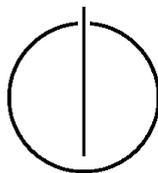


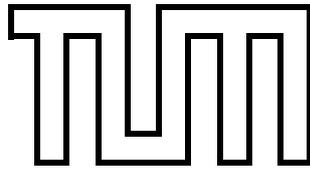
FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

**Migration of Artemis'
Architecture from Monolithic
to Microservices**

Merlin Mehmed





FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Migration of Artemis' Architecture from
Monolithic to Microservices

Migration der Architektur von Artemis vom
Monolithen zu Microservices

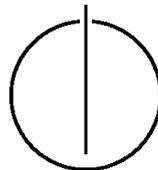
Author: Merlin Mehmed

Supervisor: Prof. Dr.-Ing. Pramod Bhatotia

Advisors: M.Sc. Evgeny Volynsky

Prof. Dr. Stephan Krusche

Date: 15.12.2021



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.12.2021

Merlin Mehmed

Acknowledgements

I would like to express my gratitude to everyone who has supported me during this thesis.

First, I want to thank my advisors Evgeny Volynsky and Prof. Dr. Stephan Krusche, for giving me the opportunity to write my thesis and always guiding me. This thesis was a challenge for me, but I had the chance to gain knowledge in fields in which I had little experience. Thanks to my advisors' feedback, I also learned how to manage and organize my tasks more efficiently and communicate more clearly. Both of them are people whom I will always respect.

I would also like to thank the Artemis development team, who helped me with many code reviews. We had a great time, and I wish them all great success in their future endeavors. They were always so kind and willing to help.

Last but not least I want to thank my family - my brother and my parents, my boyfriend, and my friends. They always supported me and sent me positive energy.

Abstract

Artemis is an open-source learning platform that uses a monolith architecture for its server application. This architecture is beneficial at the beginning of a project but causes problems in large projects. First, when developers introduce code changes, they need to build the whole application no matter how significant the difference is. This results in slow build and test phases. Therefore, developers need to wait for them to complete until they can deploy their changes, slowing down the development process as a whole. Another drawback of this architecture is inefficient scaling. Artemis supports horizontal scaling by running several Artemis instances to handle the higher load during real-time quiz exercises or exams. The current implementation does not allow scaling only part of the application, leading to inefficient use of computational resources.

This thesis sets the foundations of the migration of the Artemis architecture towards microservices. It helps speed up the development process and support efficient building, testing and scaling. It discusses the new architecture and the roles of its new components. It also describes the services extraction process from the Artemis monolith and updates to the deployment pipeline.

During the thesis, we extract two microservices from the monolith. This change allows scaling those services independently. The services have a small size, therefore, the build and test processes run fast, leading to shorter waiting times. It also has a positive effect on the availability of the system. Failure in one service will not cause failure in other services. Additionally, we create Kubernetes deployment resources for the new architecture to suit the future deployment process. Deploying on a Kubernetes cluster makes scaling and failure handling much easier as it provides autoscaling and containers self-healing.

Zusammenfassung

Artemis ist eine Open-Source-Lernplattform, die eine monolithische Server-Architektur hat. Diese ist in den Anfangsphasen eines Projekts von Vorteil, verursacht jedoch bei größeren Projekten Probleme in späteren Phasen. Entwickler müssen die ganze Anwendung erstellen, wenn sie Codeänderungen vornehmen, egal wie groß die Änderung ist. Dies führt zu langsamen Erstellen- und Testphasen. Daher müssen Entwickler jedes Mal warten, bis die Phasen abgeschlossen sind, und erst dann können sie ihre Änderungen bereitstellen, was den Entwicklungsprozess verlangsamt. Ein weiterer Nachteil dieser Architektur ist die eingeschränkte Skalierbarkeit. Artemis lässt sich horizontal skalieren, indem mehrere Artemis-Instanzen ausgeführt werden, um die höhere Auslastung bei Echtzeit-Quiz-Übungen oder Prüfungen zu bewältigen. Die aktuelle Implementierung erlaubt es nicht, nur einen Teil der Anwendung zu skalieren, was zu einer ineffizienten Nutzung von Rechenressourcen führt.

Diese Thesis legt die Grundlage für die Migrationsprozesse der monolithischen Architektur hin zu Microservices. Die neue Anwendungsarchitektur beschleunigt den Entwicklungsprozess und bietet effizientes Erstellen, Testen und Skalieren. Die Arbeit stellt die neue Architektur und ihre neue Komponenten vor. Sie beschreibt auch den Service-Extraktionsprozess aus dem Artemis-Monolithen und wie die Bereitstellungs pipeline aktualisiert wird.

In der Thesis wird vorgelegt, wie zwei Microservices aus der Anwendung extrahiert werden. Dadurch können diese zwei Elemente unabhängig skaliert werden. Sie haben eine geringe Größe, und so laufen die Erstellen- und Testprozesse schnell, was zu kürzeren Wartezeiten führt. Das wirkt sich auch positiv auf die Verfügbarkeit des Systems aus - in Ausfall von der einem Service-Anwendung führt nicht zum Ausfall der Anderen. Zusätzlich werden Kubernetes-Bereitstellungsressourcen für die neue Architektur erstellt. Die Bereitstellung auf einem Kubernetes-Cluster vereinfacht die Skalierung und die Fehlerbehandlung, da es Autoscaling und Container-Selbstheilung bietet.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 2 |
| 1.1 | Problem | 2 |
| 1.2 | Motivation | 4 |
| 1.3 | Objectives | 4 |
| 1.3.1 | Define a migration process | 5 |
| 1.3.2 | Identify microservices | 5 |
| 1.3.3 | Migrate two microservices | 5 |
| 1.3.4 | Deploy on Kubernetes | 5 |
| 1.3.5 | Implement a pattern related to microservices | 6 |
| 1.4 | Outline | 6 |
| 2 | Background | 7 |
| 2.1 | Software Architectures | 7 |
| 2.1.1 | Monolithic Architecture | 7 |
| 2.1.2 | Microservices Architecture | 9 |
| 2.2 | Microservices Patterns | 11 |
| 2.2.1 | API Gateway | 11 |
| 2.2.2 | Shared Database | 12 |
| 2.3 | JHipster | 12 |
| 2.4 | JHipster Registry | 12 |
| 2.5 | ActiveMQ Artemis | 13 |
| 2.6 | Docker | 14 |
| 2.7 | Kubernetes | 14 |
| 3 | Related Work | 16 |
| 3.1 | WETO and Plussa | 16 |
| 3.2 | Netflix | 17 |
| 3.3 | Zalando | 18 |
| 3.4 | Conclusion | 18 |

| | | |
|----------|---|-----------|
| 4 | Requirements Analysis | 19 |
| 4.1 | Current System | 19 |
| 4.2 | Proposed System | 21 |
| 4.2.1 | Functional Requirements | 22 |
| 4.2.2 | Nonfunctional Requirements | 23 |
| 4.3 | System Models | 24 |
| 4.3.1 | Scenarios | 24 |
| 4.3.2 | Dynamic Model | 25 |
| 5 | System Design | 27 |
| 5.1 | Overview | 27 |
| 5.2 | Design Goals | 27 |
| 5.3 | Subsystem Decomposition | 29 |
| 5.3.1 | Service Registry | 29 |
| 5.3.2 | API Gateway | 30 |
| 5.3.3 | Message Broker | 31 |
| 5.3.4 | User Management Microservice | 33 |
| 5.3.5 | Lecture Microservice | 34 |
| 5.3.6 | Artemis Application Server | 35 |
| 5.4 | Hardware Software Mapping | 35 |
| 5.5 | Persistent Data Management | 36 |
| 5.6 | Boundary Conditions | 37 |
| 5.6.1 | Application Startup | 37 |
| 5.6.2 | Application Shut Down | 38 |
| 5.6.3 | Failure Handling | 38 |
| 6 | Migration to Microservices | 40 |
| 6.1 | Migration Strategy | 40 |
| 6.2 | Decision on the Architecture | 41 |
| 6.2.1 | Artemis as a Gateway | 41 |
| 6.2.2 | Artemis Application in front of the API Gateway | 42 |
| 6.2.3 | Artemis server as an independent application | 44 |
| 6.2.4 | Conclusion | 46 |
| 6.3 | Decomposition into microservices | 46 |
| 6.3.1 | Creation of an API Gateway | 46 |
| 6.3.2 | Extraction of User Management Microservice | 47 |
| 6.3.3 | Extraction of the Lecture Microservice | 50 |
| 6.3.4 | Microservice Extraction Steps | 51 |
| 6.4 | Deployment | 57 |
| 6.4.1 | Kubernetes Deployment | 57 |
| 6.4.2 | Virtual Machine Deployment | 59 |

| | | |
|----------|---|-----------|
| 6.5 | Discussion | 61 |
| 6.5.1 | Findings | 61 |
| 6.5.2 | Limitations | 61 |
| 7 | Summary | 63 |
| 7.1 | Status | 63 |
| 7.1.1 | Realized Goals | 64 |
| 7.1.2 | Open Goals | 66 |
| 7.2 | Conclusion | 66 |
| 7.3 | Future Work | 67 |
| 7.3.1 | Continue with the Microservices Extraction | 67 |
| 7.3.2 | Production Kubernetes deployment | 68 |
| 7.3.3 | Migrate to micro frontends | 69 |
| 7.3.4 | Research the Availability of the Message Broker | 70 |

GUI Graphical User Interface
JSON JavaScript Object Notation
JWT JSON Web Token
REST Representational State Transfer
HTTP Hypertext Transfer Protocol
LMS Learning Management System
UML Unified Modeling Language
JMS Java Messaging Service
JDBC Java Database Connectivity
PR Pull Request
IP Internet Protocol
PVC Persistent Volume Claim
URL Uniform Resource Locator

Chapter 1

Introduction

Software architectures represent the structure and the behaviour of a system. Therefore, they are essential for each software and the success of the system.

Artemis is an open-source web application for interactive learning. It provides automated assessment tools which help to decrease the assessment efforts of the tutors [KS18]. Multiple universities actively use Artemis for their courses, some of which have more than 1000 participants¹. The current architecture of Artemis is monolithic. Therefore, a single large application incorporates all functionalities. However, Artemis has grown and changed over time, resulting in a need for a different architectural style.

This thesis sets the foundation for migrating the architecture of Artemis from monolith to microservices. This change will solve problems that the current architecture causes and will make further development easier.

1.1 Problem

The monolithic architecture deploys the code as a single process. There might be multiple instances of this process deployed for scaling reasons, but fundamentally all the code is packaged into a single process [New19].

This architectural style is excellent at the beginning of a project, but it becomes problematic as it grows. We can improve several complications caused by the current architecture.

Slow Build

The build of the Artemis application includes both the client and the server applications as a single WAR file. Then we can deploy the WAR file on a

¹<https://github.com/ls1intum/ArTEMiS>

server. Therefore, when a developer makes a change and wants to test his implementation on a test server, he needs to wait for the build to complete until he can deploy his changes. Currently, the build of Artemis in a Bamboo build agent continues for around 8 minutes. Therefore, developers need to wait for the build to deploy their changes, which slows down the development process. Suppose there is a bug that occurs only on test and production environments, and developers cannot reproduce the issue on their development environments. In that case, they need to test their implementation on the test servers. Therefore, they need to wait for 8 minutes for each change they do until the build finishes to deploy and test their changes. Fixing this bug will be highly inefficient and annoying for the developer.

Slow Server Tests Execution

An essential goal for Artemis developers is to keep the test coverage as high as possible to find early regressions in the new feature implementation. As a result, there are numerous unit and integration server tests which the Bamboo agents execute for more than 20 minutes. The Artemis developers develop new features in draft pull requests in the GitHub repository² of Artemis. Before labelling their PR as "ready for review", the developers need to ensure that all the tests pass and there are no regressions in the implementation. This leads to additional waiting, which also slows down the development process.

Scaling

More than a thousand students, their tutors, and instructors use Artemis during the semester period. The live quiz exercises and exam features supported by Artemis increase the load of the servers during such events.

In order to handle the load, the Artemis team has introduced support for several Artemis instances. Almost all universities hold their courses online since COVID-19 has been the reason to switch to online education. Therefore, the load on Artemis has also increased because many instructors have decided to use Artemis for their courses. According to statistics, Artemis conducted around thirty exams at TUM in the last two semesters.

The live quiz and exam features receive more traffic than other features. Different components of Artemis have different scaling requirements. Although only several components are experiencing higher load, the whole monolith needs to be scaled [DLL⁺18].

²<https://github.com/lslintum/ArTEMiS>

1.2 Motivation

The microservices architectural style solves the problems mentioned above. It is an approach for developing a single application as a suite of small and loosely coupled services, each of which runs in its own process and communicates with lightweight mechanisms [PMA19]. Microservices are built around business capabilities and are independently deployable by fully automated deployment pipelines [PMA19]. Their size helps for their easier maintainability and make them more fault-tolerant since if one of them fails, it will not break the whole system, which could happen with a monolithic architecture [PMA19]. Instead, the application will be still up and running, which leads to a decrease in the application's overall downtime. Therefore, this style allows designing architectures that are flexible, modular and easy to evolve [PMA19].

Compared to the monolithic architecture, the different services are separate applications. Each of them is responsible for particular business logic. Thus, developers are responsible for maintaining one or several services, and there is no need to have context on the whole codebase anymore. Furthermore, the build and test phases are fast since the scope of the application is small, and the build and tests executions are independent of unrelated features. Moreover, we can scale services independently of other services. We can scale only the services that many users will use simultaneously and not the whole application.

1.3 Objectives

The main objective of this thesis is to define and start with the migration process of Artemis from a monolithic architecture to microservices.

In order to achieve this, we will try to accomplish the following main objectives:

1. Define a migration process
2. Identify microservices
3. Migrate two microservices
4. Deploy on Kubernetes
5. Implement a pattern related to the microservices architectural style

1.3.1 Define a migration process

There are many ways to refactor an application to microservices. The first objective is to research migration patterns and define the best migration process in the case of Artemis.

1.3.2 Identify microservices

The current monolith architecture includes many different services in one large application. In order to be able to migrate Artemis to microservices, we need to identify small and loosely coupled services within the current application, which we can later implement as separate microservices.

1.3.3 Migrate two microservices

The main objective is to migrate two of the microservices successfully. We will do it by extracting them from the monolith. We will extract only two microservices due to the complexity of the task and time reasons. Since we also need to define the overall architecture, there will not be enough time to extract more microservices.

The goal is to define the two microservices as standalone services that work as a single system together with the monolith. We will also document the results in order to help the migration of the rest of the microservices.

1.3.4 Deploy on Kubernetes

Once we finish migrating the services, we would like to deploy the new architecture on Kubernetes.

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services that facilitate declarative configuration and automation. It has a large, rapidly growing ecosystem. Thus, Kubernetes services, support, and tools are widely available.

Kubernetes provides service discovery and load balancing. It can load balance and distribute the network traffic so that the deployment is stable. It also provides self-healing, which means it restarts containers that fail, replaces containers, kills containers, and makes them unavailable to the clients until they are ready to serve. What is more, it supports automatically mounting of storage systems, such as local storage, public cloud providers, and much more.³

³<https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

1.3.5 Implement a pattern related to microservices

There are many patterns related to the microservices architectural style. They are related to different fields - Data management, Transactions, Testing, Deployment. We want to implement one of those patterns since they solve the challenges that the microservices architecture introduces.

1.4 Outline

This thesis is structured as follows:

Chapter 2: Background describes concepts and technologies that are important for the thesis. It introduces the difference between the monolith and microservices architectural styles and design patterns and technologies used in the thesis.

Chapter 3: Related Work describes how other projects have migrated towards microservices and how their work can help us.

Chapter 4: Requirements Analysis discusses the current and the proposed systems and the requirements of the new architecture.

Chapter 5: System Design introduces the new architecture, its components and their responsibility.

Chapter 6: Migration to Microservices describes the whole migration process and the executed steps.

Chapter 7: Summary summarizes the work we did - the achieved and the open goals, as well as a conclusion for the architecture migration and the future work remaining on this topic.

Chapter 2

Background

This chapter introduces in detail essential for the thesis concepts and technologies. We first compare the Monolithic and the Microservices architectural styles, including their benefits and drawbacks. After this, the chapter introduces microservices patterns we implemented in the thesis. In the end, we introduce details about different technologies that we used in the thesis implementation.

