 3**	**Case 4**
Instances per service	1	5	5	1
Nodes per database	5	5	1	1
Users (ramped over 5 minutes)	3000	3000	3000	3000

Table 6.1: Test cases for the load tests that both implementations of the Ticketsystem underwent.

Four tests were done. Each test case was performed on the conventional and on the reactive Ticketsystem. This resulted in eight test runs in total. The first test case was performed with only one instance of the Ticketsystem services running. The inventory and the order service each connected to a dedicated MongoDB replica set consisting of five nodes. The second test case was run with the Ticketsystem services scaled up to five instances each. Cases three and four repeated the previous test cases but with only one MongoDB node running per database.

The load generated on the system consisted of 3000 users being ramped up linearly over a time of five minutes. The test then ran as long as the virtual users had not finished buying their tickets. The following list describes the single steps one virtual user performed during the test run, each separated by a short pause:

1. Opening the Ticketsystem's main page (the shopping cart and a list of all events are loaded).
2. Opening the detailed page for a specific event.
3. Adding one ticket to the shopping cart.
4. Opening the shopping cart and proceeding to the checkout page.
5. Placing the order.

6.2.2 Resiliency tests

The second test scenario focused on the reactive principles, mainly on resiliency. This test case was similarly programmed to the first one (see table 6.2), as there were also users, 500 in this case, attempting to buy tickets. The requests the users made are equal to those in the first test. However, it was different in terms of error handling. When a request failed for one user, this particular virtual user tried to perform the same request again after a short pause. Real world users would probably also try to perform an action again, if an application indicated failure, so this adds a simple layer of reality.

All Ticketsystem services had one single instance running. After one minute the inventory service was restarted on purpose, causing a short outage of the inventory and errors in the frontend and the order service, as those are depending on the inventory to reliably serve all data. As the tests were executed in a Kubernetes environment, the inventory service pod was simply deleted after a minute. Kubernetes automatically recreated the missing pod because the deployment of it specifies a minimum number of one instance to be available at all times. After the replacement container successfully started up, the Ticketsystem should quickly recover and return to a stable state where it can serve all requests without errors.

As in the first scenario this test case has been performed on both the conventional and the reactive Ticketsystem. All the services are scaled to one while the database nodes are scaled to five. Having the services scaled up to more than one instance would only add complexity to the test, because the deletion of one pod would not result in an outage of the inventory as there would be other instances handling the requests.

Instances per service	1
Nodes per database	5
Users (ramped over 2 minutes)	500
Time to failure	60 seconds

Table 6.2: Test cases for the resiliency tests that both implementations of the Ticketsystem underwent.

6.3 Verification and reproducibility

Creating a test scenario with verifiable and reproducible results was one of the goals this thesis aimed for. In order to be able to achieve this goal it was important to eliminate all potential instabilities during the test execution which could lead to biased results. The simplest way to ensure this was to perform several test runs for every scenario and every test variant with the expectation to get the same results each time. More importantly though was to ensure that the tests run in a static environment that does not change and is known entirely. In difference to the initial tests made in [36], where the payment method (and thus response time of the payment service) was randomised between the three possibilities, it was chosen to always use the same payment method to ensure the tests being reproducible and reduce randomisation effects. The same applies to the duration of the short pauses between each request, which had also been random earlier, but was changed to static values before the tests as part of this work were made. Keeping random values would indeed provide a layer of reality to the tests, but would lead to a less reproducible outcome.

6.3.1 Resources

Initially, it was planned to perform the tests in a Kubernetes cluster hosted by the department of computer science at the Hamburg University of Applied Sciences (HAW). Yet, as the nodes are not identical and the cluster can be accessed by every member of the department, the load would have been unpredictable and it would have been complicated to produce comparable and reproducible results. As a result of this, it was chosen to split the involved components into two parts: One part consisting of the applications required to store and visualise the measured data and not required to have a stable performance (Kibana, Elasticsearch and Curator) and the other part being the Ticketsystem services, databases and the directly related services to measure and transport the metrics.

The first part was chosen to remain hosted in the Kubernetes cluster of the department and expose Elasticsearch to be able to connect to it from outside of the cluster. The second part was moved into a dedicated Kubernetes cluster hosted by the Google Kubernetes Engine (GKE) on the Google Cloud Platform. As the GKE cluster was only occupied by the Ticketsystem services, RabbitMQ and Logstash, no other running applications could influence the performance of the machines. The cluster consisted of four nodes of the GKE instance type *n1-highmem-2*, which were controlled by a Google managed Kubernetes master server. Table 6.3 gives a more detailed technical specification of the deployed nodes and figure 6.1 gives a brief overview of the deployments in the GKE cluster. As the pods in a Kubernetes cluster are randomly distributed across all nodes, the figure only contains placeholder pods. In the actual setup, these placeholders inhibit all services deployed in the GKE cluster.

GKE instance type	n1-highmem-2
CPU	2x Intel(R) Xeon(R) CPU @ 2.30GHz
Memory	13 GB
Datacenter location / Zone	us-east1-b
Kubernetes	v1.8.6-gke.0
OS	Container-Optimized OS from Google

Table 6.3: Technical specification of a Kubernetes node in the cluster hosted at Google.

Apart from having a dedicated Kubernetes cluster it was a beneficial choice to perform the tests there, as Google has very performant load balancers which quickly distributed the requests equally across all running instances of the Ticketsystem and which was crucial for successful tests. The initial experiments on the cluster at the HAW had a bottleneck in the load balancer, which was not capable of serving the same amount of requests in parallel as the Google load balancer.

In this test setup, Logstash connected to the Elasticsearch cluster hosted in the HAW Kubernetes cluster, where the metrics were stored. This setup was able to provide recreatable, substantial results.

To ensure Gatling could always run with the same amount of resources, the tests have always been performed on the same machine. The specs of the machine used to run the Gatling tests are listed in table 6.4.

CPU	Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
Memory	40 GB
OS	macOS High Sierra 10.13.3

Table 6.4: Technical specification of the computer used to run the Gatling tests.

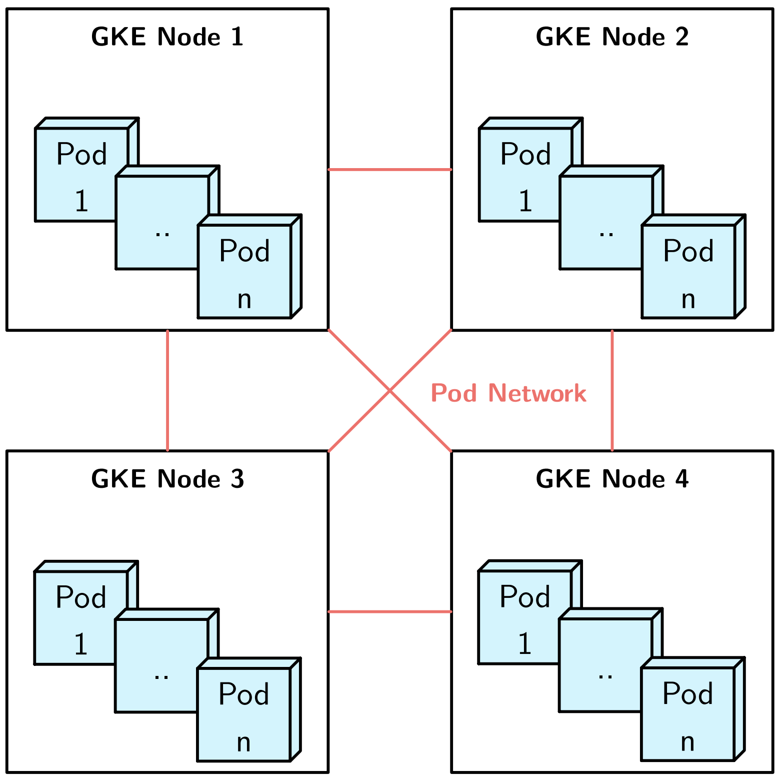


Figure 6.1: Deployments in the Kubernetes cluster hosted at Google. The connections between the nodes indicate the pod network while the placeholder pods in each node represent the randomly distributed deployments.

6.3.2 Approach

Every single test run consisted of several manual steps which needed to be done in order to always have the same starting position. The following steps describe the approach used to perform the tests.

1. Scaling the services according to the test case. Old containers are replaced.
2. Dropping the databases.
3. Creating demo data in the database.
4. Manually connecting to the Ticketsystem and performing some requests, to initialise the services and make sure everything is ready.
5. Configuring Gatling to connect to the correct URL (The GKE cluster exposed two IP addresses, one for the reactive and one for the conventional Ticketsystem).
6. Running the tests.
7. In case of resiliency tests, stopping the inventory after 60 seconds.
8. Exporting the metrics data from Kibana graphs as .csv files.

7 Test results and discussion

As part of this thesis over 30 test runs have been made in various constellations and scenarios. This chapter is dedicated to present the results of the tests performed with Gatling. At first, the results will be presented individually. At the end of this chapter, a comparison of the conventional and reactive parts will follow, as well as a discussion of the results. All results can be split up into two different views: The customer's view and the system's view. Results acquired with Gatling represent the view of a customer while the metrics recorded inside the Ticketsystem are internal metrics representing the system's view.

7.1 Load tests

The tests were performed as described in chapter 6.2.1. All four test cases were repeated multiple times on each version of the Ticketsystem to ensure the viability of the test results. As reactive programming is highly praised, an expectation was for the reactive part to perform better under load. Another expectation was higher scaled scenarios to perform better than the single-node ones. The following sections show the results that the different approaches scored in the load tests.

7.1.1 Conventional

In each conventional test case the test duration was 327 seconds. In this time 3000 simulated users made 27.000 requests with an average of 82,5 requests per second and mostly around 277 users being active in parallel. Figure 7.1 gives an overview of the responses per second and the active users in the second test case (with five instances per service and database) and shows that there was few variation during the whole test execution. Almost all requests were responded to in under 800 ms with peaks that went up to 2685 ms. The response time percentiles over time of case two are visualised in figure 7.2 and show the short peak which is the reason for the 2685 ms maximum response time. Across all four conventional test cases 108.000 requests have been made with 71 requests being answered between 800 ms and 1200 ms and 65 requests that took longer than 1200 ms to answer. None of the requests failed. In general, the results between the four conventional test cases are relatively similar with only subtile differences, while the pairs of results one and four as well as two and three show the most similarities.

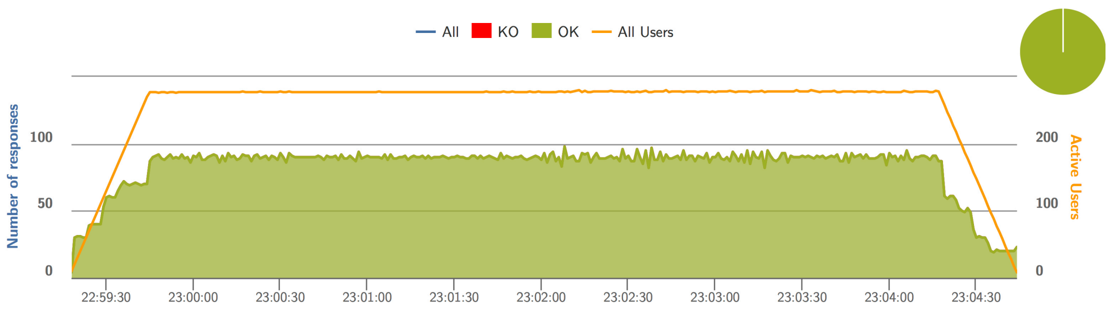


Figure 7.1: Responses per second in the second conventional load test case. The orange graph indicates the number of active users while the green area shows the number of responses per second during the test execution.

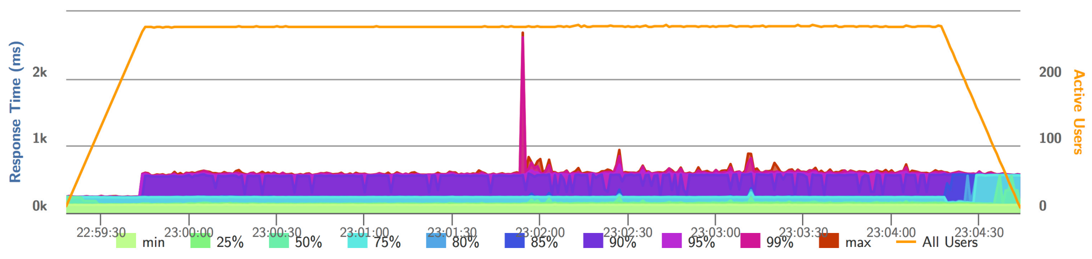


Figure 7.2: Response time percentiles over time in the second conventional test case. The orange graph indicates the number of active users while the coloured areas show the percentiles.

The mean response time ranges from 206 ms to 215 ms depending on the test case. As expected, the fourth test case with only one instance per service and database had the highest response times with a mean response time of 215 ms. The test case with the best mean response time was the third one with 206 ms, which makes it 4 percent faster than the slowest. The mean response time is not the only interesting metric, though.

As shown in table 7.1, the two test cases with five instances of the Ticketsystem services running, have fewer long-running requests than the other cases, with the second scenario with five database nodes leading as it only had nine requests which performed slower than 800 ms. Interestingly, the third test case with only one database node performed slightly better in general which might be a glimpse at the scaling performance of MongoDB. This suspicion is further substantiated by comparing cases one and four. Case one is only marginally better but has more than twice the number of requests running longer than 1200 ms in comparison to case four. Another interesting result is that, even though the fourth test case overall had the poorest results, it has the lowest maximum request time.

	Case 1	**Case 2**	**Case 3**	**Case 4**
Mean (ms)	214	209	206	215
Min (ms)	117	116	115	116
Max (ms)	2017	2685	2681	1726
50th percentile (ms)	147	145	140	148
75th percentile (ms)	235	235	235	237
95th percentile (ms)	576	573	559	572
99th percentile (ms)	640	602	583	626
800 ms < t < 1200 ms	24	7	8	32
1200 ms < t	34	2	14	15

Table 7.1: Customer's view of the load test results on the conventional Ticketsystem generated by Gatling. The table shows that test case 4 performs worse than the other test cases whereas case 3 performs better than the rest.

When comparing the test cases grouped by the scaling of the Ticketsystem services even further, impressing similarities can be discovered. Despite the small differences in response times, most of the visualisations look almost identical as to the point that case three also has a peak in response times at the same time after start as case two (compare figures 7.2 and 7.3). Test cases one and four also share response time peaks after a certain duration.

