Uni1 Application to Containers

MASTER THESIS

Nasser Eddin Nasser

Submitted on 13 July 2021

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisors:
Georg Schwarz, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 13 July 2021

License

Erlangen, 13 July 2021

Abstract

In the age of cloud computing, it is critical for software to handle any functionality increase and run on different platforms. Microservices architecture is the trend right now since it allows the creation of expandable programs.

Most of the cloud-based solutions are using this architecture thanks to its benefits in large projects. However, container technology is ideal for deploying a microservices application because it simplifies the process without sacrificing speed or efficiency.

Furthermore, it is common in a microservices project to have frequent deployments where new features are being added regularly. Nevertheless, using a microservices architecture raises cloud costs because multiple applications (services) must be deployed.

This thesis provides a new administration component for the existing Uni1 application. Moreover, the current application components have been divided into containerized components in preparation for a complete microservices switch. In addition, it provides multiple deployment approaches to reduce the costs on the cloud.

The result is a new cross-platform Uni1 version that supports multiple deployment ways. Furthermore, the thesis describes the structure of Uni1 and the integration concept, including the authentication and authorization concept.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 1
  - 1.2.1 Maintainability and Extensibility . . . 1
  - 1.2.2 Vendor Lock-in and Testability (similar environments) . . . 2
  - 1.2.3 Optimize Cloud Costs . . . 2
- 1.3 Purpose of the Thesis . . . 2
- 1.4 Methodology . . . 2
- 1.5 Structure of Work . . . 3

2 Background **4**
- 2.1 Microservices . . . 4
  - 2.1.1 Advantages and Disadvantages . . . 4
- 2.2 Docker Container . . . 5
  - 2.2.1 Advantages and Disadvantages . . . 5
- 2.3 Uni1 application . . . 6
  - 2.3.1 Uni1: Previous version (Marketplace) . . . 6
  - 2.3.2 Uni1: New version (Uni1Next) . . . 6

3 Thesis Requirements **8**
- 3.1 Authentication and Authorization Concept . . . 8
- 3.2 Administration component . . . 9
- 3.3 Deployment Concept . . . 10
- 3.4 Integration, Communication and Extensibility Concept . . . 11

4 Architecture and Design **12**
- 4.1 Logical distinction between Processe, Container, and Microservice 12
  - 4.1.1 Definitions . . . 12
  - 4.1.2 Logical Coherence and Development Stages . . . 14
- 4.2 Evaluation of Containers and Virtual Machines . . . 17
  - 4.2.1 Structure Evaluation . . . 17
  - 4.2.2 Performance and Creation Time Evaluation . . . 18

4.3 Container Technology and its Application 21
4.3.1 Container Technologies 21
4.3.2 Container Application 22
4.4 Design of Communication and Integration Mechanisms 22
4.4.1 Authentication and Authorization 22
4.4.2 Communication Technology 23
4.4.3 User Interface Integration Technology 24
4.5 Design of Admin Console component 25
4.6 Concept for a set-up in AWS 27
4.6.1 Monolithic 27
4.6.2 Microservices 28

5 Implementation **29**
5.1 Administration component 29
5.1.1 Frontend 29
5.1.2 Backend 30
5.1.3 Authentication and Authorization 30
5.1.4 Mapping user data from Cognito and Database 31
5.1.5 Communication 31
5.2 Database 32
5.3 Container 32
5.3.1 Production environment - Monolithic 33
5.3.2 Development environment - Microservices 34
5.4 Environment variable 36
5.4.1 Development environment - Microservices 37
5.4.2 Production environment - Monolithic 38
5.5 Deployment 38
5.5.1 Frontend 38
5.5.2 Backend 39
5.6 Integration 41
5.6.1 Communication 41
5.6.2 Authentication and Authorization 42
5.6.3 Backend Components 44
5.6.4 Dev Tools 45
5.6.5 UI Integration 45

6 Evaluation **47**
6.1 Authentication and Authorization Concept 47
6.2 Administration component 47
6.3 Deployment Concept 48
6.4 Integration, Communication and Extensibility Concept 49
6.4.1 Integration 49
6.4.2 Communication 49

6.4.3 Extensibility 49
6.5 Summary 49

7 Conclusions **51**
7.1 Summary 51
7.2 Future Work 51

Appendices **53**
A REST API Overview of Admin-Console Service 54
A.1 Contact Route 54
A.2 User Route 54
B REST API Overview of Dashboard Service 55
B.1 Dev Route 55
B.2 Users Route 55
B.3 Contact Route 55
B.4 Tags Route 55
C REST API Overview of Campaigner Service 56
C.1 Campaigns Route 56
C.2 Replies Route 56
C.3 Bounces Route 56
C.4 Tags Route 56
C.5 Contacts Route 56

References **57**

Acronyms

AWS Amazon Web Services

IAM AWS Identity and Access Management

ECS Amazon Elastic Container Service

S3 Amazon Simple Storage Service

KID key ID

UI User Interface

URL Uniform Resource Locator

DB Database

API Application Programming Interface

JWT JSON Web Token

JS Java Script

MVVM Model View Viewmodel

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

OIDC OpenID Connect

1 Introduction

The final goal of the Uni1 application is to switch the application from the existing monolith application to a fully microservices architecture. In this thesis, the existing Uni1 components will be divided into containers, the containerized components will be integrated, and a new containerized component will be added to handle the access rights and user accounts.

1.1 Motivation

The improvement of innovation is related to strong and trusted collaborations. Over recent decades, research and innovation in science, business, and society have grown between the University and other organizations. Friedrich Alexander University of Erlangen-Nuremberg (FAU) has a global network of large, small, private, national, and global partners.[1] That leads to a big challenge to organize, customize and filter the Interest of both sides. The group of Prof. Dr. Dirk Riehle founded Uni1, which is a tool to organize, customize, and filter the offers of the Open Source chair in FAU and the other partners. Nowadays, it is important for software to be able to treat any functionality increment and easily growth due to business requirements. In other words, the Uni1 application should be extensible and can be deployed on multiple operating systems and hardware platforms.

1.2 Problem Statement

1.2.1 Maintainability and Extensibility

In the age of digitalization, the number of applications is significantly increased. Today's apps are becoming increasingly complex and offer numerous services. As a result, adding new functions to an existing application is challenging. Especially that today's applications are developed with agile methodology. In the agile

[1] https://www.fau.eu/outreach/innovation-and-start-ups/partnerships/

world, the software must be prepared to gain new functions or eliminate existing ones.

The existing Uni1 program will receive new features, such as sending emails and managing user accounts. Uni1 is a monolithic application. When a new component is required in the Monolithic architecture, the whole application structure must be modified. Microservices architecture provides the best solution for these issues. Microservices architecture provides a functionality increment capability, allowing new components with a single functionality to be added to the application without restructuring the application.

1.2.2 Vendor Lock-in and Testability (similar environments)

Because of the success of cloud services, many applications are now hosted in a public cloud system. As a result, the application deployment in a production environment, which differs from the development environment, becomes problematic. Furthermore, When switching from one cloud to another leads to a vendor lock-in issue. Container technology is the ideal solution to this problem because it provides a platform-independent deployment.

1.2.3 Optimize Cloud Costs

An application's deployment in a public cloud can be costly. Especially if it is a microservices application consisting of numerous services, each service is contained in a container and deployed on the cloud independently. Depending on the predicted usage, multiple deployment set-ups will be possible to use in order to reduce costs. The application can be deployed as microservices if a high level of utilization and extensibility is required; otherwise, the application is deployed as a monolithic application.

1.3 Purpose of the Thesis

The purpose of this engineering thesis is to expand the current Uni1 application with a new administration feature, divide Uni1 components into containers, integrate them, and provide multi-deployment approaches.

1.4 Methodology

The process of the implementation of the thesis requirements based on the agile software development cycle. The participants in the development of the software are listed below.

Participants and Roles:

- Prof. Dr. Dirk Riehle: Sponsor and Product Owner
- Georg Schwarz: Supervisor
- Felix Müller: Software Developer
- Nasser Eddin Nasser: Software Developer

(Müller, 2021) worked in parallel on his bachelor thesis and contributed several features in the next generation of Uni1. Some tasks were developed together due to their overlap in scope. Joint tasks with Mr. Müller are labeled as such in this thesis.

Materials:

- Software developers meetings: held twice a week for the software developers to synchronize and discuss technical problems.
- Product owner and developers meetings: around once a month, all participants get together. The software developers can describe their implementation process (discuss issues). The product owner can provide new requirements.
- Communication channels: we utilized emails to schedule meetings or to discuss the requirements.
- Software project management tool: GitLab Issue Board was used to plan, organize, and visualize the implementation workflow.

1.5 Structure of Work

In chapter 2 the principles of Microservices architecture and container technology will be addressed first, followed by a discussion of the existing Uni1 application, with an emphasis on its components and additional features. The prerequisites are then specified in chapter 3. The architecture and design of the program, as well as container technologies and their application, will be described in chapter 4. The implementation will be discussed in caption 5. later, the implementation will be assessed to confirm that it meets the requirements in chapter 6. Finally, a conclusion including the future improvement is addressed in chapter 7.

2 Background

The first section of this chapter discusses the microservices architectural style, its definition, and its benefits and drawbacks. The second section explains the term of Docker container and its benefits and drawbacks. In the end, the current existing Uni1 application (the old and the new version) will be discussed.

2.1 Microservices

Microservice has no universal definition. One way to define it is as an architectural style in which an application is structured as a set of services. Each service has a single function. That deployed, scaled, and tested separately with a highly coupled loosely. (Thönes, 2015)

2.1.1 Advantages and Disadvantages

Microservices architecture style has grown in popularity in recent years. Specifically, that is very useful in the age of containerization and cloud computing. Each microservice can be developed and deployed on a different platform, using different programming languages and developer tools. Microservices communicate with one another through APIs and communication protocols, but they do not depend on one another in any other way. (Monus, 2018)

Advantages
The most significant advantage of microservices architecture is that development teams can independently build, manage and deliver each microservice independently. This form of single responsibility has additional advantages. Microservice-based applications scale faster since only a single service can be scaled independently if needed. Microservices also reduce the time to market and accelerate the CI/CD pipeline. Microservices architecture style also brings higher agility. Furthermore, isolated services have a higher failure tolerance. Overall, a lightweight microservice is simpler to manage and debug than a complicated program. (Monus, 2018)

Disadvantages
However, while independent microservices are more fault-tolerant than monolithic systems, the network is much less fault-tolerant than monolithic systems. Communication between microservices can lead to poor performance because sending messages back and forth incurs some overhead. After all, the development team must handle the microservice's whole lifecycle, from beginning to end. (Monus, 2018)

2.2 Docker Container

A container is a software unit that packages code and dependencies to move from one computing environment to another quickly and reliably. A Docker container image is a small, standalone software package that contains everything needed to run an application, including code, runtime, system tools, system libraries, and settings. ('What is a Container?', 2021)

Docker is a tool for running applications inside isolated containers. Docker consists of Docker image, Docker container and Docker engine. Docker image is a small, independent, executable software package that contains everything needed to execute an application. Docker image becomes a container when it executes on a Docker Engine. ('What is a Container?', 2021)

2.2.1 Advantages and Disadvantages

Over the last few years, Docker has become more popular because of its benefits. The major reason for this is the reduction of software costs. Former software were restricted to specific hardware or OS. Today's software can be cross-platform through using container technology as it can be deployed on different environments, operating systems, hardware, and cloud systems. Docker is not a magic solution, and it also has disadvantages as well. In this paragraph, a few advantages and disadvantages will be addressed.

Advantages
Based on ('What is a Container?', 2021) Docker technology benefits are:

- Standard: Docker containers can be transported anywhere.
- Lightweight: Containers share the machine's OS system kernel and eliminate the need for an OS per application, increasing server efficiency and lowering server and licensing costs.
- Secure: Containerized applications are more secure than deploying an application directly on the host machine because Docker provides an isolation environment feature.

Disadvantages

- Missing features: Docker containers continue to lack features such as container self-registration, self-inspection and copying files from the host to the container. ('Advantages and Disadvantages of Docker - Learn Docker', 2018)
- Data in the container: backup and recovery strategy is needed to avoid data loss when the container goes down. ('Advantages and Disadvantages of Docker - Learn Docker', 2018)

2.3 Uni1 application

The concept of the application is to simplify the process of how universities with companies participate in projects with students, that both sides can perform profit. Companies benefit through recruiting, outsourcing, and innovation resulting from the projects. Universities win new partners, earn money on the projects, and offer more attractive teaching. (Riehle, 2016)

The next generation of the Uni1 app will have additional functions, such as managing users' accounts, contacts, and emails.

2.3.1 Uni1: Previous version (Marketplace)

The Uni1 application is the core of the Marketplace component. That used to simplify the process of how universities with companies participate in projects with students. It has a client-server architecture and uses the following technologies:

Frontend: the frontend of the Uni1 uses ReactJS with Redux5 for managing the application state. ReactJS is a JavaScript framework designed by Facebook.