2.1 Software Architectures

There are many definitions for software architecture. Some architects refer to software architecture as the system's blueprint, while others define it as the roadmap for developing a system [RF20]. Architecture decisions define the rules for how to construct a system [RF20]. As said before, the need for a change in the already defined and implemented software architecture may occur after the application has gradually evolved from its initial state. Architecture matters because of how it affects the so-called quality of service requirements, also called nonfunctional requirements and quality attributes [Ric19]. There are many different architectural styles, but two are the ones that are important for us, which we describe in the following subsections.

2.1.1 Monolithic Architecture

Software application with a monolith architecture works as a single, self-contained unit [Ing18]. However, those application types are widely spread. They include interconnected and interdependent components, which results in tightly coupled code [Ing18].

We base the following benefits and drawbacks sections on the ideas and the examples in the *Microservices Patterns* book by Chris Richardson [Ric19].

Benefits

There are many benefits in the early days of a monolithic application when the application is relatively small. Some of them include the following:

- Simple to develop
Developers are focused on building a single application. There is a single code repository, and there is no need to navigate across different projects. It is clear for the developers where to add the new code and how to test it.
- Straightforward to test
It is relatively easy to develop integration tests that invoke the REST API and check the returned result.
- Straightforward to deploy
A single deployment artifact includes the application, which is easy to deploy in the production environment.

All of those characteristics eventually become a drawback over the time when the application grows.

Drawbacks

- Complex and slow to develop
As the application becomes larger, so does the codebase, which makes it hard for new developers to understand. In the end, there will be only a few developers who understand the whole application, which has several unfavourable effects, among which are resolving bugs and proper implementation of new features.
- Harder to test
The testing of the application becomes more challenging when there is a dependency between different features that wrongly depend on each other. In those cases, testers define test data for objects related to features unrelated to the one they want to test. Also, if developers upgrade essential for the project dependencies, testers might have to retest the whole application, taking them lots of time.

- Longer to develop and deploy

The development and deployment is also affected since the build process takes longer in a complex application. The usual style checks and running tests also take longer, which slows down the development and deployment processes.

- Difficult to scale

It is not possible to scale part of the application, which can lead to using too many resources rather than scaling only the part which receives more traffic.

2.1.2 Microservices Architecture

Microservices architecture is an architectural style that has gained significant importance in recent years. It builds small, autonomous, independently versioned, self-contained services [Ing18]. These services use well-defined interfaces and communicate with each other over standard, lightweight protocols [Ing18].

We base the following benefits and drawbacks sections on the ideas and the examples in the Microservices Patterns book by Chris Richardson [Ric19].

Benefits

- Services are small and easy to maintain

Each service is relatively small, which increases the performance of the developer. The codebase is small. Therefore, it is easier to find bugs. Also, the application starts faster than the one of a massive monolith. All those factors speed up the development time and the performance of the developer.

- Services are independently deployable

If one service changes, we can publish the changes by deploying only this service and eventually the other services affected by the change. Ideally, the changes will spread over a single service, but it is also a typical case that they involve several services.

- Independently scalable services

We can scale only services that require it when the need for scaling is recognised. In the case of horizontal scaling, the number of instances for different services may vary depending on the needs.

- Allows easy adoption of new technologies

It is possible to implement services using different technologies. If the team wants to introduce new technology, they can migrate an existing service or create a new one using it without the need to change the whole application. This way, they can evaluate that technology with a lower risk. It also helps when requirements are to implement specific features using a particular programming language.

- Allows autonomous teams

The whole development team can split into smaller teams, each responsible for specific service(s). Each team will have expertise in their microservice(s), which will also increase the performance of the developers. If teams are separated, the different teams should continue working together closely, being aware of critical changes.

Drawbacks

- It is challenging to find the right set of services

There is still no concrete algorithm for decomposing a system into microservices. Therefore, it is challenging to find the right services of the correct size. Poorly designed boundaries can lead to increased network communication [JPM⁺18]. A wrongly decomposed system will lead to the distributed monolith with highly coupled services. As a result, that system will have the drawbacks of both the monolith and the microservices architecture.

- The system is more complex

Distributed systems add additional complexity for the developers. They need to be aware of the different services and their responsibility. Communication between services, also known as inter-service communication, is more complex and requires additional implementation. Testing such communication is also more complex than testing a typical method call.

- Deploying features that span multiple services require additional coordination

Some changes will spread over more than one service, which will require additional coordination in the development and deployment. In addition, we need to deploy all affected services according to their dependencies, which requires a thoughtful deployment plan.

2.2 Microservices Patterns

Patterns are known as solutions to commonly occurring problems. The following two sections discuss two important patterns for the microservices architecture.

2.2.1 API Gateway

The API Gateway pattern is one of the most common patterns used in a microservices architecture. It is a component used by all clients to interact with the server. For this reason, it has to be highly available and scalable. The API gateway is the entry point for the server application and is like the Facade pattern. Similar to it, it encapsulates the application's internal architecture and handles all client requests [Ric19]. In addition, it has several essential functions, which we discuss below.

Request Routing

The API gateway's most important function is request routing. Every request is routed to the corresponding server application using a routing map. For example, the routing map may map the HTTP request type, and path to the HTTP URL [Ric19].

API Composition

Another important function is API composition. In some cases, an operation may require access to several microservices. Using API composition, the client makes a single request to the API gateway, creating several requests to the required microservices. After responses are delivered, the result is composed in a single response returned to the client.

Benefits

- The gateway encapsulates the internal structure of the server application, and the client send requests to a single entry point.
- Enhances the security of the overall architecture because it screens all incoming requests for security [SD20].

Drawbacks

- It has to be highly available and scalable.

- There is a risk that it becomes the bottleneck of the architecture.

The API gateway is part of the new Artemis architecture. It will be the entry point for the server part of the application.

2.2.2 Shared Database

Another important pattern for the thesis is the Shared Database pattern. Artemis is a large application that is important for its users. Migrating to microservices is a vast and risky task. In order to reduce the risk, we decided to use shared by the services database. Usually, microservices architecture suggests that each service has its database. In some cases, though, multiple services share the same database. This design pattern is suitable for a smaller amount of services from two to four when they share similar database tables [Sri21].

This pattern comes in great use during migration to microservices. It is hard for some monolithic applications to split the database during the migration process directly. Therefore, they initially use a shared database and split it in a later phase [Sri21].

Although a shared database is easy to implement since it is easy to integrate with the extracted services, it is a reason for higher coupling between them.

2.3 JHipster

JHipster is a free and open-source development platform that we can use to quickly generate, develop, and deploy web applications¹. It also supports microservices architectures enabling the generation of a gateway and microservice projects. Artemis is developed using JHipster, and we would like to continue using it with the microservice architecture. JHipster, which comes from Java Hipster, saves a significant amount of time providing the project configurations by its generator. It takes only a couple of minutes to create an end to end application using it.

2.4 JHipster Registry

The service registry is an essential aspect of a microservices architecture. JHipster provides its open-source implementation for a service registry, called

¹<https://www.jhipster.tech/>

JHipster Registry. It is a Java application consisting of a Netflix Eureka and a Spring Cloud Config Server [SK20].

Netflix Eureka is a client-server application. It locates services and keeps track of their count and status. Each service registers itself to the Eureka server on application startup and sends its heartbeat [SK20]. Once the service stops sending its heartbeat, the registry removes it from its database.

The Spring Cloud Config Server provides run-time configuration to all services. Services are dynamic in a microservice architecture. Depending on the traffic or other configurations, they will be started and then stopped [SK20]. For this reason, there is a need for a highly available server that holds configurations that services need to share [SK20]. The Spring Cloud Config Server provides configuration values to all services. For example, this includes the JWT secret value used to create the user tokens, which is very important for the authentication between the applications.

The JHipster Registry also serves administration purposes by providing administration dashboards to monitor and manage the registered applications, including the gateway and the services².

2.5 ActiveMQ Artemis

Microservices architecture always requires communication between the application. It can be both synchronous or asynchronous. Synchronous messaging is easy to understand since it uses HTTP thread blocking request-response calls. According to the ActiveMQ Artemis landing page³, this type of communication provides good latency in low throughput use-cases but may cause problems with scaling. Artemis supports large courses, including more than thousand students. All of them should be able to use it during lectures with no performance issues. Here comes the use of asynchronous communication. One microservice can request another without waiting for its response. The request will be placed in a message broker and consumed by the application. The good news here is that if the consumer is down for some reason, the message will not be lost and consumed once the application is up again. Artemis is already using ActiveMQ Artemis as a message broker. It is an open-source project under the Apache License. It supports asynchronous messaging with high performance. The ActiveMQ Artemis documentation states that its potential throughput is millions of messages per second and that it has the performance and feature-set to bring these gains to the applications.

²<https://www.jhipster.tech/jhipster-registry/>

³<https://activemq.apache.org/components/artemis/>

2.6 Docker

The current section about Docker refers to the official Docker documentation⁴.

Docker is an open platform for developing, shipping, and running applications. It enables the separation of applications from their infrastructure to achieve quick software delivery. Docker helps to significantly reduce the delay between writing code and running it in production.

Docker runs an application in a loosely isolated environment called a container. This isolation and security make it possible to run many containers simultaneously on a given host. In addition, the container itself is lightweight and contains everything needed to run the application, and does not depend on installed technologies on the host.

Docker supports responsive deployment and scaling. The containers can run on a developer's machine, on physical or virtual machines in a data center, and on cloud providers. In addition, the lightweight and portability of the containers make it possible to quickly manage workloads scaling up or down applications in real-time.

Moreover, Docker enables the running of several workloads on the same hardware, which is possible due to the lightweight and isolation of the containers. This will reduce the cost and make it possible to save resources.

Docker simplifies running and deploying microservices. Docker images package each service, which allows deployment on a Kubernetes cluster.

2.7 Kubernetes

The current section refers to the official Kubernetes documentation⁵.

Kubernetes is an open-source platform for orchestration containerized workloads and services. It helps to automate deployments, scale, and manage the applications. It takes care of the scaling and the failure handling of the application. If a container goes down, it can start automatically another one. It also provides many different features, one of which is load balancing. If the traffic to a container is high, it can distribute the traffic to keep the deployment stable.

In order to deploy, we need a Kubernetes cluster. A cluster consists of a collection of hosts, also called nodes, that run containerized applications. There is at least one node in a cluster. The nodes run pods which are the smallest units of work in Kubernetes [Say17]. Each pod contains one

⁴<https://docs.docker.com/get-started/overview/>

⁵<https://kubernetes.io/docs/home/>

or more containers [Say17]. Pods share the same IP address, port space, and local storage. They can communicate using localhost or inter-process communication [Say17].

Kubernetes goes hand in hand together with microservices deployment. It is because Kubernetes enables the high availability of the application. Its self-healing property by automatically restarting failed containers and automatic scaling features help for the high availability that Kubernetes offer [AVSTK18].

On the other side, it takes time to set up and configure a cluster. Since Kubernetes provides such an extensive feature set, it has a steep learning curve. Therefore, it is complex to learn and fully understand its capabilities.

Chapter 3

Related Work

There is a large amount of research on microservices. For this chapter, we focused on projects which have migrated to microservices. Unfortunately, there is not enough information about the migration of learning management systems. The first section describes the migration research of a university learning management system from Tampere University in Finland. It is not clear whether they have implemented the migration, but we have mentioned it in the related work because of the services decomposition they propose. The other examples are not related to learning management systems but are projects that have successfully implemented the migration to microservices.

3.1 WETO and Plussa

WETO, an acronym for Web Teaching Organizer, is a learning management system (LMS) developed by Tampere University in Finland. As per [NH19] which is a paper related to the migration of WETO towards a microservices architecture, its core functionalities are:

- Basic content management - creating and editing course pages, where images and files may be uploaded or linked.
- Student submission management: managing students' homework or exam submissions.
- Grade management: managing grading criteria or rules and student grades.
- Automated grading: automatic grading of the student submissions using a built-in implementation for multi-choice questions and external grader for programming tasks.

- Peer-reviewing: anonymous peer-reviewing of submissions
- Basic discussion forum: discussion feature where teachers and students can communicate issues or questions.

Plussa is another LMS responsible for authentication, grades storage, and student code evaluation.

In the paper from P.Niemelä et al. [NH19] migration of learning management systems from monolith to microservices is reviewed, taking WETO and Plussa as starting points. They identify the following core microservices: authentication, user management, course information, grading and reports, and analytic services [NH19].

Finally, they conclude that there are many benefits from the microservices architecture. However, it includes many obstacles in the realization path, which requires much work, good will and co-operation [NH19].

3.2 Netflix

We have all heard or used the services Netflix provides. Statistics from October 2021 state that Netflix has more than 210 million subscribers¹. It is clear that this amount of users generate huge traffic per day which require smart solutions. Netflix is known as one of the pioneers in microservices. The Netflix team has started to move their Monolithic architecture to Microservices Architecture in 2009. Back then, Netflix was only DVD rental company. When operating on a monolithic architecture, they have had constant server outages. [SGP19]

Currently, Netflix has more than 1000 microservices, each of which manages a separate part of the application². The entire migration has happened for around 2-3 years. Netflix moved their architecture to microservices and managed to open source many of the tools they have built and used for their new architecture, which we all take advantage of. This is why we cannot go without mentioning the work the team behind Netflix has done. They took the risk to migrate their architecture to microservices which was considered a crazy move but turned out to be a game-changer².

¹<https://www.statista.com/statistics/250934/quarterly-number-of-netflix-streaming-subscribers-worldwide/>

²<https://www.geeksforgeeks.org/the-story-of-netflix-and-microservices/>

3.3 Zalando

Zalando is a fashion store company that sells fashion products in several countries. They started as a shoe selling company and grew quickly, reaching a point when the system could not handle the load. They also faced developers' productivity issues since many people worked on the same large codebase. Another issue they faced was with new people coming to the team. They have had difficulty getting confident with the code because it has been too big.³

They have solved those issues by migrating towards microservices. However, they have faced several challenges during the process. The main challenge is a change in the team mindset both on the engineering and leadership side. Developers need to build resilient systems, especially in a microservices architecture. They need to make sure that they do not break dependant services and what should happen when the dependant service s not available. They solved the challenge by giving end-to-end responsibility to the developers - developing, testing, and operating what they have built.

3.4 Conclusion

Many projects have adopted microservices. Therefore, there are many examples of projects that have migrated their architecture, and it does not make sense to mention more. However, there is not enough information on migrating learning management systems which will be the contribution of this thesis.

³From Monolith to Microservices at Zalando, <https://www.youtube.com/watch?v=gEeHZwjwehs>

Chapter 4

Requirements Analysis

This chapter follows the structure of the Requirements Analysis Document Template in [BD10]. It analyzes the current system and discusses the requirements that need to be fulfilled by the new Artemis architecture described as the proposed system.

Section 4.1 describes the current state of the system and its architecture, while Section 4.2 proposes a new system architecture and presents its requirements. Finally, Section 4.3 describes scenarios and dynamic models to clarify the requirements using modeling techniques presented in [BD10].

4.1 Current System

The current Artemis server application is a monolith application. The server architecture uses the three-tier architectural style, as shown in Figure 4.1. As of the Object-Oriented Software Engineering Using UML, Patterns, and Java book by Bernd Bruegge and Allen Dutoit, the three-tier architectural style organizes the subsystems into three layers - interface, application, and storage. The interface layer, in our case, the web layer, includes all boundary objects that deal with the user, which is the client application. The application logic layer includes all control and entity objects, realizing the processing required by the application. Furthermore, the storage or data layer realizes the storage, retrieval, and query of persistent objects [BD10].

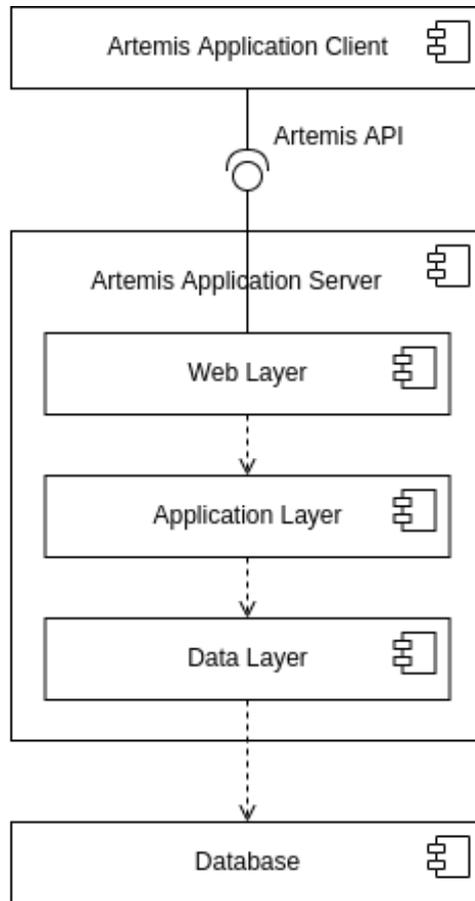


Figure 4.1: Simplified UML component diagram showing the current server architecture of Artemis. The figure is adapted from the component diagram in the GitHub Artemis documentation¹

Currently, we build the Artemis client and server in one WAR file. Bamboo is responsible for building the application. Once the build finishes, we can deploy it on one of the test, staging, or production servers. We deploy the Artemis application as a single process. This means that we have to redeploy the whole application when we want to redeploy it. Furthermore, we have to scale the whole application when we want to scale. Therefore, independent deployment and scaling are not possible.