These similarities are also present when viewing the metrics from the system's point of view. Figures 7.4 to 7.7 show that the peak visible in the response time percentile graph created by Gatling can also be seen from the system's point of view in figure 7.2. The origin of this peak can be traced back to the inventory service's database connection (see figure 7.7), as all other operations visible in the figures depend on the

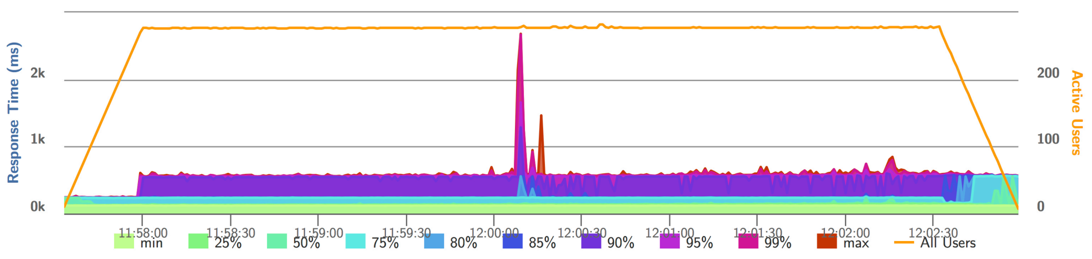


Figure 7.3: Response time percentiles over time in the third conventional test case. The orange line indicates the number of active users while the coloured areas show the percentiles.

inventory and its database. While the mean response time of the database is only increased by around 250 to 400 microseconds (see figure 7.7), the mean response time of the inventory's writing operations increases by 5 to 15 milliseconds, as well as the response times of the order service and the frontend.

A similar effect can be seen when comparing figures 7.8 and 7.9, which display the response times of the inventory service and those of its database calls in the fourth test case. While the database's response time increases by 4 to 6 milliseconds, the total response time of the inventory peaks from around 28 ms to 78 ms and only later in the test returns to its previous response time. It should be mentioned again that in both test cases this mostly affects the response times of writing operations. The reading operations from the database remain almost unaffected.

The database's response time in the fourth case (and similar in the first case) surges up by 4 to 6 milliseconds while in the second case the amount is just a fraction of it. This can probably be explained with the reduced number of available database connections. The second and third case consisted of five inventory instances with each a certain size of the database connection pool. As the first and fourth case only had one inventory instance, all incoming connections needed to share the connection pool of one inventory service instance. This resulted in longer times the requests had to wait before having been able to perform a query on the database.

By comparing the frontend's response times in figure 7.4, which are all between 5 and 35 milliseconds, with the response times Gatling measured, where the mean response time is 209 milliseconds, it is possible to draw conclusions about the transport duration, which takes a significant toll on the requests' durations. Pinging the IP address of the conventional Ticketsystem resulted in 114 ms.

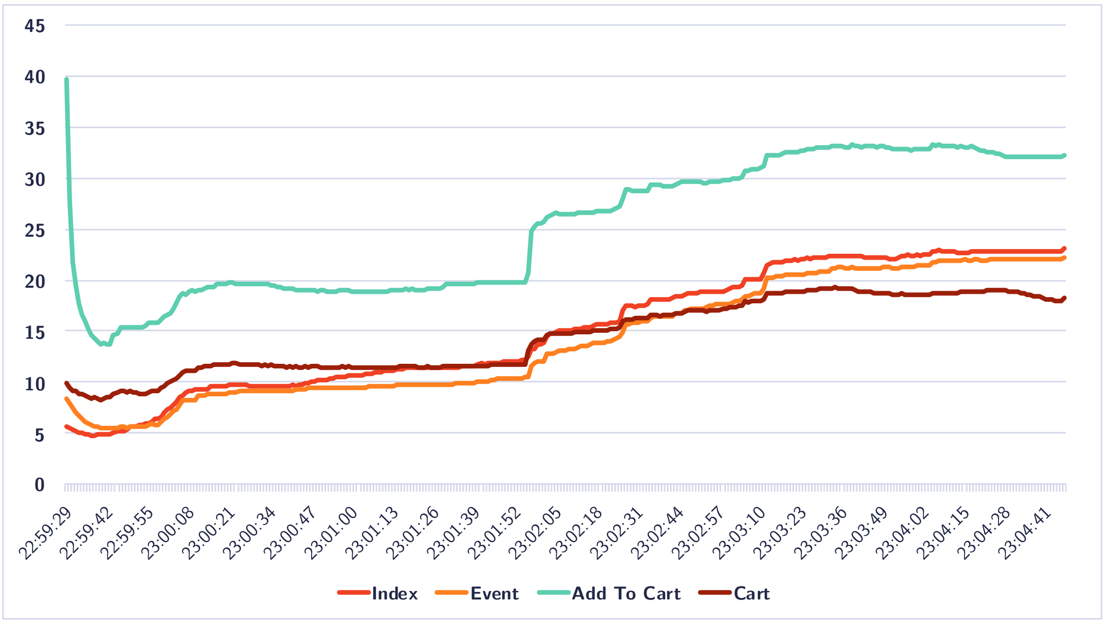


Figure 7.4: Response times in milliseconds of methods in the frontend service during the second load test case on the conventional system.

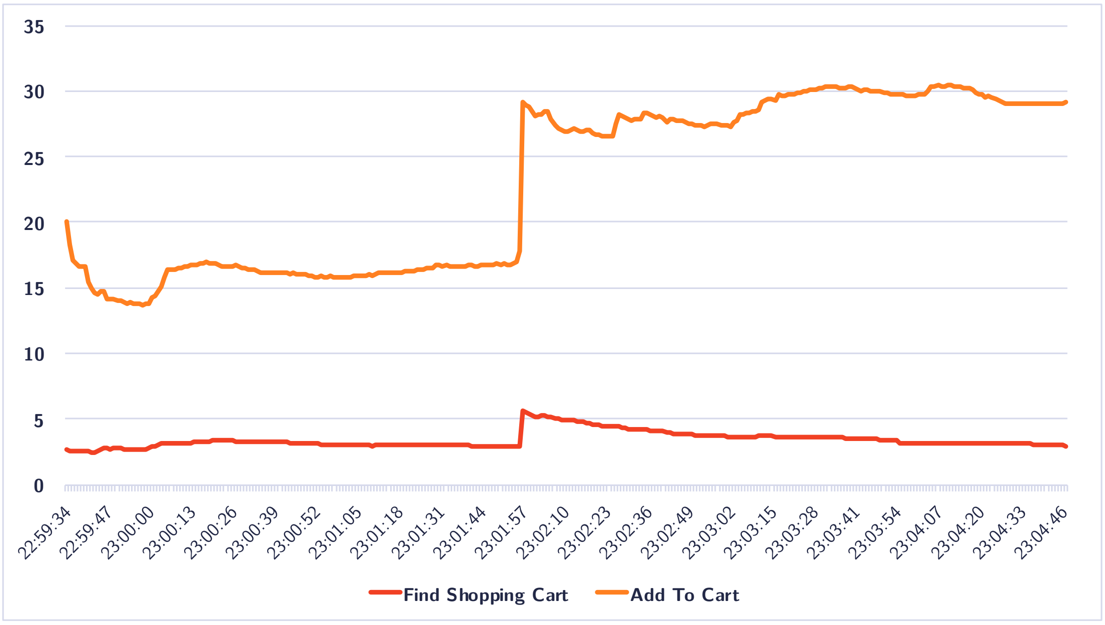


Figure 7.5: Response times in milliseconds of methods in the order service during the second load test case on the conventional system.

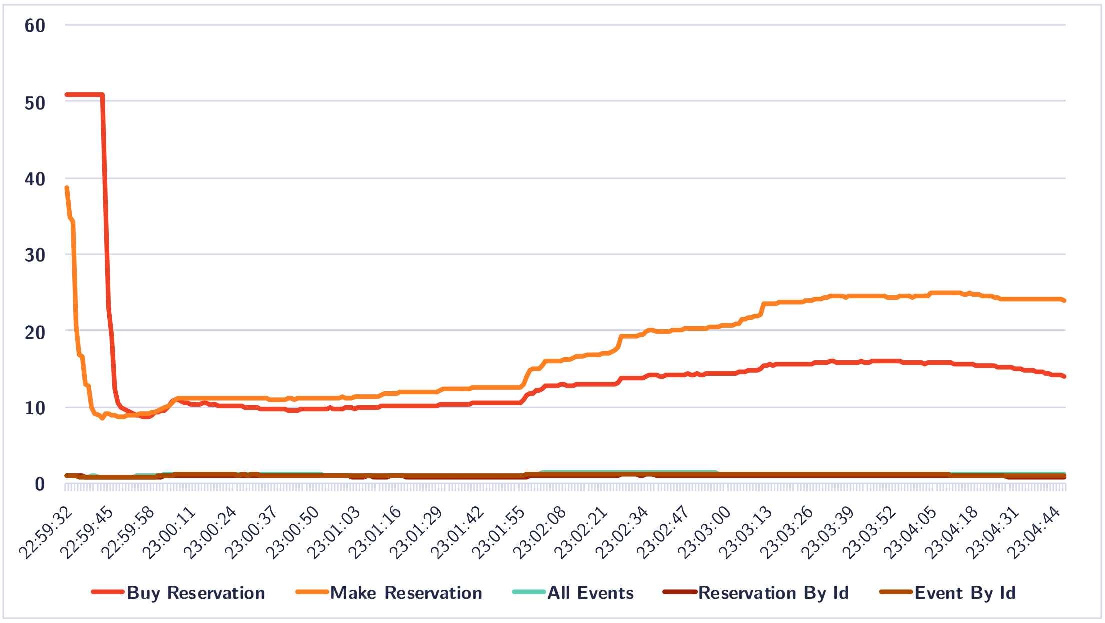


Figure 7.6: Response times in milliseconds of methods in the inventory service during the second load test case on the conventional system.

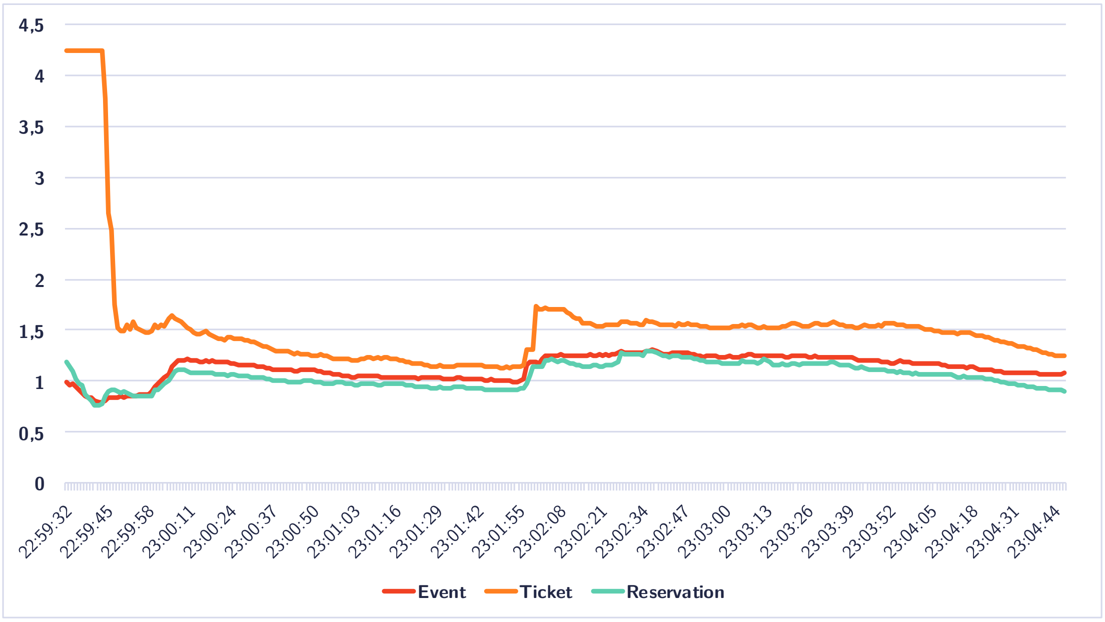


Figure 7.7: Response times in milliseconds of database calls from the inventory during the second load test case on the conventional system.

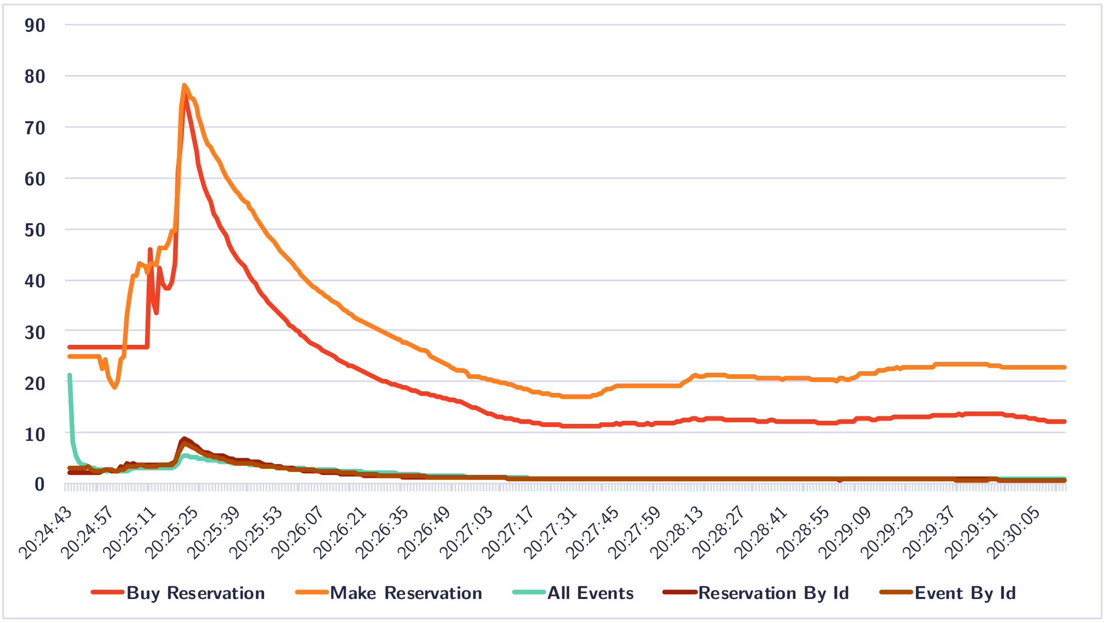


Figure 7.8: Response times in milliseconds of methods in the inventory service during the fourth load test case.

