Technologisches Gewerbemuseum
Höhere Technische Lehranstalt für
Informationstechnologie
Schwerpunkt Systemtechnik

Diploma Thesis

VIPER

Payment in Restricted Environments

Web Service Engineering
Ebenstein Michael 5BHIT

Augmented Reality Development
Fuchs Peter 5BHIT

Web Architecture
Liebmann Oliver 5BHIT

Virtual Reality Development
Matouschek Marco 5BHIT

Security Engineering
Strasser Alexander 5BHIT

Teacher: List Erhard, Brein Christoph
Realized in 2018/19

Delivery note:
8. April 2019

Accepted by:

Statutory declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Location, Date	Ebenstein Michael
Location, Date	Fuchs Peter
Location, Date	Liebmann Oliver
Location, Date	Matouschek Marco
Location, Date	Strasser Alexander

Abstract

Virtual- (VR), augmented- (AR) and mixed-reality (MR) are new and fast-growing technologies. These environments allow the user to create a completely new digital world (VR) and extend the real world with digital objects (AR). Because of the specialized hardware, headsets, and custom controllers it is not possible to use existing payment technologies inside these environments without creating inconvenience or breaking the immersion. To further the development of Cross Reality (XR)-applications and make them usable for commercial purposes, a payment solution for these applications is desired. As there are no technically mature solutions on the market, VIPER aims to develop a system that gives developers the possibility of executing payments in said environments. In order to preserve the integrity of the XR environments, the user must not be forced to remove the headset or exit the applications in any other way. All necessary information shall be entered from within the XR environments with appropriate input methods.

Kurzfassung

Virtual- (VR), Augmented- (AR) und Mixed-Reality (MR) sind neue, schnell wachsende Technologien. Diese ermöglichen es dem Nutzer, neue, digitale Welten zu erstellen (VR) und die bereits existierende, reale Welt mit virtuellen Gegenständen zu erweitern (AR; engl., to augment = vergrößern). Aufgrund spezialisierter Hardware, Headsets und angepasster Controller ist es nicht möglich, existierende Zahlungsmethoden innerhalb dieser Umgebungen zu verwenden, ohne Unbequemlichkeiten für den Nutzer zu kreieren oder das Erleben der virtuellen Welt zu unterbrechen. Um die Entwicklung von XR-Applikationen voranzutreiben und einen kommerziellen Betrieb zu ermöglichen, wird eine Zahlungsmöglichkeit für diese Applikationen benötigt. Da es auf dem Markt momentan noch keine technisch ausgereiften Lösungen gibt, ist es VIPER's Ziel, Entwicklern Zahlungen in XR-Umgebungen zu ermöglichen. Um eine fortlaufende Nutzererfahrung der Applikation gewährleisten zu können, darf der Nutzer nicht dazu genötigt werden, das Headset abzunehmen oder die Applikation auf irgendeine andere Art zu verlassen. Jegliche benötigte Information soll über angemessene Eingabemethoden in der XR-Umgebung selbst eingegeben werden können.

Acknowledgment

At this place we want to thank our teachers at the Technologisches Gewerbemuseum (TGM) and especially our advisor Erhard List and co-advisor Christoph Brein, who provided us with both technical and moral support throughout the project. We also want to thank our partner and project initiator Christian Schleining, who helped us throughout the project, offered valuable feedback and brought up new ideas. Besides the people directly involved with the project, we also want to thank all the people that have reviewed our work and provided feedback. We want to thank our main sponsor A1 Digital, who gave us access to the Exoscale cloud service, which allowed us to test and develop this project within a real-world environment.

Contents

Chapter 1

Introduction

1.1 Motivation

This project was initiated by Christian Schleining, who through his working experience with Virtual Reality (VR) has spotted the potential for Cross Reality (XR) E-commerce. The quickly rising popularity of XR and technology innovations in its field have made it a potential new marketplace for online commerce. At the same time e-commerce stores likes Amazon have experienced a huge gain in user numbers. Digital products on mobile devices and in computer games have gained similar popularity. Both e-commerce and digital products are only available on Personal Computers (PCs) or mobile devices. Thus an integration of the online commerce market for XR would enable a new way of online retail and close a market gap.

1.2 Aim of This Work

The goal of this project is to create a service which enables payment in restricted environments. This service consists of a website, which allows the user to insert payment-relevant data, like bank accounts and verification codes, a back end which handles the payment options and two applications which show the functionality of the service in both Augmented Reality (AR) and VR. Since the main problem of payment in these environments is the impossibility to input much data fast, the claimed identity cannot be verified easily. VIPER provides a solution to this problem as it uses only pin-codes and patterns to verify the user's identity. Thus the project needs to provide a service which can be implemented to XR-applications very easily.

1.3 Methodology and Approach

This thesis is split into several parts, each one associated with a certain team member. Each of these parts consists of the following three steps to complete the thesis.

Research of Existing Technologies and Requirements Payment, web development and VR and AR are very advanced technological fields. There are some technologies that simplify developing for these environments. Thus the first step is gathering information about these

technologies and research the requirements needed to develop the service and its corresponding parts. This research is very important for the later stages of the project thus they are reviewed within the first sprint.

Design Process In the next step, the design choices and possibilities are elaborated. The resulting design concepts includes diagrams and mockups . Also, the corresponding technologies for development are selected in this step. While these concepts are decisive for the implementation they still have to be reviewed throughout the implementation process and are adjusted when needed to ensure the correctness of the design decisions.

Implementation and Evaluation The last step is the implementation of the design which also includes code snippets. Since the project team consists of five person there are several dependencies between team members, which are handled via the project management method *Scrumban*. Throughout the implementation phase and especially after this step, the evaluation is done by the team, the advisers and the project's client.

1.4 Terminology

Throughout this work certain terms are used to describe the VIPER system with meanings deviating from the standard definitions.

Customer: The person who uses the VIPER service to pay for item in applications which support it.

Organization: A group or legal entity, registered at VIPER, which can receive payments via their registered applications.

Developer: The member of an organization who develops applications.

User: A person registered at VIPER. This can be either a customer or a developer.

Application: An application that is registered to an organization and has an Application Programming Interface (API) key with which it gets access to the VIPER payment API.

1.5 Structure

This thesis is split up into multiple chapters focusing on specific aspects of the project.
Chapter 2 consists of the general design concept of the product. This chapter starts off with the problem statement that led the team to develop VIPER. The different designs approaches are shown and compared and the final decision is explained.
Chapter 3 shows the business potential of VIPER and describes the business model of the project. In this chapter, the payment model and target customers are shortly explained.
Chapters 4 and 5 include the research, design and implementation of the demonstration applications for AR and VR. They also explain the difficulties that come with payment in restricted environments.

Chapter 6 covers all aspects of the back end, its architecture, the payment process and all associated technologies.
Chapter 7 explains the creation of a web application and especially the challenges that come with this process. It also states the general workflow and design concepts to realize these kinds of applications.
In chapter 8 the project management model used in the project is explained and the different sprints are shortly documented. Additionally, this chapters includes an explanation of the used tools for time and project management.
Finally, chapter 9 concludes the thesis and gives insight into the work done and provides an outlook on the future work of this project.

Chapter 2

Design Concept

2.1 Problem Statement

VR and AR, collectively called XR, offer the ability to expand and, as the name implies, augment the existing world. With XR, it is possible to dive into virtual worlds and do things unimaginable in the real world and expand reality. Unfortunately, these environments come with certain restrictions. State-of-the-art VR, and also most of the more advanced forms of AR, require the user to wear a headset, which can restrict the ability to interact with the real world. This problem is primarily applicable on VR. Thus executing payment in such an environment is difficult, since one cannot enter the required payment information, which is primarily available outside of it (e.g. a credit card number). In order not to interrupt the experience or the workflow of the user, it's crucial to ensure that the headset does not have to be taken off or the environment left in any other way. When setting up conventional online transactions, the user can either use a credit or debit card, for which they have to take out their wallet and enter the numbers on the card, or they can use online payment services like PayPal, which requires them to submit their credentials. The former option is difficult, since the user has to interact with the real world and read the payment card numbers. Another difficulty prevalent in both options is the input of information. AR and VR environments are usually controlled with special controllers, which offer little input possibilities. Thus entering the information would require the user to use a virtual keyboard and navigate it with the controllers. Since most card numbers have at least 16 digits and email addresses and passwords tend to be no different, this task can quickly become arduous and prevent users from buying things. A method for payment in these environments is needed, that requires no interaction outside them and offers security with as little input as possible. Furthermore, these inputs have to correspond to the environments abilities and workflows in order to seamlessly fit in and not disturb the experience.

2.2 Standalone Solution

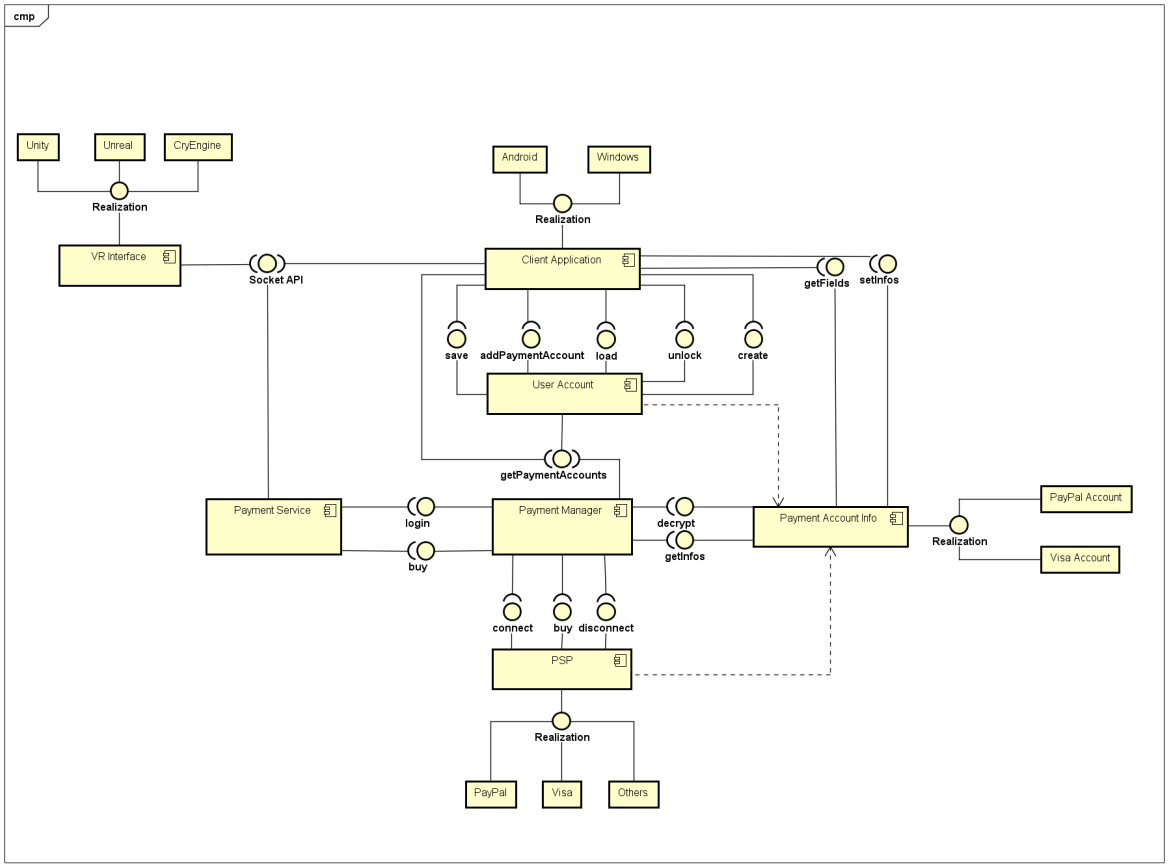


Figure 2.1: Standalone solution software architecture

The first design approach was a standalone solution, which would run as a service on each users PC. The software architecture of this service can be seen in Figure 2.1. The main class of this architecture is *Client Application* which acts as a interface for the user and needs to be implemented for each Operating System (OS), where the payment information is entered. The heart of this solution is the *Payment Manager*, a service running in the background, which offers a Representational State Transfer (ReST) interface for other applications. This service manages all user accounts, payment accounts and executes the transactions. Each VR Application would have to use a VR interface, which is implemented as a software library for different graphic engines.

The advantages of this solution are:

- No infrastructure is needed, since it is running on client machine
- Fast connection, because most parts run off-line and on the same device
- Less security threats, because only minimal data gets send from the device

The disadvantages include:

- Deployment requires the installation of the services
- Updates need new installations, since it runs locally
- Difficult to monetize, because it runs mostly offline
- Using different devices requires re-entering the payment information, because there is no central online data storage

Because the disadvantages of this design clearly outweigh the advantages it was decided not to use this solution and thus this solution is not further elaborated.

2.3 Cloud Solution

It was decided that the project should be easy to use for both customer and retailer and thus a cloud-based service solution was developed. The core components of this solution, from a user perspective, are shown in Figure 2.2 and elaborated in the following sections.

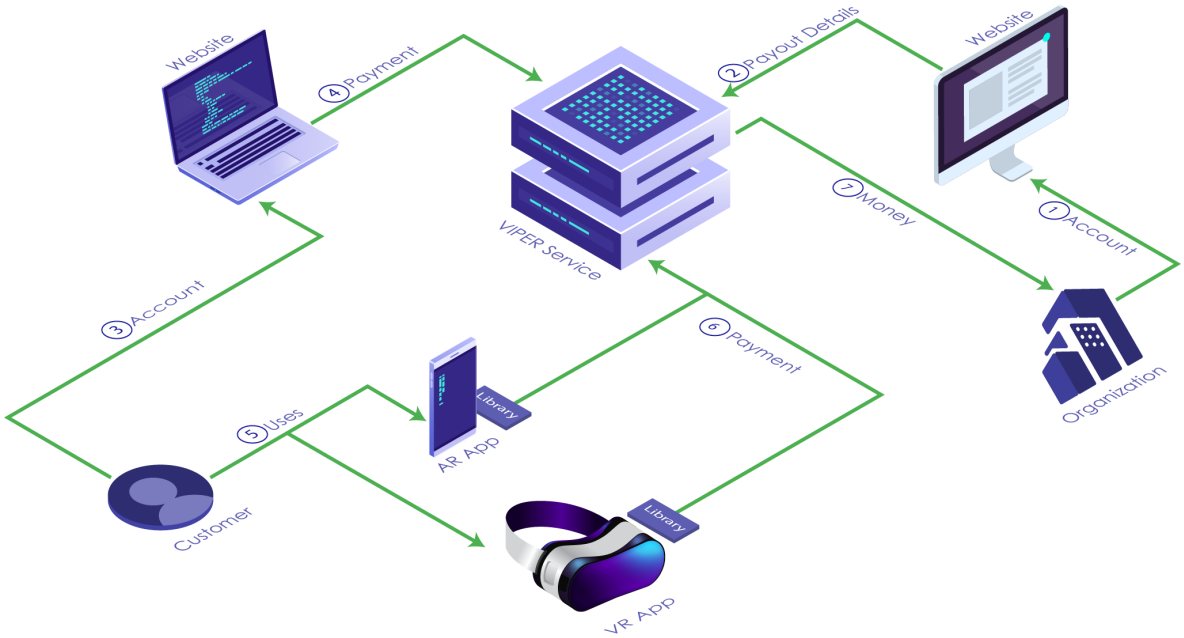


Figure 2.2: Cloud solution component diagram.
1. An Organization creates an account at the VIPER website. 2. The Organization add payment data to receive payments. 3. A Customer creates an account at the VIPER website. 4. The Customer adds payment accounts for future use. 5. The Customer uses an VIPER compatible application that implements the VIPER library. 6. Inside this application the Customer actives a payment flow, selects a previously entered payment account and enters authentication data for it. The transaction with all data needed is sent to the VIPER service. 7. The transaction is executed by the VIPER service and the money is transferred to the organization.

2.3.1 Cloud Services

The cloud service is the heart of this solution, since all other components communicate with the service at all times. The service is responsible for saving the information entered in the web application securely and providing content and data for it. The main task of this service is to use the payment information to execute transactions that transfers money from the Client to the Organization. These transactions get initiated by the client library, which is covered in the next section. Deploying the service in the cloud has several advantages:

- High availability, because of the scalability in the cloud
- Centralized data storage, availability across devices via the Internet
- Plug & Play: no installations necessary
- Monetization possible, because all transactions are centralized

The disadvantages of a cloud-base service are:

- Legal & Security responsibility, because the transactions are made on the VIPER service instead of the client's machine
- High infrastructure costs, since cloud-services are not cheap
- Payments depend on the VIPER service and are not executable without it, since the do not have any business logic implemented locally

2.3.2 Web Application

The goal of this project part was to provide a tool, which is able to implement all services provided by VIPER. This tool would define a unified resource for clients and developers to administrate their data. The decision was to create a website. Furthermore it should support a broad spectrum of devices to address a big audience. For the best User Experience (UX) the website was designed to be intuitive, accessible and responsive.

2.3.3 Client Library

The goal of this project was to make payments in AR and VR applications as easy as possible for developers. It was therefore important to develop a client library which developers can integrate in their applications to communicate with the VIPER services. Even though they client library is not necessary for communication with the back end, it minimized the development effort of developers trying to use the VIPER payment system. To allow for compatibility with a wide variety of VR and AR applications the client library is implemented using widely accepted technologies usable on a wide range of platforms.

2.3.4 Demonstration Applications

To show potential customers the possibilities of our product there are two demonstration applications. These also show developers how they can implement VIPER into their applications. Since our product can be used for both VR- and AR-applications, there will be one

application for each technology. Both of these are connected to the client library thus it is possible to provide clients with the full experience of purchasing virtual objects.

The VR-demonstration focuses on showcasing the benefit of implementing VIPER into VR-applications, by simplifying the payment process from the customers perspective. A user is placed into a virtual showroom, displaying various technical devices available for purchase. They can roam freely and select items on demand, which they can purchase after authenticating.

The AR-demonstration is furthermore split into two parts. The first part allows customers to place virtual items directly into the real environment and thus purchase them via interaction. The second part is based on scanning codes like a Quick Response-code (QR-code). When such a code is recognized the payment process is started. The AR-demonstration does not completely fit the concept of VIPER since the environment has a useful input source (the mobile device's screen) and thus rather functions as a proof that VIPER also works for AR-applications.

The payment process consists of selecting a payment account and verifying it via a code (pattern or pin) which is set in the web application. The type of the code is defined by the developer of the application.

Chapter 3

Business Potential

Almost all Payment Service Providers (PSPs) charge the Organization by receiving a cut off of each payment (see Figure 6.7.2). This makes sense, since this hides the underlying payment services from the customers and puts the overhead on the retailer's side. Customers could also be charged directly for using the platform via a subscription, but this can create the impression of having to pay twice and thus is undesired. Because of this it was decided to target the Organizations as paying customers of VIPER.

3.1 Payment Model

There are three possible payment models, in which VIPER can charge their customers.

Fixed Organization could be charged for creating an account and then use the VIPER service without having to pay again. This would be great for the Organization, since they have perfectly predictable costs. To cover the costs for running VIPER and also make profit, this fixed price would have to be set very high. This makes it difficult for smaller Organizations to use the service and also testing the service for a few months is not possible or would be expensive for VIPER. Another option is to pay a fixed price per Application, but this would also introduce the same problems and only be effective if bigger Organizations always had more applications.

Subscription Another option is to offer subscriptions for Organizations to use VIPER. These subscriptions would be fixed price and allow for a range of transaction per month, for example 1-2 million. The subscription could also be per Application, thus introducing flexibility. This would introduce calculable expenses for the Organizations and calculable income for VIPER.

Per transaction The last option is to charge the Organization by adding a small fee for each transaction. This is advantageous for it scales with the demand of the Organization and is also proportional to the underlying Organization-size. This means small Organization have to pay less and large Organizations have to pay more. Furthermore, the Organizations only have to pay if they get paid through VIPER. Because of the flexibility and fairness of this options, it was decided to use this method in the future.

3.2 Target Customers

There are multiple possible types of organizations that could be targets for VIPER.

Companies VIPER could be interesting for medium to large size companies in either the computer game industry, retail industry or any other online-present companies. Since the main income source of these companies is online-retail, they want to reduce the cost of online transactions as much as possible. This means the additional fees per transactions through VIPER are disadvantageous for them. Furthermore, they will most likely already have implemented online payment methods for theirs services and thus it may be cheaper for them to implement the functionality of VIPER themselves, introducing complete control over their system.

Small groups or individuals Small groups or even individual developers could be more interested in VIPER than companies. These targets usually have few resources and want to spent the ones they have on developing their applications. Thus VIPER poses an easy-to-use payment solution for them, without requiring any infrastructure and little implementation effort. This is why this target group should be focused on primarily.

Chapter 4

Virtual Reality

4.1 Introduction

Virtual Reality (VR) is a new and rapidly growing technology, that has started to gain traction over the past few years. It revolves around the principle of placing a player into a virtually constructed world, that aims to create a vivid experience comparable to the 'real' world. In some cases, the player is even placed in a virtual depiction of the real world, as can be seen in 4.1.

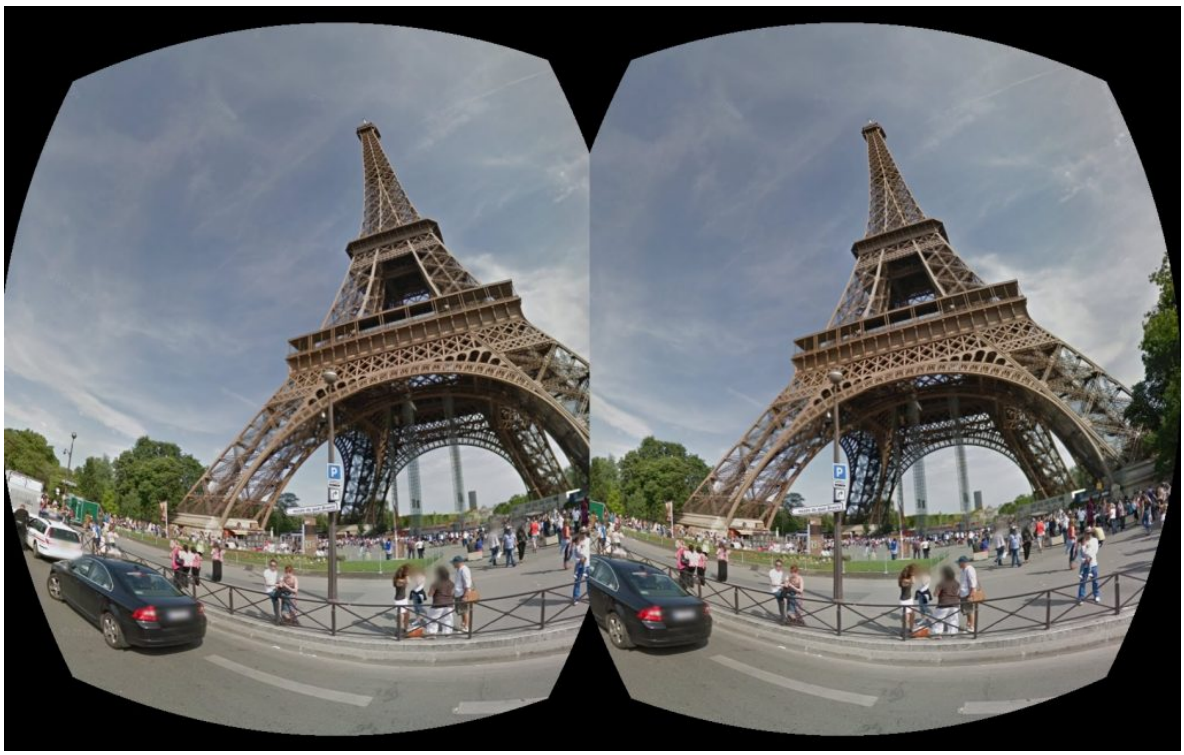


Figure 4.1: VR application showing the Eiffel Tower in Paris[159]

The current main uses for this technology besides video games are military and medical training applications, education and the virtual depiction of real world places.

Figure 4.2: A head mounted display (HMD)[160]

To create an immersive experience, the user wears a head mounted display (HMD) that covers the whole area of sight, focusing the attention onto the displays in front of their eyes. The goal is to create a Field of view (FOV) of the user that comes close to reality. The field of view is the angle at which a person is able to see and perceive. In most humans the FOV ranges from around 200° to 220°, with a binocular vision of about 110°. The binocular vision describes the area in which the sight of both eyes overlaps and what HMD manufacturers of HMDs try to maximize, as it results in a more realistic experience. The lenses used in the HMD play a big factor in increasing immersion for the user. Depending on the thickness and distance to the eyes, the FOV as well as the dimensions of the HMD can vary greatly. The current leaders of the VR market are able to achieve a FOV of about 90°, advancing steadily. [57]

4.1.1 Fields of application

Art and Cinematography
Movies and videos in VR offer a 360° view of the environment and are filmed with specialized camera equipment. Popular applications of this are the live-stream of certain events, such as the World Chess Championship, adult films and open video platforms.

In addition, a number of art museums around the world are starting to develop interactive virtual environments in which the user can roam and explore freely and look at intriguingly presented art.