Backend: the backend is developed with js and a NodeJS framework. The backend server is a CRUD server between the frontend and Database.

Database: MongoDB is used, which is a NoSql database.

2.3.2 Uni1: New version (Uni1Next)

Besides the marketplace, the next generation of Uni1 supports other functions; they are all integrated together as a set of microservices. Some of the functions

(i.e., managing users' accounts) will be implemented in this thesis (more about the implementation in chapter 5). Uni1Next components are as following:

Marketplace components: is the old Uni1. The components will be redesigned and adjusted to meet the new Uni1 logic since there were some changes in the authentication administration, DB structure, and functionality. The Marketplace components contain a frontend and backend component.

Administration components: used to manage and monitor Uni1 users' accounts and contacts. The Administration components contain a frontend and backend component.

Dashboard components: used to log in, register, change user personal information and navigate between the different services of Uni1. More about it in (Müller, 2021). The Dashboard components contain a frontend and backend component.

Campaigner components: used to create and monitor the campaign emails and their replies. More about it in (Müller, 2021). The Campaigner components contain a frontend and backend component.

Database: the backend(s) of the listed components are connected to a database used to store the data. The database is MongoDB and hosted on AWS.

3 Thesis Requirements

This chapter lists the functional and non-functional requirements of Uni1.

3.1 Authentication and Authorization Concept

To make sure that the services are secured. Authentication and authorization mechanism must be used to ensures that a user is whom its claim to be (**Authentication**) and to check if the user has access rights to a service (**Authorization**).

Role Concept Uni1 provides three services which are **Marketplace**, **Campaigner** and **Administration**, accordingly three roles are existing as the following:

> **User role:** default role (base role), user can access the Marketplace component.
>
> **Campaigner role:** user can access the Campaigner component.
>
> **Administrator role:** user can access the Admin-Console component.

The roles are independent. User type will not be considered for the roles. Suppliers and consumers do not exclude each other (professor can be a business) and can have Administrator role or/and Campaigner role or/and User role.

Requirement key points

- Authentication and authorization concept must work with monolithic and microservices deployment.
- Authentication and authorization concept is valid for all Uni1 components.
- Only authenticated users can use Uni1 services.
- The corresponding component can only be accessed by a user who has a valid access role.

- Users can only see the available components and functions based on their access rights (i.e., a user without a campaigner access right cannot see or access the campaigner component).
- Independent access roles: a user can have many access roles.
- At least one access role must be assigned to an activated user.
- A newly established user account must be assigned as a user access role by default.

User stories The following are some user stories that will help to clarify the requirement.

1. As a marketplace user, I want to have access to the Marketplace services in order to explore the available projects and propose new ones.
2. As a campaigner, I want to have access to the Campaigner services in order to start and operate campaigns.
3. As an administrator, I want to have access to the Admin console in order to allow and restrict users' access according to their needs.
4. As a user, I want to have multiple access roles in order to access different services.

3.2 Administration component

It is relevant for the admins to be able to control users' accounts and their contacts. A new component must be implemented to simplify the monitoring and managing process of users' accounts and their contacts.

Admin Console is a new Administration component added to Uni1 to manage and monitor Uni1 users' accounts.

Requirement key points

- User interface shall be user friendly.
- User interface shall be in the German language.
- Admin Console must be integrated with the other Uni1 components.
- Only users with administration rights can access it.
- Admin-Console user is capable of assigning a role to a user.
- Admin-Console user is capable of dismissing a role from a user.

- Admin-Console user is capable of releasing a contact from a user.
- Admin-Console user is capable of importing new contacts.
- Admin Console user is capable of exporting the contacts.

User stories The following are some user stories that will help to clarify the requirement.

1. As an administrator, I want to monitor and manage users' accounts in order to assign a role to a user or to dismiss a role from a user.
2. As an administrator, I want to monitor and manage users' accounts in order to activate/deactivate and delete an account.
3. As an administrator, I want to monitor and manage the contacts in order to add and remove a contact.
4. As an administrator, I want to monitor and manage the contacts in order to let users without accounts receive campaigns.

3.3 Deployment Concept

The deployment concept specifies how Uni1 deployment will occur, principally that Uni1 should support multiple deployment setups.

Requirement key points

- Dependencies between components should work with monolithic and microservice deployment.
- Consistent and isolated environment: regardless of where the components are deployed, everything remains consistent.
- Cost-effectiveness.
- Uni1 can be deployed as monolith and microservices.
- Ability to run anywhere: Uni1 components shall be able to run in different OS and clouds.

User stories The following are some user stories that will help to clarify the requirement.

1. As a user, I shall be able to access the Uni1 at any time so that I can use the application.

2. As a project sponsor, I want to choose between different deployment setups in order to minimize costs depending on the expected usage.
3. As an operator (developer), I want to have a portable application in order to run it on different kinds of deployment environments.

3.4 Integration, Communication and Extensibility Concept

The integration concept is the way of bringing all Uni1 components together into a single system that reacts as one. The communication concept involves the process of the transmission of the data between the Uni1 components. The concept of extensibility refers to Uni1's ability to gain additional components and functions.

Requirement key points

- Integration: Uni1 components must be able to collaborate together in order to give the impression of a monolithic application, even if it is a microservices application.
- Connection: Uni1 component must be able to communicate with other Uni1 components and with the outside.
- Extensibility: Uni1 shall be able to gain new components easily.

User stories The following are some user stories that will help to clarify the requirement.

1. As a user, I shall have the feeling of using one application, even if it's a microservice application, in order to get a good user experience.
2. As a project sponsor, I want to have an extensible application in order to expand it with new features.

4 Architecture and Design

This chapter presents the architecture and design of Uni1. The first section of this chapter covers the logical difference between process, container, and microservice. The second section contains different technology for virtualization that can be used in Uni1. The methods of communication and integration will then be examined. However, the design of an administration component will be investigated. Finally, the concept of AWS setup employing microservices and monolithic architecture will be investigated.

4.1 Logical distinction between Processe, Container, and Microservice

4.1.1 Definitions

Process: in computing is an instance of a program that is being executed by one or more threads. It includes the software code as well as the program's operation. Depending on the operating system (OS), a process can be composed of multiple threads of execution that execute instructions concurrently. (Silberschatz & Galvin, 2013)

Microservice: there is no universal definition of a microservice. One way to think of it is as a small application with a single responsibility that can be deployed, scaled, and tested independently (Thönes, 2015).

As shown in figure 4.1, if a new component is required in the Monolithic Architecture, the entire structure must be updated. In Microservices Architecture, a new component can be simply added and integrated with the entire system. Another advantage is that if one application service is in great demand, it can scale and be updated independently of the others.

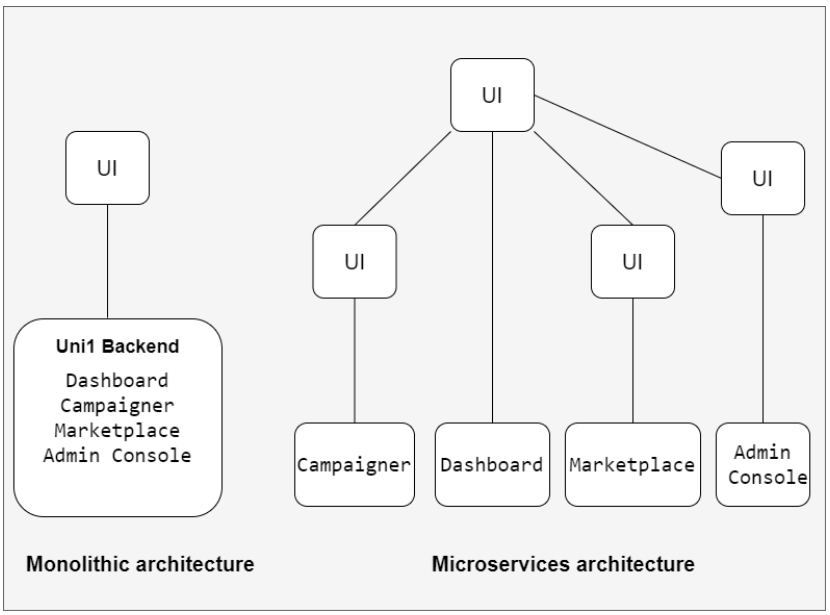

Figure 4.1: Monolithic vs Microservices

Docker container image: is a small, standalone software package that contains everything needed to run an application, including code, runtime, system tools, system libraries, and settings. ('What is a Container?', 2021)

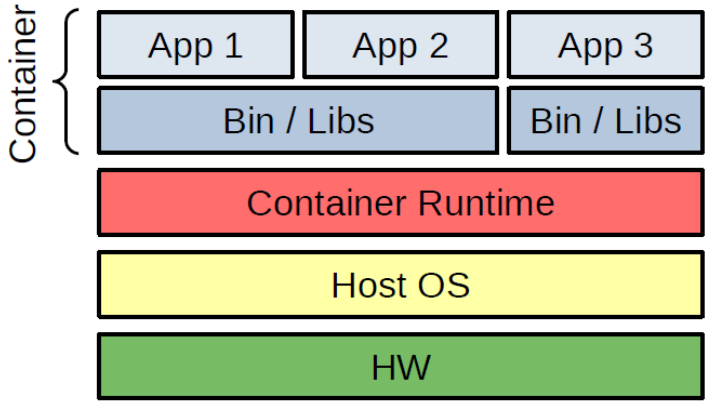

Figure 4.2: Containers architecture. (Riehle, 2020)

As illustrated in figure 4.2, containers encapsulate distinct components of application logic that are only given the resources they need.

4.1.2 Logical Coherence and Development Stages

Software development has three primary phases: Design time, Compile-time, and Runtime. The initial step is the design time, where a developer writes a source code. The second step is the compile-time, which compiles this code into the machine code to turn it into an executable program. The third stage is when the executable code is running. The table below shows the main building steps of the docker container and Microservice. Each development step will be discussed in this section for the Docker container, Microservice and Microservice with Docker.

Development Stage	Microservice	Docker Container	Microservice with Docker Container
Design Time	Codebase	Dockerfile	Codebase with a dockerfile
Compile Time	Microservice as artifact	Docker Image	Docker Image
Runtime	Microservice Instance	Docker Container	Docker Container

Design Time

Codebase The developer writes the code of the software as a codebase. The codebase of a microservice depends on the programming language of the service. It can be Java, C++, JS, or any other programming language.

Dockerfile The developer writes the code structure of a container in the dockerfile. A Dockerfile is a text document that explains how a Docker image is built. The layers have a pyramid shape, where the new layer is created on top of the previous layer. The following are the fundamental instructions for writing a dockerfile. Each instruction will create one new layer. More instructions can be found in docker docs ('Best practices for writing Dockerfiles', 2021).

- **FROM:** the definition of the base image.
- **WORKDIR:** creates a directory for the application.
- **COPY:** copy files inside the container.
- **EXPOSE:** declare ports used inside a container. For the external access, another port must be set using "-p" in the docker run command.
- **CMD/ENTRYPOINT:** the executed command(s), when running the image.

Compile Time

The second step in the software building is Compile-time, which refers to the time of transforming the programming code into machine code (i.e., binary code).

Microservice as an artifact (Executable code) The codebase will be converted to executable code using a corresponding compiler. The executable code can be, i.e., a file with .exe format or a jar file.

Docker Image Building a Dockerfile executes a Docker image. A Docker image contains application code, libraries, tools, dependencies and other necessary files for running an application.

Runtime

The run-time is when an executable code runs.

Microservice Instance After running the artifact of a microservice, a running instance of that microservice will be created.

Docker Container After running a Docker image by the Docker engine, a running instance of the image will be created, which is the Docker container. A container is an instance built according to the image as a guide at the runtime.

Microservice and Container The microservices term is an architectural style, whereas the container is a mechanism to build and run an application efficiently. A Microservice (with a docker container) is a deployment artifact of the docker image. Containers exist without microservices, even though containers provide an excellent way to deploy microservices. A Microservice may run with a container, but it could also run as a fully provisioned VM. Nevertheless, building microservices with containers (Docker containers) is a great fit. At the same time, Microservices is about self-contained systems. A Docker container provides the ideal environment for this approach. Containers secure the microservices by isolating them from one another and the underlying infrastructure. Containers are not restricted to any particular infrastructure that can operate on any device, infrastructure, or cloud. A container is an instance of a microservice (built according to the image as a guide at runtime).

In the runtime environment of a docker container, the host operating system can access computing resources and essential services such as IO and Network.