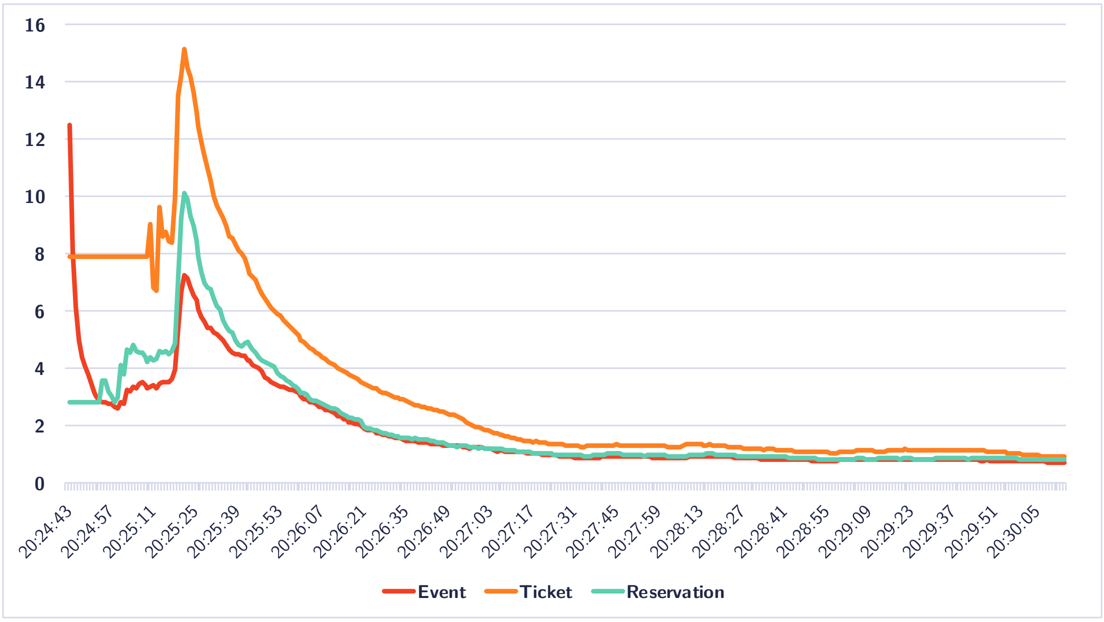


Figure 7.9: Response times in milliseconds of database calls from the inventory during the fourth load test case.

To further analyse the various steps of a requests as it processes through the system, figure 7.10 gives an exemplary overview of the *buy cart* request, which processes through every service and database in the system. The total time until a customer received a response is 576 ms. 400 ms of those are the intended sleep in the payment service. The remaining 176 ms are split up between calculations happening in the services and transport durations over the network. The sum of all annotated times results in the response time measured by Gatling.

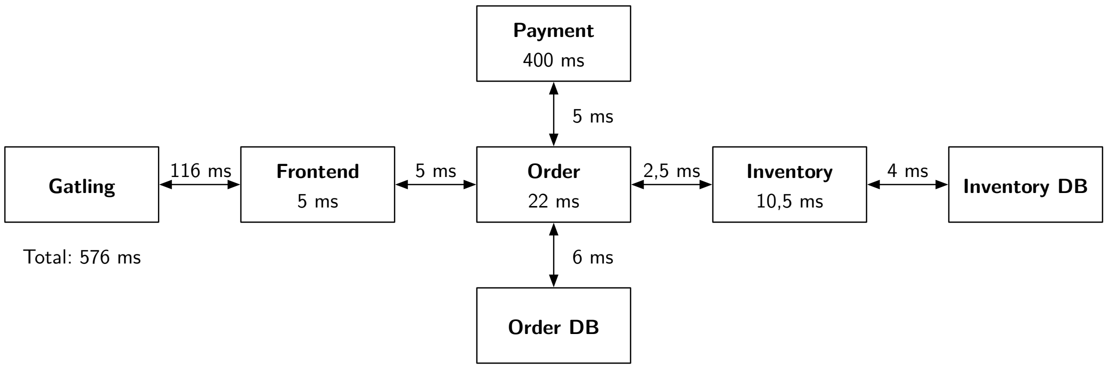


Figure 7.10: The *buy cart* request in the conventional Ticketsystem split up into the various stages it traverses. The boxes represent each stage/service while the arrows symbolise the connection between the stages. The connections and stages are annotated with their duration, as far as data is available. The transport times between the services and the databases includes the time the query takes in the database.

Summarising the load tests on the conventional Ticketsystem, it can be stated that the behaviour of the system depends more on the number of service nodes than the number of database nodes. Scenarios with more nodes generally performed better than scenarios with fewer nodes, even though the differences found during the performed tests are small.

7.1.2 Reactive

Identically to the tests with the conventional system, the tests with the reactive system all had a duration of 327 seconds with the same number of requests. Table 7.2 gives an overview of the response times Gatling measured, as well as the number of long-running requests. No requests failed and the mean response time varies from 210 ms to 219 ms. A striking value is the maximum response time of 6517 ms measured in the third test case. Reviewing the graphs, no specific source for this peak could be found and analysing the response time percentiles (see figure 7.11), it appears to be a one time peak, as the rest of the graph seems relatively monotonous and the percentile values in table 7.2 do not appear to be far more off from the other test cases.

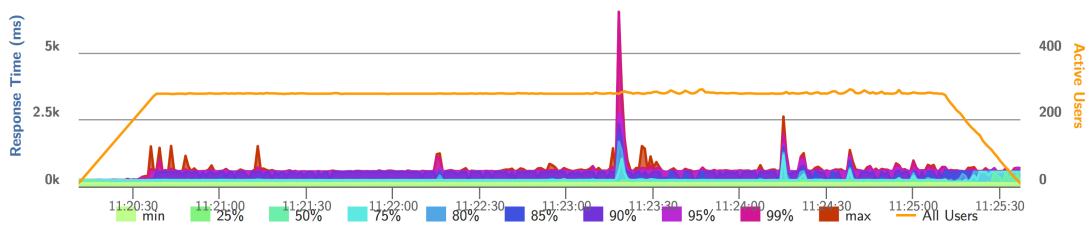


Figure 7.11: Response time percentiles over time in the third reactive load test case. The orange line indicates the number of active users while the coloured areas show the percentiles.

More interesting is the fact that the third test case in total shows the worst results. It produced by far the most long-running requests with 61 taking between 800 ms and 1200 ms and 69 taking longer than 1200 ms. In contrast to this, the second test case outperformed all other cases with a maximum response time of 960 ms and only 8 requests which took longer than 800 ms. As this is the test case with both the services and the databases scaled to five instances, this result might seem expected, but comparing this to the fourth test case, which comes second according to the response times and percentiles, it is rather noticable since both cases two and four have the same number of service instances and database instances running, which may be an indicator of a specific behaviour of the reactive MongoDB driver that was used in the implementation.

Test cases one and three show comparable similarities, apart from the peak and the statistics with the long running requests. The two pairs of test cases grouped by similar results lead to the assumption the system performing better when the number of instances per service is equal to the number of database nodes where higher scaling still outweighs smaller scaling.

Examining the results from the backend's point of view, no curiosities can be found apart from that the reactive services apparently require some time to accommodate to the load (see figs. 7.12 to 7.15).

	Case 1	Case 2	Case 3	Case 4
Mean (ms)	218	210	219	211
Min (ms)	116	116	116	117
Max (ms)	1477	960	6517	1513
50th percentile (ms)	160	147	148	147
75th percentile (ms)	235	236	237	236
95th percentile (ms)	583	562	559	556
99th percentile (ms)	625	591	637	597
800 ms < t < 1200 ms	12	8	61	35
1200 ms < t	9	0	69	3

Table 7.2: Customer's view of the load test results on the reactive Ticketsystem generated by Gatling. It shows that the second test case outperforms the other cases.

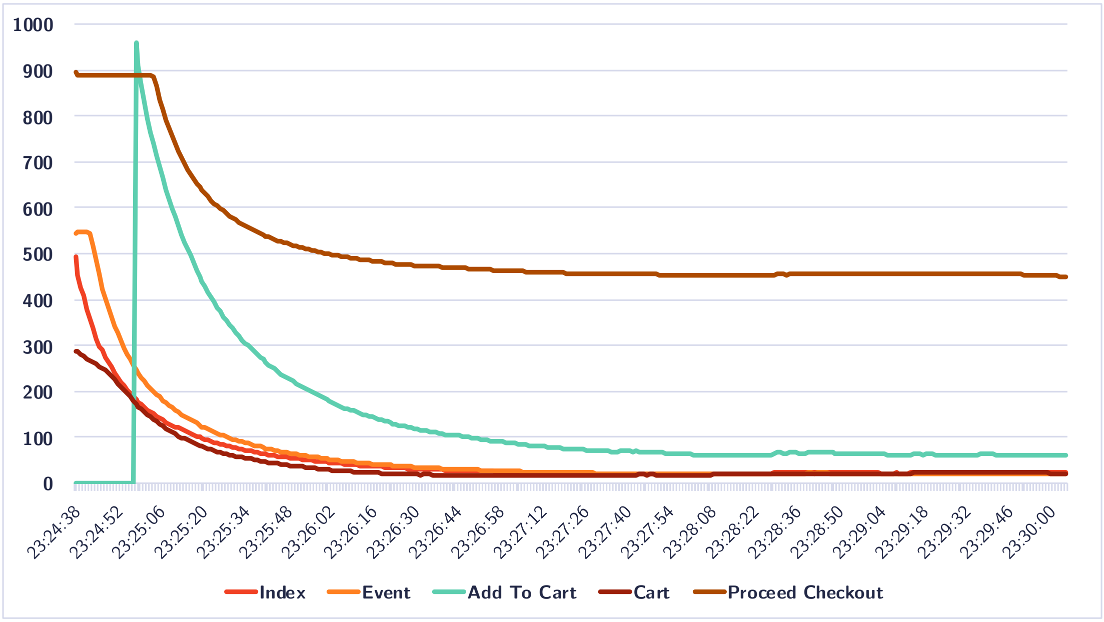


Figure 7.12: Response times in milliseconds of methods in the frontend service during the second load test case on the reactive system.

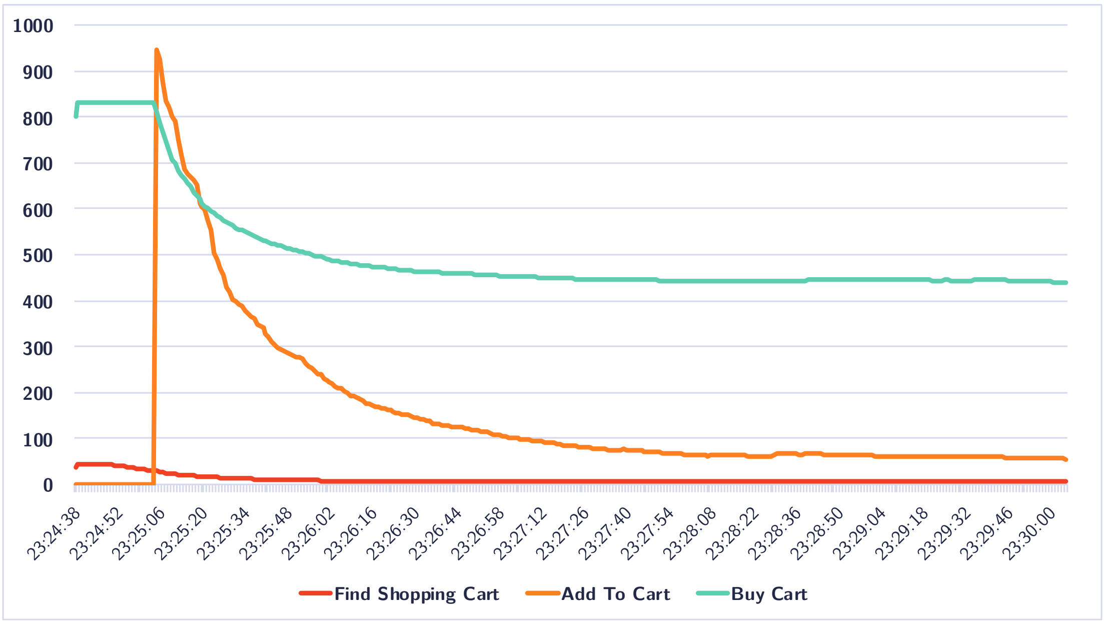


Figure 7.13: Response times in milliseconds of methods in the order service during the second load test case on the reactive system.

The services initially show a higher response time which then decreases during the test execution. This might be caused by the exponentially decaying values taken by the metrics library but can also be a side effect of the reactive MongoDB driver, as these initial peaks can be traced down to the database calls (see figure 7.15). Writing operations on the database were influenced the most. The reading operations are significantly faster, even at the beginning of the test. Although the graphs show the response times during the second load test, the general trend appears to be equal in all other test cases.

After evening out, the database's response durations are between 2 to 4 milliseconds, the response times of the inventory service take under 10 milliseconds and the total processing duration of requests takes about 20 to 60 milliseconds in the frontend, depending on the request. Obviously, processing the checkout takes significantly longer, as these requests are by purpose delayed in the payment service by 400 milliseconds. From the frontend's view, the total checkout processing takes about 450 to 460 milliseconds. This specific request has a mean response time of 566 milliseconds according to Gatling. A ping command from the terminal to the IP address of the reactive Ticketsystem resulted in an average of about 113 ms.

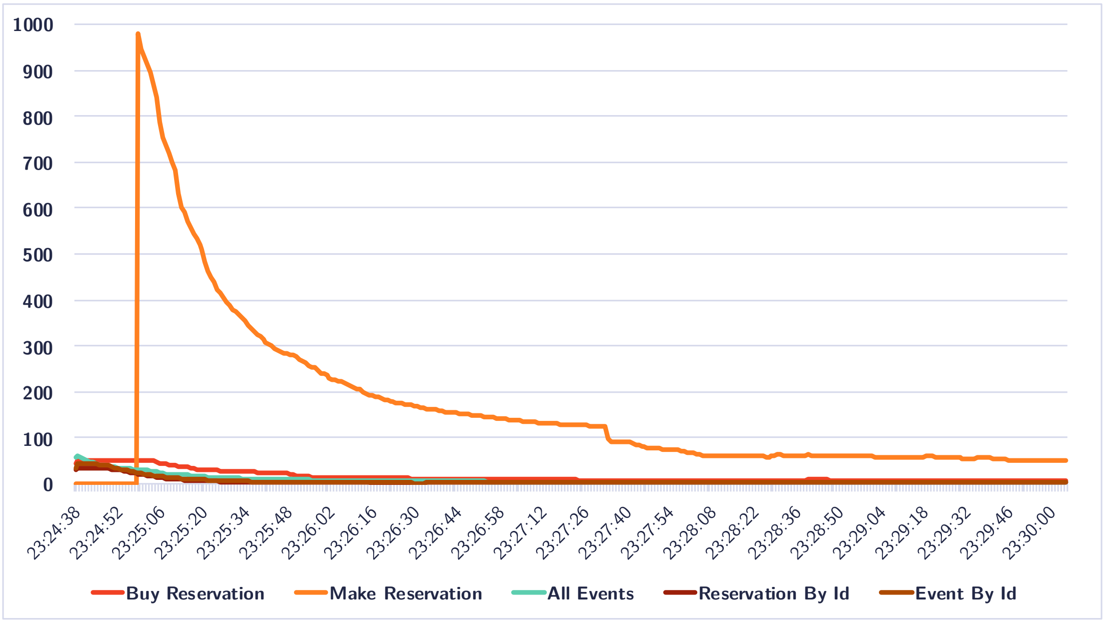


Figure 7.14: Response times in milliseconds of methods in the inventory service during the second load test case on the reactive system.