¹<https://github.com/ls1intum/ArTEMiS>

4.2 Proposed System

Figure 4.2 illustrates the proposed architecture. It decomposes the current server application and extracts two microservices from it. The first of them is the User Management microservice which is responsible for operations on users - creating, updating, deleting, and searching them. The second microservice is the Lecture service which is responsible for managing lectures and their content.

The reason behind choosing those two microservices is that they are not so deeply integrated into the Artemis application. Therefore, they must be easy to extract because they do not have high coupling with other features. Furthermore, we think that it is beneficial to start with the extraction of loosely coupled services because this will reduce the complexity of a task that is already complex enough. Since we set the foundation of the microservices architecture, we have to extract two microservices and also add additional components to the new architecture. Moreover, we will learn about microservices extraction, which will later help us use the experience we have gained to extract more complex microservices.

The new architecture also includes three additional components required by the microservice architecture. The first of them is the gateway which serves as a single entry point to the microservices. Another component is the service registry which registers all service instances and the gateway. The latter uses the registry for retrieving the location of the services. Finally, we have the message broker, which we use for inter-service communication between different services to exchange data. The microservices and the Artemis server application share the same database.

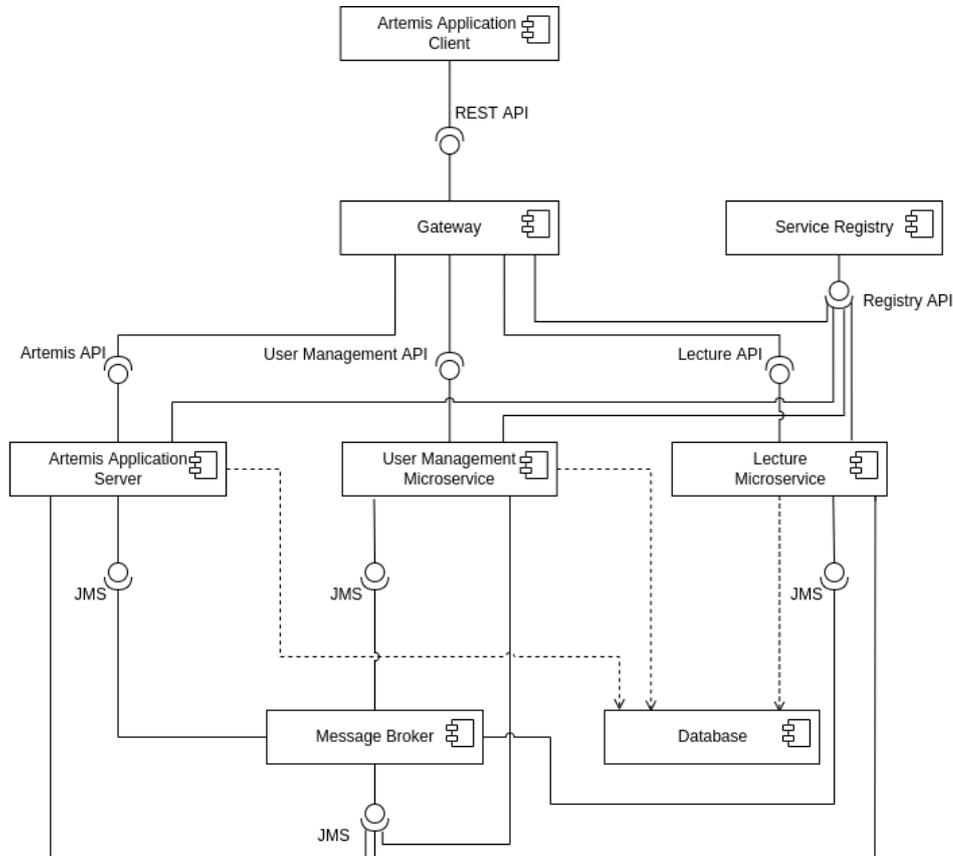


Figure 4.2: UML component diagram showing the proposed server architecture of Artemis. The Artemis client communicates with the gateway which routes the request to the Artemis server, the User Management microservice or the Lecture microservice. The Artemis server and the two microservices communicate with each other through the message broker using JMS and depend on the database. The gateway, the Artemis server application and the two microservices register themselves in the service registry which keeps track of their instances

4.2.1 Functional Requirements

This thesis is not related to implementing new features for the users in Artemis. Instead, it moves the Artemis architecture from monolithic to microservices. For this reason, there are not many functional requirements.

FR1 Retain existing features: The migration should be transparent to the user. Therefore, the existing features should not be changed, and the user should use them in the same way he has done it before.

FR2 **Show a message about problems in microservices:** Users should receive a message to either wait or try again later if the microservice he requests is unavailable or the communication between services takes more time than the specified timeout or fails.

4.2.2 Nonfunctional Requirements

The nonfunctional requirements are categorized using the FURPS+ model described in [BD10]. We omit the functionality category as it is not part of this section.

Reliability

NFR1 **Reliability:** The system should not crash if one of the microservices is unavailable.

NFR2 **Fault Tolerance:** The message broker should persist the asynchronous communication between the microservices if the consumer is not available. Therefore, no communication will be lost.

NFR3 **Security (JWT):** Each microservice should use the same JWT secret to authorize the user access.

NFR4 **Security (Unauthenticated Access):** Unauthenticated users should not have access to application-specific data.

NFR5 **Security (Inter-service Communication):** Inter-service communication should be possible only to authenticated microservices. Unauthenticated ones should not be able to send messages using the message broker.

Performance

NFR6 **Current Performance:** The new architecture should not decrease the current performance.

NFR7 **Caching Mechanism:** The Artemis server and the microservices should use the distributed cache that is already defined for the current Artemis architecture.

Supportability

- NFR8 **Extensibility**: The context of each microservice should be clear to make it easy to add new features.
- NFR9 **Maintainability** (Microservice's Size): The microservice should be small applications where it is easy to find bugs and fix them.
- NFR10 **Maintainability** (Deployment): Each microservice should have an automated deployment pipeline.
- NFR11 **Scalability**: The microservices should be able to scale both vertically and horizontally.

Implementation requirements

- NFR12 **JHipster**: We should use JHipster to generate the gateway and the microservices.

Operations requirements

- NFR13 **Docker Compose files**: We should create Docker Compose file for each new component.
- NFR14 **Kubernetes resource files**: We should create a Kubernetes deployment resource file for each component in the architecture.

Packaging requirements

- NFR15 **WAR packaging**: We should use WAR packaging for the new components.

4.3 System Models

4.3.1 Scenarios

The following scenarios show how Artemis should work after the migration. They do not propose new functionality but show that the existing functionality should remain unchanged for the user.

User Management

An admin user can create, update or delete users. Artemis supports internal and external user management depending on the application configuration. Currently, the implementation of those features are the Artemis Server Application.

After extracting the User Management service from Artemis, the gateway will forward those requests to the User Management service, which will handle them according to internal or external user management configurations. However, several cases exist when the extracted service needs to communicate with the Artemis Server Application. This communication will happen through an additional communication mechanism implemented by a message broker.

Lectures

Students can access lecture resources. Each lecture may have several units of a different type - text, exercise, video, or file. The lectures also support the uploading of attachments. Like the first scenario, the Artemis Server Application handles management and access to the lecture resources.

After the migration, the Lecture service will handle those actions. Again, communication between the Lecture service and the Artemis Server Application will happen using messaging through a message broker or HTTP requests.

4.3.2 Dynamic Model

The following diagrams show simplified example communication between the user and the system.

Figure 4.3 shows the communication procedure when a student registers himself in the system. He does that using the user interface provided by the client application, which sends the request to the API Gateway. The API Gateway decides which service to route the request to. In the case of account registration, the gateway sends the request to the User Management Service. If the service successfully creates the user, the system has to send an account activation email. This functionality is in the Artemis Server Application. For this reason, the User Management service sends a message to the Artemis Application, which sends the actual email to the user.

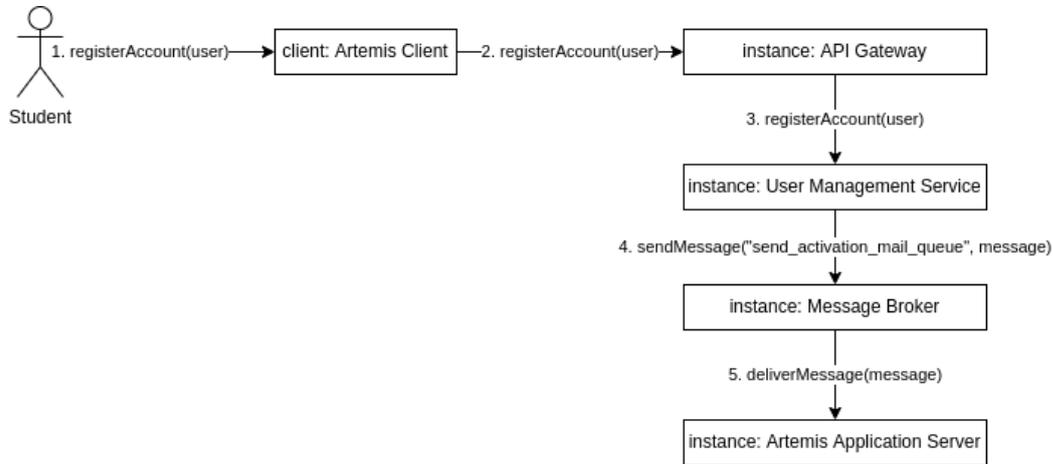


Figure 4.3: UML communication diagram depicting the communication between instances in user account registration process. A student registers himself an account in the system which sends a request to the API gateway, then the API gateway redirects the request to the User Management microservice. The microservice creates the account and sends a message through the message broker to the Artemis server application to send an account activation email.

Figure 4.4 describes the communication when a student wants to access the details for a lecture. Again, he uses the Artemis Client, which requests the API gateway. Depending on the defined rules, the API gateway decides where to redirect the request. Since it is a lecture-related request, the gateway sends it to the Lecture service.

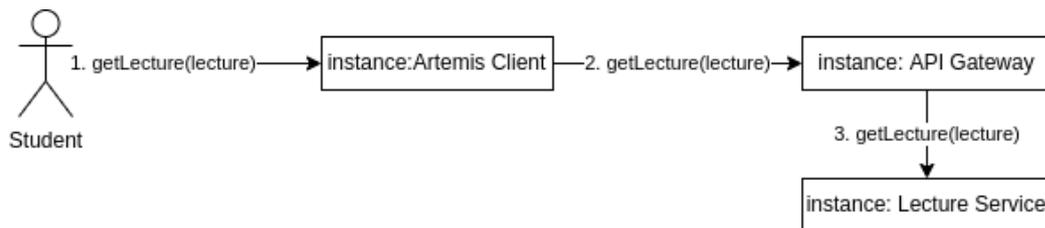


Figure 4.4: UML communication diagram depicting the communication between instances in retrieving lecture details. A student open a lecture in the Artemis client application which sends a request to get the lecture details from the API gateway, which then redirects the request to the Lecture microservice.

Chapter 5

System Design

The current chapter follows the structure of the System Design Document Template in [BD10]. It describes the proposed architecture in detail, presenting the design goals, the separate subsystems, their purposes, and other technical details.

5.1 Overview

Artemis should support several new components added by the proposed architecture and the updated existing components. The new components are the API gateway, the User Management service, and the Lecture service. One of the updated components is the message broker. Currently, Artemis uses it for WebSocket communication. The new architecture adds further broker use related to communication between the services. In addition, we also extend the use of the Service Discovery component. At the moment, we use it when we deploy multiple Artemis instances. Later, it will also be used in the development and test environment since it is essential for the new architecture.

5.2 Design Goals

Reliability

First of all, Artemis should be reliable. The system should still be available if one of the services is down. Failure in one service should not lead to failure in other services. This goal is derived from **NFR1** and has a high priority because if the students and the instructors cannot rely on the system, they would not want to use it. Additionally, we can increase the availability by

deploying multiple instances of the extracted microservices. Artemis already supports that, therefore, it should not be a problem for the User Management and Lecture service.

Fault Tolerance

Another important design goal is related to fault tolerance. In some cases, two services need to communicate with each other. An example would be when a user registers a new account in the User Management microservice. The system should send an account activation email to the user on successful registration. The Artemis application handles this action. Therefore, the User Management microservice has to send a message to the Artemis application to notify it to send an email. Suppose the Artemis application is unavailable at that moment. The message broker should persist the communication so that the Artemis instance can send the email when the instance is back. It also has significant importance because we do not want to lose communication or data. We derive this design goal from **NFR2**. The message broker handles it. If a service that needs to consume a message is not available, the message is persisted in the queue until it is delivered. If there is a problem during the message consumption, the message remains in the queue. There is a limit to the delivery times of each message. If the broker reaches the limit, it moves the message to a dead letter address queue where the message stays, but it is not the broker's responsibility to deliver it to the consumer.

Security

Further design goal is that the system should be secured. The addition of more components to the architecture requires additional security rules. Communication with other components should not be allowed for unauthorized entities or components. We derive this design goal from **NFR3**, **NFR4**, **NFR5**. It also has a high priority because we should grant access only to authorized users. Currently, we handle security through JWT tokens which remains the same in the new architecture. The system will authorize access to each microservice by checking the JWT. The critical part here is to verify the JWT by using the same secret. Here comes the Service Discovery, which has the responsibility to share the secret that the system will use to create and verify the JWTs. Therefore, the services can verify the tokens using that secret. The message broker will handle the security of the inter-service communication. Only authorized entities can connect, write and read from it.

Maintainability

Maintainability is a design goal derived from **NFR9**. It can help to easily maintain the new components in the changed architecture of Artemis. In addition, the size of the services should be relatively small makes finding and fixing bugs easier. Thus, the developers' performance will increase, and they can maintain the system faster and easier.

5.3 Subsystem Decomposition

5.3.1 Service Registry

The primary role of the Service Registry is to behave like a database for service instances to which every service and the gateway registers on application start. Then the gateway uses the registry to get this data and know the location of the services that it will redirect requests.

Another role of the Service Registry is to provide all registered applications with the same run-time configurations. For example, it provides the JWT secret the system uses to validate and generate the access tokens to all instances.

Last but not least, the registry provides an administration dashboard for all service instances, giving access to application metrics, logs, and health checks.

Figure 5.1 presents the communication between the services, the gateway, and the service registry using a UML Communication Diagram. Each service, as well as the gateway, registers in the service registry shortly after their start-up. First, the gateway fetches data from the registry about the registered service instances and their locations. Then it uses this data for request routing and load balancing.

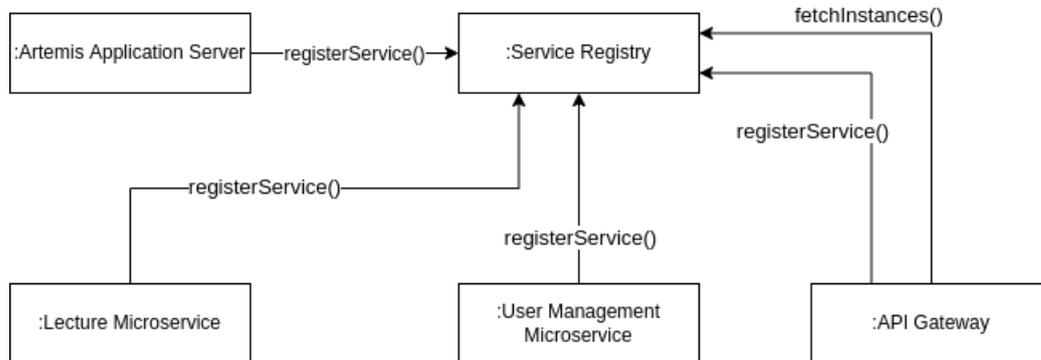


Figure 5.1: UML communication diagram illustrating the communication between the service registry and the gateway, the Artemis server application, the User Management microservice and the Lecture microservice. All components register their instances in the registry. The gateway fetches the registered instances including details about their location.