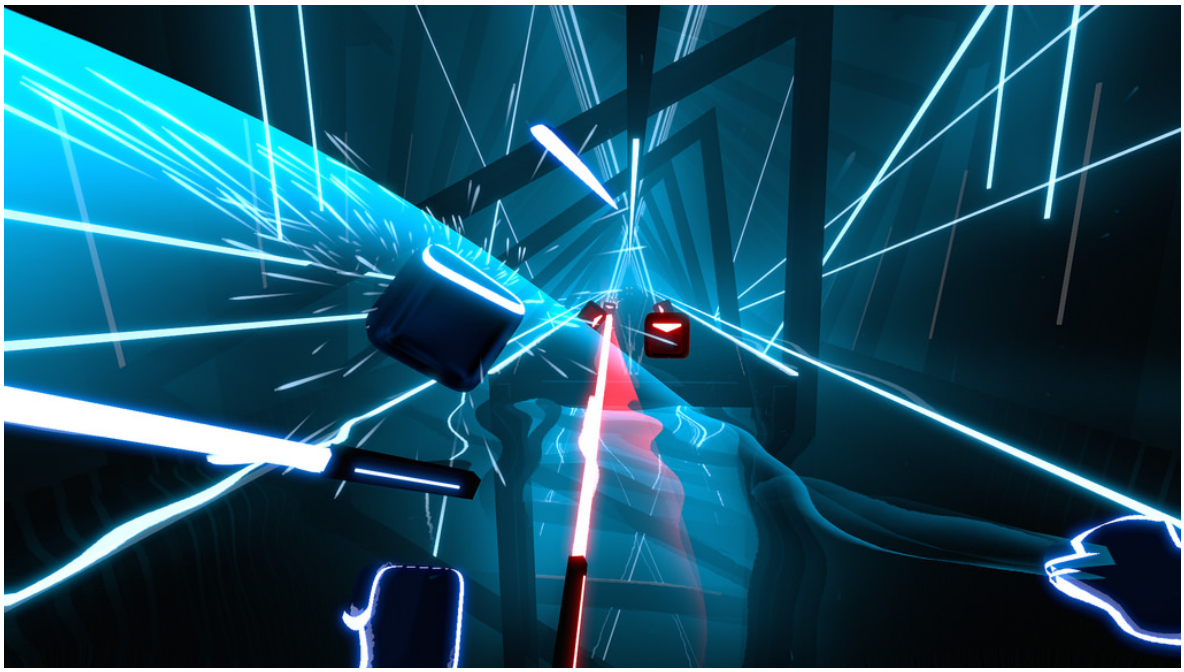


Figure 4.3: Beatsaber, a VR game [20]

Gaming
VR arose from the gaming sector and still drives the progress of this technology year by year. Popular examples such as Subnautica, Skyrim VR and Beat Saber showcase the versatility of VR in gaming and the current market size of 9.6 billion U.S dollars drew attention of many investors who started contributing to this rapidly growing technology. What started out as a fun gadget, with only a few games available, has now be come a part of the gaming industry, supported by some of the biggest game development studios.

One of the most successful applications has been 'VR Chat', a game that connects people online through virtual reality. In this game, the player can customize their character and is then able to freely roam around the world, chatting with other players along the way. Although the graphics were far from realistic, the player experienced a connection to other players that comes close to actuality. Beatsaber, as can be seen in Figure 4.3, is a game in which the user relies on his reflexes to slash approaching objects. His hand gestures are tracked by holding two specialized controllers and moving them accordingly.

Education
A great challenge in education is to maximize the motivation in students. One approach to solving this issue might be the use of VR applications as an interactive and entertaining part of lessons. Especially at an early age, where first-hand experience is a key factor in learning, students are able to see different countries, cultures and sights without leaving their classroom.

The educational opportunities do not stop there, as objects and structures, mechanisms and many other subjects can be displayed and used for teaching purposes.[168]

Healthcare
VR in health care can be used both as a training and a therapy tool. Surgeons can for example train and improve their skills in virtual reality by watching 360° recordings of previous surgeries. VR as therapeutic treatment is used for multiple conditions, including anxiety disorder, autism and Parkinson's disease.

In so-called Virtual Reality exposure therapy the patient is confronted with traumatic stimuli in order to reduce fear or stress responses. The gamification aspect of VR can also lead to increased motivation in patients to pursue therapy, especially in regards to physical activity as a treatment for obesity. [21]

Military
The military utilizes VR mostly as a training tool, offering realistic circumstances of battle situations while being cost-efficient by not consuming ammunition. The training applications go as far as to include instructional feedback, informing the trainee when to shoot and to be alerted.

VR systems are also often used in flight simulators to recreate realistic conditions and educate new pilots and enhance and train existing skills. [169]

4.2 VR Technologies

4.2.1 HMD

Standard HMD
A standard HMD has a built-in display, sensors and buttons, though the actual processing is done by the PC it is connected to. While this leads to immobility, it also offers the advantage of high computational power, resulting in a more vivid and immersive experience. How the headset is connected to the processing device depends on the manufacturer. Generally HDMI and USB ports are used to transmit data between the devices.

Compared to other HMD types, they generally come with higher resolution and refresh rate, as well as a greater FOV, though for a higher price. Current popular devices on the market are the Oculus Rift, the HTC Vive Pro and the Samsung Odyssey.

Standalone HMD
A Standalone HMD does not require a connected PC or smartphone for delivering a VR experience to the user. This means that the device itself contains a processor, battery, a display, gpu, sensors and memory to process various applications. In addition the headset does not require a cabled connection, resulting in less restriction and a more enjoyable experience for the user.

Standalone HMDs are limited by their size and therefore lack computational power and battery capacity. Despite this, they offer a balanced compromise between processing power and mobility, for a medium price.

Screenless HMD
Screenless HMDs are the most lightweight of them, with all processing and display being handled by a smart phone. This brings limitations, as current smart phones simply do not offer as high of a screen resolution and as much computational power to be able to keep up with other types of HMDs. While some manufacturers provide input capabilities such as buttons and touch pads built into the VR device, others rely solely on the smart phone to do the work.

The current market leaders of screenless HMDs are the Samsung GearVR and the Google Cardboard, the latter being an affordable minimalist solution comprised of only cardboard and lenses. [156]

4.2.2 Software and Peripherals

VR SDK
A VR Software Development Kit (SDK) builds the fundamental basis to every VR application. They assist the development process during design, implementation and testing. As the included functionality of an SDK heavily depends on the used hardware, companies in the VR sector created their own SDK for their respective hardware. For example SONYs VR system, the Playstation VR, is able to utilize six-axis motion sensing, cameras and different types of controllers only if the SDK supports these features, hence SONY created their own SDK, the PSVR Dev Kit, in order to cover the support for all these tools the hardware provides.

VR Controller
For many applications the necessity of user inputs arises. While some of the aforementioned HMDs already offer a built in button, touch pad or other similar peripherals, these simply do not cover all of the use cases regarding interaction. Therefore wireless controllers were created.

Depending on the hardware built into the respective HMD the controller is either only able to detect the direction pointed at or is able to localize its exact position relative to the worn device. This relative position can directly be translated to the position of the hand holding the controller, opening new possibilities for increasing immersion in VR, e.g. as the ability to grab objects can now be implemented into applications.

360° Camera
VR movies, videos and showrooms oftentimes require special camera equipment that is able to film at a high FOV, optimally at 360°. Depending on the application, there are different approaches for recording, the most common being two cameras with fish eye lenses connected back to back and filming simultaneously. The footage is then stitched together and converted into 360° video material using specialized software. While most products implemented two cameras filming simultaneously (see Figure 4.4, many others designed technologies with additional built-in cameras. Depending on the number of cameras used, the postprocessing and stitching of the footage becomes more and more difficult for the software to handle, resulting in possibly visible errors and longer processing time.

Figure 4.4: Samsung Gear 360 [135]

4.2.3 Development Environments supporting VR Development

Unity
Unity is a game engine that offers extensive platform support, including Android, iOS, Microsoft Windows and many more. With Unity, the user has the ability to create both 2D and 3D applications, developing primarily in C#. The written application can easily be exported to any platform desired, without the need of any changes, which makes it suitable for cross platform development.

Unity offers three different license options depending on the required features. 'Personal' is the most basic license, supporting all major development features for creating applications for free, as long as the customer's revenue does not exceed 100.000 $ per year. The priced licenses 'Plus' and 'Pro' include additional professional support, game development courses and real time statistics and user data of the developed applications. [164]

Unreal Engine
The Unreal Engine has been the choice for development of many triple-A titles over the past few years. This is most likely due to its powerful graphics and rendering capabilities, making it more efficient over other competitors in the game engine market. While it offers a block-building scheme for beginner programmers, most scripting will have to be done via C++.

In contrast to Unity, Unreal does not offer a free version, instead the developer has to pay a fee of 19.90 $ per month in addition to five percent of all revenue once the applications are made profitable. [166]

Lumberyard
Lumberyard is a game engine owned and developed by Amazon and based on the architecture of the popular CryEngine. It advertises with easy and intuitive use of AWS services and the integration of Twitch, a popular live-streaming platform. As a fairly new competitor, the offered community support is not as extensive and the documentation is not as extensive and intuitive to use compared to other engines.

Lumberyard supports C++ as well as Lua for scripting and programming. With a system called 'Gems', Lumberyard supposedly supports any current, as well as future VR devices, eliminating compatibility issues. Using Lumberyard is free, though utilizing AWS services will cost a small fee, depending on the needed features. [52]

AppGameKit
The AppGameKit is a new tool, optimized for creating 2D environments. While it supports 3D and VR applications, it still is a work in progress. The editors' functions are rather limited but should be sufficient for developing basic applications. As a scripting language AppGameKit uses its own called AGK Script, which is a type of BASIC dialect. They also provide the option to code natively with C++ libraries. In addition, AppGameKit supports VR development, though only for the Oculus Rift and the HTC Vive.

AppGameKit can be bought for a one time fee of 79.99 $, in addition the AGK VR addon is required in order to develop VR applications, which costs another 29.99 $. [12]

4.3 Concept and Objective

The core concept is to create a demonstrational application for the purpose of displaying the functionality of Viper and to act as a guideline project for developers implementing the service. In this application, the user can roam freely inside a virtual room and is able to purchase displayed objects on demand. This scenario places the user into an electronics shop/showroom, offering a few appealingly presented technological pieces. One goal is to design this room in a modern and minimalist fashion, hence only a few items at display rather than many.

4.3.1 Competition

Despite the recency of the idea that payment in VR can be made possible, some companies have started to develop prototypes on their own for tackling this issue. So far, there is no solution providing flexible, simple and extensive payment services to VR and AR applications for any system yet. While these developed prototypes implement the payment in virtual environments, universally functioning solutions have yet to be created. Some of these companies implemented individual solutions for their applications, meaning that these tools cannot be used universally.

Worldpay

Worldpay is a payment solution prototype that uses the EMV technology to process payments in VR. Through use of EMV, it is restricted to card payment services. Furthermore, it only serves as a prototype for future developments and is not a universal solution. The prototype consists of a demo that implements the purchase of different products. When purchasing, a card must be chosen, and if a certain amount of money is exceeded, a pin must be entered.

The pin input was implemented by randomly placing digits around the room, which the player then had to select. This adds an additional layer of security, as the pin cannot be recognized by analyzing the head movement of the player from an outside perspective.

AliPay VR
AliPay proposed a concept of paying in virtual reality in 2016, when presenting a prototype that features payment via nodding and other gestures. After that, a password still had to be entered for authentication, though the headset could stay on during this. Since the presentation of the prototype, no progress has been made public.

Payscout VR
Payscout developed an android app that lets the user walk through a virtual reality experience, such as a shopping mall, a video game or a VR movie and then lets them purchase (physical) goods via VISA Checkout. It is constrained by the system of the device as well as the payment option, as it only supports the option VISA Checkout on Android devices.

Conclusion

The concept of paying in VR and AR is still at its start, but the interest in developing universal solutions is growing. As of today, there is no platform that provides the required services with the support for a diversity of platforms and payment services. The aforementioned prototypes to present functioning implementations of payment processes in VR are not universally applicable solutions for existing applications.

Viper has advantages in different areas over these prototypes, starting at the range of supported payment services. PayPal is the first choice for online payments, though none of the prototypes have implemented payment via PayPal yet. They also lack hardware support, focusing on either Android or via PC functioning devices. This leads to a smaller target audience and less value on the market. The mentioned prototypes were created a year ago or earlier and since then no progress of a universal solution has been made public .

4.3.2 User Interaction

Since the user must be able to not just move, but also to interact with certain objects, a method to process user inputs had to be implemented. The options are either interaction by simply looking at objects/a general direction for a certain period of time, using a controller or utilizing the built in buttons of the HMD. While controllers provide great features, such as more intuitive use, interaction independent of viewing angle and many more, for this specific use case it proves to be unsuitable as the only interaction. Not every HMD is shipped with or even supports VR controllers, thereby only supporting them would drastically shrink the target audience reached with the demo, hence on-board button support was chosen to be implemented as well. The third method of interaction by looking at objects was not implemented, due to poor practicality and usability.

When moving the player, there are two main approaches. One of them is to simply move the player towards a chosen direction by a static, given distance, while the other allows the user to choose a location to be transported to.

4.3.3 Authentication

In VR a wide range of authentication methods can be implemented. One approach is to authenticate via biometric characteristics such as voice as an input method, face or iris scan. AliPay proposed a concept and implemented a prototype showing the authentication via iris scan and with it successfully authorizing a payment process. While it is possible to implemented, it requires specialized hardware for doing so, which not many HMDs feature at the moment. Hence the approach for authentication falls back to simpler ideas, such as to input a pin code by looking at the numbers and selecting them.

As the main selling point of Viper is to simplify the payment process and enhance the users experience, the authentication should have to be as effortless as possible, while offering

as much security and safety as necessary. Therefore entering user credentials consisting of an e-mail and a secure password is not an option, as this does not provide the usability, simplicity and rapidity for the payment process that is required. Alternatives to this are simpler authentication methods, such as pin and pattern. Even while wearing a HMD, the user is able to easily authenticate via pin using the buttons provided by the headset or the controller, as the input is much shorter and simpler compared to an e-mail and a password.

Though this also means, that the user has to be logged in automatically on start of the application. To solve this, the concept is to implement a way to log into the account manually once, which is then done automatically afterwards by saving the login information locally.

For this application specifically, the authentication using a pin input is implemented, as it shows the functionality just as well as a pattern input while being simpler to develop.

4.3.4 Mockup

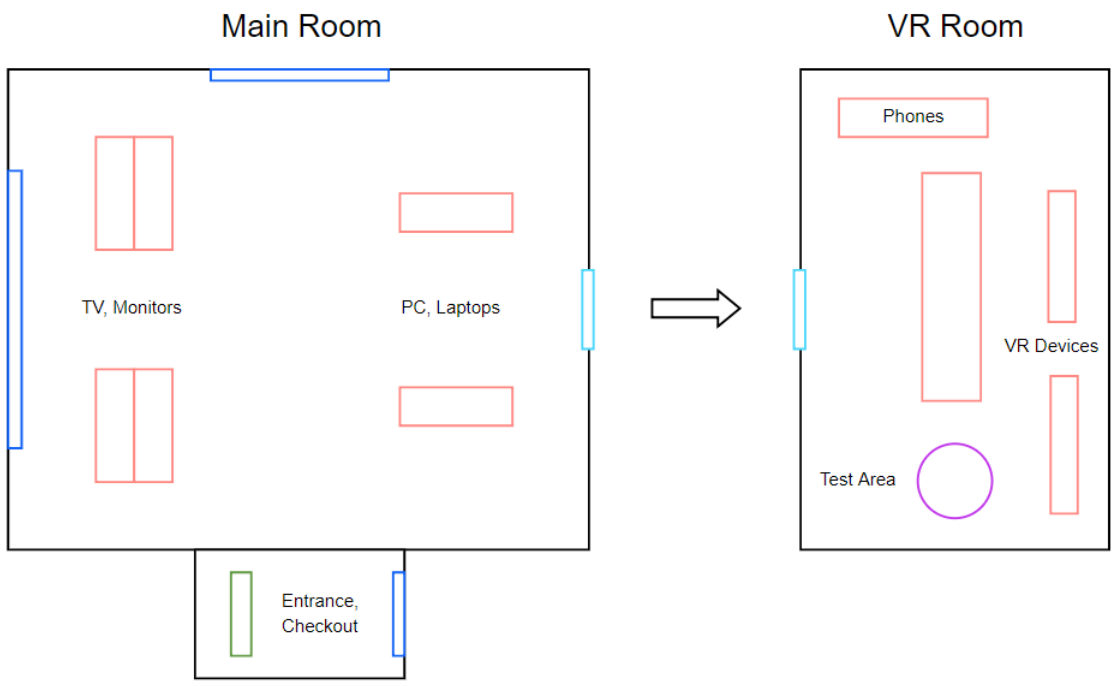


Figure 4.5: Mockup of the VR Showroom

In order to find a suiting design that fulfills the requirements of the client, a mockup was created (see 4.5). It displays a top down view of the application, consisting of 2 rooms, the main room and the so-called VR room. When starting the application, the user is placed into the entrance section of the room, which is openly connected to the main hall. When moving forward, the user is presented a section containing TVs and monitors as well as PCs and laptops. On the right of the main room a door can be found, that takes the player into the VR room, displaying phones and VR related devices.

The mockup acted as a guideline for design purposes throughout development, though adaptions were made, which will be described thoroughly later on.

4.4 Used Technologies

The purpose of this application is to show the fundamentals of the service Viper provides, while reaching the largest possible audience.

For developing the VR application, the Unity engine was used. Unity supports the development for all the largest VR technologies available and with its many features promoting quick and easy development and community support, it suited the use case for Vipers demo application. The cross platform support makes it simple to develop for multiple VR platforms simultaneously and thereby contributes to maximizing supported technologies. In addition, the used license for Unity is free and therefore does not increase the overall costs of the project. Since the popularity of Unity is partly driven by its active community, they aim to provide as much high quality documentation and educational content as possible, in order to encourage new developers to use their product. This results in simple, concise and comprehensible documentation for close to all aspects of the development process with Unity.

In order to test the application during development, a HMD was needed. The most suiting product for this use case was the Samsung GearVR. The GearVR is a screenless HMD compatible with all Samsung smart phones of the latest generations and offers the best quality for its price point of around 100€. It has a built in touch pad and buttons for user interaction, and optionally a controller can be connected to it. As it is developed by Oculus, one of the market leaders of VR technology, it is supported by their own platform SDK. The GearVR headset was used in combination with a Samsung Galaxy S9+, as it was provided by a team member and has sufficient graphics processing capabilities.

4.5 Implementation

4.5.1 Design

Splash Screen

On start of the application, the user is presented with a short splash screen, displaying the Viper logo. After a few seconds, the scene smoothly transitions into the showroom, placing the player in the entrance of the main hall.

The transition was implemented using an imported script. As shown in Listing 4.1, to start the transition the function *Fade()* has to be executed, with arguments describing the scene to which should be transitioned to, the transitional color and the duration. This function is then executed 5 seconds after start of the application, since *Start()* is executed on startup and *WaitForSeconds(5)* being yielded continues to execute it after a given amount of seconds.

```
1 public IEnumerator Start()
2 {
3   yield return new WaitForSeconds(5);
4   Initiate.Fade("main_scene", Color.black, 1.0f);
5 }
```

Listing 4.1: Splash Screen Transition

Room Layout

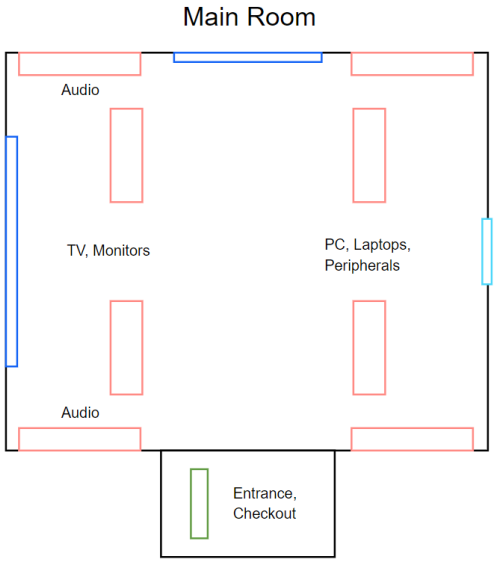


Figure 4.6: Final Layout

While the initial mockup of the room layout provided a solid foundation for designing the application, some adaptions were made during implementation, the final design can be seen at Figure 4.6. Due to time shortage, instead of two separate rooms it was decided to only create one, though with more content in it than originally planned. On the left side of the room, additional audio devices were placed to match the TVs. The right area of the room contains PCs, laptops and various peripheral devices. The player is initially placed in the entrance area, looking into the main room. The opposing wall is decorated with a poster displaying the typeface design of Viper.

The room is lit up with four discreet wall lights, of which one is placed in the entrance area and the others are distributed evenly around the main room (see Figure 4.7).

Figure 4.7: Initial View of the Showroom

Interior Design Characteristics

To create a soothing and enjoyable experience for the user, the room was furnished using light and soft colors. These colors create a sense of familiarity, turning an otherwise sterile and bare room into an enjoyable space. The textures chosen were a lightgray concrete texture for the walls, as well as a light brown wood texture for the floor (see Figure 4.8). The pedestals on which the items to be purchased are displayed are colored in an off-white tone, lighter than the walls and the floor, to highlight the objects on top.

Figure 4.8: Wall and Floor Textures

The assets used for the displayed technological devices are all acquired from external sources providing free to use 3D models.

In the entrance area, the user is able to accept the purchase of chosen objects and to authenticate via pin input, as seen in Figure 4.9. The input panels floating over the counter consist of white buttons, striving for a minimalist design. Below, a table with a cash register and a few office supplies is placed, to indicate where the items are to be purchased.

Figure 4.9: Entrance Area and Authentication Panel

4.5.2 Interaction

One of the main features of the demonstrational application is to be able to move through the showroom, in order to inspect the displayed items, to select them and finally to purchase them at the checkout. As mentioned, it was decided to move the player by a static distance each time they press a button. The basis of all interaction implemented in this demo is the Unity Raycast. With it, all types of interaction, may it be player movement, item selection or authentication can be implemented. As it proves to be the simplest and most universal solution to the task, it was used to develop all user interaction.

Regarding the functionality of a Raycast, a quick overview will be given. A Ray object receives a source position and a direction when instantiated. This Ray, if cast, is then able to intersect with other objects and detect collision. When Raycast is executed, a previously defined Ray object with a chosen reach will be forwarded in its direction, to detect possible intersection with other objects (see Listing 4.2). The hit object, if there is one, will be stored in the variable *hit*, which can then be used for further processing.

```
1 RaycastHit hit;
2 Ray myRay = new Ray(cam.position, cam.forward);
3
4 if (Physics.Raycast(myRay, out hit, reach))
5 {
6 ...
```

Listing 4.2: Using Unity's Raycast

Player Movement

To understand movement of entities in Unity, it is important to know how a player generally is implemented. The application must contain a so-called *main camera object*. The position of this object decides the content being displayed, in a way it can be seen as the eyes of the

player. When it is moved, the displayed content moves with it. All positional attributes of the camera object are stored in its *transform* component.

Implementing movement in VR starts by instantiating a ray, just as mentioned before, after which this ray is cast. The created Ray receives the position of the camera object as its origin and a chosen distance as its reach. With the function *OVRInput.Get(..)*, provided by the Oculus SDK, every user input, meaning every button click, touchpad click or drag, etc. can be checked. It was decided to use a touchpad click for interacting. If the touchpad is clicked, the position of the main-camera is translated by a given amount, after which its *y* will be reset to its previous value, to prevent the player from moving up or down, as shown in Listing 4.3. It is important to note that this code will only be executed if no object was hit and therefore the object *hit* is not defined. This was done to firstly prevent the user from moving when they interact with other objects and secondly to prevent them from moving outside of the room.

```
1
2 // Check for Button Input
3 if (OVRInput.Get(OVRInput.Button.PrimaryTouchpad)){
4   float temp = transform.position.y;
5   // Translate forward by a static distance
6   transform.Translate(-cam.forward * 3f);
7   // Set the y value to its previous value
8   transform.position = new Vector3(transform.position.x, temp,
        transform.position.z);
9 }
```

Listing 4.3: Movement Implementation

Item Selection

Inside the application, the user has the ability to interact with different elements. When the player wants to select an item, they do so by looking at the object and clicking the touchpad, assuming the distance to the object is withing the chosen reach of the Ray. By displaying a text, that a certain item has been selected, the player is notified about the action. An item can be unselected by simply repeating this process.

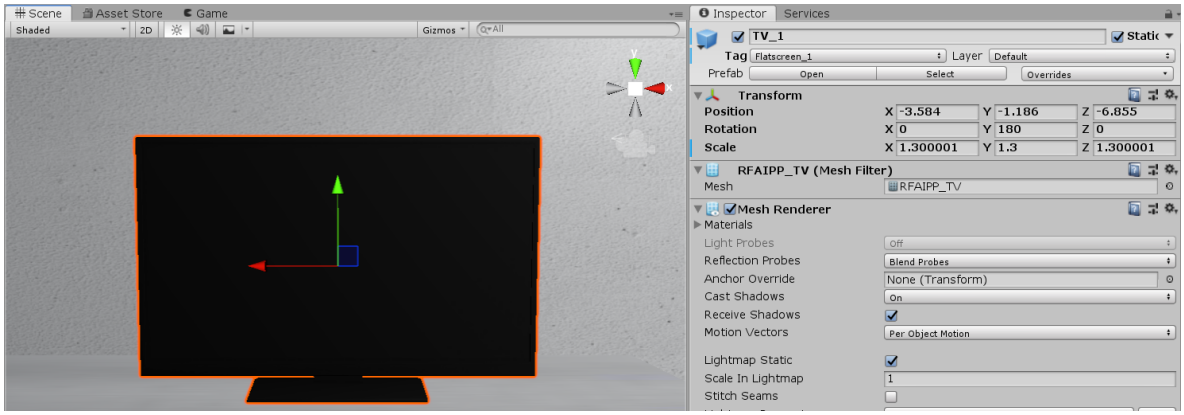


Figure 4.10: Object and Assigned Tag

Before implementing the selection of items, every object available for purchase has to be assigned a unique tag in order to identify them later on. This is done in the *Inspector* tab in Unity by adding a new tag and then assigning it to the corresponding item by choosing it out of a drop down list of all available tags (see Figure 4.10).

Just like player movement, the selection of items is implemented utilizing the Raycast functionality. To check whether the selected item is available for purchase, the tag of the *hit* object is compared to a previously defined array of strings, containing tags of objects that are not buyable, for example the default tag 'Untagged'. If the tag is not contained within this array, the game object corresponding to this tag is fetched.