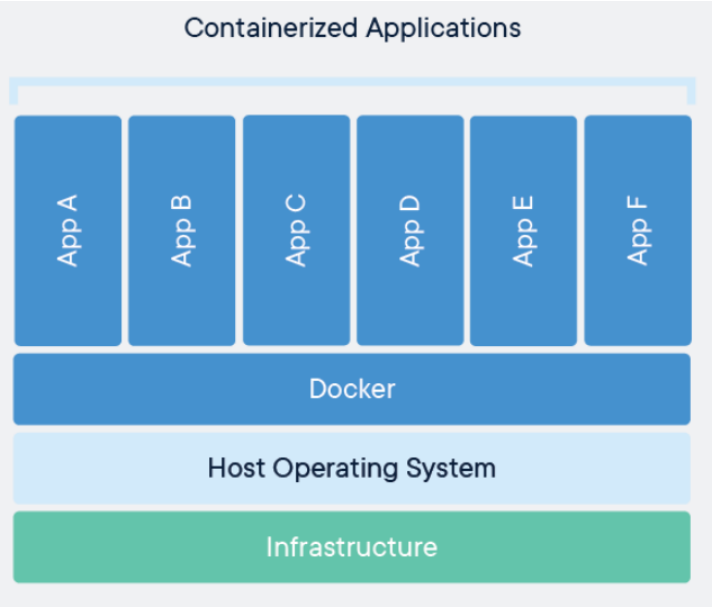

Figure 4.3: Multiple Docker containers on the same host. ('What is a Container?', 2021)

As shown in the 4.3, many containerized applications (i.e. microservices) can run on the same host. All containers share the host operating system's resources. As a result, resources are reduced and the build/run process is accelerated (because the environment packages only need to be installed once).

Process and Container It is good for the rule of thumb to limit each container to one process, but it is not a restricted rule. It makes it easier to scale horizontally and reuse containers when applications are decoupled into multiple containers. Containers, for example, can be spawned not only with a single process, but some programs may also spawn additional processes on their own. ('Best practices for writing Dockerfiles', 2021)

- A process represents a running program; it is an instance of an executing program.
- A process consists of memory and a set of data structures.
- It is possible to run many processes in a container, but It is not always a good idea.

As a summary: while the concept of the microservice is about a single functionality. Moreover, a container should only contain a single process, which leads to a single container should contain only one microservice.

4.2 Evaluation of Containers and Virtual Machines

Moving software from one computing environment to another can be very difficult. As a result, an appropriate container or virtualization technology should be used to ensure that the Uni1 application moves smoothly and reliably from one computing environment to another and from monolithic to microservices architecture.

First, we will contrast the structure of container and virtual machine technology. Second, we will compare their performance based on experiments done by (Amaral et al., 2015), where the performance was explored of two Container-based environments (Master-slave and nested) and traditional virtual machines.

4.2.1 Structure Evaluation

Virtualization technology is about using a virtual machine, as shown in figure 4.4, the VM includes an entire operating system as well as the application. On the other side, container technology shares the operating system with the applications. (Riehle, 2020)

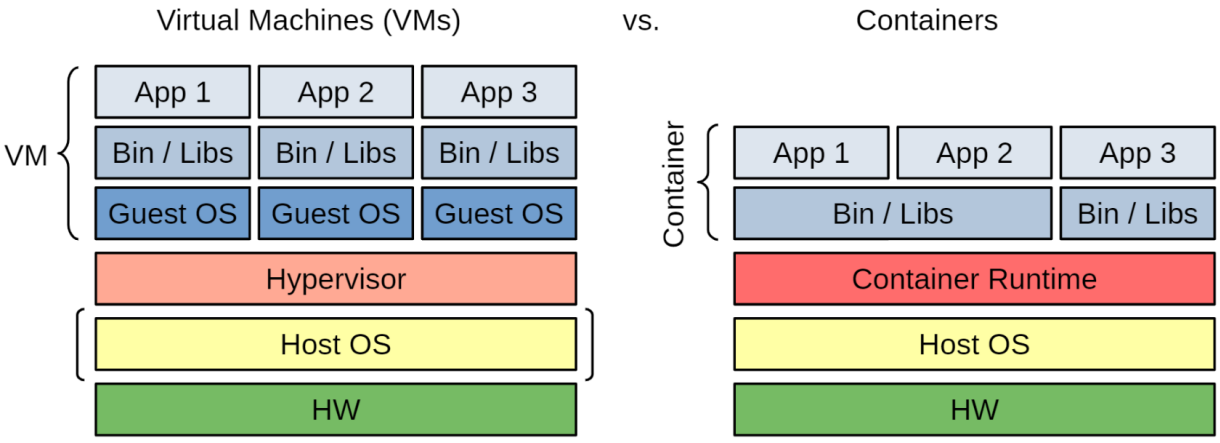

Figure 4.4: Virtual Machines vs Containers. (Riehle, 2020)

As a result: the container technology improves the deployment speed, provides a faster reboot, less resource overhead, and more lightweight in comparison with a virtual machine.

4.2.2 Performance and Creation Time Evaluation

The purpose of the experiments done by (Amaral et al., 2015) is to explore the efficiency and overhead of containers, which can be critical in the deployment of a microservices system.

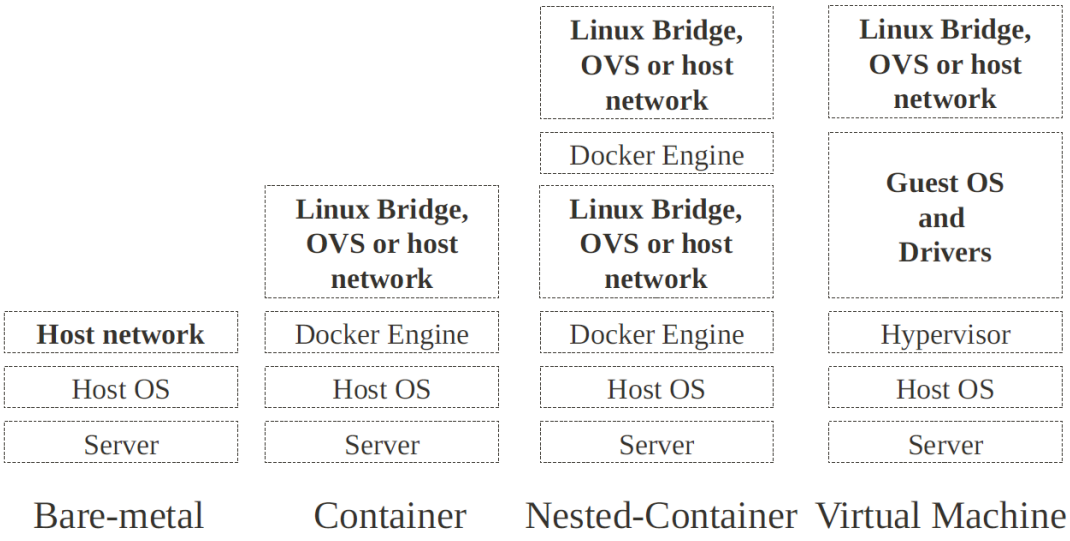

Figure 4.5: The stack of network of bare-metal, container, nested-container and virtual-machine. (Amaral et al., 2015).

The performance of the components in figure 4.5 will be compared in the experiments.

- **Bare metal**: the pure physical server.
- **Regular containers**: the master-slave architecture is made up of one container acting as the master and other containers acting as slaves that will operate the application process.
- **Nested containers**: subordinates' containers (children) are hierarchically created into the main container in the nested container (parent). The application process runs by the children, who are constrained by parental boundaries.
- **Virtual machines**.

CPU Performance Evaluation

The experiment is executed to determine the average execution time of Sysbench[1] while increasing the number of concurrent Sysbench instances from 1 to 64 of different kinds of environments: bare-metal, regular containers, nested-containers, and virtual machines. The multiple containers and virtual machines are executed, each one running a single Sysbench instance. Since there are no resource limitations in the set of containers and virtual machines, the scalability is expected to increase linearly with the number of available CPU cores.

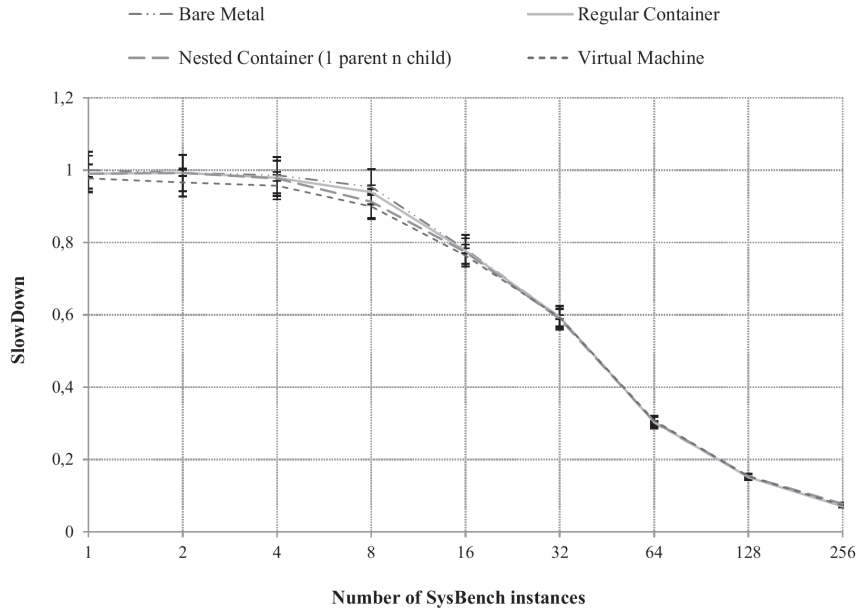

Figure 4.6: The slowdown of Sysbench with increasing number of instances relative to running a single Sysbench instance in bare-metal. (Amaral et al., 2015).

Figure 4.6 shows how the execution time of a single Sysbench decrease as the number of instances increases. As shown, the result is that all environments exhibit similar behavior, confirming that running CPU-intensive executions on containers or virtual machines has no noticeable performance effect compared to bare-metal. Containers are similar to bare-metal in that they run in the operating system natively and are covered by a lightweight layer. However, due to enhanced virtualization support in modern processors, virtual machines can now perform as much.

Conclusion: there is no significant impact on CPU performance.

[1] https://linuxtechlab.com/benchmark-linux-systems-install-sysbench-tool/

Comparing the Creation Time of Virtual Machine and Container

This experiment aims to assess the scalability of various virtual container types by comparing the creation times under different environments.

For standard containers, the elapsed time between container startup and exit is measured. For nested containers, the amount of time it takes to start and exit a nested container will be calculated, which involves loading a locally stored child image as well as starting and exiting a single child container. Lastly, the time for creating a virtual domain, starting the domain, and removing the domain will be measured for a virtual machine.

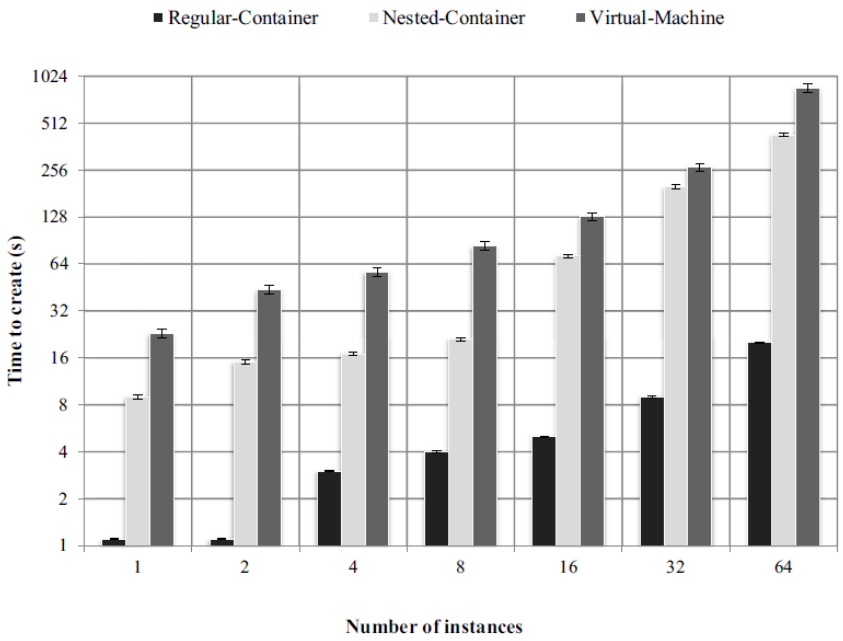

Figure 4.7: The time to create a growing number of instances of (regular and nested) containers and virtual machine. Where the nested-container is a fully initialized parent plus one child. (Amaral et al., 2015).

The result of this experiment, as shown in Figure 4.7, regular containers are the fastest solution, followed by nested containers and virtual machines. Although creating a single nested container is nearly eight times above creating a single regular container, creating nested containers is still more than twice as quick as creating virtual machines.

The extra overhead for nested containers comes from the parent container's Docker initialization, including loading a locally stored image and creating the child container. The loading of an image takes an average of 6.2s. The loading time can be reduced to 1.7s by sharing a read-only preloaded volume from parents to children that contains the child-loaded image.