5.3.2 API Gateway

The gateway component implements the "API Gateway" design pattern and acts as a single entry point to the server application. Its primary responsibility is to do request routing for the incoming requests from the client. We define the routing map in yml format in the *application.yml* file of the gateway application. The route definition is shown below in Listing 5.1. There are several definitions with the same structure: *id* defines the identifier of the application to which the gateway will route the request, the *uri* defines the link to the application. *lb://serviceId* means that the system will obtain the path to the service from the service registry. Finally, the *path* predicate defines the paths which the component supports. If there is a need to handle concrete request type, it is possible to define it by using the *method* predicate. We can add several definitions with the same id if that is needed.

There is also a separated definition for WebSocket routes because the request protocol is different. The definition itself is similar to the other ones with a difference in the *uri*. It is in the form of *lb:ws://serviceId* which means that the service path will come from the service registry, but this time it uses the WebSocket protocol.

Listing 5.1: Gateway routes definition example

```

1 routes:
2   - id: gateway
3     uri: lb://gateway
  
```

```

4     predicates :
5     - Path=/api/gateway/**
6   - id: usermanagement
7     uri: lb://usermanagement
8     predicates :
9     - Path=/api/users , /api/users/** ,
10        /api/account , /api/account/** ,
11        /api/guided-tour-settings/**
12   - id: usermanagement
13     uri: lb://usermanagement
14     predicates :
15     - Path=/api/authenticate
16     - Method=GET
17   - id: lecture
18     uri: lb://lecture
19     predicates :
20     - Path=/api/lectures/**
21   - id: artemis
22     uri: lb://artemis
23     predicates :
24     - Path=/api/**,/time,/websocket/tracker/info ,
25        /websocket/tracker/info/**,/management/** ,
26        /public/**
27   # Websocket route
28   - id: artemis
29     uri: lb:ws://artemis
30     predicates :
31     - Path=/websocket/**

```

5.3.3 Message Broker

The message broker handles the communication between microservices. It provides asynchronous communication through messages. The service that sends the message is the producer of the message, while the service receiving it is the consumer. The producer sends a message to a specific queue used to transport all the messages of a certain type. The consumer will subscribe to that queue, and when a message is placed in the queue, it will send it to the consumer. The queue naming convention defined in Artemis is *service_name_queue.action_name* where the *service_name* is the name of the microservice and *action_name* is unique name for the action which needs to be handled by the receiver i.e. *user_management_queue.send_activation_mail*.

As already described in section 5.2, messages that the consumer does not

successfully receive are kept in the queue until there is an available consumer for them. Suppose the action performed by the consumer fails and throws an exception. Then the queue still handles the message as an unreceived message and tries to redeliver it until it reaches the delivery limit. After that, it moves the message to a Dead Letter Address, where it keeps unreceived messages¹. The message broker is no longer responsible for their delivery. System administrators can use the message broker administration console to check for failing communication or find the reasons for that.

In some cases, we want to receive a result for a message sent through the message queue. This is possible by using a result queue. The naming convention for the result queue is similar to the one we already defined. We use the same queue name as the one when ending the message but adding *_result* in the end *service_name_queue.action_name_result*, i.e. *user_management_queue.send_activation_mail_result*. Here, it is essential to ensure that there is no difference in which instance should receive the result message. This type of communication can be error-prone if the developers misuse it. If there is a need of request-reply communication where it is important which instance consumes the response, we can use temporary queues.

Figure 5.2 is an example of communication between two services when a response is required. First, the Artemis Application sends a message to check whether the user group is available using the message producer. The message is put in the queue and consumed by the User Management service consumer. Once the User Management service checks whether the group is available, it uses a producer to send the response in a result queue. The broker puts the response message in the result queue and then sends it to the available consumer in Artemis to consume it.

¹<https://activemq.apache.org/components/artemis/documentation/1.1.0/undelivered-messages.html>

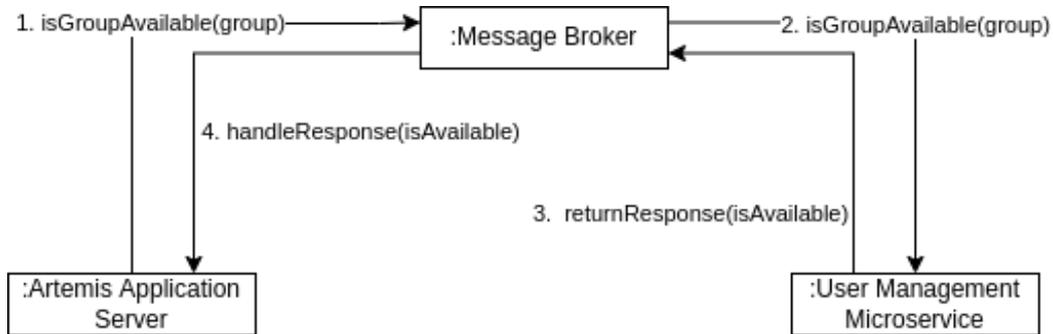


Figure 5.2: UML communication diagram showing example communication between the Artemis server application and the User Management microservice through the message broker where response is required. The Artemis server application puts a message in a queue which the message broker sends to the User Management microservice. The microservice puts the response message in a response queue which the broker delivers to the Artemis server application.

5.3.4 User Management Microservice

The User Management microservice includes functionality related to user management. It handles the creation, update, deletion, read, and searching of users. It communicates with the Artemis application server using queues in the message broker. It also communicates using REST with external User Management systems.

Figure 5.3 is a component diagram of the extracted microservice, presenting the moved or split classes from Artemis and their relationship. We have split the InstanceMessageSendService and moved only the parts relevant for our microservice. We have moved the user, account, and guided tour setting resources and the two services related to users and user creation. There is a mail service producer which sends messages to Artemis to send emails to the user. The implementation also includes consumer that receive messages from Artemis but we have omitted it for simplicity.

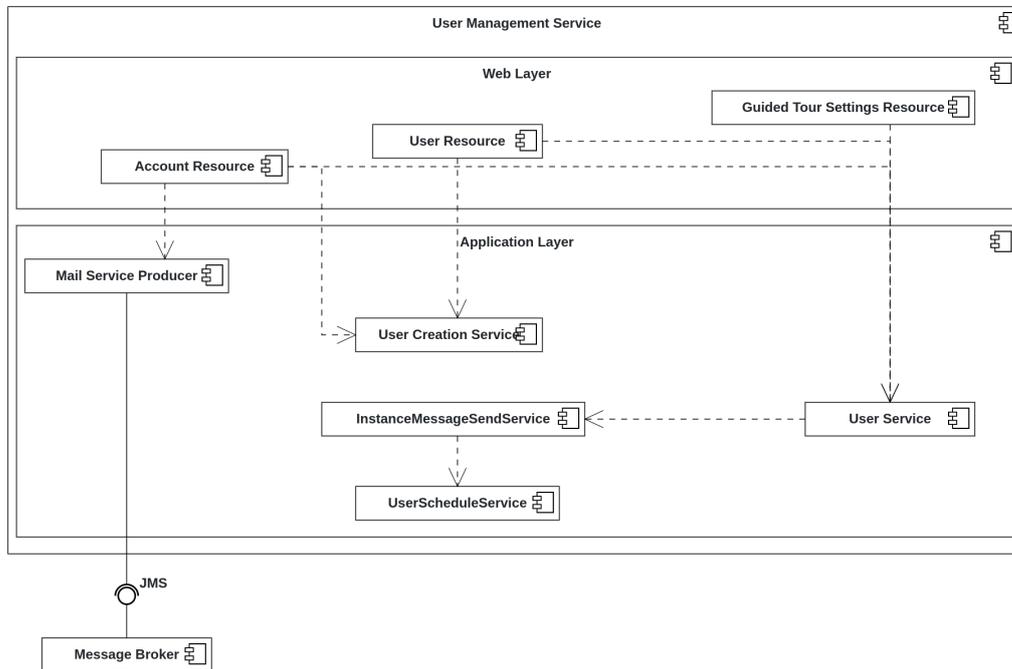


Figure 5.3: UML component diagram illustrating the User Management microservice

5.3.5 Lecture Microservice

The Lecture microservice contains functionality related to lecture features. It includes managing lectures and lecture units, including all CRUD operations. It communicates with the Artemis Application using queues in the message broker.

Figure 4.4 is a component diagram of the extracted microservice, presenting the moved classes from Artemis and their relationship. In this microservice we have moved the lecture, lecture units, exercise units, attachment units, text units and video unit resources. Moreover, we have moved the lecture and lecture units services. There is a producer which sends messages to Artemis related to retrieving details about exercises. The implementation also includes consumer that receive messages from Artemis but we have omitted it for simplicity.

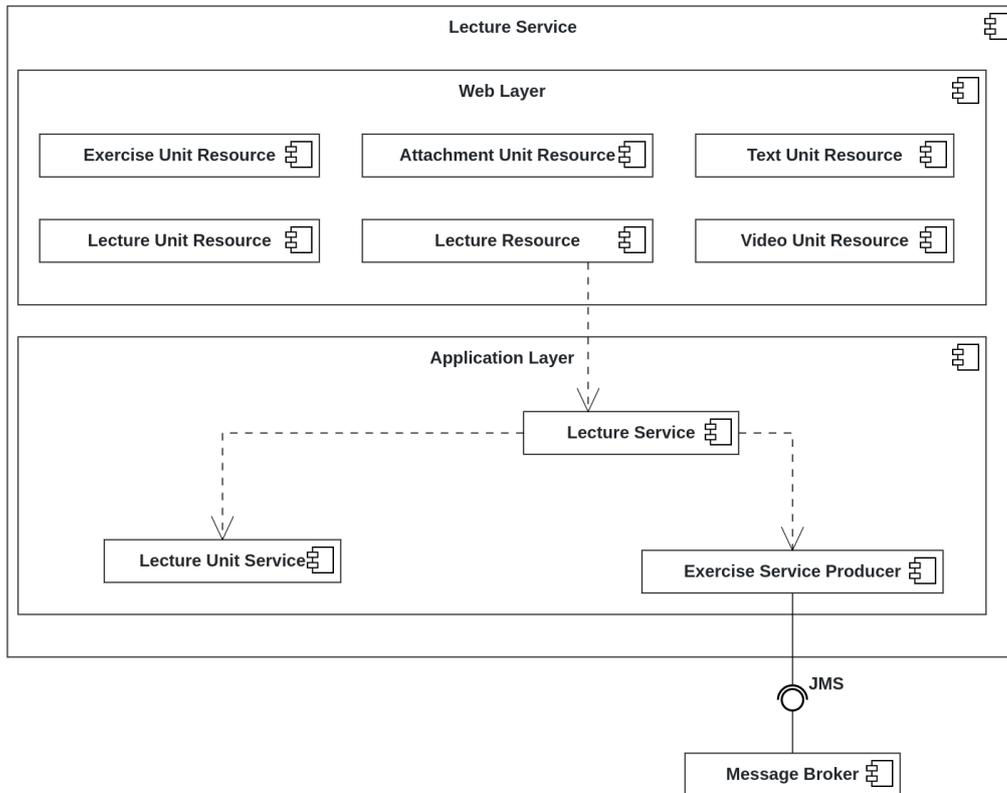


Figure 5.4: UML component diagram illustrating the Lecture microservice

5.3.6 Artemis Application Server

The Artemis application server contains the server-side logic which we have not extracted in the two microservices. It communicates with both of the microservices using the message broker. The communication with the external User Management, Version Control and Continuous Integration systems remains unchanged.

5.4 Hardware Software Mapping

Hardware software mapping describes how we assign software components to hardware components and how they communicate with each other. [BD10] We complement it with a deployment diagram that represents those relationships. [BD10]

Figure 5.5 proposes the deployment of the microservices architecture. There are three new components - the API Gateway, the User Management

Microservice, and the Lecture Microservice. In this diagram, we intentionally omit some details, focusing only on the components interesting for the thesis.

On application, start-up each service and the gateway communicates with the service registry and registers itself. Then, the gateway gets the data about all available instances from the registry. The Artemis Application Client is dependent on the API Gateway and the services. The client sends REST API requests to the gateway, which routes the request to the appropriate service by checking its routing map definition and the available instances. When communication between the services is needed, we use queues in the message broker. All services have access to the same database server.

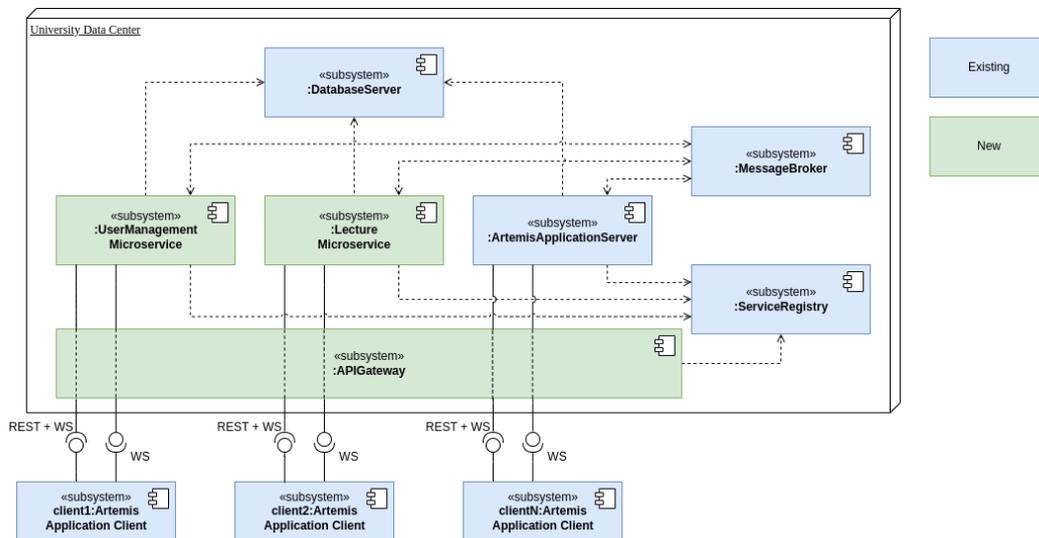


Figure 5.5: UML deployment diagram of the microservices architecture. The diagram is adapted from the deployment of the monolith application described in *Securing and Scaling Artemis WebSocket Architecture* by Simon Leiß [Lei20]

5.5 Persistent Data Management

Usually, the microservices architecture proposes a separate database for each microservice. Since this thesis lays the foundation of the migration towards microservices, we initially decided not to decompose the database. We decided that this task would be too time-consuming and risky. Furthermore, it would have complicated the migration even more. Therefore, all services currently use the same database and the same database schema to persist data. This use can be changed in the future if the database becomes the

bottleneck of the architecture. There are also other solutions than decomposing the database. One would be to create several database replicas, some of which support write actions and others read actions.

5.6 Boundary Conditions

Boundary conditions describe how the system is started, shut down and how to deal with system failures if they occur. [BD10]

5.6.1 Application Startup

Figure 5.6 illustrates the start-up dependencies between the subsystems. The only strict dependency is between the Artemis application server, the two services, and the database. That is because the services cannot start if they cannot connect to the database. The server application and the services also depend on the broker and the registry, but this will not fail the application start. There will be errors in the logs until starting those two components because the applications will try to connect to them in a defined period of time. The gateway also depends on the service registry, and it also depends on the server application and the two services to be able to complete the received requests. So again, the gateway will not fail on start if those components are not available, but it will not be able to handle any client request. It is not able to save requests for later execution. Therefore, we recommend following this order so that no user actions or data are lost.