The function *IsSelected()*(Listing 4.4) now checks, whether the hit object has been selected already and depending on this either adds it to an array of selected game objects or removes it. This array is required to keep track of all selected items, in order to purchase them later on.

```
1
2 if (Array.IndexOf(noObj, hit.collider.tag) == -1){
3   GameObject hitObject = GameObject.FindGameObjectsWithTag(hit
       .collider.tag)[0].gameObject;
4   if (!IsSelected(hitObject)){
5     selected.Add(hitObject);
6   } else {
7     selected.Remove(hitObject);
8   }
9 }
```

Listing 4.4: Object Selection

As soon as an object is selected, the panel for accepting the purchase and authentication appears in the entrance area. This is done calling a function, that sets the *active* attribute of a given object, i.e. the attribute to display that object in the scene, to either true or false, as can be seen in Listing 4.5.

```
1 void Hide(GameObject obj)
2 {
3     obj.SetActive(false);
4 }
5
6 void Show(GameObject obj)
7 {
8     obj.SetActive(true);
9 }
```

Listing 4.5: Showing, Hiding Objects

4.5.3 User Experience

User Feedback Animations

To inform the user about the selection of an item, a text is displayed briefly containing the name and whether it was selected or unselected. To look as appealing as possible, the displayed information slowly fades rather than disappearing abruptly. With the function *FadeCanvasGroup* the alpha value of a CanvasGroup component can smoothly transitioned from one value to another (see Listing 4.6). The default animation time is 1.5 seconds.

```
1 public IEnumerator FadeCanvasGroup(CanvasGroup cg, float start
    , float end, float lerpTime = 1.5f);
```

Listing 4.6: Signature of the fading method

The panel to be displayed has both a *Text* and a *CanvasGroup* component attached to it, which now need to be acquired using the method *GetComponent()*. The displayed text is then changed accordingly, depending on whether the object is contained within the array of selected items or not. The name of the item is acquired from the *hit* object, i.e. the object with which the cast ray collided. Lastly a coroutine is started, fading the fetched CanvasGroup from an alpha value of one to zero.

```
1 Text selectText = selectPanel.GetComponent(typeof(Text)) as
    Text;
2 CanvasGroup selectCG = selectPanel.GetComponent(typeof(
    CanvasGroup)) as CanvasGroup;
3 selectText.text = contained ? "Unselected " + hit.transform.
    parent.name : "Selected " + hit.transform.parent.name;
4 StartCoroutine(FadeCanvasGroup(selectCG, 1, 0));
```

Listing 4.7: Canvas Fade Implementation

Camera Pointer

As it is not always apparent where the center of the camera is located, an indicator was implemented. Googles GVR SDK provides a simple pointer prefab that is easy to use and features options to easily customize color and size. The *GvrReticlePointer* is placed as a direct child element of the *main camera object* in order to function properly. This is done by dragging the prefab into the *game object hierarchy* below the camera object. On start of the application it is then automatically rendered in the center of the view as a uni-colored circle. For the Viper demo application a white pointer was chosen, as it provides the highest contrast to most interactable objects while still retaining the subtle and comforting ambient of the showroom. The hierarchy and implementation of the pointer can be seen at Figure 4.11.

Figure 4.11: Camera Pointer

4.5.4 Payment Process

As the main feature of this application is to demonstrate the functionality of paying in virtual reality, the payment process has to be implemented. The so-called client library handles all aspects of integrating the Viper payment service into the demo by acting as the interface between application and back end.

Figure 4.12: Panel displaying the Selected Objects

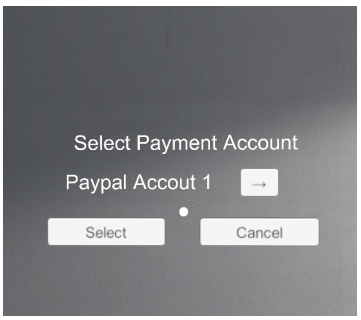


Figure 4.13: Panel for Selecting a Payment Account

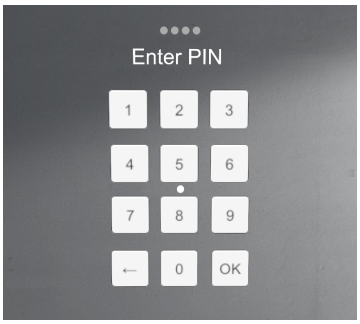


Figure 4.14: Panel for Authentication

From the user's perspective, a panel displaying all selected items will show in the entrance area. If they decide to buy they are presented with another menu in which they can cycle through all their payment accounts registered under their account. Once chosen, a pin input panel will be displayed, after which the payment will be processed. A notification informs the user on whether the payment has been processed successfully or not. It is important to note that in this application, a default account will automatically be logged in rather than the user logging in on their own. These described input panels can be seen at Figure 4.12, Figure 4.13 and Figure 4.14.

For implementing the payment services into the application, the first step is to import all Client Library source files. First they have to be placed within the *Plugins* folder of the Unity project, after which they can be imported as shown in Listing 4.8. Every used function has to be imported using this scheme.

```
[DllImport("VIPER_client", CallingConvention =
    CallingConvention.Cdecl)]
public static extern void login(string api_key, string
    identification, string password, LoginCallbackDel callback)
    ;
```

Listing 4.8: Importing the Client Library

From there on, the next step is to log into an account. The *login* function provided by the Client Library is called on start of the application. As referenced below in Listing 4.9, in this function the given user is logged in and depending on whether the user data matched a registered account will return either true or false. This value must be saved as it is used in a further step. Now the user is automatically logged in and can then proceed to purchase.

```
void Login(string name, string password){
  bool success = login_sync("hyZ  Tc5EqpJwOEspjYGYt6lVY9hRm3l"
      , name, password);
  if(success)
    print("logged in");
  instance.login_success = success;
}
```

Listing 4.9: Logging the User in

After successfully logging in, the payment accounts can be acquired. Inside the function *GetPaymentAccountsCallback*, the accounts are fetched and stored in a global variable to be accessed at a later point. Due to the format the data is provided by the Client Library, the string has to be split up into sections. Each payment account is separated using the *n* flag and attributes within are separated with a tabulator. The correct way to split the string and to store the data is shown in Listing 4.10.

```
string[] paymentAccountsArray = paymentAccounts.Split('\n');
instance.payment_accounts = paymentAccountsArray;
bool temp = true;
Array.ForEach(paymentAccountsArray, paymentAccount => {
  string[] paymentAccountArray = paymentAccount.Split('\t');
  if (temp)
    instance.chosenPaymentAccount = paymentAccountArray[0];
  temp = false;
});
```

Listing 4.10: Receive Payment Accounts

Inside the function *makePayment* orders are now being generated. In the case of Listing 4.11, these orders have the name of the selected objects to identify them, other than that they are filled with randomly generated data. The payment is now initiated via the function *make_payment* by stating the orders, the payment account chosen and an authentication string. This string must contain the correct pin code or else the payment will fail.

```
1 MakePaymentCallbackDel makePaymentCallback = new
      MakePaymentCallbackDel(MakePaymentCallback);
2 Order[] orders = new Order[selected.Count];
3 System.Random rnd = new System.Random();
4 for (int i = 0; i < orders.Length; ++i){
5   orders[i] = new Order("Item " + selected[i].name.ToString(),
        rnd.Next(1, 10), rnd.Next(500, 1000));
6 }
7
8 make_payment(orders, orders.Length, "USD", "This is additional
      info", instance.chosenPaymentAccount, authentication,
     MakePaymentCallback);
```

Listing 4.11: Execute Payment

4.6 Testing

The main approach to testing was manually, at it proves to be the best way of reviewing usability and design choices. Especially immersion must be tested in person, as no test case could rate the realism of the virtual environment accurately. Manual testing in this context means to deploy the application onto a device, after which it is tested with or without a HMD. Since the Samsung GearVR was used, this meant deploying the demo onto the provided Android test device.

Enabling VR Services
Before any VR application can be started on Android and tested with the GearVR, it first has to be signed with a so-called *Oculus Signature* file, *osig*-file in short. This file enables low-level VR functionalities that are inaccessible per default. Furthermore it is tied to only one device, which means that an *osig*-file has to be generated for each device used. After development the application can be submitted to Oculus for verification and if approved, an osig-file is provided that enables these functionalities on all devices.

Oculus provides a service for generating these signature files. As the file is tied to one device only, they require a unique identification. Android provides every device with an exclusive device id, which can be acquired using external applications such as "Device ID". When the device id is discovered, it can be submitted to the osig-generator to receive the signature file.

The applicaton is then signed by placing this file into the folder *Plugins/Android/assets*. Now the application can be started using the device of which the id was submitted.

Running the Application without an HMD

While testing using an HMD is necessary for many aspects of the application, some do not require to be immersed in the virtual room. Hence it can be useful to be able to start the application without the phone being connected to a VR headset.

It is required that at least one application with valid Oculus signature is installed on the device before this feature can be enabled. The first step is to activate the GearVR developer mode, which can be found at *Settings -> Apps -> GearVR Service -> Storage -> Manage Storage*. The VR Service version has to be pressed seven times, before the option for the GearVR developer mode is displayed. The error message 'You are not a developer!' indicates that no validly signed application has been installed on the device.

Once developer mode is enabled, the application can be run without the GearVR headset.

Chapter 5

Augmented Reality

5.1 Introduction

Augmented Reality (AR) is a new technology that proposes the idea of computing virtual objects or images (in the following concluded as “virtual objects”) into the real world. Comparable technologies include VR and Mixed Reality (MR) although they have some significant differences.

Figure 5.1: AR example with text-layers hovering over a rocket-prototype[86]

AR adds virtual objects to the top layer of the image to create the illusion of these objects being part of the real world. For example in Figure 5.1, the real world consists of the model of the rocket as well as the surrounding environment like the people and the room. Additionally, there are text-layers hovering over the real objects. These are virtual objects (here text-images) that are projected into the real world via an electronic device, in this case a tablet.

In comparison to AR, VR is fully immersive meaning that the whole environment as well as the virtual objects are computer-generated. MR on the other hand is very similar to AR but lets virtual items interact with real world items. Thus virtual items can for example appear behind real world items in MR while they can not in AR.

5.2 State of the Art

5.2.1 Fields of Application

AR has several fields of application, the most progressed are entertainment, catalogue shopping, manufacturing and maintenance. In the future there could also be several applications in the areas of robotics, medicine and military (training).[63]

Entertainment

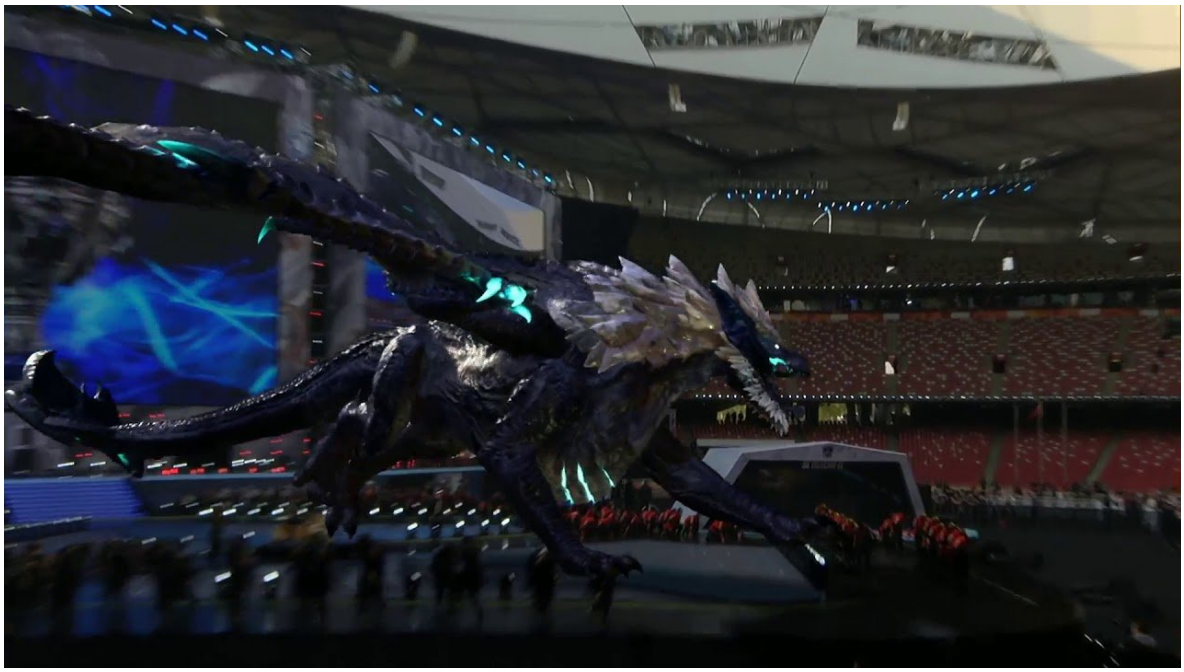


Figure 5.2: AR-dragon at Riot Games' League of Legends World Championship 2017[31]

For the last years AR has become a big player in terms of entertainment purposes, may it be games (*Pokémon GO*[116] or *Riot Games*' AR usage in its *League of Legends World Championship*-opening-ceremonies 2017 and 2018 [Figure 5.2]), advertisement or news business.

Also, the area of entertainment highlights many different implementations due to variable applications. Thus huge advertisement-augmentations require different algorithms than

the client-usable games. AR-advertisements, especially when augmented into stadiums for example, use pre-specified reference points to give the augmentation stability and keep it in the correct place.[63]

Catalogue shopping
In the earlier days, catalogue shopping was looking through a catalogue, searching for some nice furniture or looking for toys, both represented through images, hoping that it fits into the room or looks nice. However, this was not the result all the time which concluded in unsatisfied customers. Nowadays, AR solves this problem as it is capable of displaying images as three-dimensional models. The most popular providers in this field are *Inter IKEA Systems* (in the following simplified as IKEA) and *The Lego Group* (in the following just LEGO) with their AR-catalogues and -applications.

Using the process of image recognition, the AR-application of IKEA recognizes the image in the catalogue and augments the corresponding objects (mostly furnitures) over the picture. The user now has the possibility to move these objects through the room and try to fit them into the space left.

The LEGO-application works similarly. It scans the image on the catalogue, downloads the corresponding object from the database and projects it into the real world. AR-SDKs then provide the user with the possibility to interact with the object.

Manufacturing and Maintenance
One of the biggest problems of instructions is that they can be misleading when being displayed as two-dimensional images or even texts. AR helps to solve this challenge via adding three-dimensionality to the machines giving the user the possibility to see an animated version of the machine in front of them. Thus AR enables the possibility of animating sequences meaning that each step could be shown visually to the maintainer. *Boeing* already developed such a system which is "guiding a technician in building a wiring harness that forms part of an aeroplane's electrical system"[19], meaning that the company allows technicians to create cable networks visually assisted by computer-generated images.

Robotics
The algorithm for implementing a virtual object without having a predefined target (so-called Instant Tracking), Simultaneous Localization And Mapping (SLAM), is already used in robotic vacuum cleaners, more specifically in recognizing where their resting spot is and where in the room they currently are.

But AR could also help in the fields of industrial robotics (visualize what exactly the robot does) and teleoperation. In terms of teleoperation, AR could especially be helpful when the robot is far away from the user.[19] The user could be able to control the robot with a visualization of it in front of him or directly control the visualization itself. After finishing the programming, the program would then be sent to the robot and executed in real time.

Medical
Even though there are no implemented use-cases for medical purposes so far, there are lots of possibilities with AR since visualization plays an extremely important part in the medical field. Pre-operative imaging studies of the patient, such as Computed Tomography (CT) or Magnetic Resonance Imaging (MRI) scans could be visualized in front of the doctor as a three-dimensional model in real size.[19] The doctor then could look at the scan from every possible direction and improve the speed of finding the injury tremendously.

Especially in terms of minimally-invasive surgery, the doctor's ability to see inside the patient currently is very limited. AR could help with this problem by giving the doctor an X-Ray-Vision via the visualization of CT- or MRI-scans.[19]

Military
Especially when training pilots, AR is extremely helpful since it helps to simulate the surroundings of the helicopter/plane making it look as realistic as possible while still working with the real controls.[63] The goal is to give the pilot the feeling of physically being in an aircraft.

AR could also simulate opponents on battlefields helping the troops with tactical decisions as well as improving their shooting accuracy (when simulating the shots as well). After a training session AR could also help find tactical mistakes and improve the strategies even more.[63]

Even in times of war, AR could help visualize the position of enemy troops and thus give some strategic advantages. Via AR, information could be given to each soldier very easily and could even be personalized. Also, shooting assistants could be implemented, allowing the soldier to increase its shooting-accuracy and eventually increasing the strength of the whole army.

5.2.2 AR Technologies

Since the goal of AR is to augment the real world with virtual objects, the main issue of AR is that the system needs to know the position of the user and what he is looking at[140] to find the location of the projection.

This process is mostly assisted by a camera, which perceives the environment and sends the information to the system. The system then calibrates the camera, meaning it analyses the environment to discover where the camera is currently positioned and what its rotation is. This allows a precise projection of virtual objects. There are multiple approaches and therefore solutions to this challenge.

Marker-based AR
As its name states, marker-based AR is based on markers, which can range from QR-codes (Figure 5.3) over high-resolution images to three-dimensional objects.

Figure 5.3: Example QR Code for the website `https://viperpayment.com/`

Even though there is such a huge variety of possible markers the most common are simple small QR-codes (mostly 5x5 or 7x7). This is due to their good and easy readability - they are readable under almost all circumstances and within most environments. Additionally, since computers are better in finding differences in luminance (brightness) than chrominance (colour)[140]. QR-codes are mainly based on black and white squares (sometimes other colours are used as shown in Figure 5.3), thus the computer can easily identify it as such.

Marker-based AR is very easy to implement. Due to the large number of image- and marker-recognizing algorithms (e.g. [130]) and marker-based AR's stability. Once the marker is found, the electronic device knows where to compute the virtual object and does not have to calculate the position every frame. The system then calculates the pose (rotation and orientation) of the marker continuously to find the pose of the virtual object.[140]

Whilst some implementations detect markers frame-by-frame, applications can save the location of markers and use this information when locating the marker in the next frame[140].

Location-based AR

Location-based AR is an implementation that starts augmenting the virtual object when the user enters a specific geographical position and looks into a certain direction. The augmentation itself also has a fixed location, consisting of longitude, latitude and altitude, meaning that its augmented position - different to marker-based AR - is static. The AR-system has to calculate only the distance of the device's and the augmentation's locations.

Additionally, the rotation of the device is important. The augmentation should only be shown if the user is currently looking at it, meaning that only with a specific rotation the augmentation should start being processed on the user's device while otherwise there should be the normal screen.

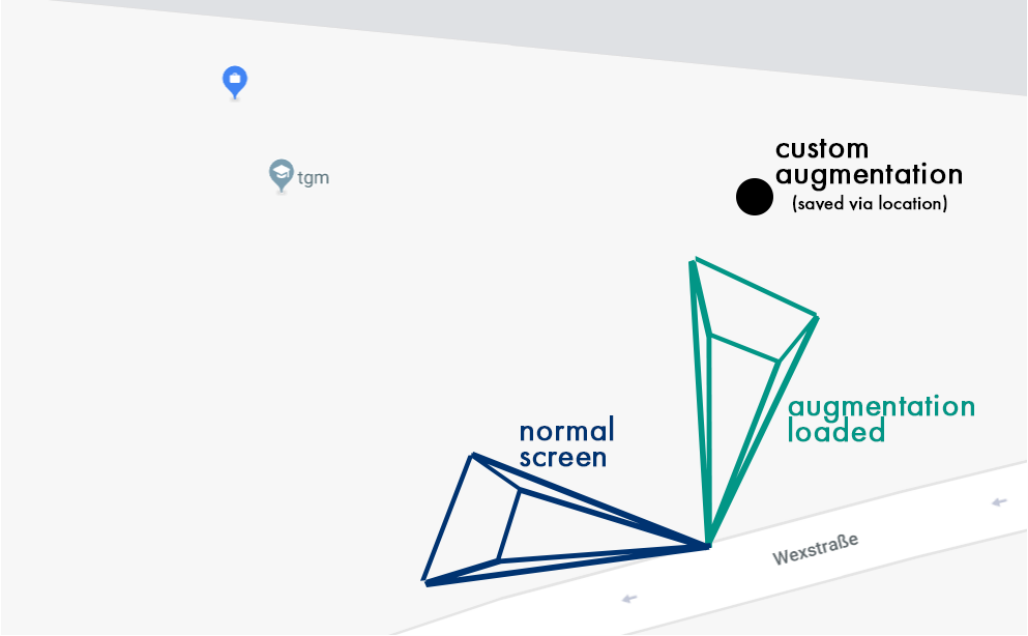


Figure 5.4: An example for geographical positions (background courtesy of *Google* Maps)

Figure 5.4 shows an example of when and when not to augment a virtual object into the real world depending on the device's rotation. Although both locations are within a distance of the augmentation, the left position's rotation is too far off to process the augmentation so the user can not see it. Due to the other device's rotation, the augmentation is within its field of vision and processed on the device.

Marker-less AR

Marker-less AR is probably one of the most difficult algorithms to implement. However, it still is one of the most known types of AR. This is probably due to its large amount of possible applications. In contrast to marker- and location-based AR, marker-less AR can be applied to almost every environment because it functions without a trigger.

The idea of marker-less AR is based on the concept of Simultaneous Localization And Mapping (SLAM). This has the goal of solving the problem of simultaneously locating the position of the device and generating its environment. This system tracks the environment for significant landmarks, such as lines or edges, and combines those into a map where it positions the camera perspectively. Based on this information, a virtual object can be computed into the real world and be visualized correctly and flawlessly.[87]

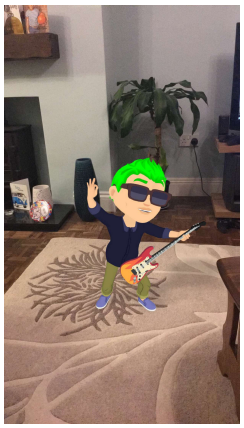


Figure 5.5: Example for marker-less AR with *Snapchat*'s Bitmoji[73]

Figure 5.6: Example for marker-less AR with *Pokémon GO*[27]

Famous applications using marker-less AR, including *Snapchat* (Figure 5.5) and *Pokémon GO* (Figure 5.6) and even the AR-SDKs most mobile devices rely on due to their integration in the OSs, *ARCore* and *ARKit*, provide a marker-less version of AR. This makes including AR into an application as simple as possible.

Complex Augmentation

Complex augmentation is probably the future of AR as it combines both marker- or location-based algorithms with marker-less algorithms. This allows features such as Extended Tracking or *Google* Glass' virtual information of local sites[49]. Even *Google*'s Cloud Anchors use a form of complex augmentation as they are combining location-based AR with marker-less tracking.

Especially when talking about future AR-applications, complex augmentation is always part of those systems. May it be Global Positioning Systems (GPSs), where a location-based augmentation is added to the world both via marker-based or marker-less tracking (e.g. projecting something relative to street highlights or anywhere on the street, but specifically on the street) or AR-systems for training in the military, which could display enemy troops based on their geographical position and other information relative to the environment (Figure 5.7).

Figure 5.7: A fictional example of using complex augmentations in a military training session

Also, IKEA's AR-catalogue is a form of complex augmentation. The image in the catalogue works as a trigger which starts the projection of the specified object. Then, marker-less augmentation is used for displaying the furniture in the room creating the Unique Selling Point (USP) for the application - checking how the furniture looks in the room itself and whether or not it fits in there.

5.2.3 AR-Devices

In contrast to VR and MR, which both are specifically headset-based, AR has multiple types of devices it can run on. Those can be separated in mobile-based and head-mounted.

Mobile AR

Mobile AR is probably the most used form of AR. It can be taken to any place.[33] Still, mobile AR is not the same as portable AR because even though a notebook can be transported from A to B and is capable of augmenting virtual objects, notebooks do not count as mobile devices.

A mobile device which runs mobile AR can be carried around easily and used in any environment. One of the best examples for this is the smartphone as it can be transported and used everywhere relatively easy. Also, tablets count as mobile devices as well since the user can operate with them almost as easily as with smartphones.

Mobile AR is supported by different AR-SDKs, those mainly being *ARCore* and *ARKit* (and the SDKs depending on them). The huge advantage of mobile AR is its possibility of being moved to a real-world-environment that cannot be transported to the device. Thus the possibilities with mobile AR-devices are way bigger than with other devices.

Head-mounted AR
Head-mounted AR-devices (also called AR-glasses) could count as mobile devices but the interaction is completely different. They have no display the user can interact with. AR-glasses have to rely on (predefined) voice- and gesture-based inputs. Additionally, they "cannot be used and worn on a daily basis for now even though this might change in the future" (as stated by Alex Kipman in an interview with *CNET*[139]). Therefore, we are going to put head-mounted AR-devices in their own category. The main challenge of paying with head-mounted AR-devices is the lack of an input possibility other than clicking or typing on a very impractical keyboard. Thus a verification similar to the VR-demonstration needs to be implemented for this type of devices.

However, the "AR"-part of the term "head-mounted AR" can be misleading as it is often rather MR than AR. The best example here is *Microsoft*'s *HoloLens*, which is capable of scanning the environment and letting the virtual objects interact with it. But since AR and MR are very similar, counting head-mounted devices as AR-devices is not that big of a deal. Other popular AR-glasses include *Microsoft*'s new *HoloLens* 2, *Magic Leap*'s *Lightwear*, the *Meta 2*, *ODG*'s *R7* and *Google*'s *Google Glass*. Even though there is a wide variety of headsets, most AR-applications are developed for mobile devices.

AR-glasses do share most of the advantages of mobile AR but the main challenges are its unavailability for consumers, which is mainly caused by its expensiveness.[71] Also, AR-glasses are currently not available for consumers due to the unhandiness as the predefined gestures are very often not recognized or wrongly interpreted.

5.2.4 AR-Software Development Kits (SDKs)

AR-SDKs are implementations of AR-systems. They can further get used in applications or different SDKs. In this section, the term "SDK" will include all three, AR-Software Development Kits, -frameworks and -libraries.

ARCore
ARCore is *Google*'s AR-implementation of an AR-SDK. The main goal of *ARCore* is to make the development of AR-applications easier. Thus it needs to provide support for some key features of AR.