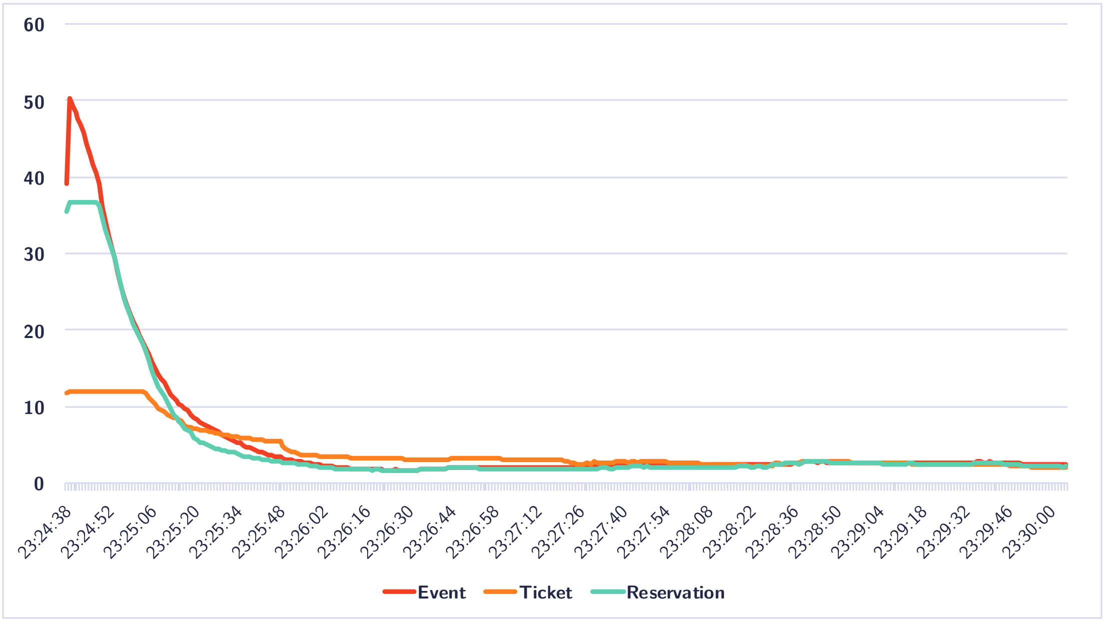


Figure 7.15: Response times in milliseconds of database calls from the inventory during the second load test case on the reactive system.

Figure 7.16 shows, analogous to the figure in the previous section, the split up response times of the various stages the *buy cart* request traverses through. The total time as measured by Gatling is 566 ms. Apart from the 400 ms delay in the payment service, most of the remaining time is spent on transport rather than on actual processing in the services.

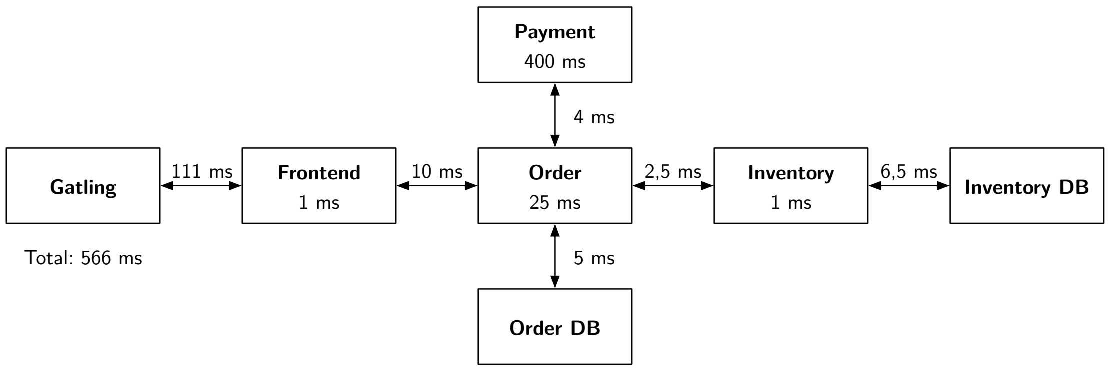


Figure 7.16: The *buy cart* request in the reactive Ticketsystem split up into the various stages it traverses. The boxes represent each stage/service while the arrows symbolise the connection between the stages. The connections and stages are annotated with their duration, as far as data is available. The transport times between the services and the databases includes the time the query takes in the database.

Summarising this subsection, it can be said that the reactive approach requires some time to accommodate to the load it is experiencing. The various test cases turned out to behave somewhat expected. The second test case performed best, while the other test cases showed an unexpected behaviour, as it appears that homogenous scenarios with equally scaled databases and services perform better than heterogenous scenarios. Test cases with more database nodes showed less long-running requests than the scenarios with only one node.

7.2 Resiliency tests

The resiliency tests were performed according to the description in chapter 6.2.2. In contrast to the load tests, only one test case with one scaling scenario was performed. The pod running the inventory service was deleted after the first 60 seconds of the test. Kubernetes then automatically started another inventory service pod. As reactive software needs to be resilient, the expectation was, that the reactive system can cope better with the situation of a missing service, especially because it was meant to be resilient, whereas the conventional approach only had basic error handling without special attention towards resiliency. The following two sections present the results of the resiliency test.

7.2.1 Conventional

The duration of the conventional resiliency test case was 231 seconds. Within this time 500 virtual users were launched and performed 4858 requests with an average of 20,9 requests per second of which 4595 succeeded. The remaining 263 requests failed as a result of the restarting inventory service. In a scenario without errors the test would have required 4500 requests but the simulated users had to repeat some of the requests as a result of the outage. Figure 7.17 gives an overview of the responses during the test run. After one minute a drop in the green graph is visible, which indicates the point at which the inventory service had been killed. Some seconds later, the red graph starts to indicate error responses. Specifically the HTTP error code 500, which stands for *Internal Server Error*, has been returned in all cases. Code 500 is the default error code Spring Boot applications return, when unhandled exceptions or other unexpected errors occur. This was the case, as the outage resulted in connection timeouts and socket connection failures in all services connecting to the inventory service.

After about half a minute the errors decreased, which indicates that the substitute inventory service has finished starting up and began to handle the requests successfully. About two minutes after the start the number of active users reached 500, which then after another 30 seconds started to drop, as the first users were finishing their simulation. The first users finished the simulations 2 minutes and 25 seconds after the start.

Table 7.3 shows response time statistics that were taken during the test run. The total mean response time was 775 ms, while for all successful responses it was 239 ms. The mean response time of error responses was at 10.140 ms. This is also visible in figure 7.17. The error responses only started about 10 seconds after the inventory service stopped responding.

Apart from the 263 failed requests there were 35 requests that took longer than 800 ms to be responded to successfully, of which 15 even took longer than 1200 ms.

Figures 7.18 to 7.21 show the results recorded from the system's point of view. It can be seen that after the outage the frontend's response time jumps up to about ten

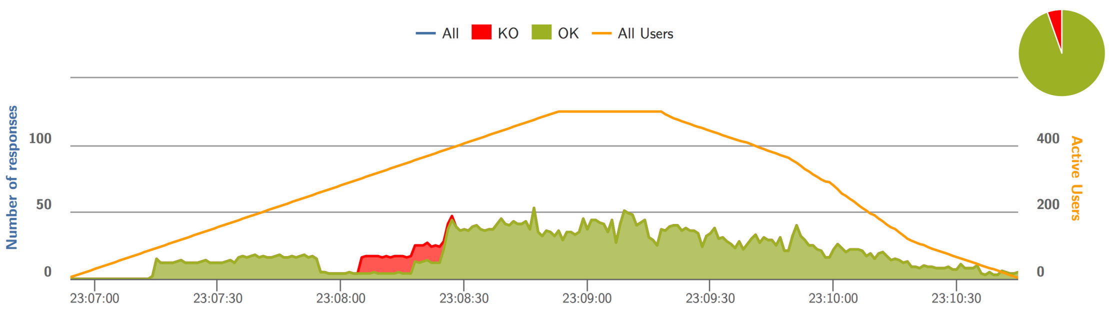


Figure 7.17: Responses during the resiliency test on the conventional Ticketsystem. The orange graph shows the active users, the green area shows the successful responses and the red area shows when error responses were received.

Mean total (ms)	775
Mean successful (ms)	239
Mean error (ms)	10140
Min (ms)	117
Max (ms)	10328
50th percentile (ms)	181
75th percentile (ms)	251
95th percentile (ms)	10129
99th percentile (ms)	10145
800 ms < t < 1200 ms (successful)	20
1200 ms < t (successful)	15
Failed	263

Table 7.3: Customer's view of the resiliency test results on the conventional Ticketsystem generated by Gatling. The rows showing the count of long-running requests only show numbers accounting for requests that were successful.

seconds (see figure 7.18). This can be followed down the call stack to the order service (see figure 7.19) which fails to contact the inventory service. The frontend also struggles to make successful calls directly to the inventory, which is also visible by following the graphs in the figure, although the *Event* and *Index* requests appear to fail faster than the *Add To Cart* request.

The time of the inventory's outage can be seen in figures 7.20 and 7.21 where the graphs are disconnected for about 20 seconds. After the inventory started up again, the response times of its writing operations increase a lot in comparison to the period before the shutdown, probably caused by the high and immediate load it experiences.

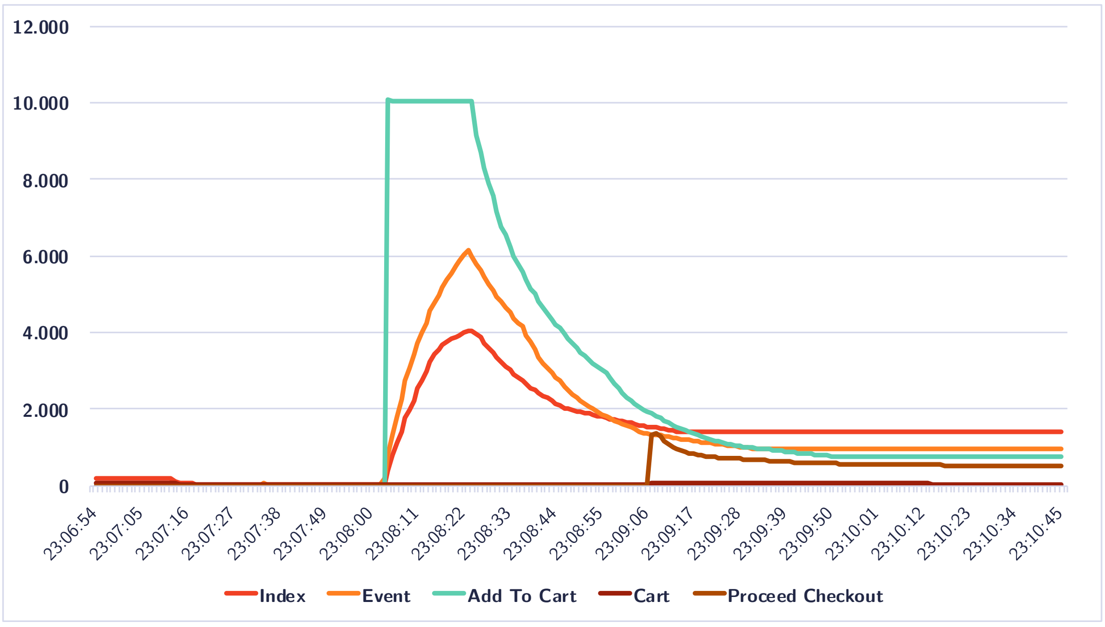


Figure 7.18: Response times in milliseconds of methods in the frontend service during the resiliency test on the conventional system.

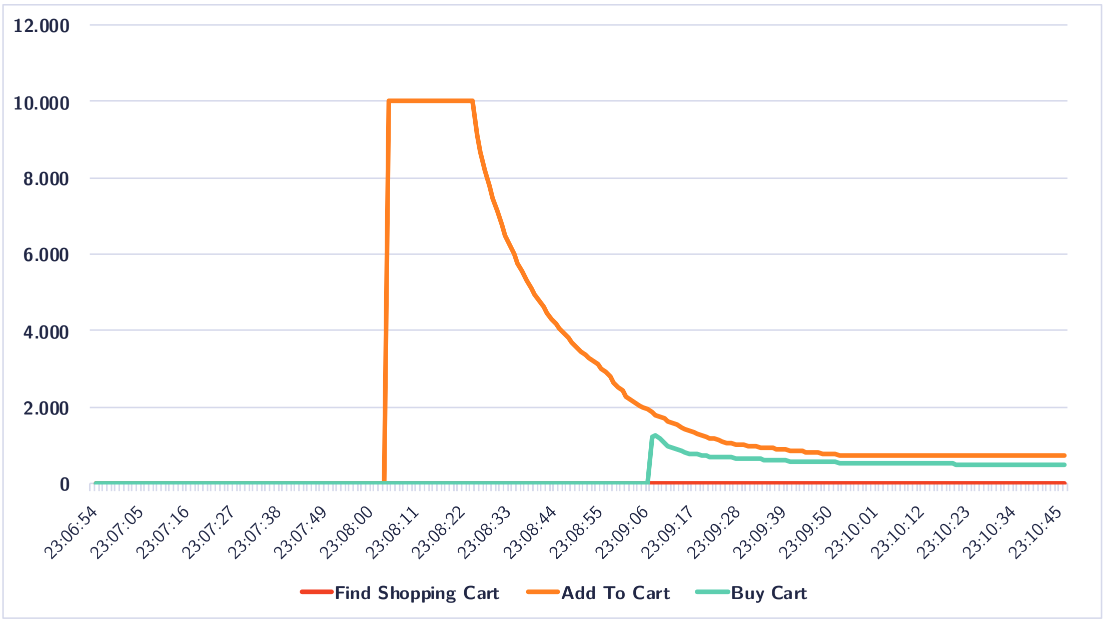


Figure 7.19: Response times in milliseconds of methods in the order service during the resiliency test on the conventional system.