Conclusion: creating a regular container is the fastest approach, then nested container and lastly, virtual machine. Creating a regular container is sixteen times faster than creating a virtual machine and eight times faster than creating a nested container.

4.3 Container Technology and its Application

Many people mistakenly assume that Docker is synonymous with container. However, container technology is available decades before Docker. Docker is a Linux container-based extension (LXC). (Riehle, 2020)

4.3.1 Container Technologies

Based on the result of the previous section 4.2, Docker container technology was chosen as a container technology for Uni1. Furthermore, Docker has a vast community and is compatible with AWS.

Operating system (Base Image)
The backend components are all built with node JS framework. Docker provides several images of Node, which are the full Node version or a light version (i.e., alpine). As shown in figure 4.8, the size of Node 12 is 918 MB and the alpine is only 88.9 MB. That leads to that the size of the Admin-console container with Node 12 is 1.06 GB, and with alpine is only 219 MB. After comparing both versions, we decided to use the alpine version.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
adminconsole-node12	latest	04e9f4327370	17 seconds ago	1.06GB
adminconsole	latest	894b90853afb	3 minutes ago	219MB
node	12	6f6922527b4a	10 hours ago	918MB
node	12-alpine	a5fe84a24ee3	10 hours ago	88.9MB

Figure 4.8: Size comparison between container built with node:12 and built with node:12-alpine

Environment variable
The usage of environment variables provides a convenient way to set application execution parameters without rebuilding the application. The backend component(s) requires variables such as the DB URL, Cognito pool ID, and AWS (configuration and credential variables). Some variables are optional, like the listening port.

There are several ways to set an environment variable to a containerized application:

- Pass a variable in the run command with -e parameter.
- Define environment variables in Dockerfile with ENV.

4.3.2 Container Application

Depending on the computing environment, two container technologies will be used. Only the backend will be containerized in both environments, while the frontend components and database will not be containerized. The frontend components will be deployed on AWS Amplify. The Amplify will be connected with the master branch of the Gitlab repository. The frontend components will be built and run on AWS Amplify automatically after modifying a component in the master branch.

Development environment - Microservices
It meets the microservices architecture. Each backend service is dockerized in a docker container. The services can run separately from the corresponding dockerfile, or all services can run together through docker-compose. This technology can be used to run backend component(s) locally when developing/running the frontend components locally. However, the database is hosted by AWS, and it is connected to the backend components.

Production environment - Monolithic
All backend services are dockerized in one docker container that will be deployed on AWS Elastic Beanstalk.

4.4 Design of Communication and Integration Mechanisms

In a microservice application, it's critical that all parts of the application interact with one another to provide the best performance and the expected data to users.

4.4.1 Authentication and Authorization

To ensure the services of Uni1 are secured, an authentication and authorization mechanism is used. That mechanism is independent of the protocol or programming language. Uni1 application will be deployed on AWS. Thus AWS Cognito will be used for authentication and authorization. AWS Cognito is a user management, authentication, and access control service, that uses OpenID Connect (OIDC) [2].

[2] https://openid.net/connect/

Authorization Amazon Cognito uses user pools, where the users are stored (with their roles), and the access groups (roles) are defined.

Authentication When a registered user login, a web token will be stored in the browser, and it will be sent by each request to the received service.

Roles concept As required 3.1, Uni1 provides three access roles that are completely independent and are not based on a hierarchy relationship. User type would not be considered for the role, i.e., vendors and customers do not preclude each other, and they can have Administrator position or/and Campaigner role or/and User role.

> **User role:** default role (base role), user can access the Marketplace component.
>
> **Campaigner role:** user can access the Campaigner component.
>
> **Administrator role:** user can access the Admin-Console component.

4.4.2 Communication Technology

The services in Uni1 use a RESTful API request/response communication. The communication technology and protocol used in a development environment vary from those used in a production environment.

Development environment - Microservices
Each backend service (server) is compact inside a Docker container, which has an external port (host port) and an internal port (container port). The external port uses for communication outside the container and the internal port can be used to communicate with the other components inside the container if they exist. The outside port will be mapped to the inside port. As shown in figure 4.13. In our design, each service is containerized in a single container, and thus the communication is available only by the external port.

Production environment - Monolithic
The backend services (servers) are deployed as one docker container application on AWS Elastic Beanstalk and use a single port "to save costs". In order to improve the connection security and ensure that the data are not easily stolen, HTTPS will be used for the external connections, which will encrypt the data through SSL. As shown in figure 4.12. In our design, the backend services are not communicated together, but if needed, the backend components can communicate together using the internal port.

4.4.3 User Interface Integration Technology

The user interface is the space where physical and digital worlds meet. It is a critical component of any interactive application. Designing this interface can be a difficult task because it needs to reflect the nature of the app and its target users. In an ever-growing microservices application, the challenge of building a UI becomes more complex. Different techniques to integrate the UI in a microservices application will be covered in this section based on (Newman, 2015).

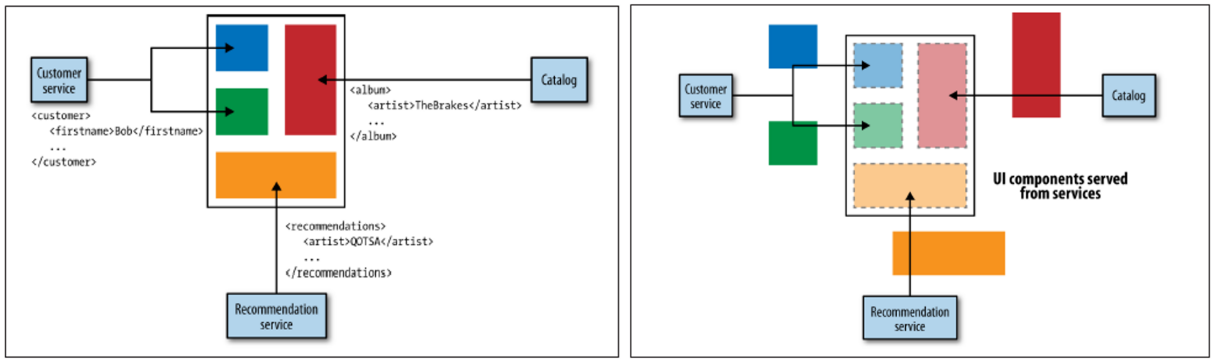

Figure 4.9: UI integration approaches: API and Fragment composition. (Newman, 2015).

API composition

The first way is to have a single user interface. As shown in figure 4.9, the user interface is connected directly with the services. The services provide the data needed by the UI. It is easy to implement but hard to deal with new services or a change in a service.

Fragment composition

Another way to integrate the UI of microservices is by sending the UI components from the services to the UI. As shown in figure 4.9, a single UI exists, and the services provide their UI components to the UI. This approach provides a dynamic UI. Having a new microservice or change in a microservice can be easily integrated with the UI. But it also gives the impression of a heterogeneous application.

Backends for frontends

The third way to integrate the UI of microservices is by having different user interfaces regarding the client type or target device (mobile application or web application). As shown in figure 4.10 each UI has a backend. The backend of

a corresponding UI deals as a middleware between the services and its UI. This approach can be mixed with one of the approaches mentioned above. However, it gives the impression of a homogeneous application. However, it is also complex and more components (backends and frontends) must be developed.

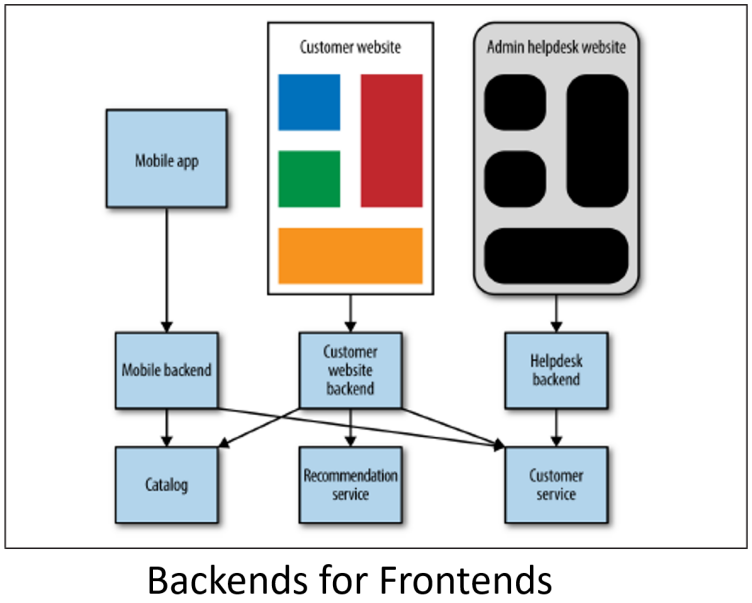

Figure 4.10: UI integration approaches: Backends for frontends. (Newman, 2015).

Uni1 User Interface

In order to win the impression of a homogeneous application, backends for frontends will be used with API Composition, where each service got its own UI and its backend. Each UI of a service uses the "API composition" to get the data from the corresponding service. At the same time, the UI of the Dashboard component is used for integrating the various user interfaces where each component has a separate interface interacting with its respective backend component.

4.5 Design of Admin Console component

The architecture of the administration component is a traditional client/server model, as shown in figure 4.11. The website serving as the client and interacting with the backend server. The two components are completely separate and can be developed and deployed separately. A RESTful API is used for communication between the two components.

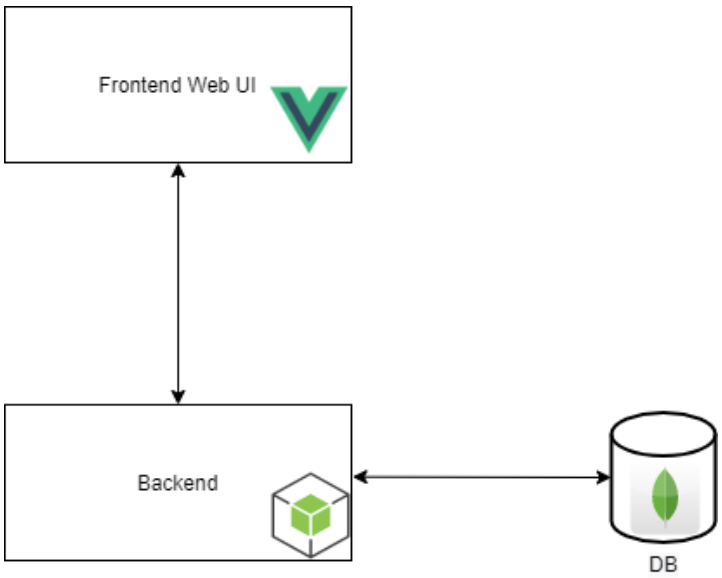

Figure 4.11: Admin-Console Component Architecture

Frontend

The frontend of Admin-Console is a website application, uses Vue.Js with Vuex for managing the application state. Vuex acts as a centralized storage area for all Admin-Console components that ensure that the state can only be changed in predictable ways. Vue.Js is an MIT-licensed open-source Model view view model(MVVM) frontend JavaScript framework for creating single-page applications and user interfaces.

Backend

The backend of Admin-Console is a NodeJS/ExpressJS application. ExpressJS is a routing library that is used to build a RESTful API on top of NodeJS. NodeJS is a Javascript framework for writing server-side applications.

Database

The shared DB will be connected to the backend component of the Admin Console. Important user account data, such as user contacts and user title, will be saved in collections in the Uni1 database cluster. The database is MongoDB, which is hosted on AWS. The connection is made using native MongoDB drivers. The database's URL must be configured as an Environment variable.

4.6 Concept for a set-up in AWS

The concept of the deployment phase will be addressed in this section for both architectures (Monolithic and Microservices). The backend is where the deployment differs in the two architectures. Only the monolithic architecture is deployed in this thesis, while the second will be partially implemented (not deployed on AWS). The frontend components (Admin console, Dashboard, Campaigner and Marketplace) will be deployed on AWS Amplify, and the backend component is deployed on AWS Elastic Beanstalk.

The key difference in deployment between architectures (Monolithic and Microservices) is in the backend components. Based on the deployment concept requirement 3.3, that declares the possibility to have many deployment possibilities in order to decrease costs.

4.6.1 Monolithic

All Backend components (Admin console, Dashboard, Campaigner and Marketplace) are containerized in one docker container, which will be deployed on AWS Elastic Beanstalk [3]. In order to ensure that our backend server is well protected, HTTPS protocol is used for the communication. The figure 4.12 shows how Uni1 backend components will be deployed as a Monolithic application.