Its key capabilities to integrate virtual content into the real world are motion tracking, environmental understanding and light estimation.[15] To make this possible, *ARCore* implements a Simultaneous Localization And Mapping-algorithm which helps the system computing the device's location as well as building its own understanding of the real world. It tracks so-called Feature Points and uses them to compute the device's change in location.[15] Additionally, *ARCore* supports marker-based tracking via images (and also Extended Tracking) and shared experiences via *Google*'s Cloud Anchors.

ARCore provides support for both *Android*- and *iOS*-devices. The compatibility gets determined by the current OS-version. To run on *Android*, the device must at least support *Android* 7.0 (although some devices are only supported if they run *Android* 8.0), in case of *iOS*, the device must at least run *iOS* 11.0.[16]

ARKit
ARKit is *Apple*'s implementation of an AR-library. Even though it supports only *iOS* devices it still is the biggest competitor to *ARCore*. This is due to the market share of about 22.85%[100]

worldwide. This concludes into over a billion AR-capable devices including every *iPhone* since the 6s.

ARKit primarily focuses on shared and consistent experiences of both marker-based and marker-less tracking. Especially the focus on consistent AR-experiences is unique over the different SDKs. This means that for example a puzzle could be solved in AR and after pausing the user could return to it without needing to redo the whole puzzle.[18] Also, shared AR-experiences enable a whole new market, since they allow users to interact with each other and for example give game-developers the possibility to create multiplayer-games. Next to *ARKit*, only *ARCore* supports the option of "multiuser-AR". Those changes came with the release of *ARKit* 2 in November 2018.

Wikitude

Wikitude is one of the newer AR-SDKs. It includes an implementation of both *ARCore* and *ARKit*. Since this provides *Wikitude* with a functioning base-system, the focus lies on enhancing the algorithms provided by *ARCore* and *ARKit* while still supporting both Operating Systems, *Android* and *iOS*. Especially when it comes to object and scene recognition, *Wikitude* already launched several updates in which those two algorithms where enhanced.[109] Also, *Wikitude* includes support for the *Windows Phone*-OS.

One of the most important aspects of *Wikitude* is its big variety as it supports different marker-based algorithms, location-based tracking, marker-less tracking and also supports complex augmentations such as extended tracking. To make these features work, the phone's OS needs to be at least *Android* 4.4 or *iOS* 9.0. The market share is currently at 83% (*Android*)[99] and 94% (*iOS*)[151], which is a pretty high number of potential customers.[110]

Additionally, *Wikitude* supports a very large number of development environments, including *Unity*, native *Android* developing on *Android Studio*, native *Apple* development on *Xcode* and other application development frameworks such as Cordova, Xamarin or Titanium. They even developed their own development environment as *Wikitude* Studio. Also, *Wikitude* provides support for AR-glasses such as *ODG*'s *R7*[172].

What's also very important for cross-platform-development is *Wikitude*'s ability to be developed in Hypertext Markup Language (HTML) and JavaScript (JS), which can be run on all different Operating Systems since they all provide support for browsers and therefore said technologies.

Vuforia

Vuforia is pretty similar to *Wikitude*, as it also supports all three mobile OS', *Android*, *iOS* and *Windows Phone*. To achieve this, *Vuforia* also includes support for *Android Studio*, *Xcode* and Visual Studio (VS) and their self-developed *Vuforia* Studio.[171] *Vuforia* runs on every mobile device running *Android* 4.4+ (83% market share[99]), *iOS* 9+ (94% market share[151]).

A huge plus of *Vuforia* is its compatibility with *Unity* since it is the only AR-SDK that is implemented in the installer. This support definitely helps *Vuforia*'s popularity as its probably the go-to-option for most developers due to its easy accessibility.

What is also very unique for *Vuforia* is its compatibility with the *HoloLens* (and the *HoloLens* 2), which is one of the few possibilities to create applications for *Microsoft*'s newest MR-product. *Vuforia* also supports marker-based and marker-less tracking, but, other than *Wikitude*, no location-based tracking. This removes some of the potential of *Vuforia* as most of the future applications are expected to include a form of location-based AR.

OpenCV

OpenCV is probably one of the most famous libraries when it comes to computer vision and graphical display. It has the whole package ranging from the detection for augmentation (Image Recognition, Face Detection, Simultaneous Localization And Mapping) to augmenting virtual objects to the image. Additionally to this, *OpenCV* is also used by several of the proposed SDKs.

The advantages of *OpenCV* come with its open-source-availability and its large user-base leading to even more popularity and more implementations. *Android* and *iOS* support is given as well as support for the desktop-Operating Systems *Windows*, *MacOS* and *Linux*. The goal of *OpenCV* is to increase "computational efficiency and with a strong focus on real-time applications"[115].

Comparison of SDKs

	ARCore[15]	***ARKit***[17]	***Wikitude***[172]	***Vuforia***[171]	***OpenCV***[115]
***Unity* support**	++	++	+	++	∼
Free-to-use	+	+	∼	∼	+
Marker-based AR	+	+	+	+	+
Marker-less AR	+	+	+	+	+
Location-based AR	∼	-	+	-	-
***Android* availability**	+	-	+	+	+
***iOS* availability**	+	+	+	+	+
***Windows Mobile* support**	-	-	+	+	-
Shared experiences	+	+	-	-	-
Consistent experiences	-	++	+	-	-
Extended tracking	+	-	+	+	-

Table 5.1: Feature comparison of different SDKs

As seen in this table, *Wikitude* is probably the most diverse SDK as it supports every feature except for shared experiences. Also, *Google*'s *ARCore* has a large diversity since it just does not provide support for *Windows Mobile*-devices and extended tracking. *Vuforia*'s and *ARKit*'s number of features are also very comparable but clearly less than *ARCore* or *Wikitude*. While *Apple*'s *ARKit* does not provide support for extended tracking and *Windows Mobile*-devices, *Vuforia* includes no support for shared and consistent experiences. Also, both of these SDKs do not provide support for location-based AR.

5.2.5 Development Environments

Unity

Unity is an editor for creating games and applications. It is currently the leading development engine for AR- and VR-applications. *Unity* supports two- and three-dimensional development and, which is unique for most development environments, supports deployment for most popular platforms including *iOS*, *Android*, *Windows Phone*, Windows, MacOS, Linux and many more. *Unity* basically supports development and deployment for all platforms across mobile, desktop, AR, VR, console, Television (TV) and the web.[157] Also, they provide a User Interface (UI) which includes an all-in-one-format meaning that only one program needs to be created for all different platforms.

An important aspect of AR-development is *Unity*'s physics engine, which is capable of realistic simulations of real-world physics. Thus augmented virtual objects look and behave more natural giving a better user experience. Also, *Unity*'s implemented UI-engine makes the creation of User Interfaces more simple and creates the possibility of consistent designs, which makes the application more recognizable and decreases the amount of learning processes the user goes through as it was stated in [114].

***Wikitude* Studio and *Vuforia* Studio**

Both *Wikitude* Studio and *Vuforia* Studio are browser-based-editors explicitly created for developing AR-applications. Both editors do have very similar features, which are based on marker-based projection, more precisely augmentation with two- and three-dimensional markers. Those markers can then be extended by multiple virtual elements, for example labels, images, videos, three-dimensional objects and buttons.

Still, they are not the same editors and have some different features, differencing especially in the required file-types (where *Wikitude* Studio uses its own file names for markers). Also, *Vuforia*'s solution is offline while *Wikitude*'s requires an internet connection.

However, both editors require a purchased licence for deployment. Since the goal of both SDKs is to support as many different Operating Systems as possible, both Studios support export for each mobile OS.

Android Studio

Android Studio is *Google*'s editor for *Android*-development. It includes both, a text-editor as well as a visual editor for graphical design. When developing native *Android* applications, *Android Studio* is the most used development editor as it is developed by *Android*'s developer team and featured on its official website.

Additionally, *Android Studio* has an included emulator and a debugger for the deployed application which makes testing of both the application and deployability extremely simple. Also, *Android Studio* includes built-in auto completion for *Android* development, resulting in a better programming experience, and making working with editor simpler.[53]

Xcode

Xcode is for *iOS* what *Android Studio* is for *Android* - the most known and used editor for *iOS*-development. Just as *Android Studio*, it provides the user with a text editor, which is powered by several programming-supporting algorithms, and a visual demonstration of the written code via *OpenGL*. Additionally, *Xcode* includes an assets catalogue, which helps to

group the resolutions of images on the project and thus reducing the project size, a quick-fix-option, static analysis and also provides the user with Continuous Integration (CI). Also, *Xcode* consists of an emulator to test the application in a more realistic environment.

5.3 Design Concept of the Implementation

Since state of the art proposed several possible solutions to each problem the team had to decide what to use for this project.

5.3.1 AR Technology

The first decision was to implement two different applications - one supporting marker-less AR and the other one supporting QR-codes. These applications were forged into one single application which was made available for potential customers. Marker-less AR was chosen due to its popularity, which comes from famous applications like *Snapchat* or *Pokémon GO* that mainly rely on this technology. Additionally, its usability in almost every environment and not requiring a tracker played a huge role in the decision.

Especially in comparison to marker-based AR, the usability in every environment is extremely important since there was no interest in forcing the user into having to print an additional image or even buy a model just to be able to test the application. The argument of different environments was also the reason marker-less tracking was chosen over location-based tracking as this method is bound to a specific location.

A QR-code-implementation was chosen as a secondary application as it demonstrates different real-world use-cases. The main concept was giving the user two different options - buying virtual objects (via the first application) and real objects (via a QR-code).

5.3.2 Development Environment

It was very important to build an application for both *Android* and *iOS*-users to give every AR-developer the possibility to experience the product. Thus *Unity* was the chosen development environment. The challenge with most of the other environments, especially *Android Studio* and *Xcode*, is the focus on either *Android*- or *iOS*-devices. Also, *Unity* provides the option of adding multiple scenes into one application and thus the possibility of forging both applications into one single application.

Another important aspect of the decision was the market share of the chosen technology. *Unity* states on their website that over two-thirds of XR-applications are developed with *Unity*[157]. The other quite unpopular options, *Wikitude* Studio and *Vuforia* Studio, do not have that much developer-base, meaning that developing for such a service do not fit the goal of making VIPER available for as many developers as possible.

5.3.3 AR-SDK and -Device

To fit the concept, mobile AR was given priority for two reasons. First, there are a lot more mobile-AR-applications available and second it is way easier to implement a QR-code-scanner and an AR-application on a mobile device than on a head-mounted device. Since the amount of mobile AR-applications and therefore developers is bigger, VIPER's target market is as well. A head-mounted device would have fit better for VIPER's main target group of developments

for devices with no useful input source. Still, there are not many developers for head-mounted devices and the acquisition of such a device is very expensive.

For the project it was important that the SDK supports *Unity*, so the *Wikitude*-SDK was used as the framework. Due to the implementation of both *ARCore* and *ARKit*, the SDK provides a good base as it shows that VIPER can be used with every popular SDK. Although *Vuforia* works quite similar and has better *Unity* support, *Wikitude* was chosen, mainly because of its better documentation and better device support. Especially when testing *Vuforia*'s demo-application on the smartphone model "OnePlus 3T", which according to *Vuforia* was supporting, the application did not work while *Wikitude*'s demo had no errors when running.

The only issue with *Wikitude* was its price. For development, the project used a trial version and applied for a Startup-version.

5.3.4 Mockup for an Implementation

Before starting to build the application, the team decided on its looks. Thus a mockup was created which represents the ideas of the application. It shows the process of purchasing an object with its different components.

Figure 5.8 in this case mocks the start menu while Figure 5.9 and Figure 5.10 show the possibilities of displaying an object and switching between different objects. Figure 5.11, Figure 5.12 and Figure 5.13 show the different steps of the payment process. As shown on these figures, the idea was to use very similar layouts to decrease the number of learning processes and also increase the recognition factor.

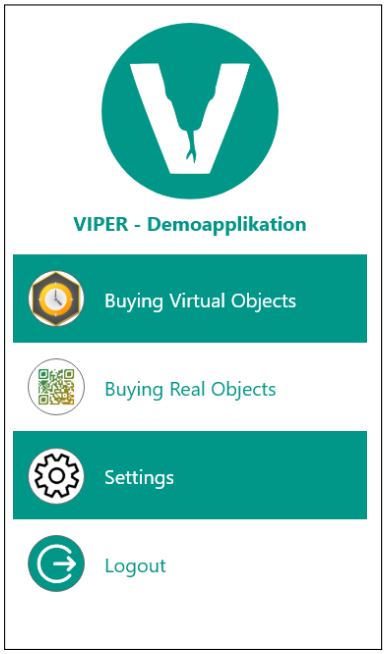


Figure 5.8: Mockup of the start menu

Figure 5.9: Mockup of the application displaying the first virtual object

Figure 5.10: Mockup of the application displaying the second virtual object

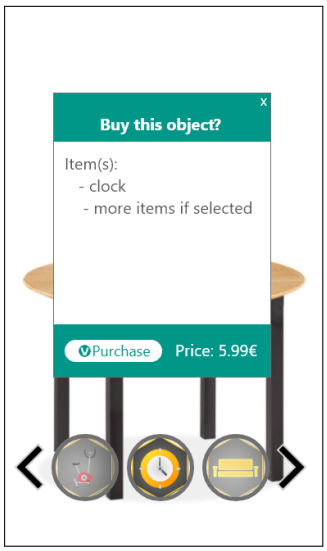


Figure 5.11: Mockup of the buying screen for the second virtual object

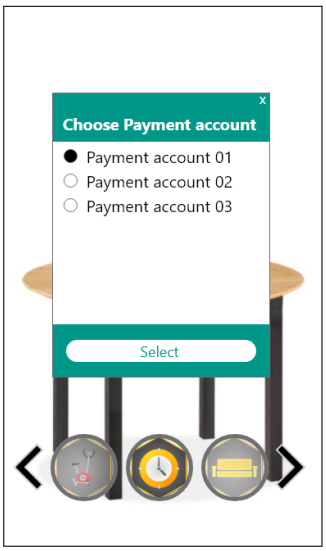


Figure 5.12: Mockup for the payment-accounts screen

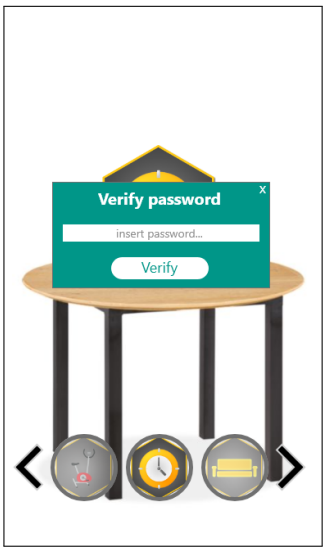


Figure 5.13: Mockup of the verification-screen

5.4 Implementation

5.4.1 Login and Start Menu

Design

The login and start menu screens are the first visible screens for the user. They were designed as simple as possible to remove large learning processes (Figure 5.14 and Figure 5.15).

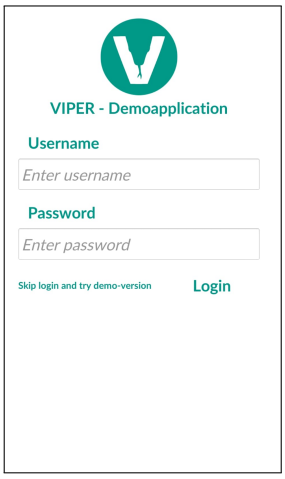


Figure 5.14: Login screen of the application

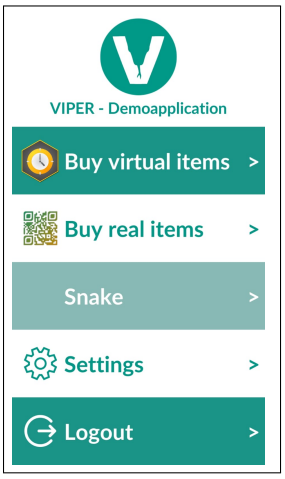


Figure 5.15: Start menu of the application

As visible on Figure 5.14 and Figure 5.15, only three colours were used: Teal, as discussed in subsection 7.2.2, white and black, which were used to add some contrast to the primary colour and still keep a simple design that is not too colour-heavy. Also, both screens were implemented through *Unity*'s UI to stay as consistent as possible.

Functionality
The main functionality of the login panel and start menu is to give the user the possibility to log into his account and start the different parts of the application: the settings-page, the first application and the second application.

Login The login functionality was implemented through the client library. In "StartMenu.cs" the method `LoginSync` calls the client library's login_sync-method, which logs in the user (Listing 5.1).

```
1 bool LoginSync(string name, string password) {
2   return Payment.login_sync(
3     "hyZTc5EqpJwOEspjYGYt6lVY9hRm3l", name, password);
4 }
```

Listing 5.1: LoginSync-method that logs in the user using its name and password

Start Menu Navigation To give the user the possibility to navigate through the different parts of the application, the different scenes implemented into *Unity* were loaded using the *ChangeScene*-method in *MenuController.cs* (Listing 5.2).

```
1 public void ChangeScene(Button sender) {
2   #if UNITY_5_3_OR_NEWER  // >= \emph{Unity} 5.3
3     SceneManager.LoadScene(sender.name);
4   #else                   // < \emph{Unity} 5.3
5     Application.LoadLevel(sender.name);
6   #endif
7 }
```

Listing 5.2: ChangeScene-method that allows the user to load a new scene and change to it

***Android* Accounts**
To enhance the User Experience (UX) on *Android*-devices, some of the functionality of *Android*'s *Account-Manager* was imported (Listing 5.3). This code allows the application to save information like the user's name and his password with a specific account type to an authenticator (which is implemented in every *Android* device since version 2.0). When restarting the application, it checks whether a user with the specific account type exists and if this is the case signs the user in automatically. In *Settings.cs*, an *AndroidAccount*-object is created and then saved in the attribute (`static AndroidAccount Account;`).

```
if (Application.platform == RuntimePlatform.\emph{Android}
  && (!Settings.IsLoggedIn() || Settings.AccountChanged))
  if (Settings.Account.CheckAccount()) {
    VIPERAccount acc =
      Settings.Account.GetAccount(Settings.CurrentAccountIndex
         );
    bool success = LoginSync(acc.Name, acc.Password);
    if (success) {
    Settings.ShowMessage("Successfully logged in as "+acc.Name
       );
    Settings.SetSession(acc.Name);
    } else
      Settings.Account.RemoveAccount(acc.Name);
    Settings.AccountChanged = false;
  }
```

Listing 5.3: Automatically log in the user if an account is saved at *Android*'s *Account-Manager*

It calls the Java-methods `boolean addAccountExplicitly(Account account,string password, Bundle userdata)` (add account), `boolean removeAccountExplicitly(Account account)` (remove account) and `Account[] getAccountsByType(string type)` (get all accounts), which are implemented through the Java-plugin *viper.aar*. Getting a single account returns the index of the object in the array.

To add the *Account-Manager* to the application, the file *viper.aar* must be placed into the "Plugins"-folder. This file includes all the requirements needed for adding a VIPER-account.

5.4.2 Settings

In the settings-dialogue, users can manage their payment accounts in the application. The user can select one of the saved payment-accounts (as explained in the *Android* Accounts-section) as the payment account. The settings also include options like hiding the tutorial and either purchasing one or multiple items.

For displaying the settings, *Unity*'s UI-elements and the colours discussed in subsection 7.2.2 were used. There are two versions of the settings: one version for *Android* with the *Account-Manager* (Figure 5.16) and one without it for all other devices (Figure 5.17).

These settings will be used throughout the whole application and can be accessed via their static variables:

```
public static bool PurchaseMultipleItems {get; set; }
public static bool VirtualTutorialFinished { get; set; }
public static bool AccountChanged { get; internal set; }
public static int CurrentAccountIndex { get; internal set; }
```

Listing 5.4: Implemented functionality of *Android*'s *Account-Manager* (with create, remove and read)

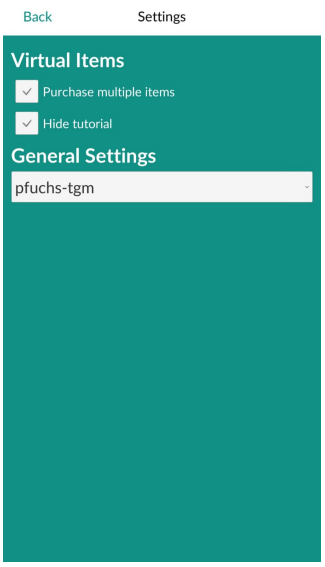


Figure 5.16: Settings screen on an *Android* device (notice the general settings)

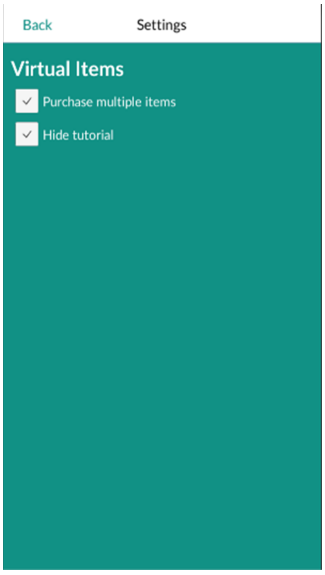


Figure 5.17: Settings screen on any device but *Android*

5.4.3 Demo Application for Purchasing Virtual Items

The first application implements the main functionality of this demo application. It allows the user to import virtual items into the real world to give him the possibility of purchasing these elements via VIPER.

SDK-Implementation
First, the chosen SDK, *Wikitude*, had to be implemented. Thus the *Unity*-SDK was downloaded from the website (wikitude.com/download)) and implemented via importing the "Wikitude.unitypackage"-file. Also, the example-project ("Examples/WiktiudeUnityExample") was added to the *Unity*-project. This allowed a starting point from which the application was built.

Design
The design is an adaptation of the example project. The main differences are some removals of unneeded content as well as some colours which were changed to match the application. The result can be seen in Figure 5.18 (in comparison to the old design on Figure 5.19).

Functionality
Most of the functionality is already introduced with the implementation of *Wikitude*'s example-project. This includes detecting a surface panel, dragging objects on that panel and scaling and moving that object.

Figure 5.18: Design of the first application

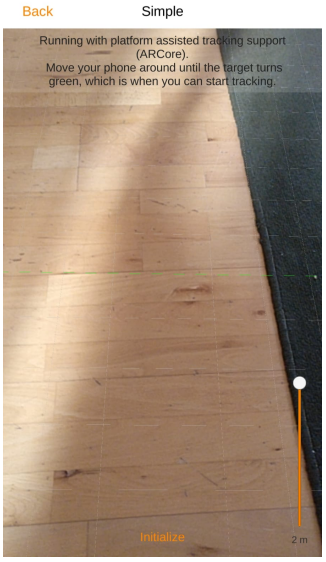


Figure 5.19: Old design provided by *Wikitude* (the key can only be used for the demo, it is unusable for our demo)

The possibilities of clicking on an object and adding an outline-shader (*OutlineShader.shader*) were added to let the user see, which objects are currently selected (Figure 5.20). The user can also rotate the object, deleting one or all objects, chose if he wants to display only one or multiple objects and reset the grid when no item is on it. The application includes a tutorial which gives the user a short introduction so he knows how the application works.

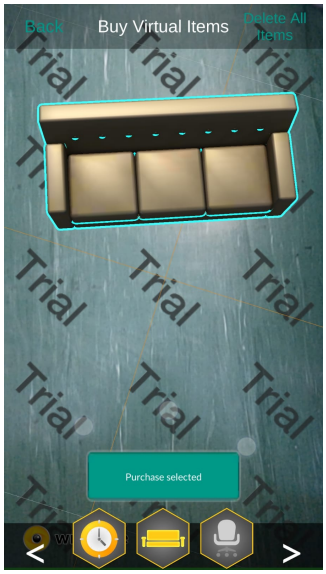


Figure 5.20: Displaying a selected item (in this case a couch)

***Wikitude*-SDK**
The features introduced with *Wikitude*'s SDK are the key technologies for using this application. Especially its implementation of marker-less tracking is the key feature for adding virtual objects which happens to be the key feature of this application. Also, *Wikitude*'s implementation of moving and scaling provided an idea on how these technologies could be implemented into the application.

SLAM The SLAM-algorithm is implementing the functionality for marker-less tracking. It basically scans the environment for unique points, also called features or landmarks. They are saved by the algorithm to create a map of the environment. With this map, the algorithm then calculates where its source (mainly a camera) is currently positioned. Enlarging the map and calculating the map for the environment happens simultaneously which concludes in its name "Simultaneous Localization And Mapping".

Wikitude now searches for landmarks that are on the same level. With that points, the algorithm creates a plane on which virtual elements can be placed (e.g. the grid being visible in Figure 5.18 and Figure 5.19). This reduces the three-dimensional space (with XYZ-coordinates) from a real-world-environment to a two-dimensional space (with XY-coordinates). The object can only be moved on this two-dimensional plane thus there is no confusion with moving an object up or down on the Z-axis.

Of course, the efficiency of these algorithms is depending on the environment. A very feature-rich environment (means that it has a lot of unique points) helps to find a plane far more easily since its calculation is easier with more feature points. This also leads to a more stable plane.

Move, Scale and Rotate Objects Moving, scaling and rotating an object uses *Unity*'s raycasts (Figure 5.21) to interact with the virtual objects.