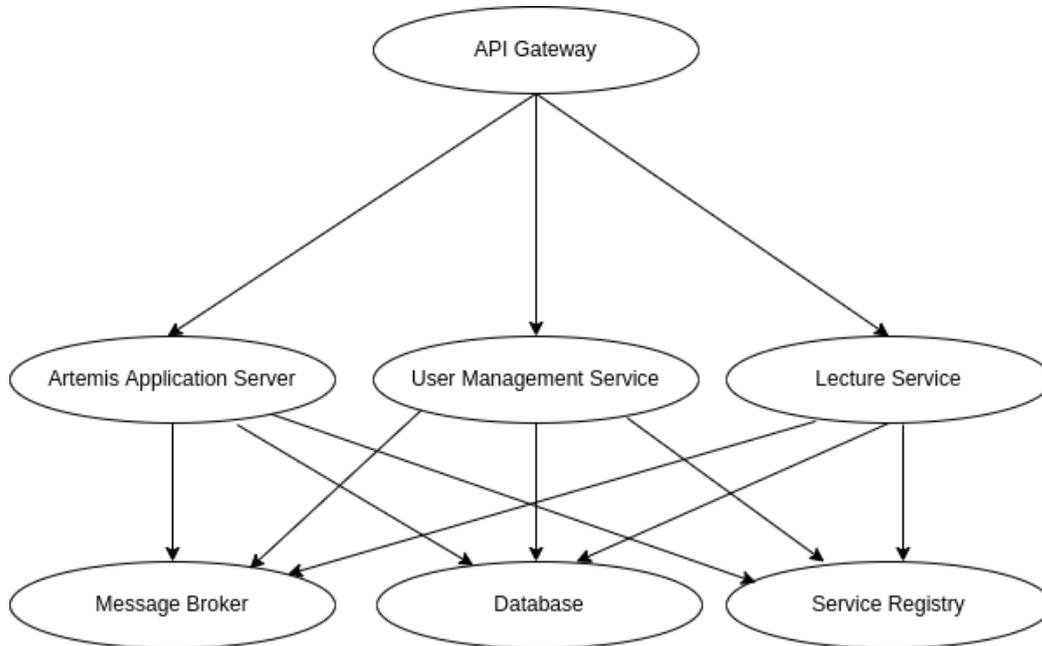


Figure 5.6: Dependency Graph describing the order to startup the subsystems. The gateway depends on the Artemis application server and the two microservices. The Artemis application server and the two microservices depend on the message broker, the database, and the service registry. The diagram is adapted from the dependency graph for the monolith application described in *Securing and Scaling Artemis WebSocket Architecture* by Simon Leiß [Lei20]

5.6.2 Application Shut Down

In order to stop the application, we can use the reverse order of the startup. First, we should stop the gateway so that the clients accept no more requests because otherwise, they will not get a response and will be lost forever. After that, the Artemis server and the two microservices followed by the service registry, message broker, and database.

5.6.3 Failure Handling

If we have started all of the applications and an error occurs, the first thing to do is open the service registry and check the status of the service instances. We should first check whether all of them are up and running or any of them has stopped working. The next thing to do is to check the application logs that have crashed. If they all seem to be okay, we check all the logs. We can start from the logs of the gateway application. There we find which

request has failed. If it is related to users, the problem must be related to the user management service. If the problem is related to lectures, then the problem is related to the lecture service. The problem should be related to the Artemis server application for all other cases. Knowing this information, we can check the logs of the respective application. The application logs will help to find the reason for the error.

We can use the ActiveMQ Artemis Console to track lost messages. For example, it allows to see messages waiting for delivery for each queue. Also, we can use it to access dead messages that the broker delivered several times, but the consumer did not consume them successfully. This could help to identify problems with inter-service communication and even with the microservices.

Chapter 6

Migration to Microservices

6.1 Migration Strategy

There are several migration approaches defined as migration patterns. Each of them is used accordingly to the concrete situation. There are no universally "good" ideas. [New19] Artemis is being actively developed at the moment by around 20 people. It is not possible to stop the development and rewrite the application. This thesis will also not enhance the feature set of Artemis. Thus, we cannot implement new features directly as microservices. Therefore, we should incrementally refactor Artemis by extracting services from the monolith.

Strangler Fig Application is a modernization strategy. The name comes from a rain forest plant called Strangler Fig, which grows upward around a tree trying to get more sunlight while the tree slowly dies [LML20]. This concept represents the gradual refactoring of an existing application by creating a new system around the old one and letting it grow until the old one is gone [LML20].

This is the strategy that we use during the migration process of Artemis. We will create a new application made out of extracted from the monolith services. The strangler application will gradually grow while the monolith becomes smaller over time. Figure 6.1 shows exactly this process. The monolith application will become smaller with each microservice we extract while the count of microservices grows.

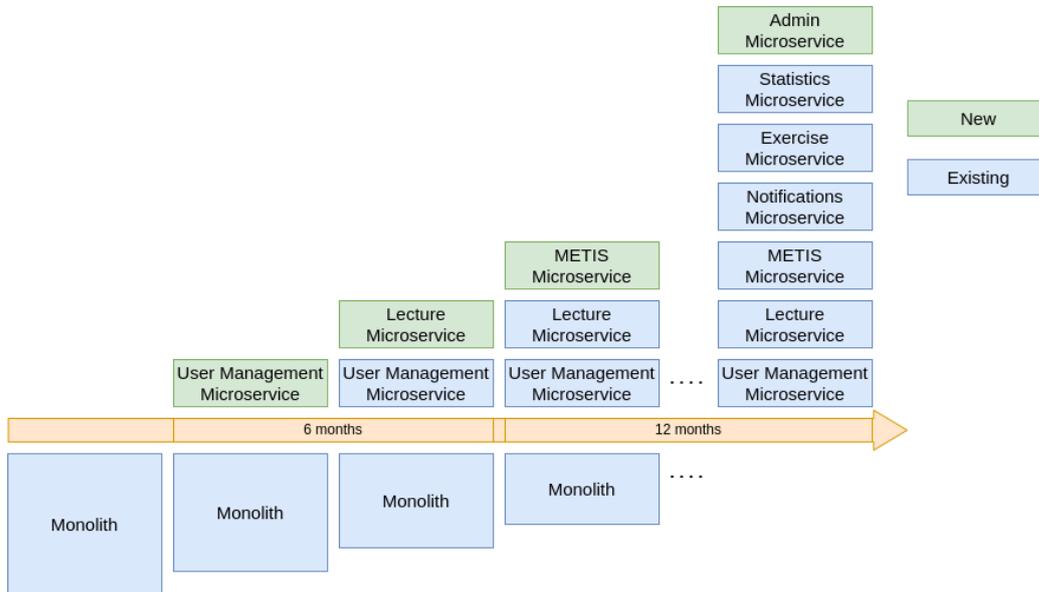


Figure 6.1: Migration Strategy for migrating the Artemis server application to microservices. The diagram describes the microservices that could be extracted over 1.5 years. The diagram is adapted from the “Strangler application” migration strategy described by Chris Richardson in [Ric19].

6.2 Decision on the Architecture

We develop Artemis using JHipster, which also supports Microservices Architecture. It enables the generation of gateway and microservice applications. There are several ways to organize a microservices architecture, and we had to choose which one is the best for our case.

The Artemis application is located in a single repository and is a single JHipster project, containing both client and server parts. As already said, there are many people actively contributing to the project. This fact restricts us from splitting the monorepo and the existing JHipster application. Therefore, we had to think about possible solutions.

We discussed the following approaches with their benefits and drawbacks:

6.2.1 Artemis as a Gateway

This approach updates the monolith Artemis application to include JHipster Gateway features. Since the generated by default JHipster Gateway includes server code for authentication, converting the monolith to a gate-

way is technically possible. New microservices will be extracted from the Gateway application until there is no more server code to be extracted there. We illustrate the approach in Figure 6.2.

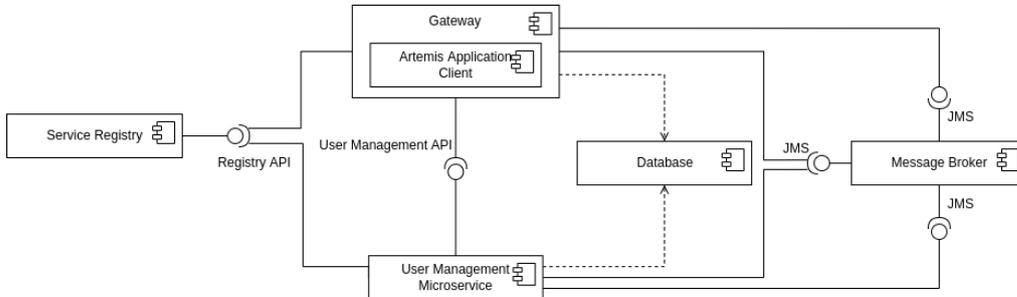


Figure 6.2: Proposed architecture where Artemis application serves also gateway features. It handles the server requests and redirects only the requests related to the microservices.

Benefits:

- JHipster supports the approach.
- Easy way to start with the migration since we will do little project configuration changes and almost or no changes to the current code.

Drawbacks:

- The API gateway is an API management tool supposed to sit between the client and the collection of services. Therefore, this approach will conflict with the API gateway definition.
- The API gateway is a component that has to be highly available. Transforming the monolith into an API Gateway will decrease the availability since there are more places where the application may fail. The routing functionality may fail due to failure in Artemis, and vice versa Artemis application may fail due to failure in the routing or other gateway-specific functionality.

6.2.2 Artemis Application in front of the API Gateway

This approach includes the creation of a JHipster API gateway without any client and server code. The gateway, as supposed, will be responsible for request routing and load balancing.

As illustrated in Figure 6.3, the Artemis application will stay unchanged. The gateway will not route the Artemis Server requests. It will route only extracted to microservices endpoints. The microservices extraction will gradually remove server code from the Artemis application until an application where only the client code is left is reached and the API Gateway does the routing for all of the microservices.

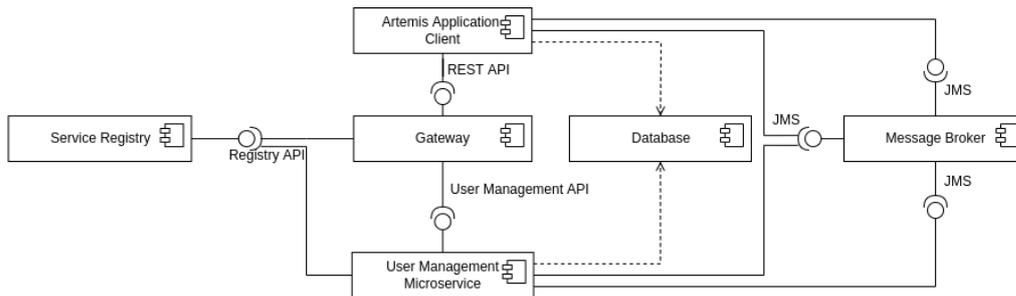


Figure 6.3: Proposed architecture where the Artemis Application is in front of the gateway. It routes only requests to the extracted microservices.

Benefits:

- Jipster still supports and allows to generate a gateway without client code.
- The API Gateway is responsible only for what it is supposed to do.
- We expect no changes to the Artemis application’s code and project configurations.
- We expect no changes in the deployment of the Artemis software component.

Drawbacks:

- The gateway will not route Artemis server requests which we have not extracted in a microservice. Therefore, the gateway will not be able to do load balancing for the Artemis server instances, resulting in a need for an additional load balancer in front of the Artemis server.
- The gateway is supposed to be a single entry point for the server-side of the application, which is not fulfilled in this architecture

6.2.3 Artemis server as an independent application

The approach is illustrated in Figure 6.4 and includes the creation of a JHipster API gateway between the client and the services. The gateway will be responsible for request routing and load balancing. The Artemis application code will remain unchanged. However, we will change the build procedure. We will build the Artemis server in a WAR file that does not include the client application. Moreover, we will build the client application with the gateway, which will also serve it.

Artemis client requests will always go through the gateway, which will route them to the correct service. We will extract services from the server code, gradually reducing the Artemis server application until no further extraction is possible.

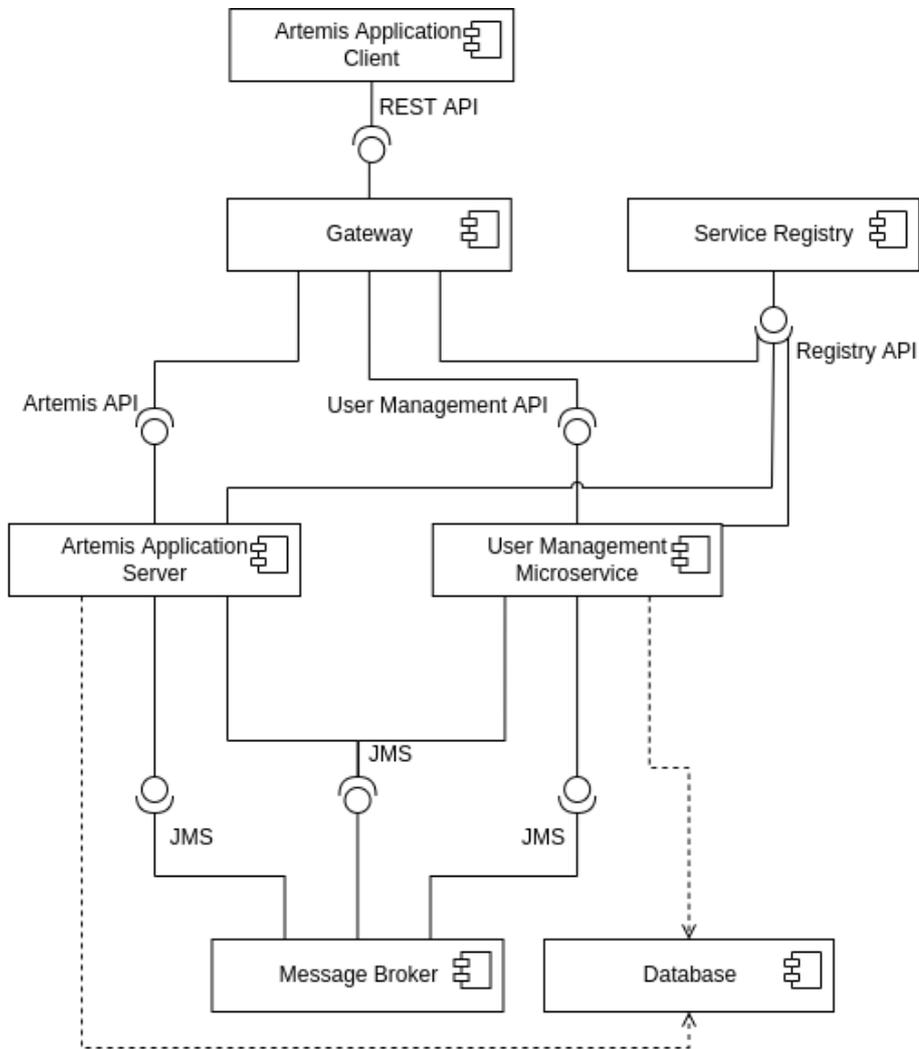


Figure 6.4: Proposed architecture where the Artemis server is an independent application which registers to the service registry similarly to the extracted User Management microservice. The client sends the requests to the gateway which redirects them to the proper server application.

Benefits:

- Supported by JHipster
- No changes in the client and server code
- A relatively easy approach to start with

- The client and the server are built and run separately
- The gateway is a single entry point to the server-side of the application

Drawbacks:

- Developers will be confused why the client application is located in the same project as the server application but is built together with the gateway application

6.2.4 Conclusion

We decided to continue with the approach defined in section 6.2.3 because it has the most benefits and most minor drawbacks and is also most relevant to the microservices architecture. The client uses the gateway as an entry point for the server-side, and we cannot access it without the gateway. We build the Artemis client and server separately. Therefore, they can also be deployed and scaled individually.

6.3 Decomposition into microservices

6.3.1 Creation of an API Gateway

JHipster provides the great feature to generate a Gateway for a microservice architecture. As already said, the API Gateway is a crucial component of the microservice architecture that has to be highly available and scalable. Except, routing client requests to the microservice, the JHipster generated gateway can include a client application and handle the user authentication. [SK20]

Our requirements are different, though. We want to package the client application in the gateway, but we do not want to move the client code to the Gateway application structure. Also, we do not want to change the Artemis server. Therefore we do not want to handle the authentication in the gateway. Thus, we need to change the generated application according to those requirements.

We first removed the database configurations and the authentication and user management features from the gateway. After that, we updated the build configurations to run the client build and copied the build files to the gateway's build resources. This way, the client application packaging is possible in the Gateway application.

We already said that the gateway needs to be highly available since it is an entry point for the server architecture, and every single request goes through it. The JHipster Gateway handles this requirement by using Spring WebFlux. The Spring WebFlux framework complements the Spring MVC and allows reactive programming in Java applications. WebFlux is a non-blocking framework built to take advantage of multi-core processors and handle massive numbers of concurrent connections with a small number of threads. This helps the system to scale with fewer hardware resources. To compare, Spring MVC uses synchronous blocking architecture with a one-request-per-thread model¹. The gateway was generated by JHipster using WebFlux by default. The gateway is supposed to handle all the input traffic. As per Matt Raible and "Reactive Java Microservices with Spring Boot and JHipster" article linked as ¹, there is a general rule of thumb that WebFlux will make a difference compared to Spring MVC if you have >500 requests/second which will most likely be fulfilled by Artemis when running exams, quizzes for large courses (i.e. EIST). There is no change to any configurations or REST endpoints in Artemis. We enable WebFlux only for the gateway project. The gateway redirects all requests from the client to the Artemis application, which still uses Spring MVC. The difference is that when the gateway redirects a request, it can handle another request with the same thread instead of waiting for an answer. Once Artemis sends back a response, the gateway will know about that, and it will finalize the request and send the response to the client.