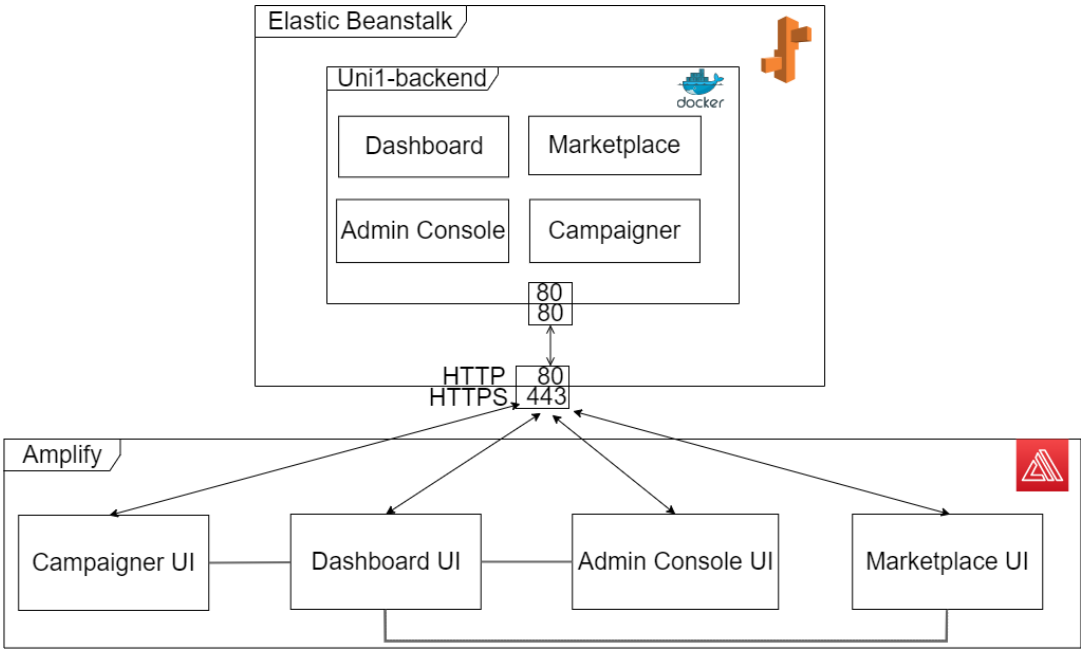

Figure 4.12: Uni1 deployment as a Monolithic application

[3] https://aws.amazon.com/elasticbeanstalk/

4.6.2 Microservices

There are several ways to deploy the backend components as microservices on AWS. In my opinion, the easiest way is by deploying the Backend components as a Multicontainer Docker on one single Elastic Beanstalk instance.

Based on AWS docs ('Using the Multicontainer Docker platform (Amazon Linux AMI) - AWS Elastic Beanstalk', n.d.), the standard Elastic Beanstalk's pre-configured Docker platforms support only a single Docker container per Elastic Beanstalk environment. Although Elastic Beanstalk allows to set up an environment in which the Amazon EC2 instances can run Docker containers concurrently. Elastic Beanstalk uses Amazon Elastic Container Service (Amazon ECS) to organize container deployments to multi-container Docker setups. Amazon ECS provides capabilities for managing a cluster of Docker-enabled instances. Elastic Beanstalk handles Amazon ECS functions such as cluster creation, task definition, and task execution. The environment's instances all run the same set of containers. The figure 4.13 shows how Uni1 components can be deployed as Microservices applications.

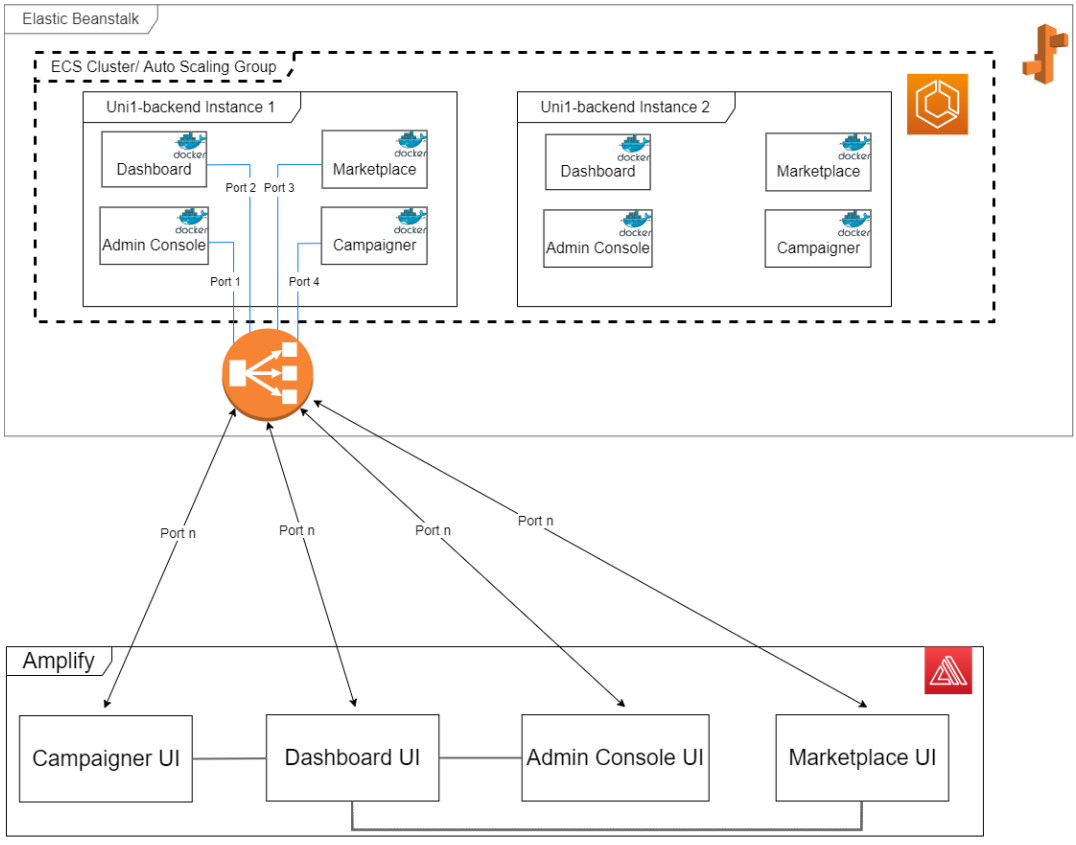

Figure 4.13: Uni1 deployment as a Microservice application

5 Implementation

This chapter presents the implementation of the Uni1 application. The implementation is based on the architecture and design discussed in the previous chapter 4. Because some parts of the system were intertwined, some parts of the uni1 were implemented by (Müller, 2021) and some parts were implemented by me. We also tried to make the most use of each one's knowledge to deliver the best implementation. The first section of this chapter explains the new Administration component. The second section provides the database. Then the implementation of the containers will be discussed. The fifth section provides the way to set the credential keys. However, then the deployment process will be explored. In the end, the integration mechanism will be covered.

5.1 Administration component

This section provides an overview of the technologies used to implement the Administration component and goes over a few implementation issues. The components are reachable under https://console.uni1.de.

5.1.1 Frontend

The frontend is a website application, uses Vue.Js[1] with Vuex[2] for managing the application state. Vue.Js is an MIT-licensed open-source Model–View–ViewModel (MVVM) frontend JavaScript framework for creating single-page applications and user interfaces. To improve code quality and make it more reliable and easier to refactor, Typescript[3] is used. Typescript is an open-source programming language developed and maintained by Microsoft. Admin-Console provides several administration services to mange user accounts as following:

- List all users.

[1] https://vuejs.org/
[2] https://vuex.vuejs.org/
[3] https://www.typescriptlang.org/

- List all contacts (including users from the old DB).
- Export all contacts (as CSV and XLSX).
- Import new contacts.
- Delete a contact.
- Disable user account.
- Enable user account.
- Delete user account.
- Assign a role to a user.
- Remove a role from a user.
- Release contact from a user.

5.1.2 Backend

The backend is a nodeJS[4] server, that uses the open-source software, under the MIT License, Express.js[5] framework. The backend server only reacts if the frontend triggers it by sending a valid request. It works as a middleware (interface) between the Admin-console frontend and users' data in AWS Cognito[6] and DB. For instance, to get all users' data, the backend will first collect users' data from AWS-Cognito and DB; second, it will merge the data and send it to the frontend.

5.1.3 Authentication and Authorization

Administration backend component uses Aws Cognito [7] to ensures that users are who they claim to be (**Authentication**), and to check the access rights to the administration functions (**Authorization**).

The main part of the credential verification (**Authentication**) is done in the frontend (by the Dashboard component) via the Login function. After a successful login, a JWT [8] will be generated by Cognito and stored in the client browser. The JWT will be sent to the backend in the header for each API request. The backend will check its validation (by the middleware 5.6.2) through its included Key ID. If it is valid, the access right will be checked by the included roles, then grants or denies permission will be given to continue the process that the frontend requested (**authorization**).

[4] https://nodejs.org/
[5] https://expressjs.com/
[6] https://aws.amazon.com/cognito/
[7] https://aws.amazon.com/cognito/
[8] https://jwt.io/

5.1.4 Mapping user data from Cognito and Database

To prevent data replication, only a portion of the data is stored in the database, while the remainder is obtained directly from Cognito through AWS SDK [9]. Indeed, via this Cognito, Admin-Console obtains the users' name, email, roles, status, phone number and gender. The rest information is stored in the database, including the users' title, contacts and tags. The data will be merged together. The database connection will be described later in section 5.2.

5.1.5 Communication

The interactive mechanism between the frontend and backend established through REST API. Rest API is an application programming interface that conforms to specific architectural constraints. Because the Admin console provides two different data types, two routes are defined based on the requested data (user and contact). All the currently available endpoints of Uni1 components are listed in the appendix A B C. The database connection will be described later in section 5.2.

The following resources are available:

- /contact/releaseUserContact
- /contact/addEmailsToContacts
- /contact/deleteContact
- /contact/getAllContacts
- /user/listAllUsersInGroup
- /user/removeUserFromGroup
- /user/adminAddUserToGroup
- /user/adminEnableUser
- /user/adminDeleteUser
- /user/adminDisableUser
- /user/getAllUsers

[9] https://aws.amazon.com/tools/

5.2 Database

In order to store data (i.e., user contacts), we used MongoDB Atlas[10] to create a new database and we deployed it on AWS as a NoSQL database, which provides a 512MB storage free of charge and can be scaled up to 5GB[11]. The connection between the backend component(s) with the database established using native drivers. Other options to connect the database with our backend(s) are MongoDB Compass and Mongo shell. In other words, the connection to the DB executed by setting the DB (cluster) URL as an environment variable, more about it in section 5.4.

The previous version of Uni1 included user's contacts. The contacts stored in the Marketplace database have been imported to the new database to avoid losing them.

The figure 5.1 shows how the DB is connected with Uni1. In monolithic architecture, the connection will be created once. While in the microservices, each component connects with the DB.

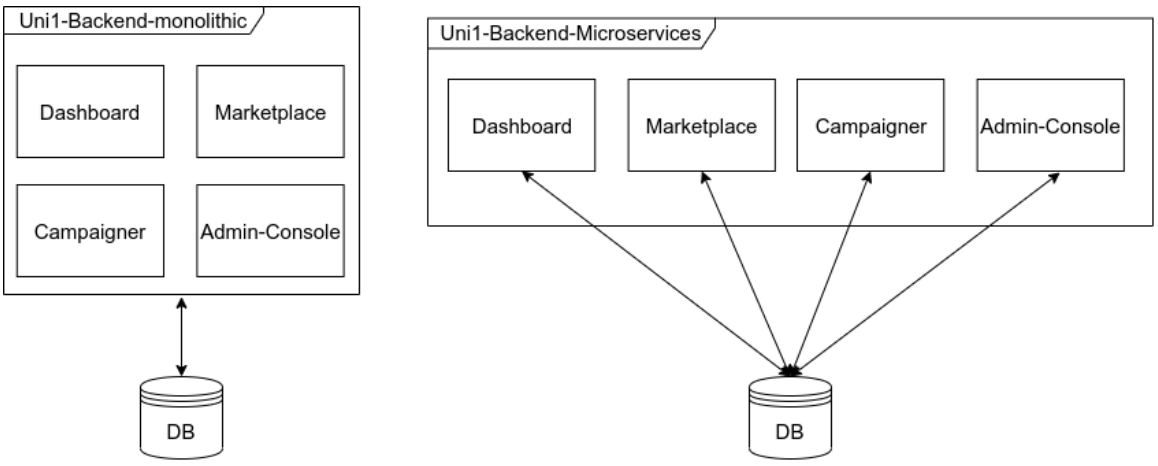

Figure 5.1: Database connection in Monolithic and Microservices

5.3 Container

In this section, we will present the implementation of the containers of Uni1 backend components. Because one of the requirements mentioned that the Uni1 backend could be used as a Monolithic and as a microservice, two implementations were implemented. The implementation of the Monolithic has been deployed on AWS as the architecture 4.12. The microservices architecture has been used in the

[10] https://www.mongodb.com/
[11] https://www.mongodb.com/pricing

development environment. In section 5.5, we will present how the microservices can be deployed on AWS.

As discussed in the previous chapter 4, the use of containers will provide many benefits. The backend containerized in order to win the benefits that containers provide, such as flexibility, lightweight, and easy deployment.