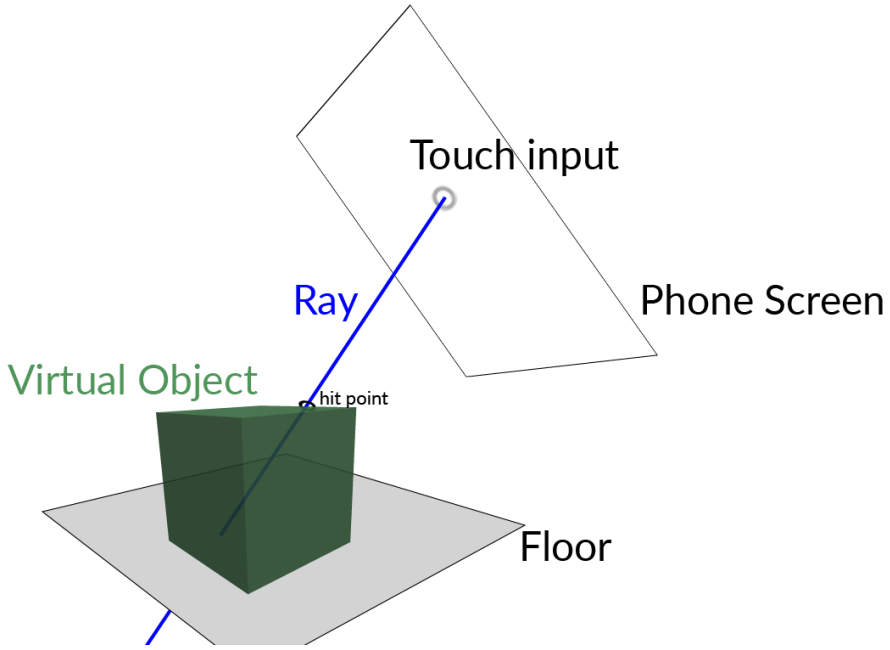


Figure 5.21: Functionality of a raycast [104]

When one finger is placed on the screen, the algorithm responsible for the moving an object is called. It casts a ray at the environment and, when hitting an object, saves the hit object as an attribute. When moving the finger around, new rays get cast to whose position the object gets translated to (Listing 5.5).

```
1 var position = cameraRay.GetPoint(enter);
2 position.y = 0.0f; // the height [here stated as y-axis] -> on
      plane
3 _active.position = Vector3.Lerp(_active.position, position,
     0.25f);
```

Listing 5.5: Translate an object from one *position* to another given by the movement of the user

Scaling an object needs two fingers placed on the screen. One finger needs to be placed on the object itself, the other one next to it, creating two points on the screen. The algorithm now calculates the distance of these two points and scales the object relative to the change of distance. Less distance means the object scales down (zooming out), more distance means the object gets scaled up (zooming in) as visualized in Figure 5.22.

Rotating works very similar to scaling since the user needs to put two fingers on the screen, the first one positioned on the object and the second one next to it. When initially touching the screen with two fingers, the application saves the distance between those fingers as a line. Now, every frame, a new line between those fingers is formed leaving an angle between those lines. With this angle, the rotation of the virtual element gets updated (Figure 5.23 and Figure 5.24).

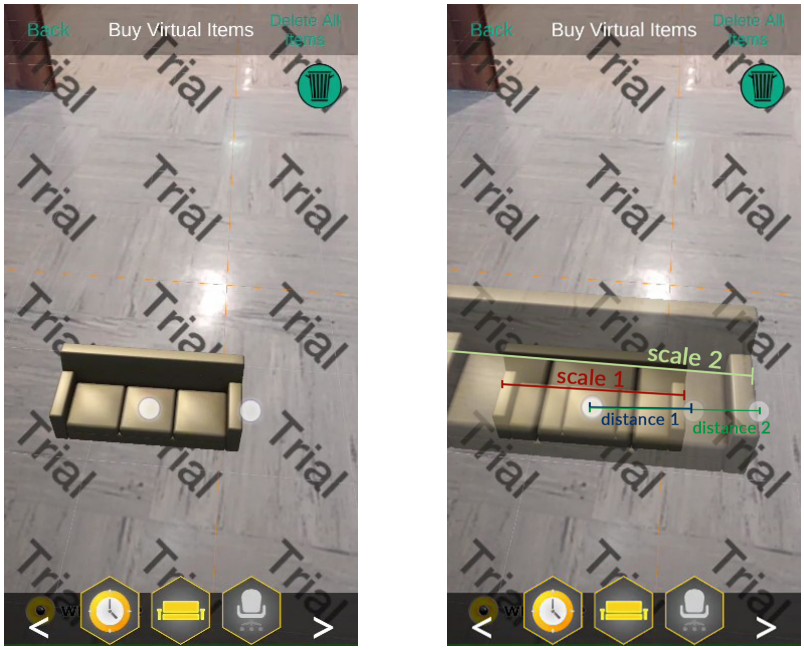


Figure 5.22: Scaling the application

Figure 5.23: Initial position of the object before the rotation happens

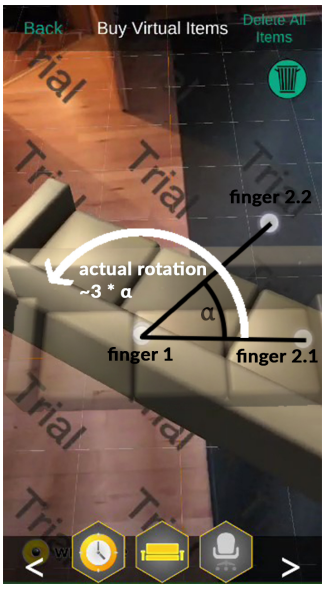


Figure 5.24: Rotation of the object after performing a rotation on the screen (with the angle made visible)

Click on Objects Like scaling, moving and rotating, clicking on an object is supported through *Unity*'s raycasts (Figure 5.21). When initially touching the screen and hitting an object, the finger's touch position is saved. While the user touches the screen the current position of the finger gets calculated and subtracted from the saved (first) one. If there is a difference, meaning that the user did move his finger, the action is not counted as a click. Otherwise, the action is detected as a click on the hit object.

This functionality is implemented in the *ClickController.cs* with the variable `bool moved;` that gets set to true when the user moved his finger during a click (Listing 5.6). When the user releases his finger and it has not been moved throughout the process the method `void ObjectClick(Transform clickedObject)` handles the event. In this case, it adds the clicked object to the active items and adds a shader to it as well as enabling the button for purchasing objects.

```
1 if (!_moved) { // only execute if finger not yet moved
2   Vector2 movedBy = touch.position - _startTouchPosition;
3   _moved = (movedBy.x != 0 && movedBy.y != 0);
4 }
```

Listing 5.6: Check if the user moved his finger while touching the screen

Delete Items Deleting items is one of the most important things for UX as having to delete all items when an unwanted item was added might cause a lot of frustration. Thus a very simple method of removing items was implemented. When an item is selected, a trash can-icon appears on the screen. When the user hovers it, the currently selected item gets deleted (Figure 5.25 and Figure 5.26).

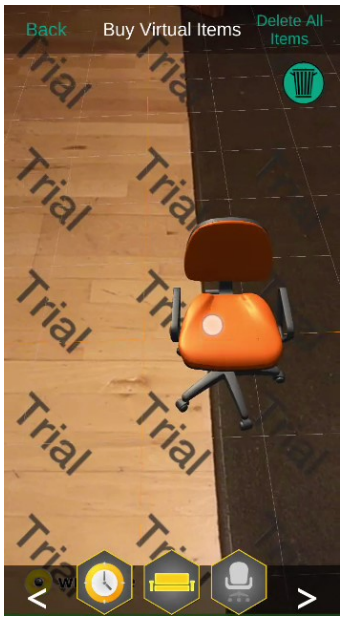


Figure 5.25: Selecting the object and seeing the trash can

Figure 5.26: Dragging the object to the trash can and deleting it

This functionality takes place whenever the user puts his finger off the screen, meaning when the *MoveController.cs* stops casting rays. Then the application checks whether or not the `Vector2 _lastPosition` was on the trash can (via `bool isOnTrashCan(Vector2 pos)`) and depending on that deletes the selected object. In either case the last position gets reset after calling the function (`_lastPosition = new Vector2(-1, -1);`) so that the user can not trigger the process unintentionally.

Display and Purchase Single Items Displaying single items was introduced to add a functionality similar to *Snapchat*'s AR-implementation where only one object at a time can be displayed (Figure 5.5). Via the settings page (Figure 5.17) this option can be en- and disabled. When starting the application, the user now has to click on the indicators rather than dragging them around (Figure 5.27). Clicking on one button then calls `void PlaceToCenter(int modelIndex)` (Listing 5.7).

```
1 int x = count; // Set x to a counter
2 // and add the listener as well as disabling the EventListener
3 b.onClick.AddListener(() => PlaceToCenter(x));
4 b.GetComponent<EventTrigger>().enabled = Settings.
      PurchaseMultipleItems;
```

Listing 5.7: Remove the possibility to drag and add the click-listener

For this method to work, all objects need to be removed before. This happens in `PlaceToCenter(int modelIndex)` where all active elements get removed (Listing 5.8). Then a new object with model corresponding to the index given as a parameter is created.

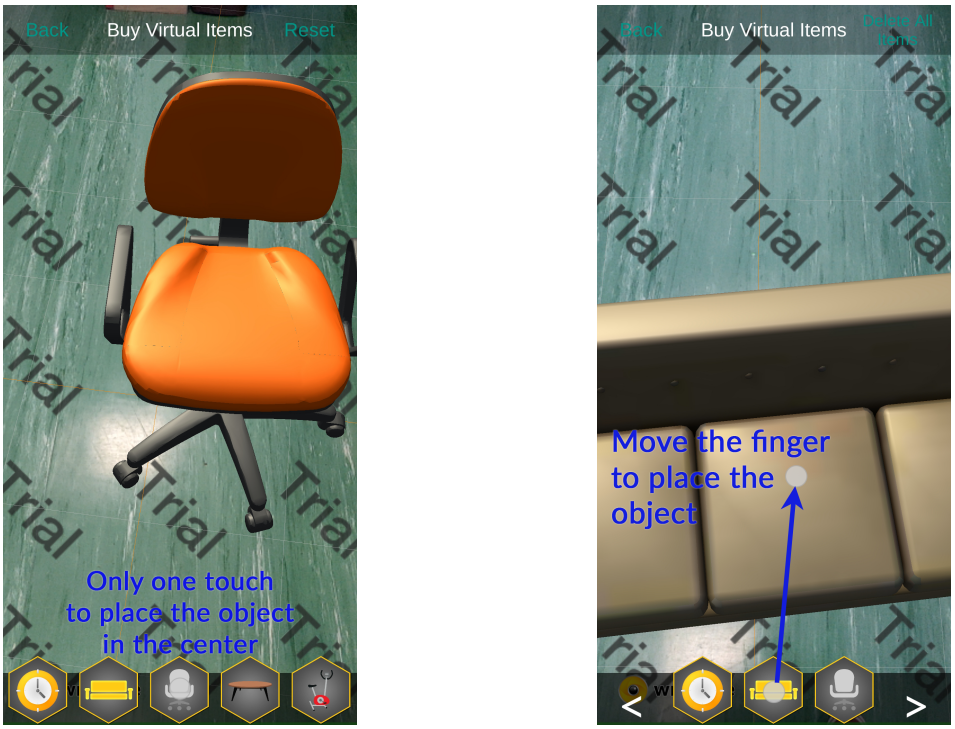
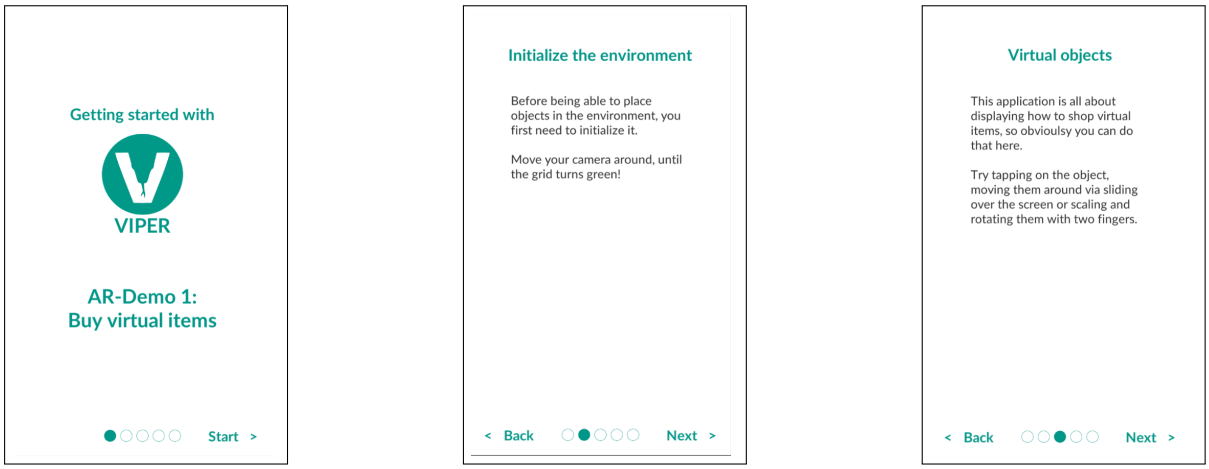


Figure 5.27: Comparison between dragging an element in and placing it into the centre via clicking

```
1 foreach (GameObject go in _activeModels)
2   Destroy(go);
3 _activeModels.Clear()
```

Listing 5.8: Remove all objects from the screen

Tutorial The tutorial is displayed when the user starts the application for the first time. It describes what the user needs to do in order to use VIPER:



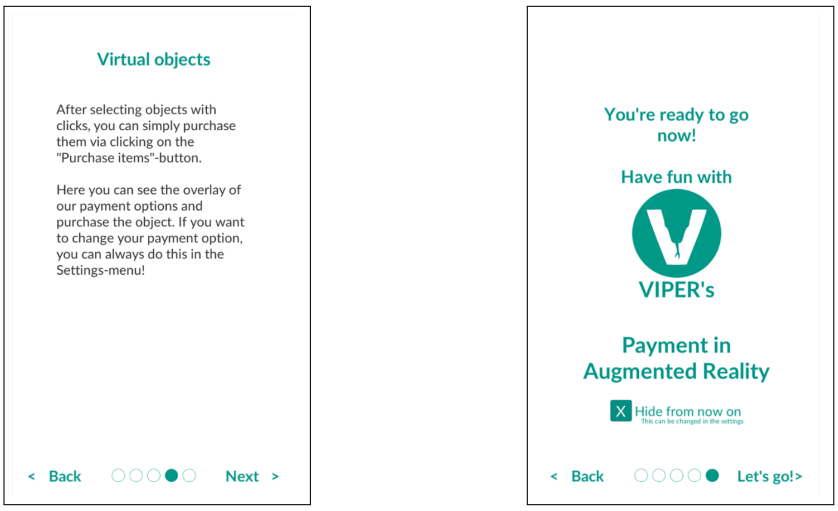


Figure 5.28: The tutorial for the first application which consists of five different pages

In the settings, the user can select whether or not he wants to display the tutorial again the next time he starts the application.

Reset the Grid Resetting the grid was a very important use-case for simple restarting especially when a scene gets lost. Without this functionality, the user would have to go back to the start menu and reload the application. This was implemented through the function `void OnResetButtonClicked()`, which resets all the current values as seen in Listing 5.9.

```
1  _itemController.Actives.Clear();
2 _clickController.PurchaseButton.SetActive(false);
3 // This line is very important since it resets the grid
4 Tracker.SetState(InstantTrackingState.Initializing);
```

Listing 5.9: Reset the grid and go back to tracking-state

5.4.4 Demo Application for Purchasing Real Items via a QR-code

The second application demonstrates how to purchase real items via QR-code and/or markers. A plug-in called *ZXing* was used that allows scanning the environment for QR-codes. When finding a code it needs to match the conditions for VIPER to be recognized as valid. Thus it needs three keys and corresponding values: *description* which is a string, *price* (in cents), which has to be a non-digit value, and *amount*, which is an integer.

Figure 5.29: Example for a QR-code which works with VIPER's QR-demo

Figure 5.29 shows an example of a QR-code for VIPER. When scanning this code the result will be `{"description": "ABC", "price": 23, "amount": 1}`, so one item called "ABC" is bought for 0.23 currencies (which is Euro by default but can also be for example US-Dollars).

The scanning result gets handled in the file *PluginController.cs* in the method `void Update()`. When a result is found, the method `void MakePayment(string data)` gets called which checks whether the result was valid (Listing 5.10). Scanning a QR-code also automatically triggers the payment process (section 1.4.7 Payment process).

```
1 try { // create an Order-object from the data
2 order = JsonUtility.FromJson<Order>(data);
3 itemController.PurchaseItems(new Order[] { order });
4 } catch (ArgumentException)
5 // if data is not an Order-object => exception
6 { Settings.ShowMessage("Not a VIPER-Code"); }
```

Listing 5.10: Check the data (saved as a string) in the recognized QR-code whether it is correct

5.4.5 Messaging

The demo application needs to send some messages to the user. These messages contain useful information about confirmations or errors. Since there is no useful implementation in *Unity*, a messaging system based on *Android*'s Toasts (messages) was added. The corresponding file for this feature is *ToastMessage.cs* as it enables the possibility of displaying a message with its internal method `IEnumerator ShowMessage(string text, float time = 2.5f)`. The return-type *IEnumerator* specifies that the method gets called in a *Coroutine* meaning that it runs on another thread. This allows pausing that thread for a time while the application in the main thread still works normally.

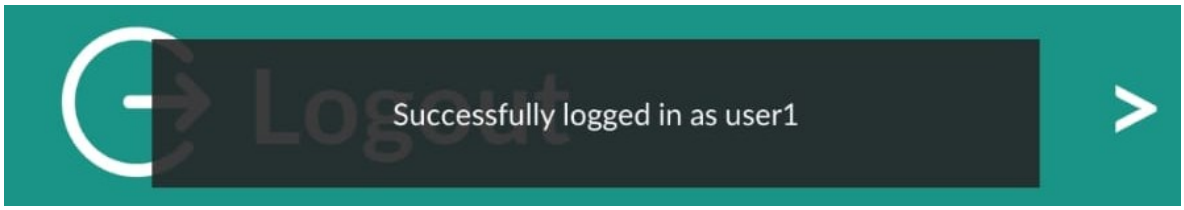


Figure 5.30: Example for the Toast message which states that the user “user1” logged in successfully

5.4.6 Payment Process

The purpose of this application is to enable the user to purchase items. Thus this process had to be made as easy and self-explaining as possible. It mainly consists of three steps:

1. display the selected elements and prices
2. let the user choose the payment account
3. verify the payment account with a code (either pin or pattern)

Display Selection
Displaying the selection especially is important for legal rights as the user needs to know what exactly he purchases and how much the price is (both individually and the total costs). A simple window showing every information the user needs to know was implemented. Figure 5.31 shows an example of this screen. Because a verification from the user is mandatory, the user must actively press a button to confirm the selection.

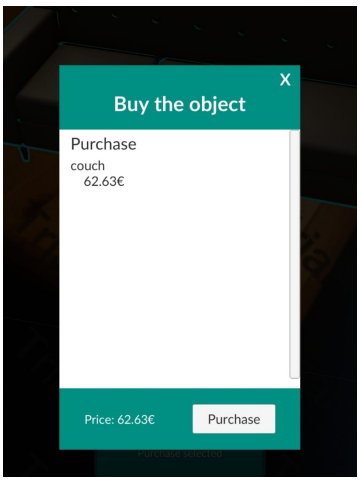


Figure 5.31: Example of the display that shows the different selections

Choose Payment Account
The user has to choose the payment account with which he wants to complete the transaction (Figure 5.32). This is allowed by a window which shows him his saved accounts. When one account is selected and confirmed it is stored as an attribute so it can get used in the payment process.

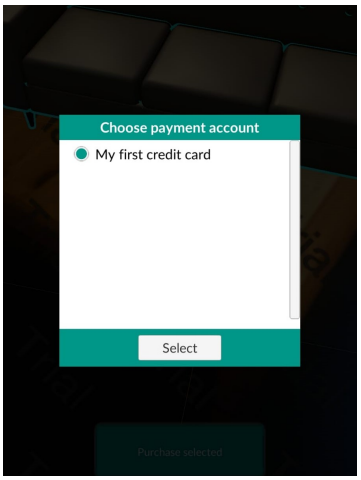


Figure 5.32: Example of selecting a payment account via the AR-demonstration. The first account is selected automatically

Verify Password

Of course when trying to purchase an item the user needs to verify his identity especially when an auto-login is enabled (e.g. via *Android*'s *Account-Manager*) as shown in Figure 5.33 and Figure 5.34. Thus there is another security level in the application namely a pin or pattern which is set on the website. This verification code can be set individually for every payment account.

The next step is to check if the given verification code was correct. If so, the transaction will be made, if not, the process aborts. The application notifies the user with a message about success or fail of the transaction. He will also be messaged when the transaction fails or the password was wrong. This is implemented via a status code that the service sends when finishing the transaction.

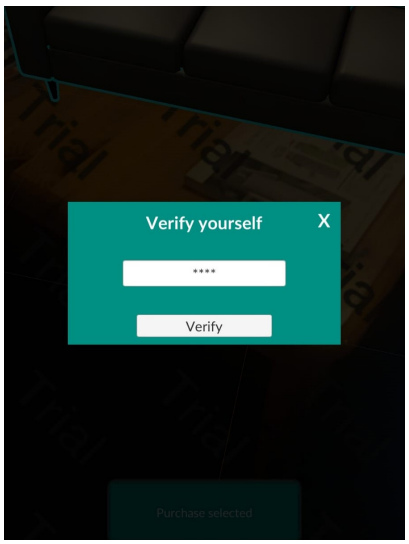


Figure 5.33: Verification screen with a pin as the verification method

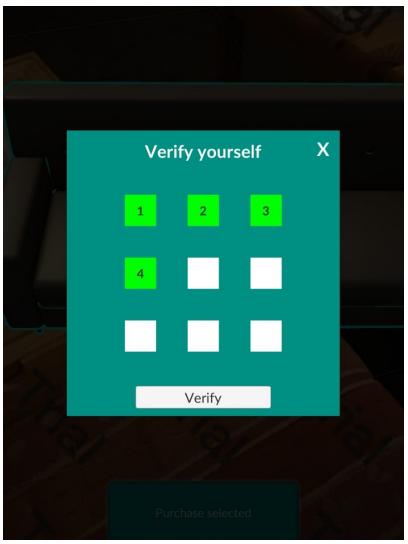


Figure 5.34: Verification screen with a pattern as the verification method

Client Library (CL)
All of the communication with the service is handled through the client library. To be fully functioning, the CL needs to be implemented via five steps:

1. Copy the CL to the project
   To realise the possibility of communicating with the service the CL needs to be implemented. Thus the library-files (*VIPER_client.dll* for *Windows* and *MacOS* and *libVIPER_client.so* for *Android* and *Linux*) are added to the "Plugins"-folder in the assets. The initialization is done automatically by *Unity*.

2. Login to the account
   Since the UX needs to be as fluid as possible the login is done at the very beginning (Figure 5.14). The CL's login-method `bool login_sync(string api_key, string identification, string password)` is called (the API-key can be seen in *StartMenu.cs* under the attribute `private readonly string api_key`).

   The parameters *identification* and *password* are filled with the credentials "user1" (username) and "1234" (password) to provide the possibility of skipping the login when clicking the *Skip login and try demo-version*-button (see the login-section). When logging in on *Android* an *Android* Account gets added automatically.

3. Select the payment account
   The next step is to get all payment accounts to give the user the possibility to select one. Since the CL stores the login-credentials (more specifically explained in section 6.13) there is no need in providing them when getting the payment accounts or making the payment. When calling the function `void get_payment_accounts(GetPaymentAccountsCallbackDel callback)` the data gets returned to a callback (`void GetPaymentAccountsCallback(string paymentAccounts, int size)`) via which all the payment accounts are displayed (as seen in Figure 5.32).

4. Make the payment
   At last, the user needs to verify his claimed identity and make the payment. This happens via the CL's method (Listing 5.11).

```
1 void make_payment(Order[] orders, int orders_length,
      string currency, string additional_info, string
      payment_account, string authentication,
      MakePaymentCallbackDel callback);
```

Listing 5.11: The payment method with its parameters

   This method consists of the items to buy (and the amount of items), the currency for the transaction, some additional info that will be displayed at the payment's receiver, the used payment account (as selected in Figure 5.32), the password the user set for this specific account and the callback-method which handles the response.

5. Handle the response
   As stated in 4., the response is handled in a callback-method (Listing 5.12).

```
1 public static void MakePaymentCallback(bool success) {
2   string message;
3   if (success) message = "Finished the transaction";
4   else message = "Could not finish the transaction";
5   PurchaseItemController.PaymentMessage = message;
6 }
```

Listing 5.12: Callback method after finishing the payment process

The callback has a success-value as a parameter which is set corresponding to the status of the payment. If it succeeds, the user will see a success-message on his screen, otherwise the message will be an error.

5.5 Testing

To make the system as stable as possible, it was very important to test the application on a regular basis. Testing an AR-system can basically be divided into manual testing and running automated system tests. Since those automated tests will be performed with fewer errors the biggest problem of AR is that it can not be tested fully automatically yet. Additionally, some functionalities were only implemented for mobile devices. Since such a device can not be simulated, its functions had to be tested manually. Both manual testing and automated testing had to be mixed to fully test the application.

AR can be used in any environment but it first needs to get initialized and find a plane it can put the virtual elements on. Since this cannot be automated with a simple algorithm, the automated tests will only include functional testing like executing methods and checking the results. Thus only unit testing and no end-to-end-testing are included in the automated tests.

5.5.1 Manual Testing

To make manual testing as widely accepted and reduce the number of errors (e.g. forgetting to test a feature) as much as possible, a testing protocol was introduced (Table 5.2) which defines the targeted output. Thus the actual and defined output can be compared very easily.

	Input	**Expected Output**	**Actual Output**	
1	Start the application	See the login screen	Login screen visible	Y
2	Click the "Skip login and try demo-version"-button	Start menu + welcome message	Start menu seen and short information about active user	Y
3	Click "logout"-button	Return to login screen	Login screen visible	Y

Table 5.2: Manual testing protocol - a full version of the protocol can be seen in the appendix (Table 9.1)

* Insert multiple objects before input
** Insert object and click on it before input
*** Repeat steps 19-21 before input
**** Also do steps 19-22 + 23 afterwards
“Y” ... test succeeds
“N” ... test fails

5.5.2 Automated Testing

For automated testing, there are two technologies that need to be considered: *Microsoft*’s VS and *Unity*’s Test Runner, which is based on the *NUnit*-framework[165]. Automated testing was implemented with *Unity*’s built-in test runner.[165] This allows to simply test all the functions defined in the different files without much more work to do. Also, *Unity* implemented the possibility to test one code on multiple platforms like *Android* or *iOS* and helps to standardize the automated tests even more.