6.3.2 Extraction of User Management Microservice

The first microservice we extract is the User Management microservice. Its domain is to manage users and their accounts by providing them the functionality to reset passwords and register themselves and updating settings for the guided tour that Artemis supports.

Knowing the context of the microservice, we can start with the extraction. First, we need to generate a new JHipster project. We do it using the configurations in Figure 6.5.

¹<https://developer.okta.com/blog/2021/01/20/reactive-java-microservices>

```
Which *type* of application would you like to create? Microservice application
What is the base name of your application? usermanagement
Do you want to make it reactive with Spring WebFlux? No
As you are running in a microservice architecture, on which port would like your server to run? It should be unique to avoid port conflicts. 8087
What is your default Java package name? de.tun.in.www1.artemis.usermanagement
Which service discovery server do you want to use? JHipster Registry (uses Eureka, provides Spring Cloud Config support and monitoring dashboards)
Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
Which *type* of database would you like to use? SQL (J2, PostgreSQL, MySQL, H2, Oracle, HSQL)
Which *production* database would you like to use? MySQL
Which *development* database would you like to use? MySQL
Which cache do you want to use? (Spring cache abstraction) Hazelcast (distributed cache, for multiple nodes, supports rate-limiting for gateway applications)
Do you want to use Hibernate 2nd Level cache? No
Would you like to use Maven or Gradle for building the backend? Gradle
Which other technologies would you like to use? API First development using OpenAPI-generator
Would you like to enable internationalization support? Yes
Please choose the native language of the application English
Please choose additional languages to install German
Besides JUnit and Jest, which testing frameworks would you like to use?
Would you like to install other generators from the JHipster Marketplace? No
```

Figure 6.5: Configuration for the generation of the User Management microservice

Once created, we then need to align the project configurations with Artemis. The first thing we need to do is to update the Artemis *settings.gradle* file and include the new project in the project structure. After that, we can share dependencies between projects and reuse source code. We then modify the *build.gradle* file of the user management service and change dependency versions using the ones used in Artemis and update tasks definitions applying the *tasks.gradle* file with reusable tasks. We also use source sets to define sources that we will reuse from Artemis. The sources include some configuration files, i.e. Liquibase and database configurations, the database entities, repositories, and data transfer objects. We also remove duplicated configurations from the user-management project to reduce code duplication. Next, we update security configurations, JWT configurations and filters, and the token provider to be similar to the implementation in Artemis. Finally, we also update the application resources. Accordingly to the changes, we also need to update the automatically generated unit tests.

After we have aligned application configurations, we can start with extracting features from Artemis to the new service. We first need to make sure which classes implement the features we want to move and which other classes are using the ones that we will move. If many classes use the one we want to move, we may need to split the class or implement inter-service synchronous or asynchronous communication.

Once we know which classes we are going to move, we can copy them and their tests. In some cases, existing integration tests can break because they also test features not included in the new service. Thus, we need to update the tests accordingly, making sure to keep high test coverage. We add *@Deprecated* annotation to all of the classes or methods that we move to find them easily later when we remove them from the Artemis server application.

Figure 5.3 illustrates a component diagram of the user management microservice. The endpoints that we extracted are listed in Table 6.1.

| Request Type | Endpoint |
|----------------|--|
| POST | api/register |
| GET | api/activate |
| GET | api/authenticate |
| GET | api/account |
| GET, PUT | api/account/password |
| POST | api/account/change-password |
| POST | api/account/reset-password/init |
| POST | api/account/reset-password/finish |
| GET, POST, PUT | api/users |
| GET | api/users/search |
| GET | api/users/authorities |
| GET, DELETE | api/users/{login} |
| PUT | api/users/notification-date |
| PUT | api/guided-tour-settings |
| DELETE | api/guided-tour-settings/{settingsKey} |

Table 6.1: Endpoints extracted in the User Management microservice

After copying the resources and services, we identify the need for broker communication. The User Management service has to send messages to Artemis in order to send emails to the user on account registration or password reset request. Therefore, we implement a producer in the User Management service and a consumer in the Artemis Server application.

The next step is to update the gateway routes definition by adding the routes that we defined in 6.1. We show the exact definition in Listing 5.1.

After integrating the gateway and the new service, we can continue manually testing the existing features. If we do not find regressions, we can continue with the next step to remove the duplicated code from the Artemis server application.

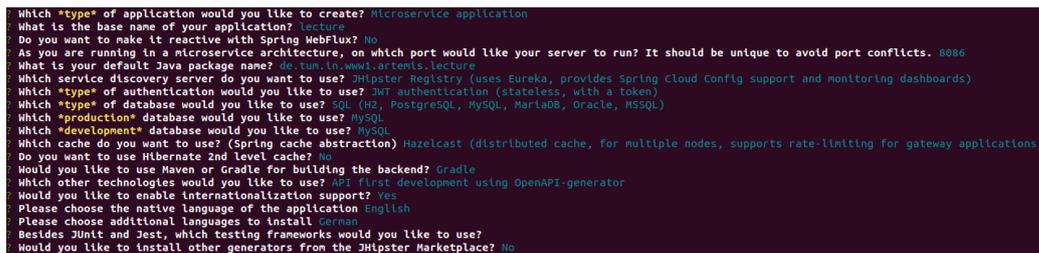
The removal of the copied code from Artemis is the last one but also error-prone. After deleting the copied classes, we might have to implement additional implementation of service communication, or we can even notice that we have wrongly moved over some of the classes or methods. The good news here is that we have not yet removed anything from the monolith application. Therefore, we can quickly correct our mistakes by removing the code from the extracted microservice. Besides, more integration tests may break. Therefore, we need to take care of fixing them as well.

Once we complete the Artemis refactoring, we should again test the application manually. If everything works, we are good to go.

6.3.3 Extraction of the Lecture Microservice

The lecture service is the second service extracted from Artemis. The scope of this service includes managing lectures and the lecture units included in a lecture, among which are video units, text units, attachment units, and exercise units. The video units are linked or embedded videos as part of a lecture. The text units allow to write a text in a markdown editor and link it to the lecture. The attachment units allow uploading file attachments as part of the lecture. Finally, the exercise units are links to exercises in the context of the given lecture. Figure 5.4 illustrates a component diagram of the Lecture service.

After we have defined the scope of the microservice, we can continue with the creation of the JHipster microservice application. Figure 6.6 represents the application generation configurations. They are similar to the ones used for the user management service. The port and the package name are the only different configurations.



```
Which *type* of application would you like to create? Microservice application
What is the base name of your application? Lecture
Do you want to make it reactive with Spring WebFlux? No
As you are running in a microservice architecture, on which port would like your server to run? It should be unique to avoid port conflicts. 8086
What is your default Java package name? de.tuhh.in.www1.artemis.lecture
Which service discovery server do you want to use? JHipster Registry (uses Eureka, provides Spring Cloud Config support and monitoring dashboards)
Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
Which *type* of database would you like to use? H2, PostgreSQL, MySQL, MariaDB, Oracle, MSSQL
Which *production* database would you like to use? MySQL
Which *development* database would you like to use? MySQL
Which cache do you want to use? (Spring cache abstraction) Hazelcast (distributed cache, for multiple nodes, supports rate-limiting for gateway applications)
Do you want to use Hibernate 2nd level cache? No
Would you like to use Maven or Gradle for building the backend? Gradle
Which other technologies would you like to use? API-first development using OpenAPI-generator
Would you like to enable internationalization support? Yes
Please choose the native language of the application English
Please choose additional languages to install German
Besides JUnit and Jest, which testing frameworks would you like to use?
Would you like to install other generators from the JHipster Marketplace? No
```

Figure 6.6: Configuration for the generation of the Lecture microservice

After creating the project, we applied the same configuration changes as we did for the User Management microservice. The next task is to copy the implementation and test classes. We copied all classes together during the extraction of the User Management service. This time, we decided to try an iterative approach by copying a small group of classes with low coupling. This approach will simplify testing and overall extraction by decomposing the task into smaller tasks. We define the following iterations: first, copy the management of attachment units, then the text units, the video units, the exercise units, overall management of lecture units, and in the end, management of lectures.

We start each iteration by copying the implementation and test classes. Then identify whether we need to implement communication between services in the context of the classes we copied. If not, we then check whether we need to refactor the copied code, i.e., if tests are breaking, we need to refactor them. Then we need to update the gateway routes if we have moved any

endpoints. At the end of the iteration, we test our changes. If there are regressions, we need to refactor the code. If not, we can continue with the next iteration.

We show the whole list of endpoints that we moved to the Lecture microservice in Table 6.2.

| Request Type | Endpoint |
|--------------|--|
| GET | api/courses/{courseId}/lectures |
| POST, PUT | api/lectures |
| GET, DELETE | api/lectures/{lectureId} |
| GET | api/lectures/{lectureId}/details |
| GET | api/lectures/{lectureId}/title |
| GET | api/lectures/{lectureId}/attachment-units/{unitId} |
| PUT, POST | api/lectures/{lectureId}/attachment-units |
| GET, POST | api/lectures/{lectureId}/exercise-units |
| GET | api/lectures/{lectureId}/text-units/{unitId} |
| PUT, POST | api/lectures/{lectureId}/text-units |
| GET | api/lectures/{lectureId}/video-units/{unitId} |
| PUT, POST | api/lectures/{lectureId}/video-units |
| PUT | api/lectures/{lectureId}/lecture-units-order |
| DELETE | api/lectures/{lectureId}/lecture-units/{unitId} |

Table 6.2: Endpoints extracted in the Lecture microservice

The final task is to remove the extracted code from the Artemis application server. Here we need to add additional communication with the Lecture microservice when the user requests the deletion of a course that includes lectures. In this case, the Artemis application sends a message to the Lecture microservice to delete the lecture. On the other side, the microservice returns a response whether the action was successful so that Artemis can continue with the deletion of the course. Since we use a shared database, we need a response that the microservice has deleted the lecture. Otherwise, if we try to delete a course without having deleted the lectures, this would lead to an exception on a database level which violates the foreign key constraints.

6.3.4 Microservice Extraction Steps

After extracting two microservices, we can generalize the steps needed to extract a microservice. We also think that the iterative approach that we used and described for the Lecture microservice is a better extraction approach. It is simpler because we decompose the whole task into smaller subtasks by

copying small parts of the implementation. This also results in easier testing and less chance to skip a test case.

We present a UML activity diagram (Figure 6.7) to illustrate the steps needed to extract a microservice using an iterative approach. We have grouped the tasks into three groups by the effort required to finish them. The groups are low, medium, and high effort. Each group represents the overall time consuming and complexity of the task. However, we cannot specify how much time they take because it will be different for each extracted microservice. We describe details about the different steps below the diagram.

6.3. DECOMPOSITION INTO MICROSERVICES

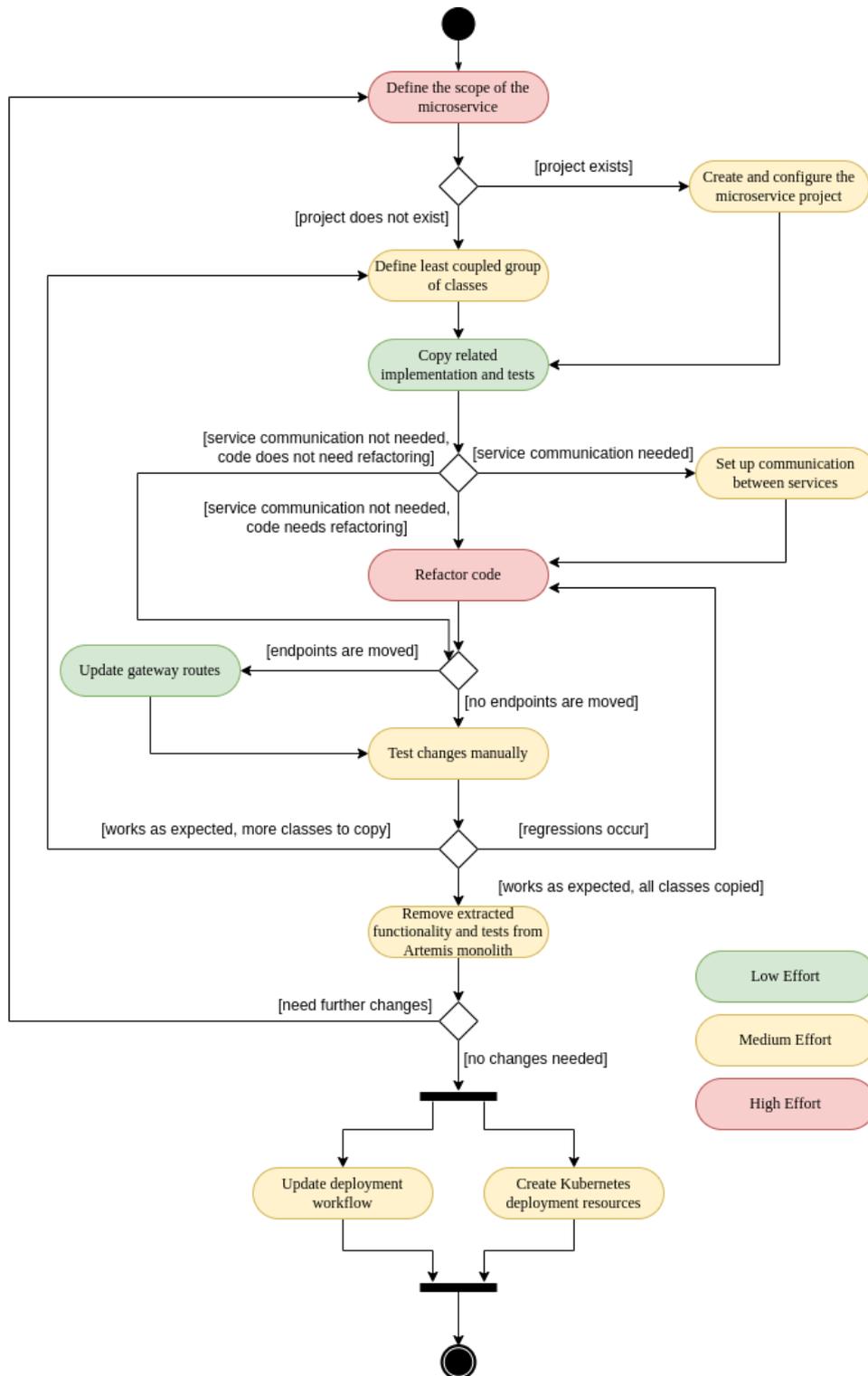


Figure 6.7: UML activity diagram illustrating the steps for extracting a microservice from the monolith.

1. Define the scope of the service

The first step is to define the scope of the service. We must define which features we want to extract and skim through the classes implementing those features to check the coupling in the current implementation. In case there is a high-coupling, we either consider whether the coupled features can be part of the microservice or refactor the existing code. This task is labeled as high effort because it takes time to define the scope, and it is also crucial for the future of the microservice. We need to make sure that there is a loose coupling between the microservice we will extract and the rest of the microservices and the monolith application.

2. Create and configure the microservice project

We use JHipster to generate a project of type microservice, using JHipster Registry as service discovery, JWT authentication type, MySQL database, Hazelcast cache, and Gradle as a build tool. After the project generation, we need to add the new project in the Artemis project structure and reuse dependencies versions and Gradle tasks. We can also reuse some of the configuration files defined in Artemis, i.e. *LiquibaseConfiguration*, *DatabaseConfiguration* and *JacksonConfiguration*. We do it by defining them as source sets. Similarly, we also add as source sets all database entities and repositories. This approach helps to reduce code duplication. Later, we can extract the common classes to a common project shared between all microservices. The next configuration step is to update security configurations and filters to provide the same authorization implementation as Artemis. This also leads to the need to update the generated by JHipster tests. The generation of the microservice project is easy and quick, but the configuration takes some time because we need to align with the Artemis application. Therefore, we label this task as a medium-effort task.