Depending on the previously discussed container technologies in the Architecture and Design chapter 4.3.1, the Docker container will be used.

Docker is a (PaaS)-based OS-level virtualization platform for delivering applications in containers. Containers are self-contained from one another and bundle their own set of applications, libraries, and configuration files. Containers use fewer resources than virtual machines because they all share the services of a single operating system kernel. (Riehle, 2020).

5.3.1 Production environment - Monolithic

All backend services are dockerized in one docker container. The Dockerfile is in the backend directory in the repository. Listing: 5.1 shows the Dockerfile of the backend that used in the deployment on AWS elastic beanstalk. The Marketplace component is included but does not work in this Dockerfile because it will be rewritten in the future after changing some logical functions in Uni1, so we decided to exclude it from the Production environment. In the following subsection 5.3.2 a solution to run Marketplace on Docker will be spotted.

```
FROM node:12-alpine
# Create app directory
WORKDIR /usr/src/app
# Copy both package.json and package-lock.json
COPY package*.json ./
RUN npm ci --silent
# Bundle app source
COPY . .
# Set NODE_ENV to production
ENV NODE_ENV production
EXPOSE 5000
CMD [ "node", "./app.js"]
```

Listing 5.1: Dockerfile of Uni1 backend

5.3.2 Development environment - Microservices

Each backend service has its own Docker container. The dockerfile of each service is located inside the backend directory in the containers branch in the Gitlab repository.

Admin Console & Dashboard & Campaigner
The three services have the same structure, as a result, the docker files of them have the same steps with some differences in the copied files and run command. The following Listing is the Dockerfile of Campaigner backend component.

```
FROM node:12-alpine
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm ci --silent
# Bundle app source && Copy shared libraries (i.e. Cognito middleware)
COPY ./campaigner ./campaigner
COPY ./shared ./shared
EXPOSE 5002
CMD [ "node", "campaigner/app.js"]
```

Listing 5.2: Dockerfile of Campaigner backend server

Marketplace
The Marketplace component has its own structure because it's built with a technology, which uses a linked library between the frontend and backend. Symlink cases problem with Docker, in order to avoid it, the shared library will be copied, then installed and then linked, as shown in the following Dockerfile 5.3.

```
FROM node:12-alpine
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm ci --silent
COPY ./marketplace ./marketplace
COPY ./shared ./shared
COPY ./uni1-api-clients ./uni1-api-clients
RUN cd uni1-api-clients \
    npm install \
    npm link \
    cd .. \
    npm link uni1-api-clients
EXPOSE 5004
CMD [ "node", "marketplace/app.js"]
```

Listing 5.3: Dockerfile of Marketplace backend server

As it is notable in the Dockerfile of Marketplace, the **RUN** command contains multiple commands instead of writing **RUN** for each command. The reason is that each RUN command creates a new layer, which leads to more layers containing information that are no longer exists and thus increase the size of the image. Writing the dockerfile as shown in 5.3 (single RUN command) reduces the image size from 223MB to 221MB.

Docker Compose

Running each Docker container manually becomes time-consuming and difficult to manage. Docker-compose [12] is a powerful tool to start all Uni1 backend services together or separately, which uses a YAML file to configure the services. Then with a single command, all services can run. The docker-compose file is located in **Containers** [13] branch in Gitlab repository.

```
version: "3.9" # docker compose version
services:
  adminconsole:
    image: adminconsole
    build:
      context: . # the path to the directory containing the Dockerfile
      dockerfile: adminconsole.Dockerfile
    ports:
       - "5001:5001" # Host port : Container port
    env_file:
       - .env # set the environment variables from .env file

  campaigner:
    image: campaigner
    build:
      context: .
      dockerfile: campaigner.Dockerfile
    ports:
        - "5002:5002"
    env_file:
      - .env

  dashboard:
    image: dashboard
    build:
      context: .
      dockerfile: dashboard.Dockerfile
    ports:
```

[12] https://docs.docker.com/compose/
[13] https://gitlab.com/profoss/uni1/uni1next/-/tree/containers

```
      - "5003:5003"
  env_file:
    - .env

marketplace:
  image: marketplace
  build:
    context: .
    dockerfile: marketplace.Dockerfile
  ports:
      - "5004:5004"
  env_file:
    - .env
```

Listing 5.4: Docker Compose of all backend components for a development environment

Another way to pass the environment variables will be discussed in the next section 5.4.

5.4 Environment variable

It is never a good idea to set a password and credential keys directly in a configuration file in the codebase. Instead, reference the credential keys to an environment variable. The following are the primary advantages of using environmental variables:

- Easy configuration: worry only once when the variable set for the first time. We only have to update the environment variable when we need to change a key.
- Better security.
- Easy to change the value of the variable without rebuilding the app.

The following Environment variables must be defined to run the Uni1 backend:

- DATABASE_URL: the URL of the Database.
- USER_POOL_ID: the user pool Id of Cognito, it can be set automatically after setting amplify.
- AWS_REGION: the region of the AWS provider, it can be set automatically after setting amplify.
- AWS_ACCESS_KEY_ID: AWS IAM Access Key ID.

– AWS_SECRET_ACCESS_KEY: AWS IAM Secret Access Key.

The following Environment variables can be defined in the backend; otherwise, they have a default value:

– PORT: the port number that the service(s) will be using.

– NODE_ENV: set the environment of the backend: testing or production. By default, it's development.

5.4.1 Development environment - Microservices

In the development environment, the environment variables can be set directly by the OS or in a **.env** file. A **.env** file is essentially a plain text document. It should be located at the root of the project. It has the structure of a key-value pair to specify the variables and their corresponding values.

Docker Compose

The .env file can be used inside a docker-compose as it shown in the docker-compose file 5.4. Since Docker compose 3.4, the environment variables can be defined at the top of the docker-compose file, then can be shared with the services as it is shown in the docker-compose file 5.5. It is also possible to define a non-common variable direct for a specific service.

```
version: "3.9"

x-common-variables: &common-variables # define common variables
  DATABASE_URL: mongodb+srv://XXXXXXX
  USER_POOL_ID: XXXXXXX
  AWS_REGION: XXXXXXX
  AWS_ACCESS_KEY_ID: XXXXXXX
  AWS_SECRET_ACCESS_KEY: XXXXXXX

services:
  adminconsole:
    image: adminconsole
    build:
      context: .
      dockerfile: adminconsole.Dockerfile
    ports:
       - "5001:5001"
    environment: *common-variables # pass the common variables to this
       service
```

Listing 5.5: Docker Compose: setting common environment

5.4.2 Production environment - Monolithic

Because the deployment on AWS executed in CI, it is essential to pass the passwords and the credential keys to the Pipeline. Gitlab offers CI/CD variables. Through Gitlab CI/CD variables, we can define environment variables that can be used inside the Pipeline and avoid hard-coding variables in the .gitlab-ci.yml file.

5.5 Deployment

The implementation of the deployment phase is addressed in this section for both architectures (Monolithic and Microservices). The backend is where the deployment differs in the two architectures. Only the monolithic architecture is deployed on AWS[14]. While the second is not fully implemented. In this section, a guide on how to deploy Uni1 as Microservices on AWS will be addressed. The frontend components will be deployed on AWS Amplify, and the Backend Component is deployed on AWS Elastic Beanstalk. The deployment of the Uni1 Monolithic application and frontend components is done by (Müller, 2021), while the network, database setup, and the set of environment variables are executed in this thesis.

5.5.1 Frontend

Each Frontend component of Uni1 application (Admin console, Dashboard, Campaigner and Marketplace) is deployed on AWS using Amplify. AWS Amplify[15] is a collection of tools and services that help to build scalable full-stack applications. Amplify works with common web frameworks like JS, React, Angular, Vue.

Deliver
Amplify provides a fully managed Continuous Deployment service, which allows Amplify to connect with the Gitlab repository. By updating the master branch on Gitlab, the CI/CD from Amplify will run.

Protocol
To improve the security for frontend components, only HTTPS is permissible, which means that all frontend components requested and responded data is conveyed via HTTPS protocol.

[14] https://aws.amazon.com/
[15] https://aws.amazon.com/amplify/

5.5.2 Backend

The key difference in deployment between architectures (Monolithic and Microservices) is in the backend components. Based on the deployment concept requirement 3.3, which declares the possibility to have many deployment possibilities in order to decrease costs. The backend components will be deployed as a single docker container; in other words, the components will be merged together in one application. This article will discuss the implementation of backend deployment as a monolithic application, as well as a suggestion of a possible technique to deploy backend components as microservices.

Monolithic

All Backend components (Admin console, Dashboard, Campaigner and Marketplace) are containerized in one docker container, which will be deployed on AWS Elastic Beanstalk [16]. AWS CloudFormation [17] is used to generate an Elastic Beanstalk instant. AWS CloudFormation is a service that assists in modelling and setting up the AWS resources and environment variables, i.e., S3 Bucket, Cognito User Pool ID, Database URL. We define all the resources in our backend template (cfnBackendTemplate.json) in git repository inside aws folder.

- **Deliver**
  The delivery of the backend executed by the GitLab CI service. The CI will be triggered by modifying the backend code in the master branch. The CI will run the following steps:
  - uploadBackendS3: upload the Backend application to S3.
  - cfnBackend: create Elasticbeanstalk using CloudFormation. CloudFormation will take care of storing and configuring the resources.
  - createEbVersion: create new app version.
  - deployEb: deploy on Elastic beanstalk as one docker container.

  After the CI finish, the backend components will be deployed on Elastic Beanstalk, run as one docker container, and stored in the Amazon S3 bucket.
- **Protocol**
  In order to ensure that our backend is well protected, HTTPS protocol is used for the communication. We requested a trusted certificate for our domain ***.uni1.de** with AWS Certificate Manager [18]. Then we set the certificate in the CloudFormation template.

[16] https://aws.amazon.com/elasticbeanstalk/
[17] https://aws.amazon.com/cloudformation/
[18] https://aws.amazon.com/certificate-manager/

The protocol for routing traffic to the backend instances is HTTP. On the other hand, the protocol of the listener is HTTPS. As shown in Figure 4.12 the communication with the backend components from outside is only available via HTTPS, while the Elastic beanstalk will convert the HTTPS request to HTTP.

Microservices

As it has been discussed in the Architecture and Design chapter 4.6.2. There are several ways to deploy the backend components as microservices on AWS. One way to deploy Microservices on AWS is by deploying the Backend components as Multicontainer Docker on a single Elastic Beanstalk instance. Amazon Elastic Container Service[19] (ECS) should be used to run and manage Docker applications across the logical group of EC2 instances.

Create a cluster Before the ECS task definition, a cluster in ECS must be created. It can be created directly from the ECS website.

Create ECS task definition ECS task definition can be set directly from the ECS website or a Docker-compose file.

As it has been shown before, a docker-compose is used in the development environment to run all the backend components (each component as a docker container). Docker Compose can be used to create the ECS task definition file. ECS task definition specifies each container's properties(i.e., CPU, memory requirement, network, and port settings).

Using container-transform[20] tool, the Docker-Compose file can be converted to a Dockerrun.aws.json definition file.

Deploy to Elastic beanstalk

Before deploying the containers, the following pre-request are needed:

- **AWS Docker configuration:** the file that describes how to deploy the Docker containers as one Elastic Beanstalk application **Dockerrun.aws.json**.
- **Load Balancing:** configure multiple Elastic Load Balancing listeners on a multi-container Docker to harmonize inbound traffic with HTTPS protocol.
- **Docker images**

The backend folder should contains the following files:

[19] https://aws.amazon.com/ecs/
[20] https://github.com/micahhausler/container-transform

```
backend
|
|-Dockerrun.aws.json
|
|-adminconsole
|  |-app.js
|
|-marketplace
|  |-app.js
|
|-campaigner
|  |-app.js
|
|-dashboard
|  |-app.js
```

The platform branch in the .gitlab-ci.yml file must be configured. Inside the .gitlab-ci.yml file, in the deployEB step, the Platform as **Docker** and Platform branch as **multi-container Docker running on 64bit Amazon Linux** must be set. A step by step tutorial can found on the AWS website [21].

5.6 Integration

The dashboard is in charge of the integration of the frontend components. In this section, the term "backend integration" refers to the capacity of the backend components to run as a single unit and how to run a single component of the backends. However, the app.js in the root backend folder used to integrate backend components, and it can be configured from package.json. This section also addressed the communication between the components as well as the implementation of authentication and authorization.

5.6.1 Communication

The communication of the heterogeneous components implemented through the standard Restful API. Where the frontend components communicate with the backend component(s) through HTTPS over a Rest API.

The communication between the backend component(s) executed through the database.

[21] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_docker_v2config.html

5.6.2 Authentication and Authorization

Nodejs provides a function called 'Middleware' which can access the request object, the response object, and the following function. A middleware (called Cognito-middleware) has been implemented and attached to each route (path) in the backend servers, which has access to each HTTP request and response and is used to verify authentication and authorization for each incoming request.