A test generally looks like this:

```
1 [Test]
2 public void TestMakePaymentCallbackSetsMessageCorrectly() {
3   Payment.MakePaymentCallback(true);
4   Assert.AreEqual("Successfully finished the transaction",
5     PurchaseItemController.PaymentMessage);
6   Payment.MakePaymentCallback(false);
7   Assert.AreEqual("Error! Could not finish transaction",
8     PurchaseItemController.PaymentMessage);
9 }
```

Listing 5.13: Example of a test in *Unity*’s Test Runner

Tests are labelled with the `NUnit.Framework.Test`-class and executed in alphabetic order. Assertions are taken via *NUnit*’s `NUnit.Framework.Assert`-class which consists of static methods including checks like equality or whether an element is set to true or false.

Also, *NUnit* includes the classes `NUnit.Framework.OneTimeSetUp` and `NUnit.Framework.SetUp` which are called before all tests and before each test respectively. Thus setting up the testing environment with setting default values can be done like this:

```
1 [OneTimeSetUp]
2 public void SetUp() {
3   // Setup code
4 }
```

Listing 5.14: Example of a setup-method which is called at the very beginning

Chapter 6

Back End and System Design

This chapter elaborates the back end design and shows how the VIPER service was implemented. Choices, concerns and implementations for integration, deployment and payment related services are explained.

6.1 System Architecture

6.1.1 Architecture Patterns

In this section different architecture patterns are compared and the differences elaborated.

6.1.1.1 Monolithic Architecture

"A monolithic architecture is one in which a software application is designed to work as a single, self-contained unit. Applications that have this type of architecture are common. The components within a monolithic architecture are interconnected and interdependent, resulting in tightly coupled code." [74]

An example of a monolithic application can be seen in Figure 6.1. The application consists of a data access layer, which communicates with the different databases and is used by other parts to save, edit, read or delete data. The business logic is implemented in multiple service modules. On top of these services, a interface layer offers a website for different device to interact with the application. The monolithic design is software-wise very similar to the standalone solution presented in the previous section and thus the reuse of the existing design is possible. Since the parts in monolithic applications are interconnected, the software can be designed in a way that most effectively ties the application together, resulting in a performance increase, as mentioned by Ingeno [74]. Furthermore, deployment is easy and only a single application needs to be deployed and managed. Horizontal scaling can be achieved by deploying copies of the application and using a load balancing proxy to ensure high availability.

On the other hand, monolithic architectures are difficult to maintain when building large applications as Ingeno already noted in his book [74]. Since the code is tightly coupled and interconnected, changing it may result in unforeseen behaviour in other parts of the software. This makes tracing and fixing bugs difficult and disencourages making changes

or updating the software to include new features. Also resources may be used ineffectively, for the different components cannot be differentiated and thus the whole application needs to be handled as a whole. Whereas this may be advantageous for deployment management purposes, but when it comes to testing an application or CI this means all tests and steps have to be executed and not only those that have changed.

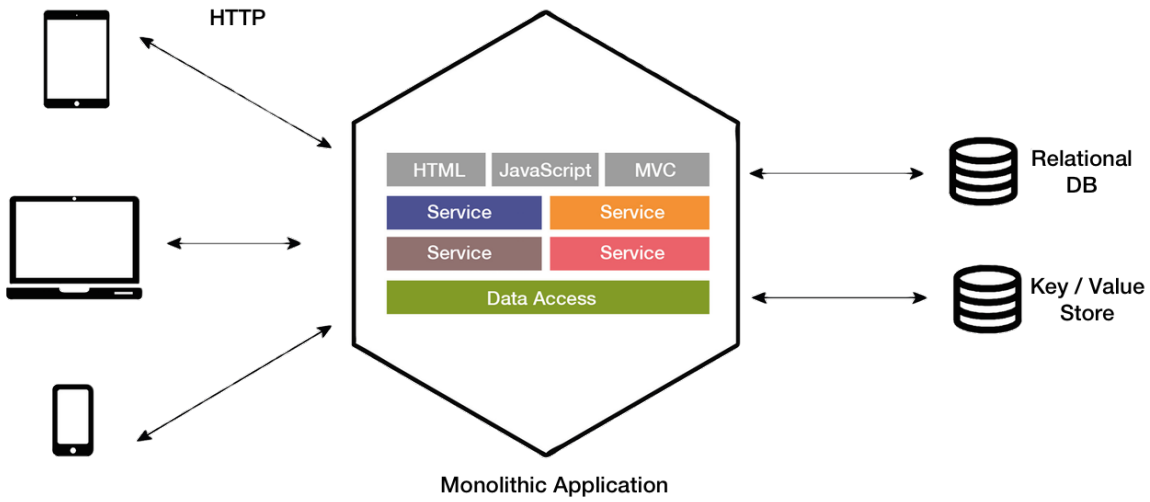


Figure 6.1: Example of a monolithic application [42].

Furthermore, it is difficult to use different technologies or change them later on because of the tight coupling. In addition, the large size of the codebase increases the difficulty to understand the software, which is especially disadvantageous for collaboration and new team members who need to learn about the existing code. The workflow may suffer from the codebase size, which requires the development environment to load many, mostly unnecessary, components and thus slows down the performance of the system and the developer. Also splitting up task and focusing only on certain aspects is difficult, since team members have to know how the other interconnected parts work. This requires large teams and management to ensure everyone has all the necessary informations, which also need to be documented extensively.

6.1.1.2 Microservice Architecture

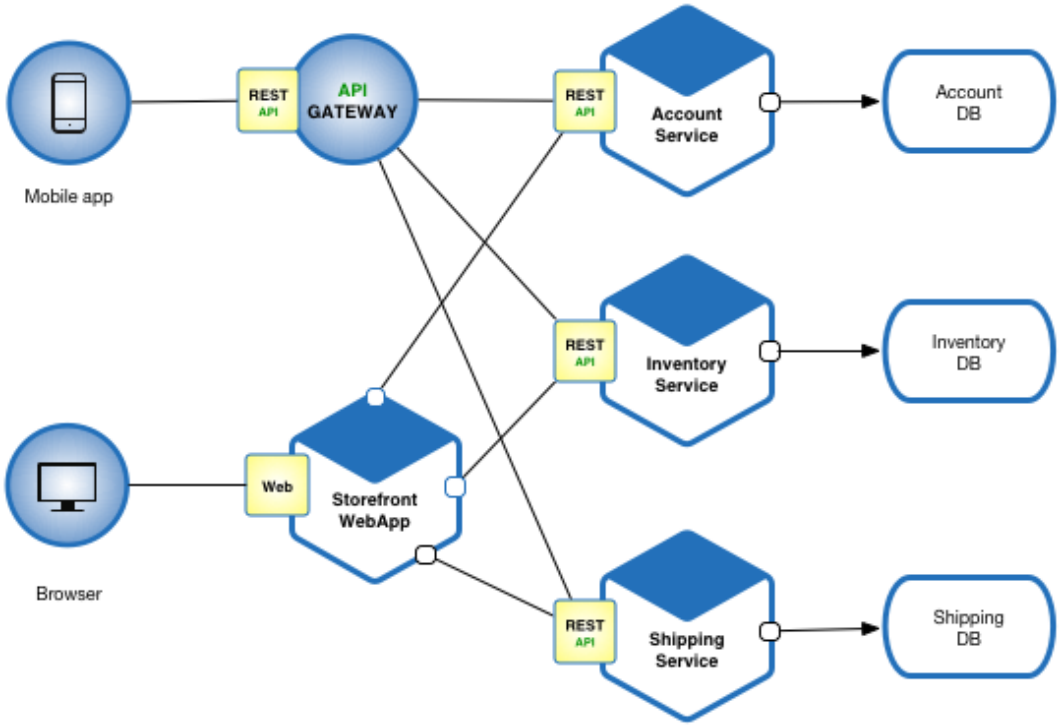


Figure 6.2: Example of a mirco-service architecture [97].

The microservice architecture is a strong contrast to the monolithic approach and consists of "small, autonomous services that work together" [111]. A service in this context is a piece of software that implements some logic and offers an interface accessible to other services. The microservice system show in Figure 6.2 consist of 5 services: A gateway that routes ReST request from outside the system to the responsible services inside, a WebApp service which offers a website to interact with the system and a account,inventory and shipping service which handle account information, inventory details and shipping data respectively and store it in appropriate databases. In his book about microservices architecture, Newman lists the following principles for microservices [111]:

Small, and Focused on Doing One Thing Well
This principle is a direct response to the size and complexity problems of monolithic applications. Microservices group together business logic into a coherent piece of software that is responsible for all matters inside this domain. This yields the advantage that bugs can be easily traced back to one specific service and can be fixed there. A service should be small, but not too small. To choose the right size for a service is one of the great design challenges associated with this architecture. There are many approaches to this problem; Newman argues, that most developers can tell when an application is too big and thus should reduce the size of the service until he does not consider it too big anymore. Another rule-of-thumb is to target a two week development time for a service and align the size to this time constraint.

Autonomous
A microservice should work independent from all other services. It should be deployed as a single application and may even run on its own OS or server. The communication between services is implemented via network calls to avoid tight coupling and increase the versatility of the architecture. Any changes to a service should be independent of any other service. This autonomy is a great advantage for development purposes, since the services can be developed in complete isolation and thus split up between programming teams. This reduces the required size of a team and thus the organizational problems associated with large teams. This also means that the services can be scaled as needed and only required parts may be deployed multiple times. Different technologies can also be used easily, since the only requirement for a service is a network interface, which is available in most technology stacks or a wrapper can be build in another technology. This furthers the ability to use the right tools for a specific tasks and most efficiently and effectively solve a problem. Also new technologies can be adopted quickly, which is more secure and enables the system to be up-to-date with current developments. This principle also makes the system resilient to failures, since all services have to work independently and thus will not fail all at once like a monolithic system would. Teams can also work without any knowledge about how the other services work, all the team need to know is the API definition of each service they require.

The single dependency on the ReST API is a big challenge, since once the definition of the API is used, it is very hard to make changes to it. Many other services depend on the definition and changing it requires other services to adopt it, which contradicts the autonomous principle. A solution for this problem is to version the API and only make other services adapt the new definition when needed, without interrupting the existing system. The versioning of an API is an overhead for the architecture, since the versions either have to run simultaneous in a service, which contradicts the first principle, or have to be deployed on multiple instances, which increases the management and routing complexity and cost of the system.

6.1.2 System Design

6.1.2.1 Requirements

The ultimate goal of this project is to enable payment inside XR applications. For the reasons stated in chapter 2 it was decided to use a cloud-based architecture. The cloud service needs to be available at all times for transactions to be executable and thus requires high-availability. To cover all possible devices and applications, the service has to offer a common and widely used interface format. Since this project is planned to continue in the future to include more payment options and possibly Business Intelligence (BI) interfaces among other ideas, an extension and change friendly design is desired.

6.1.2.2 Design Overview

It was decided to use a microservice architecture, because of its flexibility and extension friendly design. Furthermore the cloud-service can be easily split up into multiple, independent microservice.

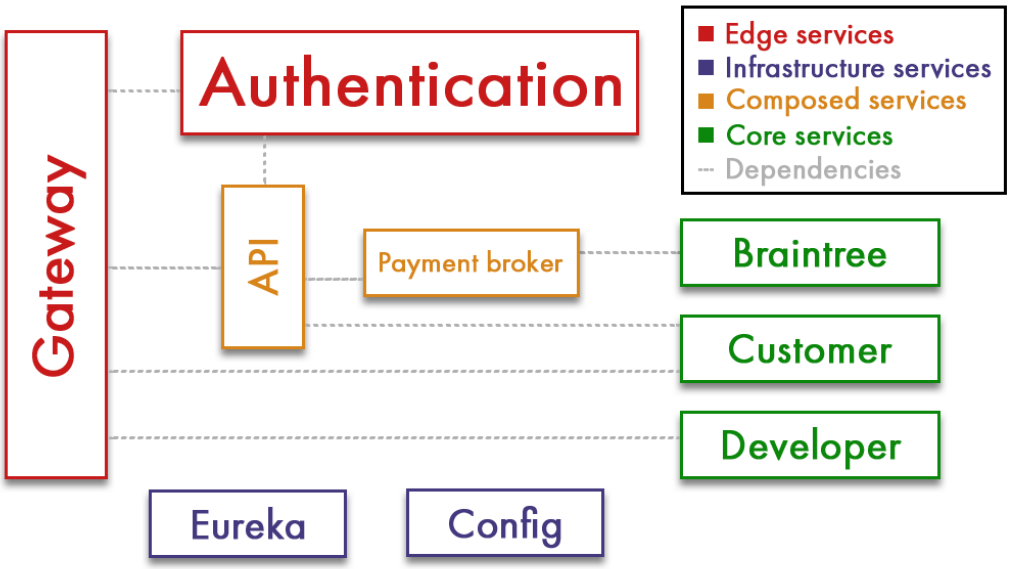


Figure 6.3: Overview of the Viper microservices.

Figure 6.3 shows the microservice design of VIPER, which consists of multiple services and a gateway. The blue-coloured services are called infrastructure services and provide basic functionality required by all other services. The most important service of this kind is the *registry* service (called Eureka in the diagram). Every service registers its name and ip address there and can query all other services registered. The *config* service is used to distribute configuration files for each service, without having to update them manually. More details about these service can be found in subsection 6.6.2 and subsection 6.6.1 respectively.

The red-coloured services are called Edge services and act as a gateway for all other services. The Gateway service receives all incoming requests and forwards them accordingly. The Authentication service is used for all authentication and security purposes. More details can be found in subsection 6.5.2 and subsection 6.5.3 respectively.

Orange-coloured services are called composed services, since they do implement little functionality and mostly forward request. The act as a layer of abstraction for other micro-services. The API service is responsible for forwarding all API related request sent by an application. The payment-broker service handles all payment related request. Details can be found in subsection 6.5.6 and subsection 6.7.6.

The green-coloured services are arguably the most important services, since they implement the actual business logic and are called core services. The organization and customer service are responsible for mainly website request and handle all organization and customer related data. The braintree service is a service used to execute transaction and store payment accounts. For details see subsection 6.8.3, subsection 6.8.2 and subsection 6.7.7 respectively.

6.2 Implementation Technologies and Frameworks

6.2.1 Java and Spring

Using Java The microservice based back end infrastructure requires the use of a technology that is capable of creating microservice, in other words small web-servers. Because all team members are familiar with the programming language Java, which offers huge support for enterprise cloud services, it was decided to use a Java-based technology for implementation. This eliminates the time required for learning a new language and gaining experience with it.

Using Spring There are two major enterprise cloud platforms for Java, namely Java Enterprise Edition (JEE) and the Spring Framework. Again, most team members had experience working the Spring Framework and thus it is preferred. The Spring Framework supports many useful components for building microservice systems and was therefor chosen for the VIPER back end.

6.2.2 Spring Cloud and Netflix OSS

6.2.2.1 Introduction

"Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems [...]. Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. [147]"

As Java and Spring were chosen for the VIPER back end, Spring Cloud, which has a lot of components helping with the development of microservice architectures, was used for many parts of the system. Many Spring Cloud components are either Netflix Open Source Software (Netflix OSS) components or were built based upon them. The media-service provider Netflix was one of the first companies to build large-scale microservice infrastructures and has therefore created many tools for doing so. These tools are published as Netflix OSS and provide different services and communication frameworks. In the following sections all Spring Cloud and Netflix OSS components used in the VIPER microservice system are described in more detail.

6.2.2.2 Usage of Deprecated Spring Cloud Components

Some of the components used in this project are already deprecated because a new version of Spring Cloud was released during the project time. When the planing phase of the project started, Spring Cloud Finchley.SR1 was the current Spring Cloud version. Relatively early in the project a new version with minor changes - Spring Cloud Finchley.SR2 - was released. As there were no breaking changes for the already developed services everything was update to use the new version. At the end of the project another new version of Spring Cloud - Greenwich.RELEASE - was released, which introduced some major changes. The release notes included a section about multiple Netflix OSS components used in this project entering maintenance mode [61]. This section also included suggested replacement for these deprecated components. The two main components entering maintenance mode where Ribbon and Zuul, with Spring Cloud Load Balancer and Spring Cloud Gateway being the suggested replacements. Especially the switch from Zuul to Spring Cloud Gateway would have brought some significant improvements for the back end system. With the biggest change being that Spring Cloud Gateway uses non-blocking APIs and supports long lived connections like Websockets [144]. The reason these new components were not integrated into the project is the late release date, as Spring Cloud Greenwich.RELEASE was released just a few weeks before the end of the project.

6.2.2.3 Spring Cloud and Netflix OSS Components Used in the Project

Config Spring Cloud Config includes configuration service and client components. The configuration service can read configuration files from a centralized location and distribute them to all clients. [148] These clients are all other services in the microservice system. This allows for dynamic changes to the configuration properties of all services at runtime and eliminates redundant configurations. The configuration for a Eureka instance, for example, has to be created only once and can then be distributed to all service.

Eureka Spring Cloud Netflix Eureka is a service discovery framework with two main components: Eureka Server and Eureka Client. The Eureka Server allows clients to register and periodically broadcasts a list to all registered clients, containing all other clients [137]. This allows the clients to make request to other services without having to know their IP addresses. This is very important as the IP address can change when a service is redeployed. The Eureka Client component is used by every microservice (except the config service) and registers to the Eureka Server on startup, allowing all services to discover each other and work together [136].

Zuul Spring Cloud Netflix Zuul is the API gateway and router of the microservice system. It is the only part of the system that is exposed to the outside world over the Internet. Zuul gets Hypertext Transfer Protocol (HTTP) requests and forwards them to the appropriate service according to its configuration. The use of Eureka allows Zuul to discovery these services automatically. In addition to the routing capabilities, Zuul allows for the creation of filters, which modify or block in- and outgoing traffic. [131] In the VIPER microservice system this is used for authentication.

Ribbon Spring Cloud Netflix Ribbon is a client-side load balancer. It is used for requests between services and decides which service a request should go to if multiple instances of a service are available. To get a list of all available services Ribbon uses Eureka. Apart from using Eureka's list of all service, ribbon periodically checks the availability of other services and changes requests to these services according to the configured strategy. [29] In this project Ribbon is never used directly, but rather just as a result of using Feign.

Feign Spring Cloud OpenFeign, which will be referred to as just Feign throughout this work, is the HTTP client used for requests between services. Feign uses Ribbon to decide on the target of a request. [38] Its main strength is the convenience and simplicity of making HTTP requests, as only a few annotations are need to set this up. Additionally, all objects are serialized and deserialized to and from JavaScript Object Notation (JSON) automatically.

6.3 Data Persistence

6.3.1 Introduction

With the use of a microservice architecture data has to be stored distributed to multiple services. When choosing one or multiple database management systems (DBMSes) for the system, many aspects such as read and write performance, resilience and security have to be considered. The microservice architecture allows for the use of different solutions for different services. This enables every service to use a DBMS and underlying database architecture optimized for its specific needs. The initial plan was to develop database solutions custom to each microservice. But when comparing the possible options it became clear that the benefits of such an approach would be negligible compared to the increase in complexity of the entire system. Thus, it was decided to develop a single database solution that could be replicated and used in any service (see subsection 6.3.3).

The following section lists all microservices that need to persist data and the properties of this data.

Authentication Service

Stored Data

- All information that is necessary for customers and developers to authenticate and log in
- API authentication information of all registered applications

Read/Write Since log-in and payments occur much more often than changes to the authentication information or the creation of new users, the database load of the authentication service will mainly consist of read operations. Furthermore, many of the write operations are not time critical and can be executed asynchronously by the underlying service.

Braintree Service

Stored Data

- The Braintree API access information of the Braintree account on which an organization receives payments
- Payment account information of customers that are registered at Braintree

Read/Write Read operations will occur with every executed payment. Write operations only occur when the payment information of an organization or customer is changed. Write operations are again not time critical.

Customer Service

Stored Data

- All personal customer data
- Available payment methods of customers and their authentication method

Read/Write Personal information and the payment methods are not changed frequently. The main operations are reads when requesting customer information or payment methods. Write operations are not time critical.

Developer Service

Stored Data

- All information of an organization
- Personal information of developers

Read/Write As with the other services described above, the load consist mainly of read operations. These occur when requesting organization, application and developer information. Writes only occur when organizations, applications or developers are created or edited. Compared to the other services the total database load of the developer service is low because it is not involved in the execution of payments.

Payment Broker Service

Stored Data

- History of all successfully executed transactions

Read/Write The majority of operations are writes because for every successfully executed transaction all information that defines the transaction is stored in the database. Read operations only occur when a user requests their the transaction history.

6.3.2 Database Architecture

For the implemented microservice architecture several database architectures are possible.

One Single Central Database The use of one single database (as shown in Figure 6.4) which holds all of the data needed in the back end system is the simplest approach. All microservices would have access to all data stored in a single database. This approach cuts down on the work needed to secure the data as there is only a single place in which it has to be secured but this also leads to some inflexibilities. If there is even a single piece of information that needs to be stored in an encrypted database, depending on the DBMS used, all data might have to be encrypted. This could of course lead to decreases in performance. Another such limitation becomes clear when thinking about the actual data stored in the

database. Some of the data might be best represented in a graph database, some of it might be too unstructured to be stored in an relational database. There can be a lot of variety in the data and the use of a single database would make it impossible to optimize the way data is stored. In the current state of the project, in which the system is just a prototype and the functionality is relatively limited, this might not be a problem, but thinking into the future a single database is definitely too limiting for the system. Thus, even if this solution would be easy and fast to implement, it is not the right choice. Additionally, it would violate one of the basic ideas of a microservice architecture - the complete separation and loose coupling of individual services. With a single database a change to one service could lead to changes in the database and if all services use the same database, some or even all of them would have to be changed.

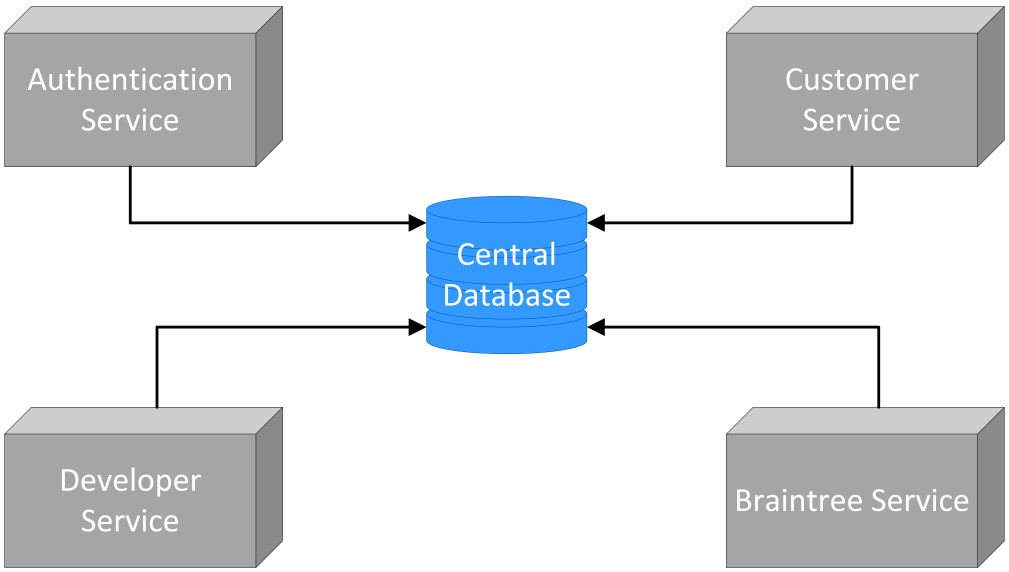


Figure 6.4: One single database that is used and shared by all microservices

One Single Central Persistence Service Instead of having a central database that is directly accessed by all microservices, this database could be wrapped in a microservice of its own (as shown in Figure 6.5). This would eliminate some of the problems of a single database whilst still being fairly simple to implement. Having an additional layer of abstraction for accessing the database makes it easier to perform changes to the database without the need to also change all microservices. This would also bring back some loose coupling and encapsulation of similar behavior that is important in microservice systems. But ultimately this approach still does not change the fact that the data might be too diverse to be efficiently stored within the same DBMS.

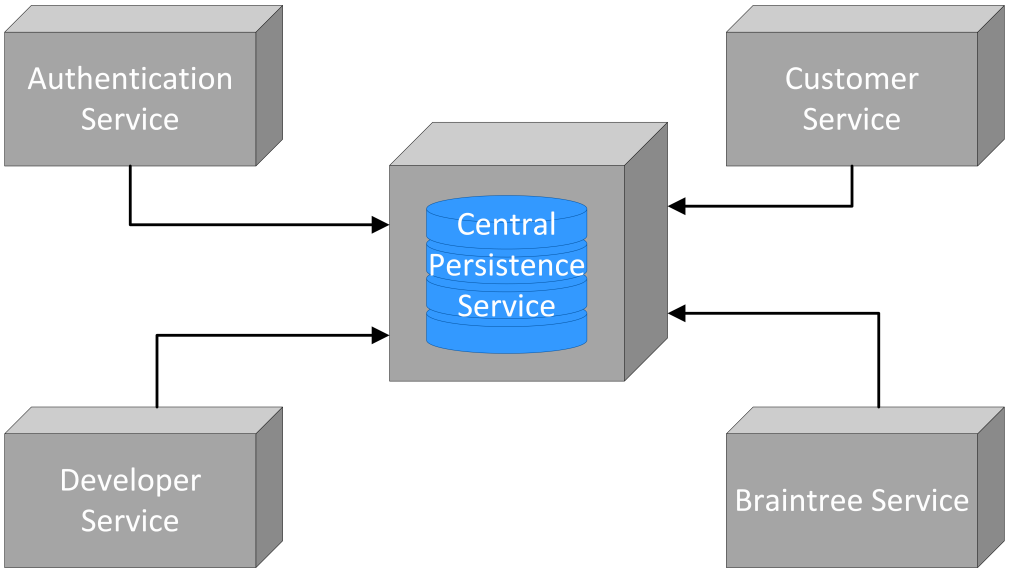


Figure 6.5: one single persistence service that is used and shared by all microservices

One Database per Service When looking at the basic principles of microservices the approach shown in Figure 6.6 is very sensible, which is the reason it was chosen for the VIPER back end. Every microservice has its own database, which can only be accessed by this service. This gives a lot of freedom in development. Different microservices can use different DBMSes depending on the data they stored. Every service has by design only access to the data it should have access to without the need of any sort of access control. This approach makes it very easy to make changes to the database, as there is only one service accessing it. Depending on how this architecture is implemented it can also lead to significant improvements of security. With the approach that was chosen, every service that needs a database has an entry for it in the docker-compose file. With this configuration the database is deployed automatically on the same machine as the microservice. This eliminates the risk of exposing data when it is sent between the service and the database as everything happens on the same machine.