3. Define least coupled group of classes

We know the scope of the service, and we know which classes we need to move. In order to start the iterative microservice extraction approach, we need to start with the classes that have the lowest coupling among those which we will move. In this step, we need to define the group of classes that we will copy in this iteration. It could be a group of one, two, or more classes. We also need to add the test classes to the group. The important part is that the group implements features that we can test. This task is another medium-effort task because we need to find

the easiest classes to copy, which can be more complex if the scope of the service is large.

4. Copy related implementation and tests

After defining the group of classes, we can continue copying them. After each copying of a class, we can use the *@Deprecated* annotation, including a comment, to mark the copied code. This is also helpful for other developers who want to change this code so that we do not accidentally remove changes in the code we copied. Since this task includes only copying of classes, it is easy to execute a low-effort task.

5. Set up communication between services

This step is executed if there is a need for inter-service communication. It happens when one class uses another class that is not in our microservice's scope. Therefore, we need to identify what communication type we need in this case. We use synchronous communication when we do not expect a response, or if we do, we need to ensure the proper implementation of the communication. For example, we implement asynchronous communication by implementing message broker producers and consumers and sending messages over a queue defined for specific needs. In other cases, when we decide that we can create a new endpoint for the specific use, we can use synchronous communication. Synchronous communication can be done via HTTP requests, directly requesting the service, or using API composition in the gateway. The effort for this task depends on how much communication we need to implement. We label it as a medium effort because the implementation of producers and consumers is straightforward but is more complex than just copying code.

6. Refactor code

Several cases require code refactoring. The first of them is when we have implemented communication between services. In this case, we need to refactor the two microservices that communicate to use the consumers and producers, which we implemented in the previous step. Another reason to refactor the code is failing tests. For example, some of the integration tests may fail because they also test features that are now in another service. Therefore, we need to ensure that all tests pass by refactoring the tests. Furthermore, we may need to refactor some of the implementation classes if we need to split them and move only part of their functionality. Since there are several reasons to refactor

the code, we label this task as a high-effort task because we may face all the reasons and need extensive refactoring.

7. Update gateway routes

If the classes we copied include endpoints, we want to route incoming requests for those endpoints to the new microservice. In order to do it, we need to update the gateway routes definition in the *gateway/application.yml* file as described in Section 5.3.2 and Listing 5.1. This is a low-effort task because updating the routes definition is straightforward and includes several examples.

8. Test changes manually

Now we can manually test the features we have copied. We need to make sure we test all the features we have moved. This is a medium-effort task because testing can be straightforward if we have tiny groups of features in each iteration or very extensive in the case of large groups. If some of the features do not behave as they do in the monolith application, we need to refactor the code and fix the issues. In case we have covered all iterations, we continue with step 9. Else we can continue with the next iteration starting again from step 3.

9. Remove extracted functionality and tests from the Artemis monolith

The next step is to remove the extracted functionality and tests from Artemis. Here, we use the *@Deprecated* annotations we added before, and we can easily find the classes which we need to remove. Removal of the classes may also result in a need to implement additional inter-service communication. It can also break tests referring to the extracted functionality. Furthermore, we can find out that the defined scope of the microservice is wrong, and we need to update it. In this case, we go back to step 1. to update the scope. Else, we have finished the service extraction and can continue with the update in the deployment pipeline. This is another medium-effort task because it can either be quick and easy or take a considerable amount of time.

10. Update deployment workflow

In order to deploy the new microservice on a server, we need to update the deployment workflow and add the new component. We need to create or update the Dockerfile of the microservice and configure the create, push and pull operations of the Docker image of the new component in the GitHub workflow, which we will describe in the next

chapter. We also need to update the Docker Compose file of the microservices architecture, which we will also describe in the next chapter. We label this task as a medium-effort task because it might be more time-consuming for people who do not have experience with GitHub workflows, and it also has to be tested.

11. Create Kubernetes deployment resources

A parallel task to the previous one is the creation of Kubernetes deployment resources. The previous task can also be obsolete in the future when Artemis fully migrates to Kubernetes deployment. In order to deploy the new component on Kubernetes, we need to create deployment resources for the new microservice. This includes the creation of a new workload, config map, secrets configuration, and persistent volume claim (PVC) if the microservice needs data volume.

6.4 Deployment

The new architecture requires changes in the deployment process. This section explains the work we did regarding deploying the microservices architecture. We want to deploy Artemis on Kubernetes, but this is not possible right away. We explain the reasons why in the following subsection. Since it is not yet possible to deploy on Kubernetes, we create a deployment pipeline to deploy on a virtual machine described in the second subsection.

6.4.1 Kubernetes Deployment

Unfortunately, it was not possible to deploy on a Kubernetes cluster during this thesis. There were challenges to creating and configuring the Kubernetes cluster on TUM's infrastructure. The Artemis team currently works on providing an on-premise Kubernetes cluster for deploying the Artemis application [Lin21]. There are several challenges that the Artemis team faces. The first of them is that the deployed containers need both IPv4 and IPv6 IP addresses in order to provide intra- and inter- Kubernetes communication [Lin21]. They also need to choose a solution for persistent storage and implement a backup strategy [Lin21]. Moreover, they need to handle the certificate management process, and support secure HTTPS connection [Lin21]. Those are only part of the challenges that the Artemis team faces. Matthias Linhuber describes them in his thesis proposal cited as [Lin21].

For this reason, we deployed Artemis and the new microservice architecture on a local Kubernetes cluster. We create a local cluster using k3d,

which is a lightweight wrapper to run the minimal Kubernetes distribution². According to the k3d documentation², we can easily create and remove Kubernetes clusters with k3d. The clusters run on Docker. We also use Rancher, which is an open-source cluster management platform³. It provides a user interface that helps us to start quickly, stop, (re)deploy, scale, and get details about our applications [SK20]. Another tool we use is kubectl. It is a command-line tool that allows running commands against a Kubernetes cluster⁴. Using kubectl, we can deploy our multi-component architecture with a single command. It is possible due to the Kustomize tool, which is the native configuration management for Kubernetes and built-in kubectl⁵. This tool helps to reuse the resources in order to deploy them in different environments [Mar20]. Kustomize uses a kustomization.yaml file, which declares paths to all resources that kubectl needs to apply on the cluster.

The resource files define Kubernetes objects which are persistent entities in the Kubernetes system⁶. They describe which applications are running, how they are configured and how they behave⁶. There are different types of objects that we use in our Kubernetes deployment.

The first of them is the workload. It is an application that runs on Kubernetes pods which are the smallest units of work on Kubernetes. There are several types of workload resources among which we use Deployment and StatefulSet. The Deployment is a high-level resources which makes it easy to deploy and update applications. We define the configurations in the Deployment resource and Kubernetes takes care of the low-level actions [Luk18]. We use deployments for stateless applications which includes the Artemis monolith, the microservices and the gateway. Other workload type we use is the StatefulSet. It takes care of stateful resources such as the database. StatefulSet provide persistent hostname to each replica with a unique increasing index [BBH19]. This is useful when we remove replicas because we know which pod will be removed. We create StatefulSets for the JHipster Registry and the MySQL database.

Furthermore, we use load balancing objects. The object type we use to expose the gateway, the service registry and the ActiveMQ Artemis console is Ingress. Ingresses are equivalent to virtual hosts which is a mechanism to host many HTTP websites on a single IP address [BBH19]. When we define the Ingresses we also define the host that we will use for our services.

²<https://k3d.io/v5.2.1/>

³<https://rancher.com/>

⁴<https://kubernetes.io/docs/tasks/tools/#kubectl/>

⁵<https://kustomize.io/>

⁶<https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>

We use the following form `{applicationName}.artemis.rancher.localhost`, i.e. `jhipster-registry.artemis.rancher.localhost`.

Other objects we create are the Persistent Volume Claims. They are storage resources that we link to the workloads to store persistent data. We create PVCs for the MySQL database StatefulSet and the Artemis server application Deployment. We define both read and write access and allocate 3GB of memory. This will be insufficient for a production environment. Therefore, we should change it when deploying on production.

All applications require application-specific environment variables. We define them using ConfigMaps. ConfigMaps are key-value pairs that we link to the specific workload they configure. There are separate files for each workload. They contain values for the database URL, the JHipster Registry URL, the host of the ActiveMQ Artemis instance, and other configuration data. ConfigMaps help us deploy the same image on different environments by updating their values [BBH19].

Other similar objects are the Secret. They also define key-value pairs, but they keep sensitive data. Therefore, we use them to store passwords. For example, we use them to store JHipster Registry and the message broker passwords. Kubernetes stores the secrets base64 encoded [BBH19].

Those are the objects we have defined in our local cluster. Using them, we have a running on Kubernetes microservices architecture. We can quickly deploy all components on the cluster using the `kubectl apply` command and the Kustomization file. Moreover, if we update one of the objects and rerun the same command, it will alter only the object we want to update. Kubernetes will not recreate the rest of the objects since they already exist and have not changed. On the other side, we notice that the Artemis application runs slower on the local Kubernetes cluster. The machine we used for development has 16GB of RAM and 2.60GHz Intel Core i7-6700HQ processor with eight threads. The OS installed on the machine is Ubuntu which is a Linux distribution. Therefore, we should measure the performance of the Artemis application on a production cluster to make sure we do not impact the current performance.

6.4.2 Virtual Machine Deployment

Since deploying on Kubernetes is not possible yet, we deployed the microservices architecture on a virtual machine. We do it using GitHub actions. GitHub actions are a sequence of actions that create a GitHub workflow which allows us to automate development workflows in a project repository [CH21]. Workflows are triggered when a developer creates a pull request, creates commits to the pull request, and when the pull request gets merged

[KWGT21]. We should create workflows in YAML, and we should place them in the `.github/workflows/` directory in the project repository [KWGT21].

There are already defined workflows in the Artemis repository. They are responsible for building the application, running the server and client tests, and static code analysis. In order to deploy the new architecture on a test server, we need to create a new workflow that will deploy the required components using Docker images.

Before doing that, we should create a Docker Compose file, including container configurations for all components we want to deploy. Therefore, we create a file including the service registry, the gateway, the Artemis server application, the User Management microservice, the message broker, and the database. We also need to adapt the Dockerfiles for the gateway and the microservice.

Now that we have the Docker Compose file ready, we start creating the new workflow. It consists of several actions, also called jobs. The first is to build WAR files for the Artemis application server, the gateway, and the User Management microservice. Then we upload the artifacts so that we can use them in the following job ⁷.

The following job downloads the uploaded artifacts and builds the `artemis-service`, `artemis-gateway`, and `artemis-usermanagement-service` Docker images. Next, it adds a tag to the image, which we use when deploying artifacts from a specific PR. In the end, it logs in to the GitHub Container Registry and pushes the images we build.

The last job is the deployment job. We have an additional test server (`artemistest6`) allocated for deploying the microservices architecture. The other available test servers still deploy only the monolith application. The first thing we do in the deploy job is to compute the tag we need to use to pull the Docker images. The tag will be `pr-` when deploying artifacts created in a pull request or `latest`, if we want to deploy the develop branch. Then we check the lock of the test server. We need the lock to be able to deploy on the test server, and we check it to make sure that another PR is not using it. Then we need to install the required software for establishing a VPN connection to the test server. In the end, we run `docker-compose` commands first to stop the already running containers, pull the new images and finally start the containers using the new images.

⁷<https://github.com/actions/upload-artifact>

6.5 Discussion

This section discusses interesting findings of the new microservice architecture and limitations it causes.

6.5.1 Findings

The first finding is that microservice extraction is time-consuming, especially for developers who do not have previous experience with the extraction of microservices. Furthermore, it is even more complex if the developer does not fully understand the features that the newly extracted microservice will implement. Therefore, it will take a significant amount of time to migrate the whole application. Additionally, it takes time to thoroughly test the new microservice, which is another reason to use the iterative approach for extraction which we described in section 6.3.4 and Figure 6.7.

Another finding is that the role of the message broker grows. The monolith uses the broker for WebSocket communication, but the microservices architecture extends its usage. As we already described, we use the broker to implement inter-service communication. Therefore, we need to make sure that the broker is highly available to avoid losing messages and communication delays.

6.5.2 Limitations

There are also several limitations which the microservices architecture leads to. The first is that the local development environment becomes more complex than before, and developers need to run several applications in their development environment. They always need to run the service registry and the gateway. They also need to start the Artemis monolith, the User Management microservice, and the Lecture microservice. If they want to test changes to one of the microservices or the Artemis monolith, they do not need to start all three applications but only the one they develop. They also need to start the message broker in order to use the communication between services. To simplify the start of all applications, we can create an IntelliJ Compound configuration that allows us to run several applications together. Another complexity for the developers is that they need to understand the new architecture and the role of each component. They also have to understand the scope of each microservice in order to be able to identify which microservice they need to work on to implement their changes. Moreover, it is hard to implement integration tests that test more than one microservice. There are no such tests currently, which we should improve in the future.

Additionally, the deployment is more complex as well. There are more components that we need to deploy and scale. It is also essential to start the applications in the correct order described in section 5.6.1, to avoid losing user requests.

Chapter 7

Summary

This chapter summarizes the work that was done in this thesis. We discuss the status including realized and open goals, we conclude the thesis contribution and outline future work ideas.

7.1 Status

The section describes the status of the work done in the thesis. We assess the completeness of the requirements realization. We group them in three categories:

- Not implemented
- ◐ Partially implemented, additional work is needed
- Fully implemented and tested requirements

| Requirement | Status |
|--|--------|
| FR1 Retain existing features | ◐ |
| FR2 Show a message about problems in microservices | ○ |
| NFR1 Reliability | ● |
| NFR2 Fault Tolerance | ● |
| NFR3 Security (JWT) | ● |
| NFR4 Security (Unauthenticated Access) | ● |
| NFR5 Security (Inter-service Communication) | ● |
| NFR6 Current Performance | ○ |
| NFR7 Caching Mechanism | ◐ |
| NFR8 Extensibility | ◐ |

| Requirement | Status |
|--|--------|
| NFR9 Maintainability (Microservice's Size) | ◐ |
| NFR10 Maintainability (Deployment) | ◐ |
| NFR11 Scalability | ◐ |
| NFR12 JHipster | ● |
| NFR13 Docker Compose files | ● |
| NFR14 Kubernetes resource files | ◐ |
| NFR15 WAR packaging | ● |

Table 7.1: Status of requirements implementation

7.1.1 Realized Goals

We tried to keep the existing features [FR1] without introducing regressions during the migration of the two microservices. We have implemented the migration of the microservices. However, the implementation of the Lecture microservice is still in review. Furthermore, we need to implement the last step in the migration process - removing the duplicate code from the Artemis monolith. Therefore, since we have not fully finished the migration yet, we have partially implemented this part.

The main benefit of microservices is reliability [NFR2]. If one of the microservices is down, the others will continue working. It is the same in Artemis. For example, if the User Management microservice fails for some reason, the other features that are not related to managing users will continue working.

Fault Tolerance [NFR3] is another goal we realized. The message broker we use ensures that the messages are kept in the queue until it delivered them. ActiveMQ Artemis handles the message persistence using a journal which consists of a set of files on the disk¹.

Security is another important topic for each web application. We managed to fulfill all requirements about security. The microservices, the Artemis application, and the gateway [NFR3] share the same JWT secret, which the applications use to authorize the requests coming from the client. Moreover, we have secured each microservice and the gateway. As a result, unauthenticated users do not have access to application-specific data [NFR4]. The communication between microservices and the Artemis server application is also secured [NFR5]. They cannot send a message through the message broker if they have not authenticated themselves to the broker.

¹<https://activemq.apache.org/components/artemis/documentation/1.0.0/persistence.html>

The Artemis application uses a distributed cache the available Artemis instances share. Since the microservices and the Artemis application share the same database, the microservices should also use the distributed cache [NFR7]. We have partially implemented this goal because we have configured the microservices to connect to the cache, but we have not tested the implementation's correctness. Therefore, we might need to add additional implementation to fulfill this requirement.

We have tried to fulfill the extensibility [NFR8] and maintainability [NFR9] requirements of the microservices by providing them with clear names and making them small with easy-to-understand scopes. However, we have not measured them, which is why we mark them as partially implemented.

Another maintainability requirement is related to the deployment pipeline [NFR10]. We should have an automated deployment pipeline for the new microservice architecture. We have partially implemented this requirement. We deployed the new architecture components - the gateway, the User Management microservice, the service registry on a test server, but we have not deployed it on other environments. Moreover, we still need to add the Lecture microservice to the pipeline.

Scalability is another essential requirement for Artemis [NFR11]. The scalability of the Artemis server application remains unchanged. Theoretically, we can also scale the gateway and microservices, but we need to do additional work there since we have not tested it.