As shown in figure 5.2 the middleware can either terminate the HTTP request or pass it to the following middleware or function.

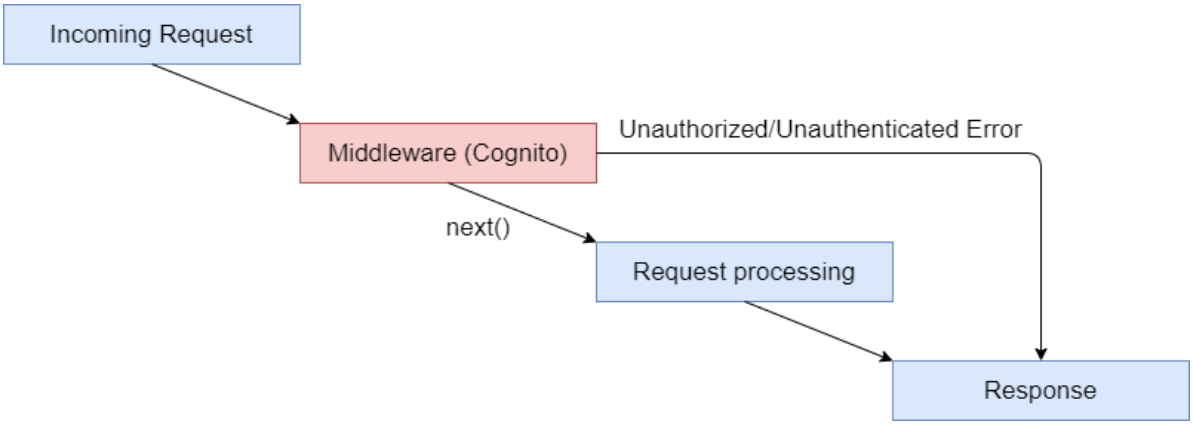

Figure 5.2: Cognito middleware

Use Case

The diagram in figure 5.3 visualizes the sequence of a message flow in the Uni1 system between the Admin-console frontend and Uni1 backend (regardless of the backend architecture). This sequence diagram captures a user's behaviour with administrator access rights and successfully logs in and accesses the admin console UI. When a user logs into the user interface dashboard, a JWT will be stored in the browser's cookie. The JWT contains the Cognito user's initial information, as well as their access groups, as shown in 5.6. Each request will include a token that will be sent to the backend. The backend's middleware will verify the token's validity using the Key ID it has included. If it is legitimate, the included positions will verify the access rights; if grant permission granted and the request will be processed; if refused, permission refused, the request will terminate and an error will be sent back.

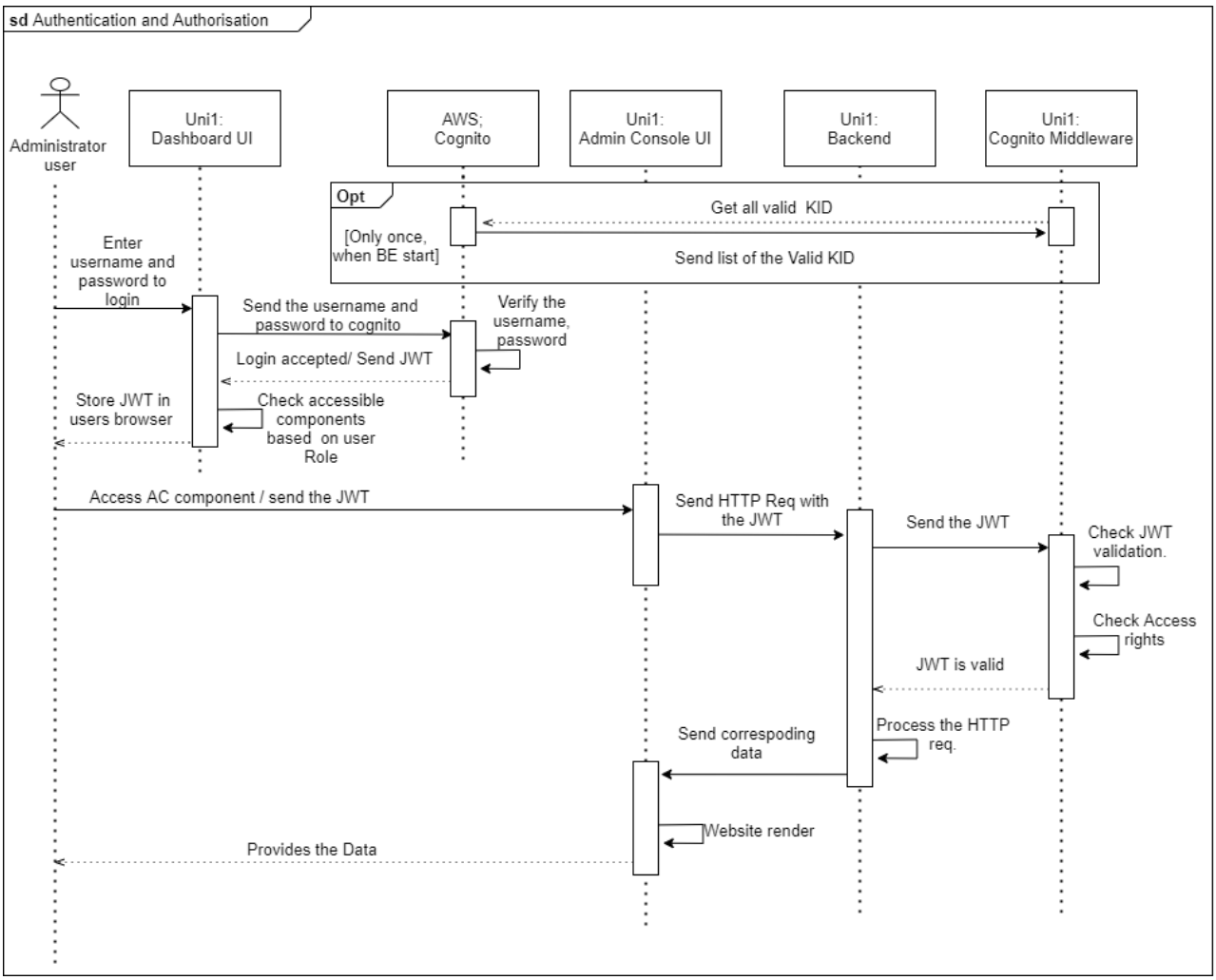

Figure 5.3: SD: the Authentication and authorization in Uni1

```
"header":{
  "kid": "XXXXXXX/XXXXXXXXXXXXXXXXXXXXXXXXXXXX/XXXXXX",
  "alg": "RS256"
},
"payload:"{
  "sub": "53eb85f3-24f2-4f39-8bcd-XXXXXXXXX",
  "cognito:groups": [
    "Campaigner",
    "Administrator",
    "User"
  ],
  "event_id": "c2260145-05f7-4cc3-8e92-b8c475a47476",
  "token_use": "access",
  "scope": "aws.cognito.signin.user.admin",
```

```
"auth_time": 1618775114,
"iss": "https://cognito-idp.eu-central-1.amazonaws.com/...",
"exp": 1618933792,
"iat": 1618930192,
"jti": "7c5ea677-14fd-4d7a-960d-f29f194b4ffa",
"client_id": "4io7r9u2o1etb2172qhilimjgf",
"username": "Max" }
```

Listing 5.6: Cognito JWT

5.6.3 Backend Components

Each of the Backend components can run as a single unit or as a part of one component. The backend components can run through Docker and docker-compose (as explained in 5.3), or directly from npm, as package.json illustrated in Listing 5.7. Multiple "run" and "test" instructions are executable.

```
{
    "name": "uni1next",
    "version": "1.0.0",
    "description": "REST API",
    "main": "app.js",
    "scripts": {
        "start": "node app.js",
        "start:dev": "nodemon app.js",
        "startadmin": "node adminconsole/app.js",
        "startadmin:dev": "nodemon adminconsole/app.js",
        "startcamp": "node campaigner/app.js",
        "startcamp:dev": "nodemon campaigner/app.js",
        "startdash": "node dashboard/app.js",
        "startdash:dev": "nodemon dashboard/app.js",
        "startmp": "node marketplace/app.js",
        "startmp:dev": "nodemon marketplace/app.js",
        "test": "cross-env NODE_ENV=testing mocha",
        "testadmin": "cross-env NODE_ENV=testing mocha
            ↪ \"adminconsole/test/\"",
        "testcamp": "cross-env NODE_ENV=testing mocha
            ↪ \"campaigner/test/\"",
        "testdash": "cross-env NODE_ENV=testing mocha
            ↪ \"dashboard/test/\"",
        "testshared": "cross-env NODE_ENV=testing mocha \"shared/test/\""
    },
    "dependencies": {...},
    "devDependencies": {...} }
```

Listing 5.7: package.json

Through the multiple commands, we were able to set multiple environments. The first environment is development, where compiling with errors was allowed, and without the need to manually (re)start the node server, the node server will be restarted automatically. The second environment is production, where errors are forbidden. That improved the production environment and significantly decreased the developer's time and effort necessary to execute the program.

5.6.4 Dev Tools

A simple tool has been implemented to create "Tags". It can be accessed on the Dashboard UI. It can only be seen and used by users with administration rights. The dev tools should be removed once the Marketplace is rebuilt.

5.6.5 UI Integration

As discussed in the design and architecture chapter, there are multiple ways to integrate the UI of microservices. The implementation of the UI integration of Uni1 is similar to "Backends for frontends" with "API composition", where each service got its own UI and its backend. Each UI of a service uses the "API composition" to get the data from the corresponding service. At the same time, the UI of the Dashboard component is used for integrating the various user interfaces. Where each component has a separate interface interacting with its respective backend component. As illustrated in figure 5.4, each component appears on the dashboard as a button; the user is brought to the corresponding URL component "which is a subdomain of uni1.de" by clicking the component. The accessible component will be displayed, depending on the logged-in user access right. The URL configuration of each frontend component can be found in a JSON configuration sheet under the config folder on the Gitlab repository.

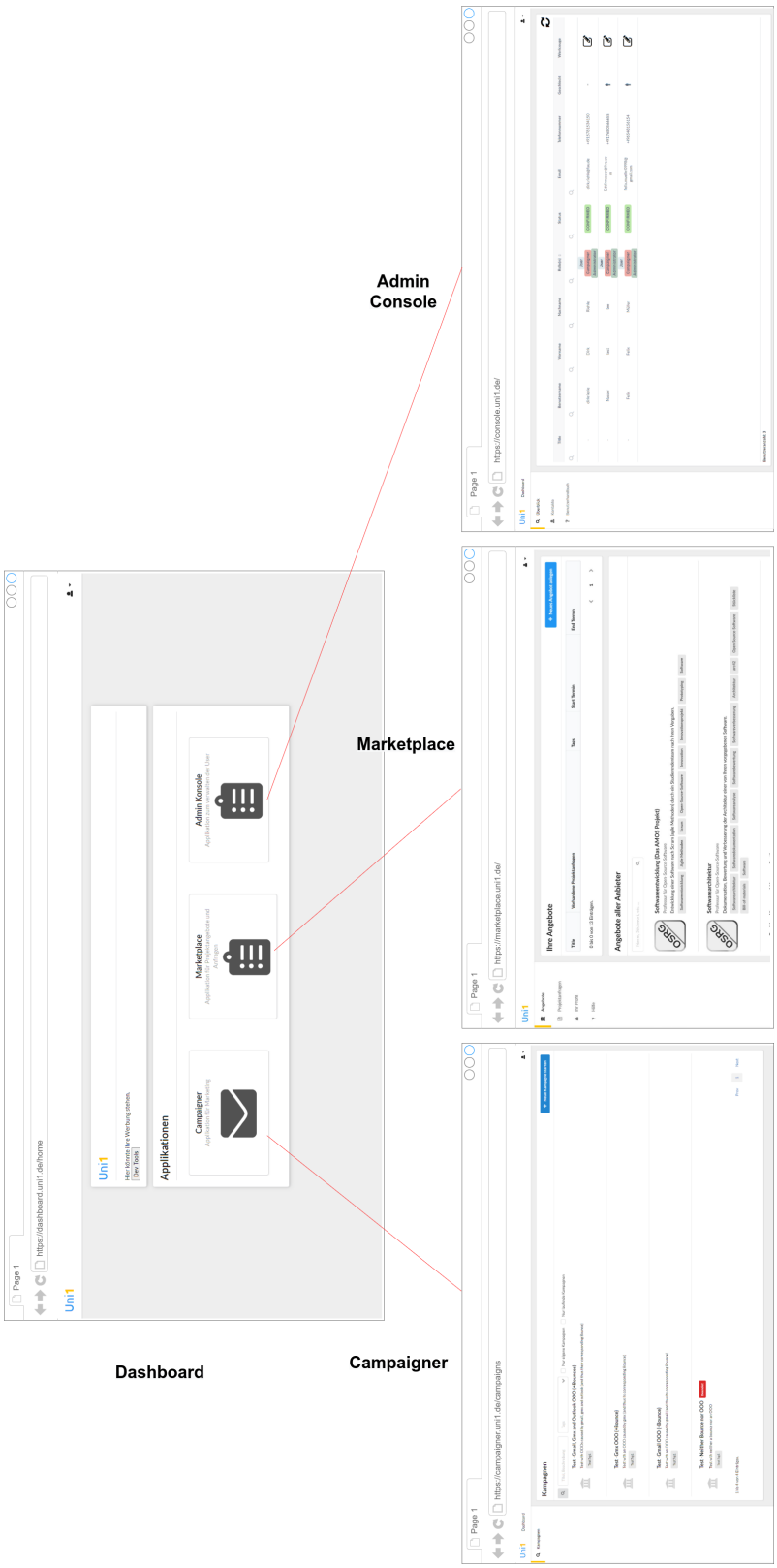

Figure 5.4: A screenshot of the Dashboard UI for a user with fully access right

6 Evaluation

This chapter evaluates the implementation of the thesis requirements in chapter 3 and discusses whether they are met.