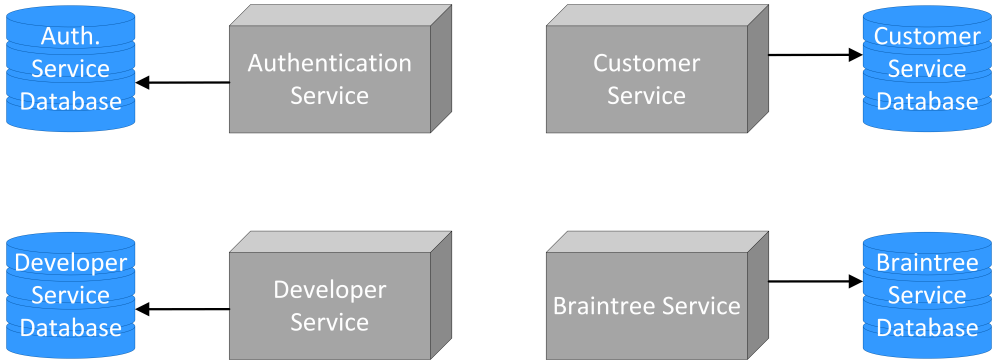


Figure 6.6: every service has its own individual database which can only be accessed by this service

But having all the data distributed over multiple services also comes with some downsides. For many requests, the service that gets the request needs some piece of information from another service. In many cases this is not a problem, as that is what microservices are made

for. Every service only does what it is meant to do and delegates everything else to other services. But sometimes it is just a lot faster to have duplicates of data stored at different services, to lower the number of calls needed deal with a request. This ensures compliance with the principle that microservices should not be chatty if it can be avoided [111, p. 65]. One such example of data being duplicated was much of the data that was originally just meant to be stored in the authentication service. Data like the email addresses and usernames of users is authentication information, which is the reason it was initially just stored within the authentication service. But moving forward it became clear that these pieces of information were needed for many requests to the customer and developer services and that it was very inefficient to always request them from the authentication service. Thus, changes were made to also store information directly at the customer and developer services.

Multiple Databases per Service Expanding on the ideas above, having a single database per microservice might still be limiting in some ways. The data within a single service might be too diverse to be stored in the same DBMS in which case it might be beneficial to have multiple databases for a single microservice. This would also allow for the possibility of having a cache, which could be realized with a key-value store like Redis, which works in-memory and is therefore a lot faster, as this paper comparing main-memory and disk-resident databases shows [2]. Even tough this would be a great addition to the database architecture the time constraints of the project did not allow for it. But using an in-memory cache should definitely be considered as a future improvement. Having data that is too diverse to be stored in the same DBMS was not a problem in this project, but to remain future-proof the solution that was chosen - having one database per service - can be easily expanded to an architecture with multiple databases.

6.3.3 Databases Selection

Initially the idea was to use different DBMSes for different services. But as the data of all services was relatively similar, performance gains would have been hardly noticeable. Thus, it was decided to use the same DBMS for all services.

SQL or NoSQL As a single DBMS had to be select for all services it was clear that SQL databases would be too inflexible to store all the different types of data. The characteristics of NoSQL databases described in this 2012 paper [36] clearly make them better suited for a microservice architecture than SQL databases. NoSQL databases are more easily scalable, which is an important aspect and one of the main benefits of microservices. These databases also fit the style of development better as an agile project management method was used which requires the ability to be flexible and be able to make changes to the existing system. NoSQL system are are superior in this aspect because different from SQL database they have no fixed schema or data format, making changes very easy.

The Type of NoSQL Database As described in this paper [36] there are multiple types of NoSQL databases - key-value databases, document stores, column databases and graph databases. For the data stored in the VIPER back end a document store is the most suitable database type as its data model fits the data that needs to be stored very well. Key-value databases would be too limited in their capability to store unstructured data and references between different data points. The use of column databases would be possible but their

data model would not allow for optimization of most of the read operations that would be performed. A similar problem rules out the use of graph database as the strengths they have working with data that has many relationships would not be relevant with the data that needs to be stored, especially since many of the relationships between different data points span across service boundaries.

Selection of a NoSQL Document Store One important limitation for the selection of a DBMS was its support by Spring Data as it was important for easy and fast development to have a solution for the communication with the database that would integrated well with the rest of the service code. This limited the possibilities to MongoDB, Couchbase and ArangoDB [149]. Looking at the functionality of MongoDB [102], Couchbase [32] and ArangoDB [14], there is no clear benefit to any of them for what was needed at the back end, thus letting the selection of a DBMS come down to previous experience of the team. Therefore MongoDB was selected as the DBMS of the microservices.

6.3.4 Security

Much of the security the system provides comes as a result of the chosen architecture and used principles. For example, the database is deployed on the same machine as the microservice that is using it and it does not (and cannot) get requests from anywhere outside the machine. An Exoscale firewall was used was used for this purpose (for further details see subsubsection 6.5.7.1)

6.3.4.1 MongoDB Security

The security provided by the underlying architecture should be enough to ensure that all data is secure during transit between the database and the service. Nevertheless, some additional measures were taken to make the MongoDB database secure for possible future changes. The docker-compose configuration shown in Listing 6.1 is used to set the username and password with which the database can be accesses and passes a custom configuration file in which the authorization is enabled, so that username and password have to be provided to access the database (see Listing 6.2).

```
1 version: '2.4'
2 services:
3   mongo:
4     image: mongo:4.0.3
5     environment:
6       MONGO_INITDB_ROOT_USERNAME: some_user
7       MONGO_INITDB_ROOT_PASSWORD: some_password
8     command:
9       - '-f'
10      - '/etc/mongod.conf'
11    volumes:
12      - '/home/ubuntu/mongod.conf:/etc/mongod.conf'
```

Listing 6.1: MongoDB docker-compose configuration

```
security:
  authorization: "enabled"
```

Listing 6.2: MongoDB configuration to enable authorization

6.3.4.2 Frugal Data Storage

A core principle used in this project to make development easier and the system more secure was to store as little data as possible. This also limits legal responsibility as the government or anyone else cannot ask for information that is not stored. For this reason only the information that is entered by the user and that is absolutely necessary for the system to function and some historic data of payments but no activity logs or anything that is not actually needed by the system, is stored.
The most important application of this principle was to not store any information of the payment accounts, like credit card numbers, which would have had to be secured very thoroughly and would have been a target for security attacks. Using Braintree (for further details see subsection 6.7.7) all sensitive information lies at the Braintree servers. Only a reference to this information is stored and can easily be revoked.

6.3.4.3 Future Security Improvements

As the project had time constraints and the finished product is just a prototype there are still some security improvements necessary for a production ready system.

Encryption at Rest To ensure that the data is secure even when an attacker gets access to the file system of the machine the database is running on, all data should be encrypted within the database. As this is a feature only available in the non-free MongoDB Enterprise Version [51], setting up such an encryption was not within the scope of this project but is still important for a production ready system.

Encryption in Transit With the current architecture, where the database and the microservice are on the same machine, encrypting the data in transit is not relevant and would just be an overhead. But if this changes in the future the communication between the database and the service needs to be secured. This can be achieved by having an X.509 certificate as .pem file for the MongoDB database and adding the options in Listing 6.3 to the MongoDB configuration file.

```
net:
  ssl:
    mode: requireSSL
    PEMKeyFile: /path/to/file.pem
```

Listing 6.3: MongoDB configuraiton needed to enable SSL for communication

System Activity Audit To track access and changes to the database for detection of an attack or intrusion the MongoDB auditing system should be used for a production system [103].

6.3.5 Database Access

6.3.5.1 Library Used for Database Access

As all the microservices are build using Java and Spring, *Spring Data* is used for the communication with the service database. For integration with MongoDB Spring Data has the module *Spring Data MongoDB* which can be added to a service as a Maven dependency with the code from Listing 6.4.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

Listing 6.4: Spring Data MongoDB Maven dependency

6.3.5.2 Creating the Documents

The structure of the data as it is stored in the database is defined by the classes whose objects are used to represent and work with the data within the code of a service. These classes need to be annotated with *@Document*.

There are two types of document classes: ones that are actually stored as their own collection (collections are basically the tables of MongoDB) within the MongoDB database and ones that are just embedded into other document classes. When deciding which classes should be collections and which ones should just be embedded, it is important to consider which read operations will be most commonly performed. If a lot of data is just embedded within another class all of that data is read and returned to the application with every read operation. If there is therefore a piece of data that is for example on average only needed in every tenth read request it would be wise to consider storing it in its own collection and just referencing it. But there are also performance considerations for references, as Spring Data MongoDB provides the possibility to load referenced data either lazily or eagerly. Here the same considerations as mentioned before should be made. If a piece of referenced data is needed in every or most read requests it should be loaded eagerly but if it is only occasionally needed it should be loaded lazily. Examples of how to reference other classes and load the data either eagerly or lazily can be found in Listing 6.5.

```
@DBRef(lazy = true)
private Object lazilyLoadedObject;

@DBRef(lazy = false)
private Object eagerlyLoadedObject;
```

Listing 6.5: Lazy and eager loading of referenced objects

To be able to uniquely identify any object stored in the database every class that is stored as a database collection should have a unique ID. Spring Data MongoDB has the class *org.bson.types.ObjectId* for this purpose. The field in which this ID is stored has to be annotated with *@Id* which is just short for using the annotations *@NotNull* and *@Indexed(unique = true)* (see Listing 6.6).

```
1 @Id
2 private ObjectId _id;
```

Listing 6.6: MongoDB ObjectId field

Apart from the ID other indexes can be added to a collection with the *@Indexed* annotation. To mark an unique field for indexing the annotation *@Indexed(unique = true)* has to be used (default is *unique = false*). To create a compound index with multiple fields for a single index the annotation *@CompoundIndex* has to be used. This is a class level annotation. The index is defined as shown in Listing 6.7. The argument *def* is set to all the fields that are used within the index have to be listed in the shown format.

```
1 @CompoundIndex(name = "index_title", unique =  true, def = "{'
    field1' :  1, 'field2': 1}")
2 public class ... {
3   ...
```

Listing 6.7: Definition of a compound index

All database collections of the individual services were designed using these principles. A full example of a document class implementing all of this can be found in the appendix under Listing 9.2

6.3.5.3 Database Repositories

To actually store data in the database and retrieve it again the Interface *MongoRepository* of Spring Data MongoDB is used [76]. For every document class an interface inheriting from *MongoRepository* that is annotated with *@Repository* has to be created (see Listing 6.8). As *MongoRepository* inherits from the Interface *CrudRepository* basic Create, Read, Update, Delete (CRUD) functionality (save new or updated object, delete object by id, find object by id) is already provided with an empty interface.

```
1 @Repository
2 public interface UserRepository extends MongoRepository<User,
    ObjectId> {
3 }
```

Listing 6.8: Definition of a repository interface

In some cases it was necessary to add additional methods to the interface to provide more advanced ways of finding or deleting objects. This is achieved by using a special syntax for

method names described in detail here [85] and here [173] [150]. Listing 6.9 shows a few examples of this method name syntax.

```
1 Optional<User> findByEmail(String email);
2
3 Optional<User> findByUsername(String username);
4
5 boolean existsByEmail(String email);
6
7 Optional<User> findByEmailOrUsername(String email, String
    username);
8
9 void deleteByUsername(String username);
```

Listing 6.9: Spring Data Repository methods

As return type of the find-methods the class *Optional* is used since these methods are not guaranteed to return an object. In case the object is not found an empty *Optional* is returned. The class *Optional* has the method *isPresent* to check if the *Optional* contains a value. This check should be performed before retrieving the actual object from the *Optional*.

To use these repositories they have to be added as an *autowired* field to the classes they are used in (see Listing 6.10)

```
1 @Autowired
2 private UserRepository userRepositroy;
```

Listing 6.10: Autowiring repositories

6.4 Service Communication

As the VIPER back end is built using a microservice architecture there are many requests requiring multiple services to work together to be fulfilled.

6.4.1 Synchronous or Asynchronous Communication

For any request made between services the communication can be either synchronous or asynchronous. With asynchronous communication a service makes a request to another service without waiting for a response just assuming that everything went well. In many cases this approach can be much faster than to wait for a response and therefore improve the end user experience. [111, p. 89] But it is important to note that whilst synchronous communication can almost always be used and will just be a little bit slower in some cases, asynchronous communication cannot. If it is important to know that an action completed successfully or if the service that makes the request needs some resource from the request target, synchronous communication has to be used. It will therefore not be possible to use just asynchronous communication within the microservice system.

Advantages of Asynchronous Communication Good use cases for asynchronous communication are operations in which resources are created changed or deleted but no data or information about the success of the operation has to be returned. Taking the example of a new customer account being created, many advantages of an asynchronous communication model become clear. When a new customer account is created, the initial request is sent to the Customer Web Service. It could then send out synchronous requests to all other services that need to know about this new customer and wait for them to send back a success response. But it could also just send out asynchronous messages without worrying about the success of other services. This would make the whole process a lot faster and the user experience better.

Event Based Communication A variation to this approach would be to not send the asynchronous message directly to the services, but to send out an event stating that a new customer has been created using a message queue. This would mean that the service getting the initial request does not need to know which other services have to be informed. Instead every service just listens for events and decides whether or not it has to do something because of an incoming event. This model of communication is called *choreography* in contrast *orchestration* - the model described before. With choreography the business logic of the microservice system is more evenly distributed across the services. Additionally, this approach greatly reduces coupling between services [111, pp. 89–92].

Conclusion Knowing all of that, there is still a good reason the approach of asynchronous event based communication was not used for the VIPER microservice system. Implementing such a system is very complex and there are a lot of aspects to be considered. E.g. how is data kept consistent across service boundaries without getting responses whether or not actions were successful. Additionally, the performance improvement would not have justified the effort needed to implement such a system, as most requests are processed very fast within the services receiving them, making the whole request fast enough even with synchronous communication. If there had been more time to implement a system with asynchronous

communication it might have been a smart decision to do so but with the time constraints of the project, using the much simpler synchronous communication was the right way to go.

6.4.2 Communication Technologies

This sections explains the communication technologies used for requests within the microservice system.

6.4.2.1 ReST communication

ReST Having decided on the use of synchronous communication between services, using ReST over HTTP was the obvious choice as these are the technology Spring and Netflix OSS use by default. As detailed in subsubsection 6.5.1.1 using a different technology like Simple Object Access Protocol (SOAP) would not have brought any significant advantages. But would have integrated worse with the used technology stack as it requires additional configuration to be used (which ReST does not).

JSON As a data format for the ReST requests it was decided to use JSON since this is the format that is used by Spring Web and Feign by default. Changing it would not result in any improvements, as JSON already is one of the simples formats with very little overhead. But it would have resulted in much higher development efforts, as additional configuration would have been required. That being said, the data format used in transit does not have any impact on the actual development as all data is automatically serialized and deserialized by Spring Web and Feign. Thus, using the default is a sensible approach.

6.4.2.2 Spring ReST Interfaces

The ReST interfaces of the services were implemented using *Spring Web* and their web-bind-annotations. Using these annotations allowed for fast and easy development of a big number of ReST endpoints. The classes containing ReST endpoints have to be annotated with *@RestController* and *@RequestMapping("...")*. The argument of *RequestMapping* is the base Uniform Resource Locator (URL) path of every ReST endpoint defined in that class (see Listing 6.11).

```
1 @RestController
2 @RequestMapping("payment")
3 public class PaymentController {
4   ...
```

Listing 6.11: Annotations of a ReST controller

The actual ReST endpoints are methods of a RestController class, which have to be annotated with the annotations shown in Listing 6.12, containing the URL path of that ReST endpoint.

```
@GetMapping("path/to/endpoint")
public Object getEndpoint(...) { ... }

@PostMapping("path/to/endpoint")
public Object postEndpoint(...) { ... }

@DeleteMapping("path/to/endpoint")
public Object deleteEndpoint(...) { ... }

@PatchMapping("path/to/endpoint")
public Object patchEndpoint(...) { ... }
```

Listing 6.12: ReST endpoint annotiations

The annotation determines the HTTP method of the endpoint (multiple endpoints can have the same URL path with different HTTP methods). For more information about HTTP methods see [72] and [167]. For a formal specification of the HTTP methods see [58, pp. 21–33] and [48].

6.4.2.3 ReST Requests With Feign

Adding Feign to a service All request between services were made using *Feign* which integrates very well with *Eureka* (see subsection 6.6.2) eliminating the need to know the addresses of the targeted services. Feign can be added to a service with the Maven dependency from Listing 6.13. For Feign to work properly the entry point class of the Spring application (which is already annotated with *@SpringBootApplication*) has to be annotated with *@EnableFeignClients*.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Listing 6.13: Feign Maven dependency

Creating a Feign Client To make a request to another service an interface annotated with *@FeignClient("...")* has to be created. The argument of *FeignClient* is the name (given with the configuration property *spring.application.name*) of the service to which the requests is made. The actual address of the service is found by Feign as it works together with the service registry Eureka. For every possible request to this service a method needs to be added to the interface. This method must have the same signature (apart from the method name and method arguments without annotations) of the ReST endpoint method. An example of a ReST endpoint and the corresponding Feign client method is shown in Listing 6.14 and a complete example of a Feign client interface can be found in the appendix under Listing 9.3.

```
1 //ReST endpoint
2 @PostMapping("user")
3 public Response register(HttpServletResponse response,
      @RequestBody CreateUserRequest request) { ... }
4
5 //Feign client method
6 @PostMapping("user")
7 feign.Response createUser(@RequestBody UserCredentials
      credentials);
```

Listing 6.14: ReST endpoint and the corresponding Feign client mehtod

It is important to note that the classes used as return type and as method arguments have to be equal in the endpoint method and the Feign client method. Equal in this case means that they must have the same fields, so that the object mapper can convert an object of that class to JSON and then back to a Java object in the receiving service. The only exception to this is the return type of the Feign client methods as this can always be *feign.Response*. The *feign.Response* class contains the returned object alongside some meta data like the HTTP status code. This functionality was used to communicate error types in some requests.

Using the Feign Client As the Feign client interface is a normal Spring bean, it can be used by adding an autowired field to the class in which it is needed (see Listing 6.15).

```
1 @Autowired
2 private FeignClient feignClient;
```

Listing 6.15: Autowiring a Feign client interface

Problem: PATCH Mapping Not Supported by Feign When creating Feign client methods for PATCH mappings, they will not work and just throw an exception saying *Invalid HTTP method*. To solve this, the HTTP client used by Feign has to be changed to one that does support the PATCH method. For this the Feign Apache HttpClient [56] can be used. It is automatically used if the Maven dependency is added to the service as shown in Listing 6.16. As both *feign-httpclient* and *spring-cloud-starter-openfeign* have *feign-core* as a sub-dependency, it is important to use versions of these Maven dependencies that reference the same version of *feign-core* (in the case of this project version 9.7.0).

```
1 <dependency>
2   <groupId>io.github.openfeign</groupId>
3   <artifactId>feign-httpclient</artifactId>
4   <version>9.7.0</version>
5 </dependency>
```

Listing 6.16: Feign Apache HttpClient Maven dependency

6.4.3 Security

Since sensitive data is passed freely between the individual microservices, it is important to consider the security of this data during transit.

6.4.3.1 Communication Between Exoscale Instances

In the VIPER back end every microservice runs on its own Exoscale instance with all instances being in the same data center in Vienna. This means that all traffic between the instances is limited to the data center and is never routed over the Internet. Additional security is provided by firewalls, allowing only Exoscale instance of other microservices to make HTTP requests to each other (for more details about Exoscale firewalls see subsubsection 6.5.7.1). Security could be slightly improved by only allowing communication between instances that need to communicate with each other. But as the back end system is relatively small and all microservices in the system trust each other, this is not used. Having this more complex firewall configuration would also bring back an unwanted element of coupling between the services, as every service would have to manage from which other services requests are allowed.

Exoscale provides the functionality of virtual private networks. Using this, all microservices could be inside such a private network, being able to communicate with each other without being exposed to the Internet. This would improve security of the services and eliminate the need for a firewall. The reason this feature is not used for the VIPER back end is the fact that Exoscale released a new feature call *managed private networks* [77]. Whereas with standard private networks all management of IP addresses has to be handled by the user of the network, managed private networks have a Dynamic Host Configuration Protocol (DHCP) server managed by Exoscale. As this feature is by the time of writing only available in the Swiss region of Exoscale, it could not be used for this project. Using the standard private networks instead would have been possible but would not have made much sense. Using them would have required the development of a custom solution for distributing IP addresses to the instances, which would be obsolete once managed private networks become available in the Vienna region. Since this project is just a prototype no time was spend developing something the will soon be obsolete. Instead it was decided to wait for the new feature to come to the Vienna region.

6.4.3.2 Communication Over TLS

All communication between the services happens within a single data center and nothing is routed over the Internet. Encrypting the traffic is therefore not a priority. But as it would still bring security improvements, it should be implemented in the future. The reasons for not implementing it are the time constraints of the project. Encryption can be done by forcing Hypertext Transfer Protocol Secure (HTTPS) on all ReST endpoints. This would be implemented the same way as described under subsubsection 6.5.7.2. Even though performing all communication over Transport Layer Security (TLS) would result in security improvements, it should be considered that encrypting and decrypting the network traffic is an overhead that might be unnecessary for some communication paths. If only information that is already available through a public API is shared in a request, it would be unnecessary to encrypt this traffic. The communication over TLS should therefore only be implemented where it is actually needed to improve security.

6.5 External Communication

The microservices do not only need to communicate with each other, but also with third parties over the Internet. They get requests from users using the website and from customers using a VR or AR application. These requests from AR or VR applications will in most cases come from the client library (see section 6.13).

6.5.1 Communication Technologies

Choosing the right technologies is especially important for the external communication, as these technologies not only decide how the components of the VIPER system communicate with each other but also how third parties can communicate with the system.

6.5.1.1 Using ReST

The communication with the website and the AR and VR applications happens over a public API. It was therefore important to implement it, using widely accepted technologies, usable across a wide variety of systems, platforms, technology stacks and programming languages. Using ReST over HTTP was therefore the obvious choice, as these are the technologies Spring Cloud uses by default. HTTP and ReST fulfill all the requirements stated previously. These technologies are very widely used and accepted, they can be used on almost any platform and with virtually every programing language. HTTP and ReST are especially easy to use with web applications clients using JavaScript, as the example from this website show [69]. Development of the system is very simple when HTTP and ReST are used, because every component (Spring Web, Spring Security, Feign and Zuul) used in the VIPER back end system, uses these technologies by default. This eliminates the need for any additional configuration and takes virtually no time to set up.

Of course there are other options for the external communication. One such option, that is also supported by Spring Cloud, is SOAP. But there are multiple downsides of this technology compared to ReST. It does not integrate as well with the used technology stack and would lead to a higher development effort. Additionally, it is not as widely used, especially for lightweight web applications [41] [143]. This 2013 paper [107] also shows the clear performance benefits of using ReST instead of SOAP. All of these downside lead to the elimination of SOAP and make ReST and HTTP the logical choice for the external communication. For more information about ReST see this paper [59]

6.5.1.2 JSON data format

For the data format used to share information with third parties, the same requires stated in the previous section hold. It has to be widely accepted, it should be easy to use with a wide variety of programming languages and it should ideally be human readable (for increased convenience). Given these requirements, there are two possible data formats: JSON and eXtensible Markup Language (XML). Both these formats are human readable, widely used and have libraries for virtually any programing language. Compared to XML JSON has less overhead, which decreases network load and increases performance. Additionally, JSON is the data format used by Spring Web by default. It was therefore chosen as the data format for the external communication. XML would not have provided any advantages over JSON, as feature like tags and attributes, which are unique to XML would not have been used in the

project. Moreover, using XML would have lead to an additional development effort, as it is not the Spring Web default.

6.5.2 Zuul API Gateway Service - Routing

The API Gateway Service uses Netflix Zuul for its routing capabilities.

6.5.2.1 Zuul and Eureka

Zuul works together with the Eureka Service (see subsection 6.6.2) to get a list of all available services and can route incoming traffic according to the routing configuration. When using Zuul with Eureka, it automatically creates routes to all discovered services based on their names. A service with the name *api* would therefore automatically receive all traffic sent to the gateway with the URL path */api/**. To prevent this automatic creation of routes and customize the routing behavior, all services for which no automatic routes should be created, have to be ignored with the configuration shown in Listing 6.17.

```
1 zuul.ignored-services=api,authentication,braintree,customer,
      developer,payment-broker
```

Listing 6.17: Ignore all services to prevent automatic route generation by Zuul

6.5.2.2 Zuul Routing

Having disabled automatic route generation, new routes can be created with the configuration shown in Listing 6.18. This configuration routes all traffic coming to the gateway via the URL paths *user/**, *developer/** and *api/** to the appropriate services. With this configuration the prefix of the URL path is stripped. Thus, a request to *gateway*/user/pay is forwarded to *customer_service*/pay. For the Authentication Service only a single route is exposed to the outside world - */login* - all other traffic is blocked.

```
1  # Zuul routes
2  # -----------
3
4  # authentication service
5  zuul.routes.authentication.path=/login
6
7  # customer web service
8  zuul.routes.customer.path=/user/**
9
10 # developer web service
11 zuul.routes.developer.path=/developer/**
12
13 # api service
14 zuul.routes.api.path=/api/**
```

Listing 6.18: Configuration of Zuul routes

All services without a route configuration cannot be reached over the Internet and can just communicate with the other services in the microservice system.