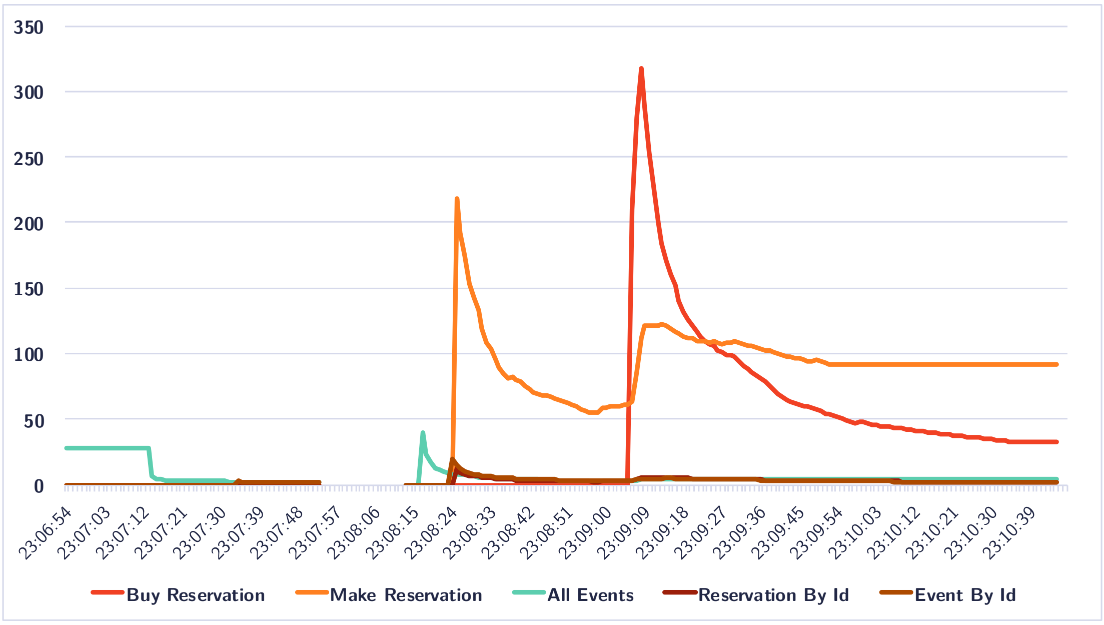


Figure 7.20: Response times in milliseconds of methods in the inventory service during the resiliency test on the conventional system. In the timespan where the service was down, no data is shown.

This can be pursued down to the database, which also takes longer to answer after the restart.

Ultimately, the services appear to stabilise at some point during the test execution, but the response times are still higher than before. Both views, the internal system's view and external customer's view, conform to each other, as the high response times are visible in the data both sides recorded. Even though the results show long response times and errors, all simulated users succeeded in finishing the test scenario and buying tickets from the Ticketsystem. In total as a result of repeated requests, the test run required 8% more requests than necessary.

Summarising the results of the resiliency test on the conventional Ticketsystem, it can be stated that the system reacted with very long timeouts which resulted in high mean response times, errors and extreme error response times. The system needed about 32 seconds to recover from the failure, but took even longer to return to a stable state. Also about five percent of the users had to deal with response times of more than ten seconds during the failure. Finally, the test succeeded and all users were able to perform the programmed routine.

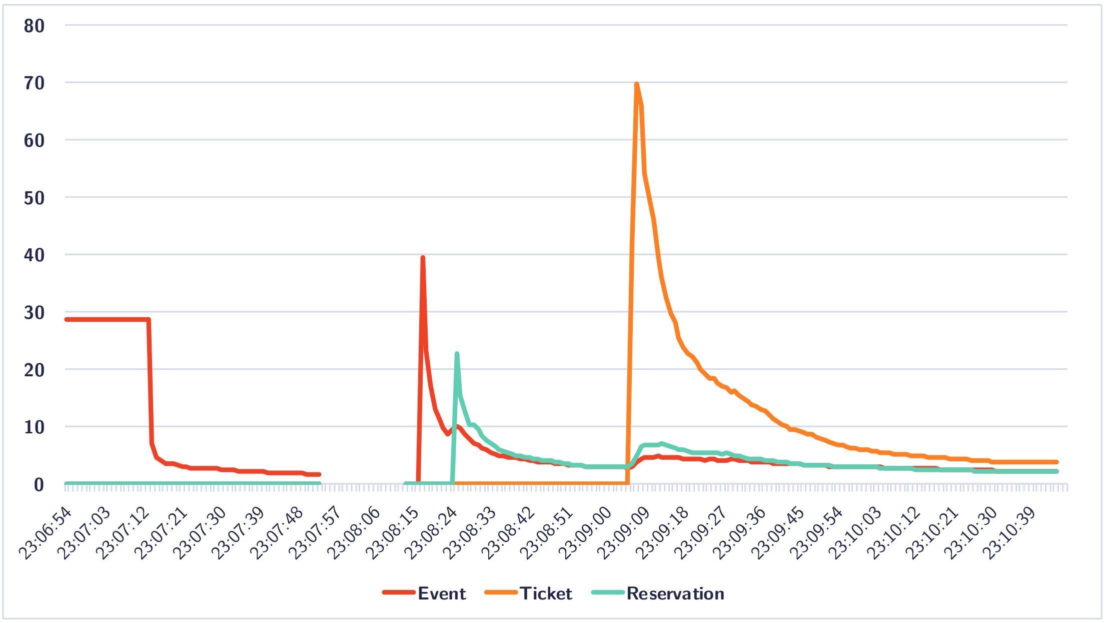


Figure 7.21: Response times in milliseconds of database calls from the inventory during the resiliency test on the conventional system. In the timespan where the service was down, no data is shown.

7.2.2 Reactive

The same scenario has been repeated on the reactive version of the Ticketsystem. The total execution duration was 231 seconds and the 500 simulated users performed 4920 requests with an average of 21,2 requests per second during this time. 287 of those requests resulted in errors due to the temporary outage of the inventory service. The outage was induced 60 seconds after the test began. The corresponding decrease of positive responses can be observed in figure 7.22. Seven seconds later, error responses started to be monitored. The reactive services are programmed to return status code 503 *Service Unavailable* (see chapter 3) when communication timeouts occur. In addition to this, the services immediately return this code, when an intermediate service, which was called during a request, returns 503. All errors received by Gatling were of this type and a consequence of the restarting inventory service.

It can be seen that the number of errors quickly rises and then begins to fall again. After 25 seconds of downtime the substitute inventory service began serving first requests and some seconds later the system returned to a fully operational state. The

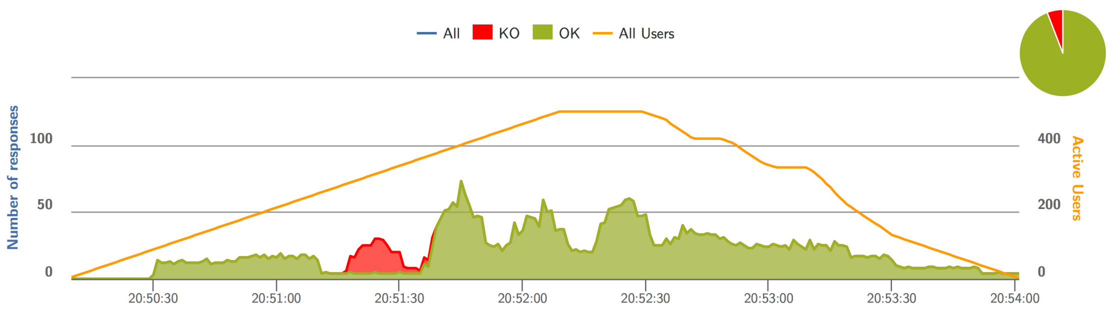


Figure 7.22: Responses during the resiliency test on the reactive Ticketsystem. The orange graph shows the active users, the green area shows the successful responses and the red area shows when error responses were received.

maximum number of 500 users were active for about 20 seconds before the first users completed their simulation and the number of users started to decrease. Thus the first users successfully finished the simulation after 2 minutes and 20 seconds. After the outage the response rates graph shows phases of many responses (with peaks at 73) alternating with phases of less responses (lows at 20). What might be misinterpreted as broken software behaviour can be explained with the graph in figure 7.23 which shows the requests Gatling made. The graph of produced requests and the graph in figure 7.22 showing the responses look almost identical, with the major difference during the timespan of the outage of the inventory service. This shows that there is no broken software behaviour involved but instead the requests were being sent in the same phases as the responses were returned.

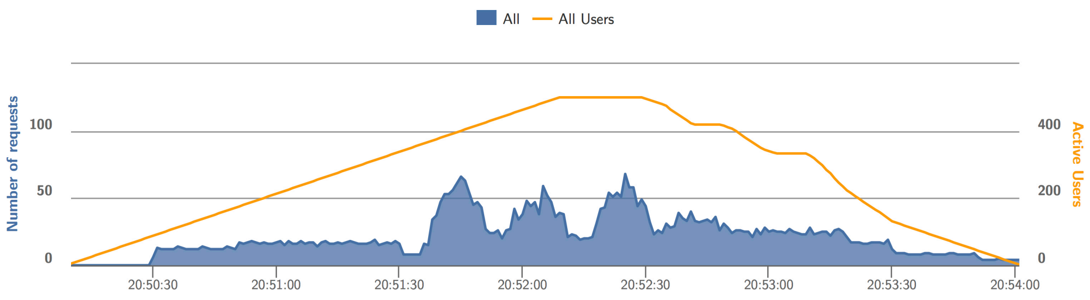


Figure 7.23: The number of requests Gatling produced during the resiliency test on the reactive Ticketsystem. The orange graph shows the active users while the blue area shows the number of requests Gatling initiated.

A detailed summary of the responses and the times they took is listed in table 7.4. It shows that the total mean response time was 372 ms. Successful requests returned faster with a mean response time of 185 ms, while errors took 3384 ms to return. The longest running request took 7120 ms whereas the quickest response returned after 112 ms. In total, all successful requests took less than 800 ms to complete. 287 requests failed and led to a repetition of the corresponding simulation step. Similar to the previous chapter, these statistics can partly be seen in the responses figure 7.22, as the errors started to appear about 7 seconds after the shutdown of the inventory service.

Mean total (ms)	372
Mean successful (ms)	185
Mean error (ms)	3384
Min (ms)	112
Max (ms)	7120
50th percentile (ms)	130
75th percentile (ms)	164
95th percentile (ms)	550
99th percentile (ms)	7120
800 ms $<$ t $<$ 1200 ms (successful)	0
1200 ms $<$ t (successful)	0
Failed	287

Table 7.4: Customer's view of the resiliency test results on the reactive Ticketsystem generated by Gatling. The rows showing the count of long-running requests only show numbers accounting for requests that were successful.

The metrics recorded in the backend (see figs. 7.24 to 7.27) correlate with the measurements Gatling provided. The outage of the inventory can be seen by the missing data in the figures 7.26 and 7.27. Shortly after the outage the response times of the system increase significantly due to the first error responses. This increase is mostly visible in the order service (figure 7.25), but also shows up in figure 7.24, which shows the response times of the frontend service. The mean response time in the frontend is lower than in the order service because the circuit breaker and error handling logic forwarded the order service's errors and thus prevented further long requests and let subsequent requests fail immediately.

The recovered inventory service required about 15 seconds to return to a stable state. During this time, the requests to the database took a significant amount of time, with a special impact on writing operations. The reason could be a slow initiation of

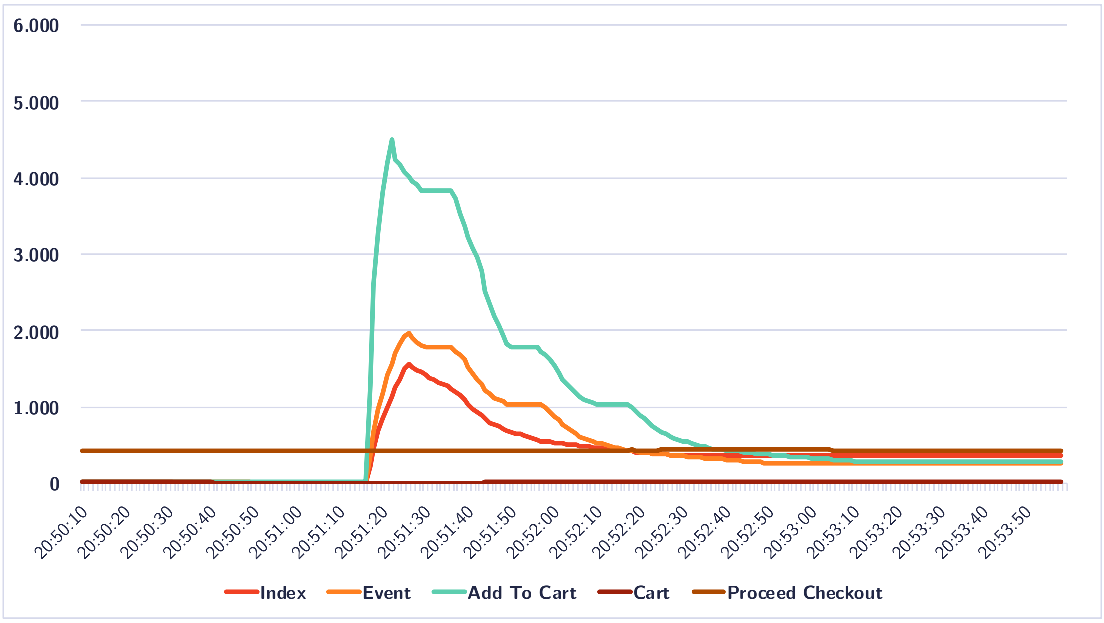


Figure 7.24: Response times in milliseconds of methods in the frontend service during the resiliency test on the reactive system.