6.1 Authentication and Authorization Concept

The first requirement was to set an authentication and authorization mechanism. Authentication and authorization both are critical components of system security. They validate the user's identity and grant access to the Uni1 application. As shown in the authentication and authorization section 5.6.2, a Cognito middleware was attached to each backend route that checks validation of the authentication and authorization for each incoming request. On the other side, the major part of the credential verifies (Authentication) executed in the frontend by the Dashboard UI component via the Login function. The dashboard is connected with AWS Cognito, which verifies the inserted username and password from the user. If it's valid (Authentication), the corresponding components will appear on the dashboard as shown in 5.4. Furthermore, a user must have at least one role. Summary the first requirement is met.

6.2 Administration component

The second requirement was to create a new component to manage and control user accounts and contacts. The new component has been developed and integrated with the other Uni1 components. As specified in the implementation section 5.1 the component has been built with the architecture and technology, which have been discussed in 4.5. The new component is called Admin Console (Admin Konsole). Admin Console has the ability to list all the accounts and contacts. Furthermore, it has the ability to release contact from an account, remove a contact, and import email(s) to contacts. Admin Console, in addition, provides the capability to assign roles to users or to dismiss roles from a user and to manage account status (active and deactivate). The user interface of Admin

Console is in German and is user-friendly. Summary the second requirement is met.

6.3 Deployment Concept

The third requirement was to have the ability to deploy the application as monolithic and as a microservices. It also mentions that it must be a cross-platform application to execute it in various environments.

Following the result of the comparison between the Docker container and a virtual machine in chapter 4. Docker containers were picked as specified in the container implementation 5.3 for Uni1 backend components. The run of the backend component (as a monolithic and Microservice app) was successfully tested on Windows 10, Linux (Debian and Kali), and AWS Elastic beanstalk.

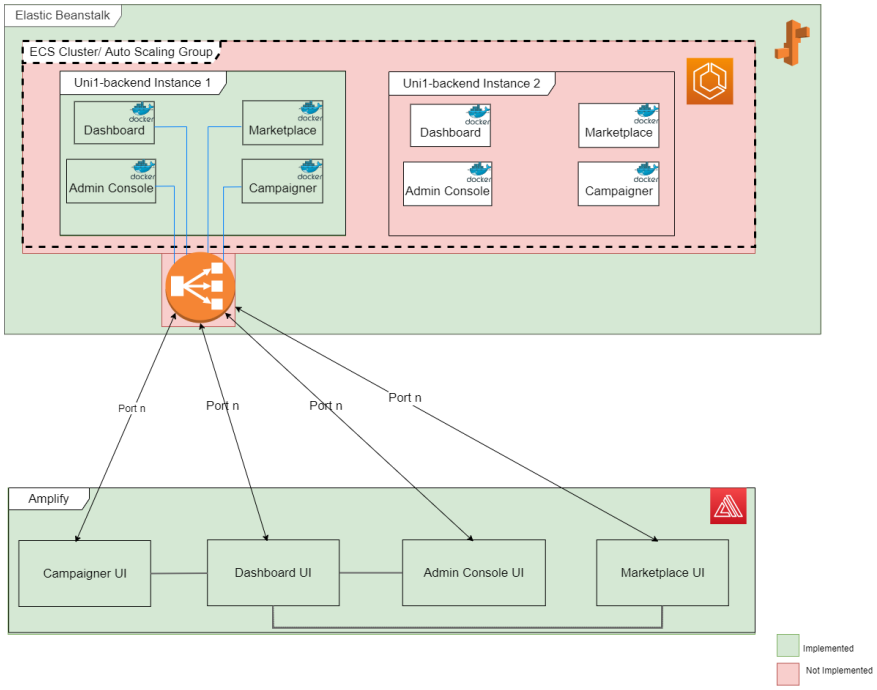

Figure 6.1: Uni1 deployment as Microservices

In the architecture and design 4.6, the concept of the deployment of Uni1 as monolithic and as Microservice were discussed. In the implementation 5.5, the implementation of monolithic backend and an example of the implementation of microservices were specified. The microservices architecture is not deployed yet, as shown in figure 6.1 some elements "coloured in red" are not implemented, but the approach to implementing them is mentioned. Summary the third requirement is met, but the microservices deployment still needs to set up.

6.4 Integration, Communication and Extensibility Concept

The fourth requirement was to have the feeling that the user is using a monolithic application despite if its microservices. It also specifies that components must have the capacity to communicate together. Moreover, it is easy to add a new component.

6.4.1 Integration

As illustrated in figure 5.4, all the frontend components are integrated together in the Dashboard UI. The user cannot notice if the application is monolithic or microservices. Also, the backend components can each run as a single unit(microservice), or all the components can run as a single unit (monolithic).

6.4.2 Communication

The communication between the frontend and backend components is executed with RESTful API over HTTPS. In contrast, there is no real connection between the backend components. The backend components communicate together through the database.

Based on the thesis (Schwarz, 2019), which listed some common pitfalls of a system based on microservices, that leads to a hidden monolith.

Sharing the same collection data on the same database between different microservices is dangerous because no explicit data ownership is stated. Furthermore, even if responsibilities are established, if one service modifies the format of stored data, other services may crash if they access the modified data. This can be avoided by using distinct databases.

6.4.3 Extensibility

It is possible to add a new component to the Uni1 by adding the new component UI (if it has a frontend) to the Dashboard UI and specify the access rights from Cognito by Amplify. At the same time, the backend component must use the backend's shared library and connect to the shared database. According to (Schwarz, 2019), having a shared library leads to a hidden monolith.

6.5 Summary

The first requirement, the Administration component, is met. The second requirement, the authentication and authorization requirement, is met. The third

and fourth requirements are not entirely met. Since the implementation contains a hidden monolith, the shared database and shared library must be avoided to transition to a complete microservice design. The reason for it is that while the microservices concept was being established, different programmers were developing other components. Due to time constraints, we chose the simplest option to share the database to complete the implementation and ensure that our application works consistently. However, the shared database must be avoided in the future.

7 Conclusions

7.1 Summary

In this thesis, we addressed the problems of building an extensible application in the age of the cloud. The purpose of this thesis was to split the existing Uni1 application into containerized components, construct a new administration component, integrate all components, prepare for a full microservices switch, and reduce deployment costs.

The new administration component was built using the traditional client-server architecture. For the frontend, the Vue.js framework with typescript used. For the backend, Nodejs used. For the connection, a RESTful API used.

All the frontend components were deployed on AWS Amplify, where the backend components were deployed on AWS Elastic Beanstalk and the database is hosted on AWS.

We compared two virtualization solutions to improve Uni1's scalability and deployment procedure (Virtual machine and Docker Container). After demonstrating that the docker container is better suited to our application, we chose it as virtualization technology, allowing us to avoid vendor lock-in and environment dependency.

To reduce the hosting costs, two deployment methods were created; depending on the expectational usage need, the backend components of the Uni1 can be deployed as a monolithic application or as microservices applications.

7.2 Future Work

From a technological standpoint, a full transition to complete microservices must involve some points. Multi databases should be used instead of a shared database. Remove shared libraries by copying them into each backend component. A method for deploying the backend components as microservices is mentioned in the implementation. Furthermore, several deployment functions are not yet

set up, as shown in figure 6.1 some components "colored in red" has not been implemented yet. As a result, when Uni1 is deployed as microservices, some work must be done to set up the missing elements.

Appendices

A REST API Overview of Admin-Console Service

A.1 Contact Route

Methode	Endpoint	Description
GET	/getAllContacts	Get all contacts
POST	/releaseUserContact	Relase an email from users account
POST	/addEmailsToContacts	Add emails to the contacts
DELETE	/deleteContact	Delete a contact by email

A.2 User Route

Methode	Endpoint	Description
GET	/getAllUsers	Get all users
POST	/listAllUsersInGroup	Get all users in a group
POST	/adminAddUserToGroup	Give a user an access right based on the group
POST	/adminEnableUser	Enable a user account
POST	/adminDisableUser	Disable a user account
DELETE	/removeUserFromGroup	Remove user from a group
DELETE	/adminDeleteUser	Delete user account

B REST API Overview of Dashboard Service

B.1 Dev Route

Methode	Endpoint	Description
GET	/tag	Get all tags
POST	/createTag	Create a tag with a tag name

B.2 Users Route

Methode	Endpoint	Description
GET	/me	Get my account data
POST	/signup	Create new user
POST	/me	Edit user data

B.3 Contact Route

Methode	Endpoint	Description
POST	/	Create contact
POST	/{contactId}	Update contact
POST	/{contactId}/subscribe	Subscribe an email
POST	/{contactId}/unsubscribe	Unsubscribe an email
DELETE	/{contactId}	Delete contact

B.4 Tags Route

Methode	Endpoint	Description
GET	/{tagId}	Get tag by id
GET	/	Get all tages
POST	/	Create new tag

C REST API Overview of Campaigner Service

C.1 Campaigns Route

Methode	Endpoint	Description
GET	/	Get all campaigns
GET	/{campaignId}	Get campaign by Id
POST	/{campaignId}/finish	Set finish to a campaign by Id
POST	/	Create new campaign
POST	/preview	Get an HTML preview of the campaign

C.2 Replies Route

Methode	Endpoint	Description
GET	/{replyId}	Get reply by Id
GET	/	get all replies
GET	/{replyId}/emails	Get if reply handled
POST	/{replyId}/emails	Set reply handled

C.3 Bounces Route

Methode	Endpoint	Description
GET	/	Get all bounces
GET	/{bounceId}	Get bounce by Id

C.4 Tags Route

Methode	Endpoint	Description
GET	/{tagId}	Get Tag by Id
GET	/	Get all tages
POST	/	Create new tag

C.5 Contacts Route

Methode	Endpoint	Description
GET	/email/{email}	Get contact by email
POST	/{contactId}/resetBounce	Reset bouncing flag
POST	/{contactId}/unsubscribe	Unsubscribe a contact
DELETE	/{contactId}	Delete contact

References

Advantages and disadvantages of docker - learn docker [DataFlair]. (2018, November 22). Retrieved June 27, 2021, from https://data-flair.training/blogs/advantages-and-disadvantages-of-docker/

Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M. & Steinder, M. (2015). Performance evaluation of microservices architectures using containers. *2015 ieee 14th international symposium on network computing and applications* (pp. 27–34). https://doi.org/10.1109/NCA.2015.49

Best practices for writing dockerfiles [Docker documentation]. (2021, May 3). Retrieved May 6, 2021, from https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Monus, A. (2018). *What are microservices? the pros, cons, and how they work* [Raygun blog]. Retrieved May 22, 2021, from https://raygun.com/blog/what-are-microservices/

Müller, F. (2021). Uni1 monolith to components.

Newman, S. (2015, February 2). Building microservices: Designing fine-grained systems. "O'Reilly Media, Inc."

Riehle, D. (2016). *Das uni1 projektkonzept (2016)* (tech. rep. CS-2016-04). Technische Fakultät.

Riehle, D. (2020). *Advanced design and programming*. Retrieved May 24, 2021, from https://github.com/dirkriehle/adap-course/blob/master/Generated/Lecture%20slides/ADAP%20B01%20-%20Containerization.pdf

Schwarz, G.-D. (2019). Migrating the jvalue ods to microservices, 18–19.

Silberschatz, A. & Galvin, P. B. (2013). Operatings system concepts, 23–24.

Thönes, J. (2015). Microservices. *IEEE Software*, *32*(1), 116–116. https://doi.org/10.1109/MS.2015.11

Using the multicontainer docker platform (amazon linux AMI) - AWS elastic beanstalk. (n.d.). Retrieved May 15, 2021, from https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_docker_ecs.html

What is a container? | app containerization | docker. (2021). Retrieved May 12, 2021, from https://www.docker.com/resources/what-container