Artemis uses JHipster to develop the monolith application, and we wanted to use it as well for the microservices architecture. Therefore, we used it to generate the gateway and the microservices applications [NFR12].

An essential requirement for the deployment of Artemis is the Docker Compose files [NFR13]. We created Docker Compose files for each new component and the whole architecture which we use to deploy on a test server. Another related to the deployment requirement is the creation of Kubernetes resource files [NFR14]. We have created resource files for the Artemis monolith and the microservice architecture. The ones for the microservice architecture are still in review, which is the reason for the partially implemented status.

The last requirement is about the packaging of the new components. We package the Artemis monolith in a WAR file. We wanted to package the microservices and the gateway in a WAR file [NFR15]. This is possible for each microservice and the gateway.

With regards to the objectives we have defined at the beginning of the thesis, we have fully completed the definition of a migration process and identifying microservices. We have described both of them in section 6.1. We have partially implemented the migration of two microservices. Because of

time reasons, the implementation of the Lecture microservice is still in review. Moreover, as we already discussed, we still need to remove the code which we copied to the two new microservices from Artemis. We will complete this task after the submission of the thesis. There is one more partially implemented objective related to the Kubernetes deployment. We have prepared the Kubernetes deployment resource files for the microservices architecture, also in review. Unfortunately, we deployed the architecture only on a local cluster while creating the resources. The last objective we have defined is implementing a pattern related to the microservice architectural style. We have realized this objective. We have implemented at least two microservices patterns. The first is the "API Gateway" pattern which defines the usage of a gateway as a single entry point to the server-side of the application. We discuss the pattern and the solution in sections 2.2.1, 5.3.2, and 6.3.1. The second is the "Shared Database" pattern which states that the microservice access and write to the same database. We discuss it in sections 2.2.2 and 5.5.

7.1.2 Open Goals

In some cases, the microservices and the communication between them may fail. Therefore, we think it is good to show the user a message about such problems [FR2] so that he can try again later. Unfortunately, we did not have time to implement this feature, and it remains an open goal.

Another open goal is measuring the performance of the new architecture [NFR6]. Unfortunately, we did not have time to fulfill this requirement. However, it will be a measurement that will be interesting to the developers.

7.2 Conclusion

This thesis sets the foundation of the migration of Artemis' architecture towards microservices. We defined the components in the new architecture as well as their roles. We also defined which implementations for service registry and message broker the microservices architecture will use. We created a gateway which is the entry-point to the server-side of the architecture. We also extracted two microservices from the Artemis server monolith application. This helped us to define the generic microservice extraction process. Other developers can use this process to extract other microservices without doing the same research. Another thing we did was to prepare the deployment resource files of the microservices architecture for Kubernetes deployment. They include all objects that need to be created in a Kubernetes cluster to

have a working application.

7.3 Future Work

This section describes the work that remains after finishing this thesis.

7.3.1 Continue with the Microservices Extraction

The microservices migration is a process that takes a huge amount of time. Therefore, it is natural that continuing the work started in this thesis is the first point of the future work section. There are still several services which we can extract from Artemis which will have an even better effect on the development pace and independent scaling. The more microservices we extract from Artemis, the more lightweight each application will be. It will most likely take long until we completely decompose the Artemis server application. However, using microservices will be beneficial for the developers, the system administrators, and the users.

Figure 6.1 illustrates microservices that we could extract from the Artemis monolith in the future. The first of them is the METIS microservice. It is responsible for the course communication between course students, tutors, and instructors via posts, post answers, and reactions. Another potential microservice is the Notifications microservice. It takes care of sending and receiving notifications and managing notification settings. Then, we have the Exercise microservice, which is the biggest microservice among all because it is responsible for managing all exercise types, the assessment of the exercises, and managing exams. Next, the Statistics microservice is responsible for creating all statistics in the application. Last is the Admin microservice, responsible for all administration actions - audit logs, course management and server logs, health, and metrics.

Other developers can from the migration steps we described in section 6.3.4 and Figure 6.7. They can follow them to migrate the rest of the microservices, which can help them start faster with the migration rather than defining new steps. They can also modify them if they find a better approach. Similarly, they can modify the scope of the proposed potential microservices. Moreover, they can define more microservices if the scope is not correctly defined.

7.3.2 Production Kubernetes deployment

Another future goal is to deploy the Artemis application on a Kubernetes cluster. We already described the challenges to do it in the university infrastructure. The Artemis team has already started working on this topic. Once the team solves the challenges and have a functioning Kubernetes cluster, they can deploy Artemis on Kubernetes for each environment. They can use the Kubernetes resource files which we created in this thesis and create copies by modifying them for each environment - test, staging, production.

They can also configure automatic scaling of the components of the architecture. Considering the current state of the architecture, the gateway and the Artemis server application are the components which will require scaling. Therefore, those two components that the team can start configuring. Unfortunately, we did not have time to configure the automatic scaling.

Additionally, the storage that we currently allocate to the persistent volume claims may not be sufficient. Therefore, the team should review it and decide whether it is enough. If not, they should reduce or increase it. The storage for the persistent volume claim of the MySQL database workload is currently 3GB which is not enough. We have defined 3 GB because we deployed on a local cluster, and the machine we used did not have enough memory to allocate more storage.

Furthermore, the Artemis team needs to review the configuration values we extracted in ConfigMaps and Secrets. There could be more values that they should add there or even some that they should move from ConfigMaps to Secrets and vice versa. The need for changes will most likely occur during deployment to the different environments. Since we deployed only on a local environment, we have most likely missed some of the configurations.

Finally, the Artemis team should also review the update strategies of the workloads for the different environments. The strategy that we used for all workloads is the RollingUpdate. This is the default strategy when creating Deployments. This strategy aims to update without downtime by first creating a new replica with the new version of the component [Mar20]. When the replica is in good health, the old version is marked as not ready and removed from the available instances [Mar20]. There are other update strategies available. For example, another available strategy in Rancher is the Recreate strategy. The Recreate strategy is the simplest one. Kubernetes will stop the old version and start the new one, and the two versions will not run simultaneously [Mar20]. It is also possible to define a custom strategy in Rancher. The Artemis team may decide to change the strategies to use another more suitable one.

7.3.3 Migrate to micro frontends

The Artemis client application takes several minutes to build and run on a development environment. We also faced an issue with slow client test execution, which we are gradually solving by optimizing the existing client tests. Therefore, the client application faces similar issues as the monolith server application. Thus, we can migrate the client application to micro frontends. The micro frontends architecture is similar to the microservices architecture concept but in terms of the client application. It involves separating the client application into smaller ones that lead to faster development and better application performance [PAMM20]. Table 7.2 describes a visionary scenario of migration of the Artemis client application to micro frontends.

| | |
|--------------------------------------|---|
| <i>Scenario name</i> | Migration to micro frontends |
| <i>Participating actor instances</i> | alice,bob,john: Developer |
| <i>Flow of events</i> | <ol style="list-style-type: none"> 1. Alice starts with the migration of the client application to micro frontends. She changes the architecture and extracts two micro frontend applications from the client application. First, she extracts a Lecture micro frontend application that is responsible for managing lectures and lecture units. Then she extracts a Discussion micro frontend application that takes care of the discussions between the course participants, including creation, answering and reacting to posts. 2. Bob is a new developer in Artemis and his task is to continue with the migration Alice already started. He also extracts two micro frontend applications from the client application. First, he extracts a Notifications micro frontend application that is responsible for showing user notifications. Then he extracts a Statistics micro frontend application that renders and shows all charts in Artemis. 3. John is the next developer who is interested in migration to micro frontends. He has the task of extracting the last micro frontend application responsible for managing the exercises, assessments and exams. He already has |

| | |
|----------------------|--|
| <i>Scenario name</i> | Migration to micro frontends |
| | examples from the work of Alice and Bob which he can use to finish the migration to micro frontends process. |

Table 7.2: Migration to micro frontends visionary scenario

7.3.4 Research the Availability of the Message Broker

The message broker is an essential part of the microservices architecture. Therefore, we need to ensure that it is a highly available component. According to the ActiveMQ Artemis documentation² which is our main source for this section, we can provide high availability by creating groups of live - backup instances. This means that each broker has one or more backup brokers. They call the first broker the live broker. It is also the only broker that servers ActiveMQ Artemis clients. The backup brokers are waiting for a failure to occur. Suppose it happens, one of the backup brokers becomes the live broker. If the live broker that has failed becomes again in good health, it has a priority over the other backup brokers, and it is the next one that will become the live broker.

There are also three strategies for the creation of backup brokers. The first is replication, where the live and the backup brokers do not share the same data storage. Instead, the backup brokers will duplicate the persistent data they will receive over the network. Therefore, when a backup broker starts up, it should synchronize the data, leading to a delay before it becomes fully operational.

The second strategy is the shared store in which the live and backup server share the same data directories. Therefore, this strategy eliminated the need to synchronize the data. However, on the other side, it requires a file system that is accessible by both the live and backup broker. Moreover, when the backup broker starts up, it needs to load the journal from the storage, which can also take some time if the amount of data is significant.

The third strategy is live only, which means there are no backup brokers.

There is an ongoing development in ActiveMQ Artemis related to Broker Balancers³ which allows distributing the load among different available brokers. This feature is still experimental, but we can keep an eye on it because it is something we can use in the future.

²<https://activemq.apache.org/components/artemis/documentation/1.0.0/ha.html>

³<https://activemq.apache.org/components/artemis/documentation/>

List of Figures

- 4.1 Simplified UML component diagram showing the current server architecture of Artemis. The figure is adapted from the component diagram in the GitHub Artemis documentation¹ 20

- 4.2 UML component diagram showing the proposed server architecture of Artemis. The Artemis client communicates with the gateway which routes the request to the Artemis server, the User Management microservice or the Lecture microservice. The Artemis server and the two microservices communicate with each other through the message broker using JMS and depend on the database. The gateway, the Artemis server application and the two microservices register themselves in the service registry which keeps track of their instances 22

- 4.3 UML communication diagram depicting the communication between instances in user account registration process. A student registers himself an account in the system which sends a request to the API gateway, then the API gateway redirects the request to the User Management microservice. The microservice creates the account and sends a message through the message broker to the Artemis server application to send an account activation email. 26

- 4.4 UML communication diagram depicting the communication between instances in retrieving lecture details. A student open a lecture in the Artemis client application which sends a request to get the lecture details from the API gateway, which then redirects the request to the Lecture microservice. 26

LIST OF FIGURES

| | | |
|-----|--|----|
| 5.1 | UML communication diagram illustrating the communication between the service registry and the gateway, the Artemis server application, the User Management microservice and the Lecture microservice. All components register their instances in the registry. The gateway fetches the registered instances including details about their location. | 30 |
| 5.2 | UML communication diagram showing example communication between the Artemis server application and the User Management microservice through the message broker where response is required. The Artemis server application puts a message in a queue which the message broker sends to the User Management microservice. The microservice puts the response message in a response queue which the broker delivers to the Artemis server application. | 33 |
| 5.3 | UML component diagram illustrating the User Management microservice | 34 |
| 5.4 | UML component diagram illustrating the Lecture microservice | 35 |
| 5.5 | UML deployment diagram of the microservices architecture. The diagram is adapted from the deployment of the monolith application described in <i>Securing and Scaling Artemis WebSocket Architecture</i> by Simon Leiß [Lei20] | 36 |
| 5.6 | Dependency Graph describing the order to startup the subsystems. The gateway depends on the Artemis application server and the two microservices. The Artemis application server and the two microservices depend on the message broker, the database, and the service registry. The diagram is adapted from the dependency graph for the monolith application described in <i>Securing and Scaling Artemis WebSocket Architecture</i> by Simon Leiß [Lei20] | 38 |
| 6.1 | Migration Strategy for migrating the Artemis server application to microservices. The diagram describes the microservices that could be extracted over 1.5 years. The diagram is adapted from the "Strangler application" migration strategy described by Chris Richardson in [Ric19]. | 41 |
| 6.2 | Proposed architecture where Artemis application serves also gateway features. It handles the server requests and redirects only the requests related to the microservices. | 42 |
| 6.3 | Proposed architecture where the Artemis Application is in front of the gateway. It routes only requests to the extracted microservices. | 43 |

| | | |
|-----|---|----|
| 6.4 | Proposed architecture where the Artemis server is an independent application which registers to the service registry similarly to the extracted User Management microservice. The client sends the requests to the gateway which redirects them to the proper server application. | 45 |
| 6.5 | Configuration for the generation of the User Management microservice | 48 |
| 6.6 | Configuration for the generation of the Lecture microservice . | 50 |
| 6.7 | UML activity diagram illustrating the steps for extracting a microservice from the monolith. | 53 |

List of Tables

| | | |
|-----|---|----|
| 6.1 | Endpoints extracted in the User Management microservice . . . | 49 |
| 6.2 | Endpoints extracted in the Lecture microservice | 51 |
| 7.1 | Status of requirements implementation | 64 |
| 7.2 | Migration to micro frontends visionary scenario | 70 |

Bibliography

- [AVSTK18] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Torero, and Ferhat Khendek. *Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned*. 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), 2018.
- [BBH19] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes: Up and Running, 2nd Edition*. O'Reilly Media, Inc., 2019.
- [BD10] Bernd Bruegge and Allen H Dutoit. *Object Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, 2010.
- [CH21] Chaminda Chandrasekara and Pushpa Herath. *Hands-on GitHub Actions: Implement CI/CD with GitHub Action Workflows for Your Applications*. APress Media, LLC, 2021.
- [DLL⁺18] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. *Microservices: How To Make Your Application Scale*. Springer International Publishing AG, 2018.
- [Ing18] Joseph Ingeno. *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing Ltd, 2018.
- [JPM⁺18] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. *Microservices: The Journey So Far and Challenges Ahead*. IEEE Software, 2018.
- [KS18] Stephan Krusche and Andreas Seitz. *Artemis - An Automatic Assessment Management System for Interactive Learning*. SIGCSE '18, 2018.

BIBLIOGRAPHY

- [KWGT21] Timothy Kinsman, Mairieli Wessel, Marco A. Gerosa, and Christoph Treude. *How Do Software Developers Use GitHub Actions to Automate Their Workflows?* 2021.
- [Lei20] Simon Leiß. *Securing and Scaling Artemis WebSocket Architecture*. 2020.
- [Lin21] Matthias Linhuber. *Towards a Kubernetes Supported Learning Infrastructure at Scale*. Master’s Thesis Proposal, 2021.
- [LML20] Chia-Yu Li, Shang-Pin Ma, and Tsung-Wen Lu. *Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System*. IEEE, 2020.
- [Luk18] Marko Lukša. *Kubernetes in Action*. Manning Publications, 2018.
- [Mar20] Philippe Martin. *Kubernetes: Preparing for the CKA and CKAD Certifications*. APress Media, LLC, 2020.
- [New19] Sam Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O’Reilly Media, Inc., 2019.
- [NH19] Pia Niemelä and Heikki Hyyrö. *Migrating Learning Management Systems Towards Microservice Architecture*. 2019.
- [PAMM20] Andrey Pavlenko, Nursultan Askarbekuly, Swati Megha, and Manuel Mazzara. *Micro-frontends: application of microservices to web front-ends*. Journal of Internet Services and Information Security (JISIS), volume: 10, number: 2 (May 2020), pp. 49-66, 2020.
- [PMA19] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. *Migrating from monolithic architecture to microservices: A Rapid Review*. 38th International Conference of the Chilean Computer Science Society, 2019.
- [RF20] Mark Richards and Neal Ford. *Fundamentals of Software Architecture*. O’Reilly Media, Inc., 2020.
- [Ric19] Chris Richardson. *Microservices Patterns*. Manning Publications Co., 2019.
- [Say17] Gigi Sayfan. *Mastering Kubernetes*. Packt Publishing Ltd, 2017.

- [SD20] Prabath Siriwardena and Nuwan Dias. *Microservices Security in Action*. Manning Publications Co., 2020.
- [SGP19] Chellammal Surianarayanan, Gopinath Ganapathy, and Raj Pethuru. *Essentials of Microservices Architecture: Paradigms, Applications, and Techniques*. Taylor Francis, 2019.
- [SK20] Deepu Sasidharan and Sendil Kumar. *Full Stack Development with JHipster - Second Edition*. Packt Publishing, 2020.
- [Sri21] Rajiv Srivastava. *Cloud Native Microservices with Spring and Kubernetes: Design and Build Modern Cloud Native Applications using Spring and Kubernetes*. BPB Publications, 2021.