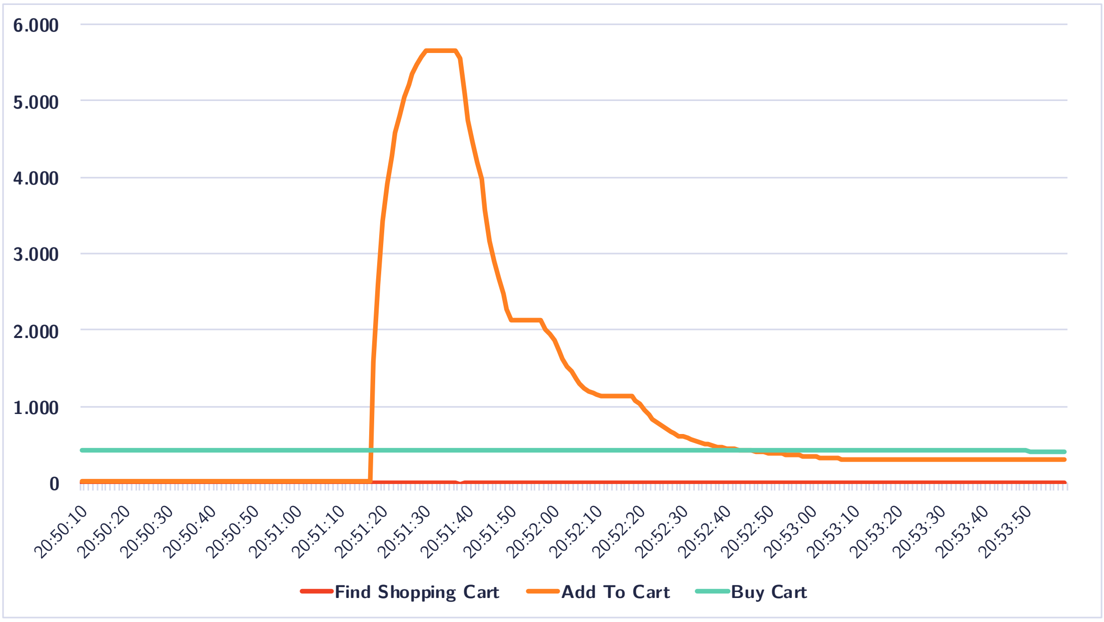


Figure 7.25: Response times in milliseconds of methods in the order service during the resiliency test on the reactive system.

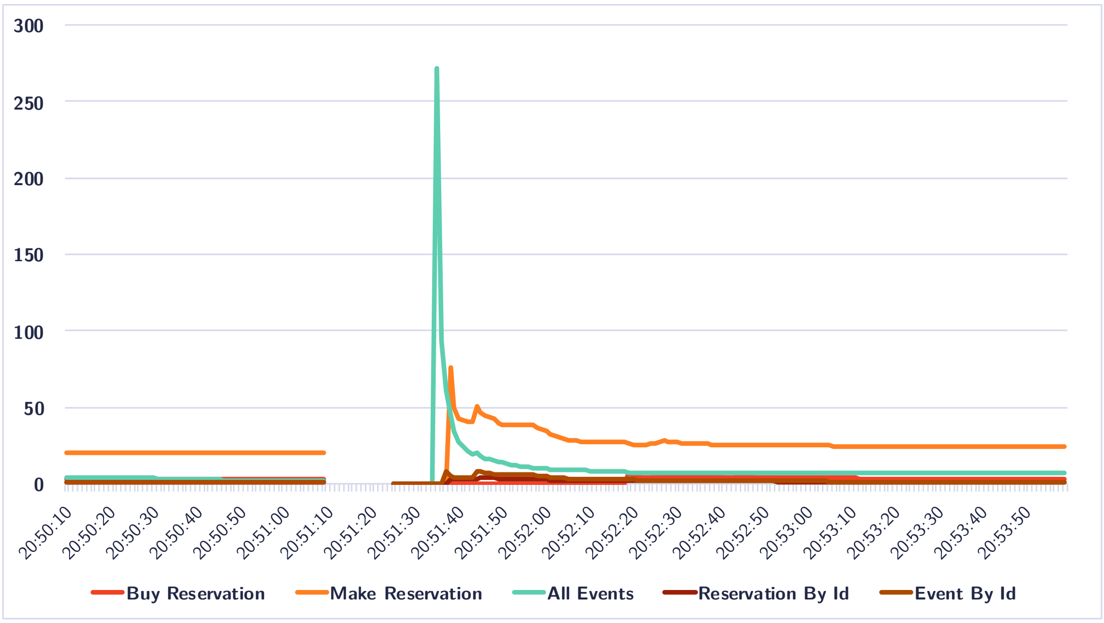


Figure 7.26: Response times in milliseconds of methods in the inventory service during the resiliency test on the reactive system. In the timespan where the service was down, no data is shown.

the connection between the service and the database combined with a relatively high frequency of requests. Eventually, the response times returned to lower values. The same recovery time was required by the other services to return to the pre-outage state, which is a result of the dependency to the inventory service.

After a short time, all services returned to a stable state and had adequate response times. Although this test showed that errors are returned to the users, all users successfully finished the simulation and achieved to buy tickets. The errors accounted for 9% of additional requests that had to be repeated in comparison to a faultless scenario.

In summary, the reactive system reacted with a few relatively long timeouts for about 1% of all users but in general kept acceptable response times, even in error cases. Additionally, the system recovered in 25 seconds to a working, and in 40 to 50 seconds to a stable state. As a total result, the test completed successfully and all users were able to perform all requests.

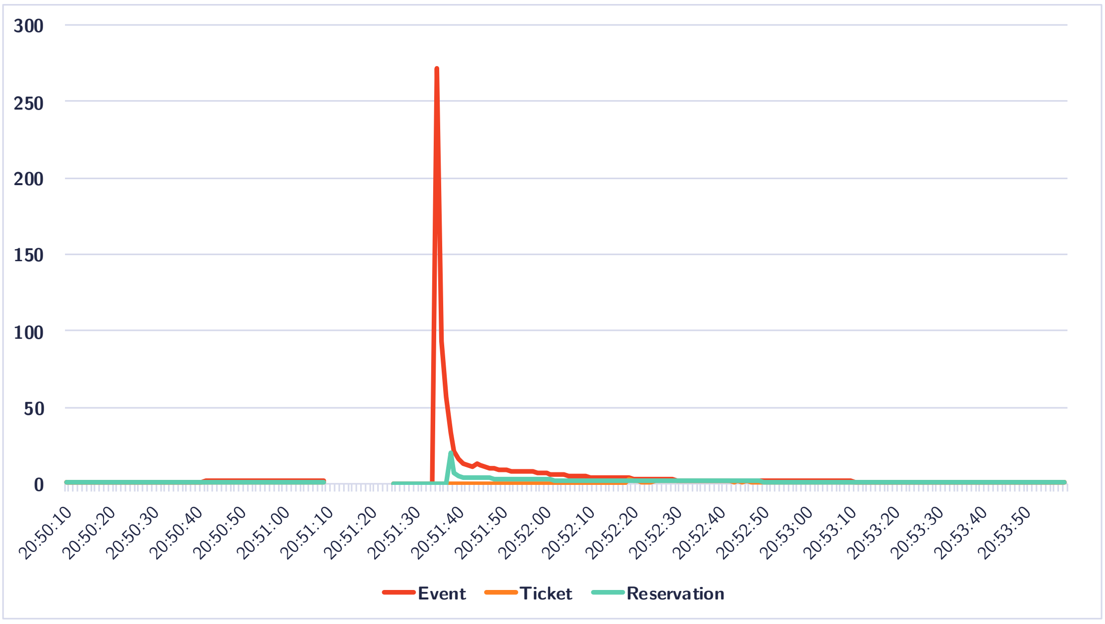


Figure 7.27: Response times in milliseconds of database calls from the inventory during the resiliency test on the reactive system. In the timespan where the service was down, no data is shown.

7.3 Comparison of the conventional and reactive results

After the results of the test cases have been presented, the following sections are going to compare the conventional with the reactive results.

7.3.1 Comparing load capacity

Beginning with the comparison of the load test results and keeping in mind the expectations that were mentioned at the beginning of section 7.1, one deflating result is that, while the reactive approach showed slightly better results in some metrics than the conventional, the conventional approach still showed a better overall performance according to mean response times. The differences are so small though, that its superiority cannot be stated with utter conviction. As to the second expectation, that higher scaled test cases show better results, the expectation was fulfilled. Again, the differences were only small, though, but in both scenarios the best performing case was one with higher scaling.

For a brief comparison, table 7.5 shows the best and worst performing test cases of each scenario next to each other, while *best* and *worst* refer to the mean response times and number of long-running requests.

	Best		**Worst**	
	Conv. 3	**React. 2**	**Conv. 4**	**React. 3**
Mean (ms)	206	210	215	219
Min (ms)	115	116	116	116
Max (ms)	2681	960	1726	6517
50th percentile (ms)	140	147	148	148
75th percentile (ms)	235	236	237	237
95th percentile (ms)	559	562	572	559
99th percentile (ms)	583	591	626	637
800 ms < t < 1200 ms	8	8	32	61
1200 ms < t	14	0	15	69

Table 7.5: Comparison of the Gatling results of the best and worst test cases which resulted from each scenario.

Starting with the best cases, most rows are only a few milliseconds apart with the only large difference being the reactive system showing a significantly lower maximum response time of almost only one third of the conventional system's value. This also accounts for the fact that the reactive system did not have any requests running longer than 1200 ms while the conventional approach had 14 of these.

It is probably not possible to proclaim one of these approaches as the better one. Such statements can only be made on the basis of specific values of the statics. For example, the conventional approach wins regarding to mean response time but the reactive approach is definitely superior to the conventional concerning maximum response time when comparing the best results. Confusingly, exactly the maximum response time shows an inverse behaviour when comparing the worst performing test cases per scenario, where the reactive approach took a maximum of 6517 ms to respond while the conventional approach only needed 1726. Furthermore, ironically the maximum response time in the worst conventional approach was the best maximum response time of all conventional test cases.

Comparing the worst test cases, it can be discovered that these also differ slightly by some milliseconds with both approaches being better than the other in some cases. The biggest differences occurred, as in the previous comparison, in the maximum response time and number of long requests. The reactive approach showed more than three times the number of long running requests and had a maximum response time of four times of the conventional approach.

Moving on to the comparison of the performance from the system's point of view and starting with the inventory's database, a small difference can be seen in the response times. In the reactive approach, the response time, once stable, settles around 2,5 ms (see figure 7.15). The conventional approach manages to perform these requests about 1 ms faster, which can be seen in figure 7.7. Looking further up the call stack, the inventory service responds to most requests in about 2 to 10 ms in the reactive approach. Only the *make reservation* request takes about 60 ms. The same request takes less than half the time to complete in the conventional system. However, the *buy reservation* call takes about 10 ms longer than the reactive counterpart, while the reading operations are answered in durations of around 2 ms. This makes the reactive inventory almost twice as fast when it comes to writing operations and only slightly slower comparing reading performance. The differences in the order service are a bit larger with the reactive system being only about half as fast as the conventional approach showing differences of 3 ms to 30 ms. The frontend shows equal performance when it comes to reading requests. The *add to cart* request is 30 ms slower in the reactive system whereas the *proceed checkout* request is 10 ms faster. In general, all reactive services needed some time to *swing in*, noticeable by longer response times, although, after some time, the performance improved.

A detailed comparison of the timeline of the *proceed checkout* request can be made by looking at the figures 7.10 and 7.16. The overall result of both requests is similar with 566 ms in the reactive system and 576 ms in the conventional part. The reactive system has highly reduced delays inside the individual services. Only the order service requires 3 ms more processing time compared to the conventional counterpart. The conventional inventory adds 10,5 ms of delay to the request while the reactive part only requires 1 ms. This covers up for the slower database response times of the reactive inventory. The figures show that the reactive approach is faster when it comes to processing time inside the services. While the conventional approach is also better in some of the values, like the processing time in the order service or the duration of calls to the inventory database, the overall performance is 10 ms slower. This is not a large difference but it is interesting when looking at the details and roots of it.

In summary, the load test does not appear to give clear results as both implementations succeed in outclassing the other in some points but are inferior in other aspects. There are subtile hints that the implementations of the different MongoDB drivers can be responsible for some of the minor differences, but further analysis would be required to prove or deny this hypothesis. It is visible, that systems with more replicas perform better in both scenarios, though.

7.3.2 Comparing resiliency

A comparison of the resiliency test results draws more interesting conclusions in contrast to the load tests. Both Ticketsystem implementations passed the test and all users managed to buy tickets, although some requests had to be repeated as a result of the outage of the inventory service. Indeed, as expected, the reactive system proved to be much more resilient and to cope with the situation better than the conventional approach. The conventional Ticketsystem was not directly programmed towards high resiliency (if it had been, there would have been no point in comparing the implementations). However, it still handled the outage really well, although it was returning internal server errors which is generally a way of expressing that an unexpected error happened and which is almost never a desired state, although true in this case.

Table 7.6 shows a direct comparison of the test results in both implementations. It is obvious that the reactive Ticketsystem outperforms the conventional in every way with one exception: The reactive approach produced 24 additional erroneous responses compared to the conventional system. Likely, the circuit breaker has to account for this number. Upon errors, it lets subsequent requests fail immediately without checking the availability of the missing inventory service constantly. Only after a grace period, it allows some of the requests to be made. This leads to a small number of false negatives right after the inventory finished starting up. However, it prevents that the service is overloaded by high requests right while starting up but might not be fully able to handle requests. Considering these circumstances, the slightly higher error count could actually be seen as a better (or at least not worse) result in this case.

Further analysis of the values in table 7.6 indicates that the reactive approach had a twice as fast total mean response time. Considering only the successful responses, the mean response time is still faster by almost 30%. Errors returned even thrice as fast as in the conventional approach with a mean time of 3350 ms compared to 10140 ms. A major difference can be found when comparing the mean error response time with the maximum response time. Apparently most erroneous requests took about 10 seconds in the conventional system, as there is only a difference of 188 ms between the mean error response time and the maximum response time. The reactive system's mean error response time is less than half the time of the maximum response time. This shows that a lot of the failed requests only took a very short time to return. When comparing the response time percentiles, it can be seen that the 95th percentile still shows 10129 ms in the conventional approach while the reactive approach is at 550 ms. For the upper five percent of users this makes a significant difference, which is much in the favour of the reactive approach. Looking at the count of long requests, it can be seen that the conventional approach has 20 requests that took between 800 ms and 1200 ms and 15 requests longer than 1200 ms while the reactive implementation had no long running requests at all (the maximum successful response time was 682 ms).

Viewing the graphs that show the response times from inside of the system, some similarities but also differences can be seen between the two approaches. It can be observed that, while both implementations showed a high initial response time right after the restart, the conventional approach took longer to recover to the same healthy state it had before it was killed (see figures 7.20 and 7.26). Especially the reactive inventory service and its database show only very narrow peaks in the timeline whereas the conventional inventory has broader peaks, hence longer periods of high response times. The same effect, although less distinct, can be observed on the frontend and the order service.

Both approaches proved to finish the scenario successfully and all virtual users were able to complete the tests after the same time. The reactive system could achieve this with a twice as good mean response time and thrice as good error response time

	Conventional	**Reactive**
Mean total (ms)	775	372
Mean successful (ms)	239	185
Mean error (ms)	10140	3384
Min (ms)	117	112
Max (ms)	10328	7120
50th percentile (ms)	181	130
75th percentile (ms)	251	164
95th percentile (ms)	10129	550
99th percentile (ms)	10145	7120
800 ms $<$ t $<$ 1200 ms (successful)	20	0
1200 ms $<$ t (successful)	15	0
Failed	263	287

Table 7.6: Comparison of the Gatling metrics which were recorded during the resiliency test.

though and it was also able to have the first users complete five seconds earlier than the conventional Ticketsystem. The only aspect the conventional system managed to be better at, is the number of errors, which is lower by about 9% with 263 in comparison to 287.

7.4 Discussing the results

The previous sections have shown the results of the performed tests as well as a comparison between the results the conventional and the reactive Ticketsystem scored. This section is going to discuss the comparison.

Beginning with the load tests, it seems rather obvious that there is almost no difference at all. Both implementations of the system have pros and cons in some constellations and some points, but no substantial and striking differences are present. This leads to the conclusion that a reactively (or conventionally) programmed software is not simply *better* because of the paradigm it has been coded with. Reactive programming in this context of a web application in microservices can therefore not be seen as a general solution towards better performance in normal operation scenarios.

Considering the resiliency tests though, this is where reactive programming appears to be brilliant at. Although, according to the reactive manifesto, which states that *"(...) any system that is not resilient will be unresponsive after a failure."* [2], and the conventional Ticketsystem was responsive after the failure, the much better response times of the reactive Ticketsystem prove that reactive programming indeed helps to increase responsiveness in fault scenarios. Furthermore, responsiveness, according to the reactive manifesto, defines that a system *"(...) responds in a timely manner if at all possible."* and responsive systems *"(...) focus on providing rapid and consistent response times (...)"* [2]. That said, the over 10 seconds of mean error response time in the conventional Ticketsystem probably does not count as rapid and responsive and thus deprives the conventional approach of resiliency. Regarding the higher error count of the reactive system it was already stated that the circuit breaker is probably to account for. It should be added that reactive systems and circuit breakers follow a *fail-fast* design [12, 53], which demands this kind of behaviour. Consequently, the higher error count should not be mistaken as a drawback of circuit breakers but it is rather the intended design of this component to ensure responsiveness and stability. Finally, a lower error count with a mean error response time of over 10 seconds is with a high certainty not better than a slightly higher error count but with mean error response times of only one third of that length.

In conclusion to the tests performed as part of this chapter, it can be stated that reactive programming is not more performant in general, but it is more stable and performant in cases of errors. This improvement has a direct impact on the users of a system and could, in the example of a ticket shop, decide about gaining, losing or keeping customers and thus revenue.

8 Conclusion

This chapter will summarise the insights gained and the experiences made during the implementation of the Ticketsystem, the preparation and execution of the tests and their results and try to use this knowledge to prove or refute the hypotheses made in chapter 1.4.

8.1 Reflecting the project and its architecture

Planning microservices, there is almost always a different way to split up a system into separate services and the chosen approach might have been a different one as well. Besides different microservices there would also be the possibility to split the frontend into many parts individually served by the responsible services instead of having a dedicated service for the entire frontend. This section will discuss and reflect the decisions made in the Ticketsystem project.

8.1.1 Ticketsystem

The chosen microservices proved to be viable, are easily maintainable and all have their individual concerns. They can run without requiring the other services, although normal operational behaviour of course depends on the availability of other services.

Spring proved to be a viable framework for creating microservices as it supports all required features such as creating HTTP servers and clients, sharing the sessions using the Redis session cache and providing various drivers for databases, some of those even in both reactive and non-reactive fashion.

Although MongoDB had a very good performance, it might have been better to use a different database, preferably a relational one like PostgreSQL. This would allow to reference foreign tables and keys and perform queries which are more precisely tailored toward the data wanted. This is actually the approach that was implemented in [36] and which worked very well. Spring currently does not offer a reactive driver for PostgreSQL though, which would have been required to implement the reactive Ticketsystem in a similar way to the conventional one. This limited the choice to Cassandra, Couchbase and MongoDB, as Spring offers reactive drivers for those [54, 55]. Cassandra's concept of different keys, indices and partitions was not a viable approach for the use case of the Ticketsystem. As the final choice was between Couchbase and MongoDB, MongoDB was chosen due to its popularity. Furthermore, MongoDB is easily scalable, replicable and executable in containerised environments. Reflecting this choice, Couchbase might have been an interesting alternative.

Considering the architecture no seriously wrong choices have been made. There would have been alternative ways to program the system and other databases to try out. However, the chosen setup turned out to have no explicit flaws.

8.1.2 Test architecture

The test architecture already proved to be working in [36]. After working with this test setup longer, it appeared like it could have been easier or more lightweight in some points. Kibana is a convenient visualisation tool and was great to observe measurements in near realtime during the test executions. The additional option to export the data as .csv files for further analysis or visualisation was very helpful, especially to create vector images of the measurements for this thesis. Nevertheless, running Elasticsearch in a containerised cluster required a lot of configuration. Additionally, Elasticsearch needs many nodes to run in a robust cluster, which, in this case, resulted in a seven node cluster consisting of three master, two client and two data nodes. A different setup with other tools (e.g. Grafana [56] in combination with a different database) might have been less bulky. The chosen solution allowing the Ticketsystem services deliver their metrics to an external endpoint proved to be good and easy to construct. That way, potential new services can be added without the need to reconfigure the external applications.

Apart from the relatively complex and heavy setup, the test architecture proved to be working, although it could be improved in some ways such as by reducing the bulkiness or lowering the number of required external services.

8.2 Challenging the hypotheses

After the results of the automated tests being presented in chapter 7, it is possible to look back to the hypotheses established in chapter 1.4 and examine whether they proved to be true.

8.2.1 H1: Performance benefits

"Reactive programming has benefits over conventional programming patterns regarding performance (possibly under various loads)."

The first hypothesis can be discussed with the results acquired by the load tests (see chapter 6.2.1). It was shown that the reactive Ticketsystem's performance did not differ much from the conventional approach. Some of the metrics appeared to be better in the reactive approach and some were better in the conventional system, no clear *winner* could be determined yet. Even various scaling setups did not show any significant differences. Comparing some individual test cases, it could be found that the internal data processing of some services was faster in the reactive approach and that the inventory's database responded marginally slower, which in total resulted in a difference of only some milliseconds of response time. With values in this scale, a lot of random factors like varying latency between the Ticketsystem and the system executing the tests could account for such a small disparity.

Based on the available data and metrics, no remarkable performance differences could be found between the two approaches, hence the findings are refuting the hypothesis H1, although the performance in error cases needs to be discussed separately (see section 8.2.3). As the Ticketsystem is a very CRUD based application, an interesting experiment would be to re-run a similar test in a more data processing oriented application, as will be suggested in the following chapter 9.

8.2.2 H2: Software development process

"Reactive programming changes (and possibly improves) the software development process."

The second hypothesis can be related to based on the experiences made during the implementation process. While the experiences might be partly subjective, this section will present the findings as neutrally as possible.

The entire software development process consists of the actual programming, continuous integration and continuous delivery and includes writing tests, creating a development environment and learning to use libraries or frameworks. During the development of the Ticketsystem only some of these points were actually influenced by the reactive programming approach. The unaffected parts were continuous integration, continuous delivery, the creation of a development environment as well as the requirement to learn to use the selected tools. Those aspects were required in both implementation approaches and did not change between the implementations.

A noticeable difference was the programming process and the writing of tests. Reactive programming forces the developer to imagine the data flow in the application in a different way. Conventionally, when calling a method it will be executed immediately. In reactive programming methods return a stream. These methods also return immediately, but they are more like a builder pattern to create reactive streams which can be used to get the desired data. The actual data flow only starts when the returned reactive stream is subscribed to. Without subscribing, no data is being processed (apart from the reactive stream objects). This leads to less readable code in cases where the data processing is not entirely linear but e.g. a method needs to return a stream which combines data from two different streams. Undoubtedly, programmers can get used to it but it may be uncommon for beginners in reactive programming. Reactive streams also have a strong influence on unit tests. The JUnit test framework used does not wait for asynchronous method calls during a test case. As a consequence, assert statements inside of callback methods passed to the subscribe method of a reactive stream will not be evaluated. In order to solve this, some of the reactive stream frameworks like RxJava or Reactor offer testing suites which can be used to test the results of streams. Another way to achieve the same reaction is to use a blocking subscribe method on the reactive streams. These effectively wait until data arrives.

Hypothesis H2 is correct regarding the fact that reactive programming changes the software development process. Yet, whether this change can be seen as an improvement is probably more a matter of the individual programmer's taste.

8.2.3 H3: Software stability

"Reactive programming improves the stability of a software."

This hypothesis relates to the resiliency tests from chapter 6.2.2. The reactive Ticketsystem proved to handle the error case more gracefully than the conventional approach. Even though some of the processes had comparable durations, like the replacement time of the missing inventory service or the time the first user needed to finish the test, the response times of the reactive system were many times better than in the conventional approach and it required less time to return to a healthy state. In a system that relies on its customers and needs to deliver a flawless customer experience, fast response times are crucial. If some part of a system is faulty, it will almost always be possible to handle this appropriately and in a timely manner. Reactive programming and reactive streams help the developer focus on such parts of the system and give motivation to use patterns like circuit breakers as possible solutions.

In consequence, it can be said that hypothesis H3 has been proved by the resiliency tests with the reactive approach of the Ticketsystem.

8.2.4 H4: Error handling

"Reactive programming improves error handling."

Reactive streams offer a vast amount of operators to modify the stream or to trigger special behaviour based on various conditions. Many of these operators are dedicated to error handling. In Reactor streams they e.g. allow to map to a different `Mono` or `Flux`, when the current stream emits an error. Timeout operators can throw errors if a stream does not emit a value in a certain time. Circuit breakers can be used as an operator to create shortcuts to errors after multiple consecutive errors occurred. The streams also allow to add custom exception handler methods by passing the type to catch and a corresponding handler function. A stream can handle the same exception multiple times in different positions. As an example, one can imagine a stream which first needs to fetch data from a database and then map this data to another value. The fetching of the data and the mapping can fail and throw the same exception in an error case. In Reactor, such stream could be constructed as seen in listing 8.1.

```
database.getData()
  .doOnError(ExampleException.class, this::databaseErrorHandler)
  .map(this::someMappingFunction)
  .doOnError(ExampleException.class, this::mappingErrorHandler)
  .subscribe(this::dataHandler);
```

Listing 8.1: Reactive exception handling with the Reactor library. The same exception can be caught twice in a stream.

To achieve the same error handling without reactive streams, developers would be forced to use either nested try-catch statements as seen in listing 8.2 or consecutive try-catch statements as in listing 8.3. The number of lines compared to the reactive approach show that error handling is at least cleaner and therefore probably better readable in reactive streams than in conventional approaches. Additionally, the example with the nested exception handling might catch errors caused by the data handling method (which might not be intended), while the consecutive approach could run into null pointer exceptions when no additional care is taken.

```java
try {
  Object data = database.getData();
  try {
    Object mappedData = this.someMappingFunction(data);
    this.dataHandler(mappedData);
  } catch(ExampleException e) {
    this.mappingErrorHandler(e);
  }
} catch(ExampleException e) {
  this.databaseErrorHandler(e);
}
```

Listing 8.2: Nested exception handling with try-catch statements.

```java
Object data;
try {
  data = database.getData();
} catch(ExampleException e) {
  this.databaseErrorHandler(e);
}
Object mappedData;
try {
  mappedData = this.someMappingFunction(data);
} catch(ExampleException e) {
  this.mappingErrorHandler(e);
}
this.dataHandler(mappedData);
```

Listing 8.3: Consecutive exception handling with try-catch statements.

Similar to the second hypothesis the truth of this hypothesis can be partly influenced by the developer's taste and is probably depending heavily on the individual case. Still, as the example in this section shows, the error handling methods of reactive streams are inviting developers to use more error handling and it feels less like an obstacle during the whole development. At least this is the experience made during the implementation of the Ticketsystem. Following this explanation, the fourth hypothesis can be seen as partly true. Error handling might not be improved, but implementing error handling is less of an obstacle in reactive streams and allows a precise control of the data flow. Thus the assumption can be made that by using reactive streams there is a higher chance of implementing precise error handling - and that could indeed be seen as an improvement.

8.3 Completing the comparison

As already stated in chapter 2, conventional applications are not entirely the opposite of reactive and can also contain reactive elements. An example of this is the resiliency test, which showed, that even the conventional Ticketsystem is resilient in some way, although it is not able to compete with the reactive approach. Other reactive aspects such es elasticity, which includes the possibility to scale applications dynamically, are not only based on programming. The infrastructure around the software needs to support this. As the Ticketsystem has been run in containers on a Kubernetes cluster, this was possible, even in the conventional approach.

Of course, a reactive software needs to be developed towards these goals, but partly depends on its environment, which needs to be chosen adequately.

This project has shown that reactive programming and containerised environments fit together naturally and that in some aspects reactive programming outperforms conventional programming, while the approaches are equal or only slightly off in other situations. A strong impression left by reactive programming is that it is well suited towards good and thoughtful error handling and leads to a more streamlined view of the data flow in an application, while conventional programming proved that it does not automatically violate all aspects of reactive programming and that conventionally developed application should not immediately be dropped in favour of reactive programming, as it could be seen that these also run very successfully in containerised environments.

Looking back at the example of the Elbphilharmonie ticket shop in chapter 3.1, it can be assumed that it was not developed reactively. However, maybe a reactive approach would have improved the resiliency of the system and could have shortened or even prevented the outages.

9 Possible further research

The results and insights gained as part of this thesis may, as already stated in chapter 3.4.2, not, or not in their entirety, be applied to other scenarios where other languages, frameworks, environments or application types are involved. Based on what has been found out, this section will propose directions of possible further research, which may give interesting insights and might, combined, lead to a further understanding of the real and measurable differences and consequences of reactive and conventional programming in general.

9.1 Application types

As already mentioned in chapter 8.2.1, the Ticketsystem is a quite CRUD based application. It is possible that a comparison of reactive and conventional programming produces other results in different application types.

9.1.1 Not CRUD

The first idea is to perform similar tests on a non-CRUD based application, perhaps using CQRS with longer data processing pipelines which do more than retrieving data from a database, wrapping it in various objects and returning it to the end user. The different models and more processing could have an impact on how a reactive approach would behave and how reactive streams fit into the programming of such an application.

9.1.2 High performance or real-time applications

Another application type that could lead to interesting results are high performance applications or applications which need to do real time data processing. Examples for this could be the processing of live measurements and analysis of environmental data, such as pollution or weather data, to create precise warnings in real time. In such an application type, performance is most critical as a lot of processing has to happen in a short time. Furthermore, a lot of the work could probably be modelled with (reactive) streams and parallelised over many instances, which would work well together in containerised environments and microservices.

9.1.3 Online game servers

Servers for online games, especially massively multiplayer online games with several thousands of parallel online players need to perform many calculations based on the partly unpredictable behaviour of all players and change the virtual world's environment accordingly. These servers could suffer strongly varying loads depending on the time of the day and need to be highly available. As many of such games are paid for by subscriptions, players may have high expectations towards fast response rates, while downtimes are not acceptable and outages of parts of the game need to be handled quickly. It appears that such a scenario could be a good candidate to make a reactive vs. conventional comparison.

9.1.4 Image and video processing

One particularly calculation intensive approach is the processing of images and video data. Netflix does a lot of server side video processing in microservices to prepare their content for various streaming scenarios. Other use cases could also be imagined, like the rendering and pre-/post processing of animation movies. Today's animation movies take several million CPU hours to render (e.g. DreamWorks' *How To Train Your Dragon 2*[1] needed 90 million render hours [57]). These scenarios probably consume data from many sources and process it in many steps (e.g. rendering models, adding textures, adding illumination etc.) which could be broken down to a stream like data pipeline. Comparing the performance of such an application programmed with a reactive and a conventional approach could lead to interesting insights.

9.2 Programming languages

The Ticketsystem was programmed entirely based on the JVM with Java (a small exception is the payment service which was experimentally programmed in Kotlin in the reactive approach). It is possible that a project written in another programming language could result in different outcomes.

[1]How To Train Your Dragon 2: `https://en.wikipedia.org/wiki/How_to_Train_Your_Dragon_2`

9.2.1 Different JVM languages

A first approach could be to try out other JVM based programming languages. As Kotlin is currently gaining popularity and gets exclusive features in Spring like functional beans [58], a system written purely in Kotlin while taking full advantage of its features, could result in a very different comparison.

Another language to consider is Scala (which is also used to write Gatling tests). As an example, the Akka toolkit is written entirely in Scala and was originally created by Jonas Bonér who is the initiator and co-author of the reactive manifesto. It is possible that applications written in Scala perform differently. While leaving it with these examples there are also other JVM languages which could be tested.

9.2.2 Other programming languages

More interesting would probably be a reactive vs. conventional comparison in entirely different languages like for example Go, JavaScript in a NodeJS environment, or Swift. All these languages are currently gaining popularity and have various features which could result in different behaviours during such tests. Go, for instance, has very fast startup times and lightweight containers, which would definitely result in different measurements during a resiliency test similar to the one performed as part of this thesis. Apart from these examples, many other programming languages could be considered for different reasons with different results. The ReactiveX collection alone supports 17 different programming languages, which could be chosen from [59].

9.3 Architecture and environment

The chosen approach for this thesis was to create an application based on microservices in an containerised environment. But a comparison could also be made by changing some of the core architectural or environmental choices.

9.3.1 Different databases

As already hinted to in chapter 8.1.1 the replacement of the database or the database driver could at least change the results in applications with a high database access frequency. An idea would not only be to change the database to a different NoSQL database like, for instance, Couchbase, but also to change the type of database to e.g. a relational or a graph database. This approach would probably need a different type of application though, as the choice of database depends on the application that uses it.

9.3.2 Monolithic applications

Even though microservices are currently the *flavour of the month*, monolithic applications are still present and not in danger of extinction. As there is no network communication between the different parts of the software, method calls and timings are completely different to microservices. Additionally, the amount of resources is different. All these factors could lead to different results when comparing conventional and reactive programming in such a system. Resiliency would be interesting in such a system, though, as the outage of one instance of the software results in something comparable with the outage of one instance of each service in a microservice system. Scaling such an application would be less precise and could add a lot of unnecessary redundancy. Therefore, this approach would be interesting to examine. Furthermore, this approach is less optimised for containerised deployments, which leads to the next idea of static deployments.

9.3.3 Static deployments without containers

Deployments without containers, purely on virtual machines or dedicated servers without dynamic scaling could draw interesting results, mainly, but not exclusively in the resiliency tests. Scenarios in which one machine is drained of all its resources due to bad programming could also affect other applications running on the same system. The investigation of the effects of reactive programming in this scenario could therefore introduce useful results which could be of use for teams maintaining such applications without plans to move to other deployments but willing to improve the software or debate about migrating parts of the software to a reactive approach.

9.3.4 Low level environments

A final idea would be to develop low level software in a reactive and conventional fashion. Examples would be parts of an (or an entire) operating system, software that runs directly on micro controllers, operating system drivers for various hardware components or software that drives IoT devices. While not all aspects of reactive programming can be applied to this scenario, the usage of reactive streams and the asynchronous nature of reactive programming could be interesting to observe in tests developed for such a scenario.

List of Figures

Listings

List of Tables

List of abbreviations

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CQRS	Command Query Responsibility Segregation
CRUD	Create, Read, Update, Delete
GKE	Google Kubernetes Engine
HAW	Hamburg University of Applied Sciences
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
PVC	Persistent Volume Claim
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
URL	Uniform Resource Locator
VM	Virtual Machine

Bibliography

[1] C. Richardson, “Microservices Architecture pattern,” 2015. [Online]. Available: http://microservices.io/patterns/microservices.html (Accessed 2016-04-11).

[2] “The Reactive Manifesto.” [Online]. Available: https://www.reactivemanifesto.org/ (Accessed 2018-02-06).

[3] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices.* Pearson Education. ISBN 978-0-13-597444-5

[4] K. Henney, *97 Things Every Programmer Should Know.* ISBN 978-0-596-80948-5. [Online]. Available: http://shop.oreilly.com/product/9780596809492.do (Accessed 2016-07-03).

[5] W. Hasselbring and G. Steinacker, “Microservice Architectures for Scalability, Agility and Reliability in E-Commerce,” in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, Apr. 2017. doi: 10.1109/ICSAW.2017.11 pp. 243–246.

[6] M. Stine, *Migrating to Cloud-Native Application Architectures*, 1st ed. O’Reilly Media, Feb. 2015. ISBN 978-1-4919-2422-8. [Online]. Available: http://www.oreilly.com/programming/free/migrating-cloud-native-application-architectures.csp

[7] S. J. Fowler, *Production-Ready Microservices.* ISBN 978-1-4919-6597-9. [Online]. Available: http://shop.oreilly.com/product/0636920053675.do (Accessed 2017-02-07).

[8] S. Newman, *Building Microservices.* ISBN 978-1-4919-5035-7. [Online]. Available: http://shop.oreilly.com/product/0636920033158.do (Accessed 2016-05-21).

[9] “Documentation | Akka.” [Online]. Available: https://akka.io/docs/ (Accessed 2018-02-06).

[10] “Jonas Bonér.” [Online]. Available: http://jonasboner.com/ (Accessed 2018-02-06).

[11] “Reactive Manifesto 2.0 | @lightbend.” [Online]. Available: https://www.lightbend.com/blog/reactive-manifesto-20 (Accessed 2018-02-06).

[12] R. Kuhn, B. Hanafee, and J. Allen, *Reactive design patterns.* Shelter Island, New York: Manning Publications Company, 2017. ISBN 978-1-61729-180-7 OCLC: ocn910773340.

[13] “ReactiveX.” [Online]. Available: http://reactivex.io (Accessed 2018-02-06).

[14] “ReactiveX - Operators.” [Online]. Available: http://reactivex.io/documentation/operators.html (Accessed 2018-02-06).

[15] “spring.io.” [Online]. Available: https://spring.io/ (Accessed 2017-05-20).

[16] “Reactor 3 Reference Guide.” [Online]. Available: http://projectreactor.io/docs/core/snapshot/reference/ (Accessed 2018-02-06).

[17] “Amazon Web Services (AWS) - Cloud Computing Services.” [Online]. Available: https://aws.amazon.com/ (Accessed 2018-02-14).

[18] “Google Cloud Computing, Hosting Services & APIs.” [Online]. Available: https://cloud.google.com/ (Accessed 2018-02-14).

[19] “Basics Docker, Containers, Hypervisors, CoreOS EtherealMind.” [Online]. Available: http://etherealmind.com/basics-docker-containers-hypervisors-coreos/ (Accessed 2018-02-14).

[20] D. Bernstein, “Containers and Cloud: From LXC to Docker to Kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[21] “Docker.” [Online]. Available: https://www.docker.com/ (Accessed 2018-02-14).

[22] B. Butler, “Container party: VMware, Microsoft, Cisco and Red Hat all get in on app hoopla,” Sep. 2014. [Online]. Available: https://www.networkworld.com/article/2601925/cloud-computing/container-party-vmware-microsoft-cisco-and-red-hat-all-get-in-on-app-hoopla.html (Accessed 2018-02-14).

[23] B. Butler, “FAQ: Containers,” Sep. 2014. [Online]. Available: https://www.networkworld.com/article/2601434/cloud-computing/faq-containers.html (Accessed 2018-02-14).

[24] Docker swarm. [Online]. Available: https://docs.docker.com/engine/swarm/ (Accessed 2018-02-14).

[25] Rancher. [Online]. Available: https://rancher.com/ (Accessed 2018-02-14).

[26] D. K. Rensin, "Kubernetes - Scheduling the Future at Cloud Scale," 2015. [Online]. Available: https://research.google.com/pubs/pub43826.html (Accessed 2017-12-15).

[27] Containers, VMs, kubernetes and VMware. [Online]. Available: https://cloudplatform.googleblog.com/2014/08/containers-vms-kubernetes-and-vmware.html (Accessed 2018-02-14).

[28] Kubernetes. [Online]. Available: https://kubernetes.io/ (Accessed 2018-02-14).

[29] Most used languages among software developers globally 2017. https://www.stackoverflowbusiness.com/hubfs/content/2017_Global_Developer_Hiring_Landscape.pdf. [Online]. Available: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ (Accessed 2018-02-24).

[30] Netflix Technology Blog. How we build code at netflix. [Online]. Available: https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15 (Accessed 2018-02-24).

[31] Netflix open source software center. [Online]. Available: https://netflix.github.io/ (Accessed 2018-02-24).

[32] Netflix Technology Blog. The evolution of container usage at netflix. [Online]. Available: https://medium.com/netflix-techblog/the-evolution-of-container-usage-at-netflix-3abfc096781b (Accessed 2018-02-24).

[33] T. Berners-Lee, R. T. Fielding, and H. F. Nielsen, "Hypertext transfer protocol – http/1.0," Internet Requests for Comments, RFC Editor, RFC 1945, May 1996. [Online]. Available: https://tools.ietf.org/html/rfc1945

[34] H. True, "Sonderkonzerte nach 28 Minuten ausverkauft," Feb. 2017. [Online]. Available: https://www.abendblatt.de/hamburg/article209592791/Erneut-Probleme-beim-Ticketing-der-Elbphilharmonie.html (Accessed 2017-12-15).

[35] "Elbphilharmonie: Ticket-Chaos beim Vorverkauf," *Spiegel Online*, Jun. 2016. [Online]. Available: http://www.spiegel.de/kultur/musik/elbphilharmonie-website-nicht-erreichbar-a-1098639.html (Accessed 2017-12-15).

[36] P. Dakowitz, "Vorbereitung von Performancemessungen und Lasttests in Java-basierten Microservices," *Hochschule für Angewandte Wissenschaften Hamburg, Hauptprojekt*, December 2017.

[37] M. Fowler. CQRS. [Online]. Available: https://martinfowler.com/bliki/CQRS.html (Accessed 2018-02-17).

[38] MongoDB, Inc. MongoDB for GIANT Ideas. [Online]. Available: https://www.mongodb.com/index (Accessed 2018-02-18).

[39] Redis. [Online]. Available: https://redis.io/ (Accessed 2018-02-18).

[40] I. Haber, "15 reasons to use redis as an application cache." [Online]. Available: https://redislabs.com/wp-content/uploads/2016/03/15-Reasons-Caching-is-best-with-Redis-RedisLabs-1.pdf (Accessed 2018-02-18).

[41] Angular. [Online]. Available: https://angular.io/ (Accessed 2018-02-21).

[42] MongoDB reactive java driver. [Online]. Available: http://mongodb.github.io/mongo-java-driver-reactivestreams/ (Accessed 2018-02-21).

[43] R. Winkler. Resilience4j user guide. [Online]. Available: http://resilience4j.github.io/resilience4j/ (Accessed 2018-02-21).

[44] F. Montesi and J. Weber, "Circuit breakers, discovery, and API gateways in microservices." [Online]. Available: http://arxiv.org/abs/1609.05830 (Accessed 2017-05-29).

[45] "Capturing JVM- and application-level metrics. So you know what's going on," Dec. 2017. [Online]. Available: https://github.com/dropwizard/metrics (Accessed 2017-12-15).

[46] Maven metrics-core usages. [Online]. Available: http://mvnrepository.com/artifact/io.dropwizard.metrics/metrics-core/usages?p=1 (Accessed 2018-02-22).

[47] "RabbitMQ." [Online]. Available: https://www.rabbitmq.com (Accessed 2016-02-14).

[48] "Logstash." [Online]. Available: https://www.elastic.co/products/logstash (Accessed 2017-12-15).

[49] "Elasticsearch." [Online]. Available: https://www.elastic.co/products/elasticsearch (Accessed 2017-12-15).

[50] "Apache Lucene - Welcome to Apache Lucene." [Online]. Available: http://lucene.apache.org/ (Accessed 2017-12-15).

[51] "Kibana." [Online]. Available: https://www.elastic.co/products/kibana (Accessed 2017-12-15).

[52] "Gatling Load and Performance testing - Open-source load and performance testing." [Online]. Available: https://gatling.io/ (Accessed 2017-12-15).

[53] T. M. Jog, *Reactive Programming With Java 9: Build Asynchronous applications with Rx.Java 2.0, Flow API and Spring WebFlux.* Packt Publishing - ebooks Account, 2017. ISBN 1787124231

[54] Apache Cassandra. [Online]. Available: http://cassandra.apache.org/ (Accessed 2018-03-09).

[55] NoSQL engagement database | couchbase. [Online]. Available: https://www.couchbase.com/ (Accessed 2018-03-09).

[56] Grafana - the open platform for analytics and monitoring. [Online]. Available: https://grafana.com/ (Accessed 2018-03-09).

[57] The making of DreamWorks animation's tech wonder: How to train your dragon 2. [Online]. Available: https://venturebeat.com/2014/07/25/dragon-making-main/ (Accessed 2018-03-10).

[58] S. Deleuze. Spring framework 5 kotlin APIs, the functional way. [Online]. Available: https://spring.io/blog/2017/08/01/spring-framework-5-kotlin-apis-the-functional-way (Accessed 2018-03-10).

[59] ReactiveX - languages. [Online]. Available: http://reactivex.io/languages.html (Accessed 2018-03-10).

Index

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

I hereby declare that I wrote this work independently and unassisted and only used the specified sources of help.

Hamburg, 27. März 2018 Philip Dakowitz