6.5.2.3 Disabling Individual Routes

When using the double asterisk (.../**) in route configurations all traffic that matches the pattern is routed to a service. This might have some unwanted side effects, as all endpoints of a service would be exposed. Even those that are just meant for inter-service communication. To disallow traffic to certain endpoints the configuration shown in Listing 6.19 can be used. The three services using this configuration (Customer Web Service, Developer Web Service and API Service) do not need this protective measure. All endpoints of these services are in the public API, as either part of the website back end or part of the VR and AR application API. If the APIs of these services were to be expanded to include private endpoints in the future, the configuration from Listing 6.19 could be used to protected them from unwanted traffic.

```
1 zuul.ignored-patterns=/user/account/info,/developer/org/
      payment
```

Listing 6.19: Example of a configuration to disallow traffic to certain endpoints

6.5.3 Authentication Service

The two main tasks of the Authentication Service are to store user account information and validate the credentials of a user trying to log in. Additionally, it also stores the API keys of all registered applications.

6.5.3.1 Authentication Service Database

As described in subsection 6.3.3, the Authentication Service uses MongoDB to store all of its data. This includes all information necessary to verify login attempts of users and the API keys of all applications. The Entity Relationship Diagram (ERD) from Figure 6.7 shows the database design used to store this information. Some of the information stored in the database, namely the organization ID of developers and applications is redundant, with the main storage being the Developer Web Service. This information is stored redundantly for performance reason, as the communication between the Authentication and Developer Web Service was too chatty without this redundancy.

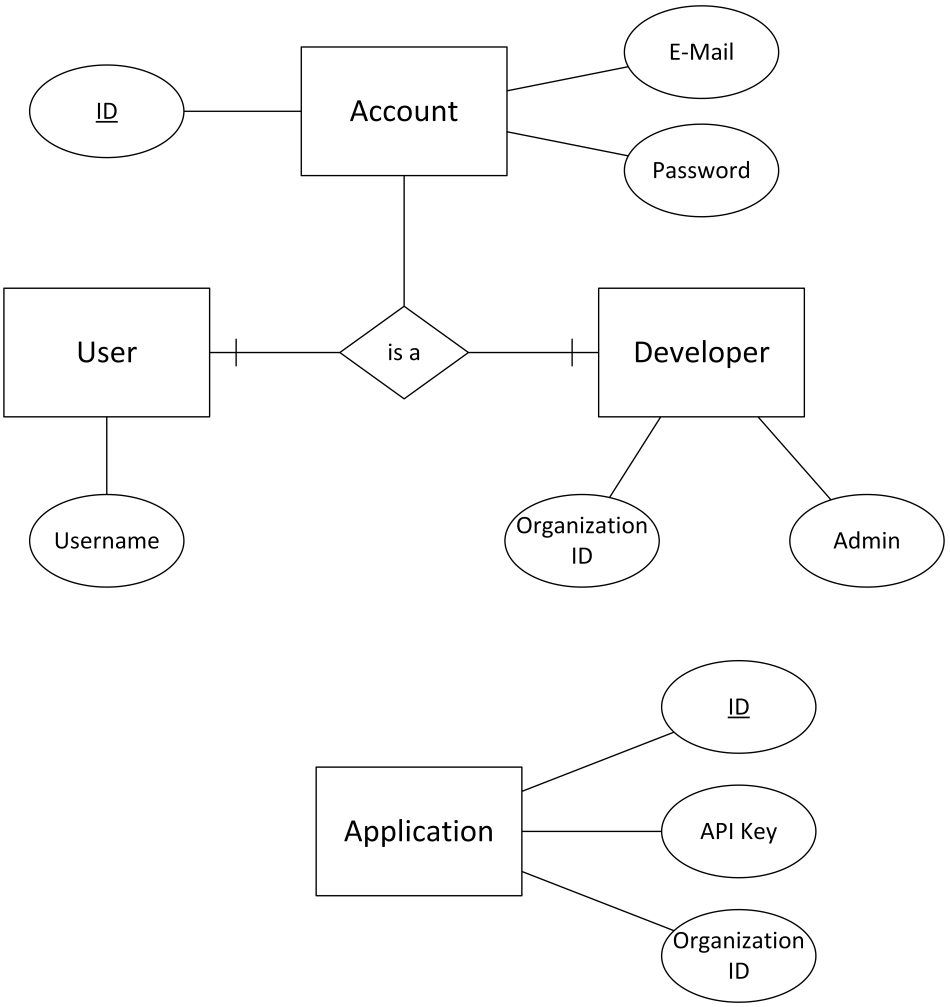


Figure 6.7: ERD of the Authentication Service database

6.5.3.2 Password Hashing

All user passwords were stored in hashed form using the Scrypt hashing algorithm (for a detailed explanation of the Scrypt hashing function see [119] and [120]). Scrypt is one of the four hashing algorithms deemed secure enough for password storage by the Open Web Application Security Project (OWASP) [84]. The other three being Argon2, PBKDF2 and Bcrypt. OWASP recommends the use of Argon2 for password hashing, but as the used security library - Spring Security - does not support this new hashing algorithm, it could not be used. If support for this algorithm gets added in the future switching to it should be considered to improve security.

Out of the remaining three choices - PBKDF2, Bcrypt and Scrypt - both PBKDF2 and Bcrypt are not recommended any more. [126] The calculation of the PBKDF2 can be optimized and speed up by omitting parts of the calculation and can potentially be parallelized and further speed up with the use of Graphics Processing Units (GPUs), as detailed in this 2018 paper [170]. Comparing Bcrypt and Scrypt, the calculation of a Bcrypt hash is less memory intensive and can therefore be cracked more easily using GPUs [90]. For these reasons Scrypt was chosen as the password hashing algorithm of the VIPER back end.

6.5.3.3 Authentication Service User registration

When a new user registers at the VIPER system, the request initially goes to either the customer or the Developer Web Service, depending on the account type. These services then send all information necessary for user authentication to the Authentication Service, where it is stored in its database. At the Authentication Service the password is hashed and the user information is written to the database, as shown in Listing 6.20.

```
1 User user = new User(newUser.getEmail(), passwordEncoder.
    encode(newUser.getPassword()), newUser.getUsername());
2 userRepository.save(user);
```

Listing 6.20: Storing a new user in the database of the Authentication Service

The *passwordEncoder* object used for password hashing is an instance of Spring Security's Scrypt implementation *SCryptPasswordEncoder*, which hashes the password and automatically adds a salt for added security (for further information about password salts see [84] and [134]).

After adding the user information to the database, the newly created user is automatically logged in by returning a JSON Web Token (JWT) in the *authorization* header, as shown in Listing 6.21. This process is explained in more detail in subsection 6.5.4.

```
1 //Create JWT
2 String token = JwtUsernameAndPasswordAuthenticationFilter.
    createJwt(user.get_id().toHexString(), new ArrayList<String
    >(){{add("ROLE_USER");}}, jwtConfig.getExpiration(),
    jwtConfig.getSecret());
3 //Add authentication header with the token
4 response.addHeader(jwtConfig.getHeader(), jwtConfig.getPrefix
    () + token);
```

Listing 6.21: Automatically logging in a newly created user by returning a JWT in the authorization header

6.5.4 Login Handling at the Authentication Service

6.5.4.1 Using Spring Security

For handling user logins Spring Security was used in the Authentication Service. This module can be added to a service as a Maven dependency with the code from Listing 6.22.

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-security</artifactId>
4 </dependency>
```

Listing 6.22: Spring Security Maven dependency

Spring Security requires a class containing all security configurations. The basic structure of this class is shown in Listing 6.23. The full security configuration of the Authentication Service can be found in the appendix under Listing 9.4.

```
@EnableWebSecurity
public class SecurityConfig extends
    WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception
      { ... }

  @Override
  protected void configure(AuthenticationManagerBuilder auth)
     throws Exception { ... }
}
```

Listing 6.23: Basic structure of the Spring Security configuration class

The configuration inside the first *configure* method contains rules for all incoming HTTP requests. The second *configure* method contains the configuration of the *AuthenticationManager*, which performs the authentication of a user. Both these configurations are explained in more detail under subsubsection 6.5.4.3.

6.5.4.2 Choosing an Authentication Technology

Choosing the right authentication technology for the external communication is of prime importance, as this can greatly affect the security of the whole system. Spring Security supports both Sessions and tokens like JWT (see [78] for more details about JWT).

Using Sessions With sessions, all data of logged in users is stored at the Authentication Service, either in memory or in a database. The logged in user just receives a session ID used to find the session data. As only a single ID is need to authenticate a user, the network load is minimal. Having all session data stored at the Authentication Service enables it to invalidate the session at any time. This improves security, as the user can be securely logged out at any time, reducing the risk of an third party maliciously using a user's account without them knowing. When considering performance, using session has some significant downsides, especially in a microservice system. All data has to be stored at the server side and has to be retrieve for every request of a user. This introduces a significant load on the Authentication Service. In the case of multiple running instances (which is important, as high scalability is one of the main benefits of a microservice architecture) it has to be ensured that all requests of a user are routed to the same instance of the Authentication Service. When using sessions the verification for every request has to be done by the same service that created the session and has therefore access to the session storage. This does not work well with the architecture of the VIPER back end, where the Authentication Service is supposed to authenticate users logging in and the Gateway Service is supposed to authenticate requests of logged in users.

Using JWTs In contrast to sessions, JWTs store all information at the client side, significantly reducing the load on the authentication service but trading it for a higher network load. JWTs contain all information necessary to identify a user (in the case of this project just the user ID and role) and are signed with a signature, ensuring that the token cannot be tampered with. When a user tries to log in, the authentication service creates a token, containing all necessary information, along with some meta data like the expiration date and signs it with a secret key. This token is returned to the user inside the *Authorization* header of the HTTP response. The user then has to store the token and pass it in the *Authorization* header of every request it makes to the VIPER back end. At the back end the Gateway Service authorizes the user requests by checking if the JWT content and its signature match. This process works very well with a microservice architecture, as no session information has to be stored at the back end and different services can be used to issue and check tokens. The downside of JWTs is the inability to revoke a token on the server side. The only way for a user to revoke a token and therefore log out, is to delete it at the client side. If a malicious third party has gotten access to the token, there is no way to invalidate it before the set expiration date. A workaround would be to keep a black-list of logged out user's tokens and check every token against this list. But this would bring back some of the downside of sessions, given the fact that this approach requires data to be stored at the back end.

Conclusion Even though JWTs have slightly worse security characteristics than sessions, given their limited ability to be revoked, they provide some significant performance improvements. Using sessions would introduce a heavy load on the Authentication Service and would substantially hinder the ability to scale up the microservice system with multiple instances of the authentication and Gateway Service. Thus, JWT was chosen as the authentication technology for all external communication.

6.5.4.3 Authenticating User Login Requests

Authentication Filter The authentication of a user trying to log in is done with the class *JwtUsernameAndPasswordAuthenticationFilter* which extends Spring Security's *UsernamePasswordAuthenticationFilter*. This class is a filter, which is performed for every request to the */login* endpoint. The configuration shown in Listing 6.24 is necessary to enable this filter and return a 401 (unauthorized) error if it fails.

```
protected void configure(HttpSecurity http) throws Exception {
  http
      // making sure to use stateless session
      .sessionManagement().sessionCreationPolicy(
         SessionCreationPolicy.STATELESS)
      .and()
      // handle unauthorized attempts
      .exceptionHandling().authenticationEntryPoint((req, rsp,
          e) -> rsp.sendError(HttpServletResponse.
         SC_UNAUTHORIZED))
      .and()
      // Add a validation filter
      .addFilter(new
         JwtUsernameAndPasswordAuthenticationFilter(
         authenticationManager(), jwtConfig))
}
```

Listing 6.24: Authentication Service security configuration to add the authentication filter

The authentication filter has two methods: *attemptAuthentication*, which attempts to authenticate a user (shown in Listing 6.25) and *successfulAuthentication*, which is executed if the authentication was successful. The *attemptAuthentication* method creates an authentication token out of the credentials provided by the user. This token is then used by the *AuthenticationManager* to authenticate the user.

```
// 1. Get credentials from request
UserCredentials userCredentials = new ObjectMapper().readValue
    (request.getInputStream(), UserCredentials.class);

// 2. Create auth object for auth manager
UsernamePasswordAuthenticationToken authToken = new
    UsernamePasswordAuthenticationToken(userCredentials.
    getIdentification(), userCredentials.getPassword(),
    Collections.emptyList());

// 3. Authentication manager authenticates the user
return authManager.authenticate(authToken);
```

Listing 6.25: Attepting to authenticate a user using their credentials

If the authentication was successful, a new JWT is created and returned in the *Authorization* header of the HTTP response, as shown in Listing 6.26. The JWT is signed using the SHA-512 hashing algorithm and a 64 character (512 bit) secret stored in the *JwtConfig* class. Even though the SHA hashing algorithm should not be used for password hashing, this paper shows [62] that the combination of SHA-512 and a random secrete of sufficient length is secure. For maximum security the secret used, is a cryptographically secure random string, generated with the random string generator of this website [121].

```
long now = System.currentTimeMillis();

// Create JWT
String token Jwts.builder()
        .setSubject(sub)
        .claim("authorities", auth)
        .setIssuedAt(new Date(now))
        .setExpiration(new Date(now + exp * 1000))
        .setIssuer("com.viper.service.authentication")
        .signWith(SignatureAlgorithm.HS512, secret.getBytes())
        .compact();

// Add token to header
response.addHeader(jwtConfig.getHeader(), jwtConfig.getPrefix
    () + token);
```

Listing 6.26: Creation of a new JWT

The complete *JwtUsernameAndPasswordAuthenticationFilter* class can be found in the appendix under Listing 9.5.

User Details Service The aforementioned *AuthenticationManager* uses the class *UserDetailsServiceImpl*, which extends *UserDetailsService*, to authenticate the provided credentials against the database. The configuration telling the *AuthenticationManager* to use this class is shown in Listing 6.27.

```
@Autowired
@Qualifier("UserDetailsServiceImpl")
private UserDetailsService userDetailsService;

protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth.userDetailsService(userDetailsService).passwordEncoder(
     sCryptPasswordEncoder());
}
```

Listing 6.27: AuthenticationManager configuration

The *UserDetailsServiceImpl* has the method *loadUserByUsername*, which takes the provided credentials and looks for a matching customer or developer in the database. If the user is found, a new User object, containing the credentials and the role, depending on the type of user account, is returned. If no matching user is found in the database, an exception is thrown and a 401 error is returned. The possible roles are *ROLE_USER* (for customer accounts) and *ROLE_DEV* (for developer accounts). Developers can also have the additional role *ROLE_ADMIN* which allows them to make certain request normal developers cannot make.

An example of how the customer account information is retrieved is shown in Listing 6.28. The complete *UserDetailsServiceImpl* class can be found in the appendix under Listing 9.6.

```
List<GrantedAuthority> grantedAuthorities = new ArrayList<>();

// Retrieve customer account from database
Optional<User> optionalUser = userRepository.
    findByEmailOrUsername(identification, identification);

// Check if customer exists
if(optionalUser.isPresent()) {
  User user = optionalUser.get();

  // Set role
  grantedAuthorities.add(new SimpleGrantedAuthority("ROLE_USER
     "));

  return new org.springframework.security.core.userdetails.
     User(user.get_id().toHexString(), user.getPassword(),
     grantedAuthorities);
}
```

Listing 6.28: Retrieving the customer account to authenticate a user

6.5.5 Authentication at the API Gateway

Like the Authentication Service, the Gateway Service also has a Spring Security filter for authentication.

6.5.5.1 The JWT Authentication Filter

Spring Security Configuration The gateway's *JwtTokenAuthenticationFilter*, which inherits from Spring Security's *OncePerRequestFilter* is, as the name implies, executed once for every incoming request. The gateway also has a Spring Security configuration class with a structure comparable to what is shown in Listing 6.23. The configuration from Listing 6.29 is necessary to enable the filter and return a 401 (unauthorized) error if it fails.

```
protected void configure(HttpSecurity http) throws Exception {
  http
      // making sure to use stateless session; session won't
         be used to store user's state.
      .sessionManagement().sessionCreationPolicy(
         SessionCreationPolicy.STATELESS)
      .and()
      // handle unauthorized attempts
```

```
7         .exceptionHandling().authenticationEntryPoint((req, rsp,
              e) -> rsp.sendError(HttpServletResponse.
              SC_UNAUTHORIZED))
8         .and()
9         // Add a filter to validate the tokens with every
              request
10        .addFilterAfter(new JwtTokenAuthenticationFilter(
              jwtConfig), UsernamePasswordAuthenticationFilter.
              class)
```

Listing 6.29: Spring Security configuration of the Gateway Service enabling the JWT authenticaiton filter

Authentication Process Inside the *JwtTokenAuthenticationFilter* the JWT is parsed and all claims (e.g. the expiration date) as well as the signature are checked. If these checks are successful, the user ID stored in the token is taken and put into Spring Security's *SecurityContextHolder* for further processing. The most important steps of this verification process are shown in Listing 6.30. The complete *JwtTokenAuthenticationFilter* class can be found in the appendix under Listing 9.7.

```
1  // Validate the token
2  Claims claims = Jwts.parser()
3      .setSigningKey(jwtConfig.getSecret().getBytes())
4      .parseClaimsJws(token)
5      .getBody();
6
7  String id = claims.getSubject();
8
9  // Create auth object
10 UsernamePasswordAuthenticationToken auth = new
       UsernamePasswordAuthenticationToken(id, null, authorities.
       stream().map(SimpleGrantedAuthority::new).collect(
       Collectors.toList()));
11
12 // Authenticate the user
13 SecurityContextHolder.getContext().setAuthentication(auth);
```

Listing 6.30: Validation of the JWTs

6.5.5.2 Zuul Authentication Filter

After Spring Security's JWT authentication filter, another filter is executed. This filter is built using Zuul's filtering framework and has two purposes: blocking requests a user is not allowed to make and adding a custom *ID* header to every request. The decision which endpoints a user is allowed to access is made based on the the role stored inside the JWT. If a user makes a request to an endpoint they are not allowed to access - e.g. a customer making

a request to an endpoint of the Developer Web Service - the request is blocked by Zuul and a 403 (forbidden) error is returned. The implementation of this filter can be found in the appendix under Listing 9.8 (in the *authorize* method). If the request was not blocked by the filter a custom *ID* header, containing the user ID of the user making the request is added, as shown in Listing 6.31. The value of this header lets downstream services know which user made the request they are receiving.

```
1 RequestContext ctx = RequestContext.getCurrentContext();
2 HttpServletRequest request = ctx.getRequest();
3
4 ctx.addZuulRequestHeader("ID", request.getUserPrincipal().
      getName());
```

Listing 6.31: Adding a custom ID header to every request

6.5.6 API Service

The API Service is the back end of all applications using VIPER and the smallest service in the VIPER microservice system. It only has two ReST endpoints: */accounts* for retrieving a list of all payment accounts of the currently logged in customer and */pay* for executing payments. The reason for making this a distinct service, even though the functionality is so minimal, was the fact that this combined functionality does not fit to any of the other services. Additionally, there are plans to expand the service's functionality in the future.

The API service uses the same endpoint at the Customer Web Service, that is used by the website to retrieving a list of all payment accounts. At the */pay* endpoint, the API Service first checks the provided API key and resolves it to the organization ID and application ID, before forwarding it to the Payment-Broker Service for further processing.

6.5.7 Security

Many security aspects of the communication with third parties over the Internet have already been covered in the previous sections (mainly subsubsection 6.5.2.3, subsection 6.5.3, subsection 6.5.4 and subsection 6.5.5). This section will expand on the ideas of previous sections and try to resolve some additional security concerns.

6.5.7.1 Exoscale Firewall

The firewall provided by Exoscale is one of the main lines of defense for the VIPER microservice system. The firewall allows all microservices to communicate with each other, without being exposed to the Internet. This is achieved by putting all Exoscale instances that contain microservice into same security group. This security group is then configured to block all incoming traffic from the Internet to all ports. For Exoscale instances inside this security group communication over the ports 8000 (inter-service communication), 8761 (communication with Eureka) and 8788 (communication with the Config Service) is allowed.

To allow the API gateway to communicate with the outside world, an additional security group was created, which explicitly allows incoming traffic on port 80 and 443 for this single service. Having this firewall configuration in place eliminates any possibilities of

unwanted, unauthorized requests to any of the microservice (except the gateway). But as already mentioned in subsubsection 6.5.2.2, since the Gateway Service is exposed to the Internet, incorrect configuration of Zuul routes could still pose a security threat by exposing private endpoints of services.

6.5.7.2 Using HTTPS

One of the most important measures to keep communication secure is encrypting said communication. Thus, all communication over the Internet is encrypted by the use of HTTPS.

Spring Security HTTPS Configuration The API Gateway Service only allows request to port 443 using HTTPS. This is achieved with the Spring Security configuration shown in Listing 6.32.

```
protected void configure(HttpSecurity http) throws Exception {
  http
      .requiresChannel()
      .anyRequest().requiresSecure()
}
```

Listing 6.32: Spring Security configuration to only allow communication over HTTPS

SSL Certificate To use encrypted communication over HTTPS, an SSL certificate is required. As the system developed in this project is just a prototype, a free SSL certificate, issued by *Let's Encrypt* [88] was used. This certificate was provided in the form of a *.pem* file and had to converted to a PKCS#12 keystore (for more detail see [34]), to be used with the Java application. The configuration in Listing 6.33 is required to enable the use of TLS in Spring, using the SSL certificate of the created keystore.

```
# port
server.port=443

# SSL
server.ssl.enabled=true
server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:keystore/viper.p12
server.ssl.key-store-password=some_secret_keystore_password
server.ssl.key-alias=viper
```

Listing 6.33: Spring SSL configuration

6.5.7.3 CORS

To prevent unwanted traffic to the back end services, no other website than the official VIPER website should be able to make request it. Thus, a Cross Origin Resource Sharing (CORS)

configuration was created to only allow requests originating from a *viperpayment.com* URL. The Spring Security configuration shown in Listing 6.34 was used to achieve this goal. This configuration allows all requests originating from *viperpayment.com* or *www.viperpayment.com* with any URL path. It also contains configurations to allow and expose the *Authorization* header, which is required for authentication to work properly.

```
1 public CorsConfigurationSource corsConfigurationSource() {
2   CorsConfiguration configuration = new CorsConfiguration();
3   configuration.setAllowedOrigins(Arrays.asList("http://
       viperpayment.com", "https://viperpayment.com", "http://
       www.viperpayment.com", "https://www.viperpayment.com"));
4   configuration.setAllowedMethods(Arrays.asList("GET", "POST",
        "DELETE", "PATCH", "OPTIONS"));
5   configuration.addAllowedHeader("content-type");
6   configuration.addAllowedHeader("authorization");
7   configuration.addExposedHeader("Authorization");
8   UrlBasedCorsConfigurationSource source = new
       UrlBasedCorsConfigurationSource();
9   source.registerCorsConfiguration("/**", configuration);
10  return source;
11 }
```

Listing 6.34: Spring Security CORS configuration

6.6 Infrastructure Services

Infrastructure services are services within the microservice system, that do not contain any business logic, but rather manage all other services and help them to work together. In the VIPER back end these are the Config Service, which distributes configuration files and the Eureka Service, which allows all microservices to find each other.

6.6.1 The Configuration Service

The Config Service is built using Spring Cloud's *Config Server*. All other microservices use *Spring Cloud Config* to receive configurations from the Config Service.

Configuration of the Config Service The Spring Cloud Config Server functionality can be added to a service by adding the Maven dependency from Listing 6.35. Additionally, the *@EnableConfigServer* annotation has to be added to the main Spring Boot application class.

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-config-server</artifactId>
4 </dependency>
```

Listing 6.35: Maven dependency of the Spring Cloud Config Server

The Spring Cloud Config Server needs a Git repository from which the configuration files can be loaded and distributed. The location and access to this Git repository is configured with the configuration shown in Listing 6.36.

```
1 spring.cloud.config.server.git.uri = https://git.viperpayment.
    com/viper/service-config.git
2 spring.cloud.config.server.git.username=some_username
3 spring.cloud.config.server.git.password=some_password
```

Listing 6.36: Config service Git repository configuration

Configuration of the Clients The clients of the Config Service, which are all other microservices, need to add the Maven dependency from Listing 6.37 and the configuration shown in Listing 6.38. It is important to put this configuration into the *bootstrap.properties* file and not *application.properties*, which is the standard Spring configuration file. *application.properties* is replaced once the configuration from the Config Service is loaded. In addition to the Config Service address, the *bootstrap.properties* file has to contain the name of the Spring application to tell the Config Service which configuration files should be distributed.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

Listing 6.37: Maven dependency of the Spring Cloud Config client

```
# Spring application name
spring.application.name=customer

# IP and port of the config service
spring.cloud.config.uri=http://194.182.173.3:8788
```

Listing 6.38: Configuration of a Spring Cloud Config client

6.6.2 Service Registry with Eureka

As Newman mentioned in his book, automatic service discovery becomes increasingly important, as a microservice system grows [111, p. 397]. The VIPER back end uses Netflix's Eureka (for more details see [111, p. 403]) for this purpose. Eureka has two parts: the Eureka Server and the Eureka Client.

Eureka Server Setup and Configuration The Eureka Server allows clients to register and periodically broadcasts a list of all registered clients. To add the Eureka Server functionality to a service the Maven dependency from Listing 6.39 has to be added. Additionally, the *@EnableEurekaServer* annotation has to be added to the main Spring Boot class.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</
     artifactId>
</dependency>
```

Listing 6.39: Spring Cloud Netflix Eureka Server Maven dependency

To allow communication between the Eureka Server and Eureka Clients they have to be in the same region and zone. The zone configuration of the Eureka Server is shown in Listing 6.40. The corresponding zone configuration of a Eureka Client is shown in Listing 6.42. The complete Eureka Server configuration can be found in the appendix under Listing 9.9.

```
# zone configuration
eureka.client.region=at-vie-1
eureka.client.availability-zones.at-vie-1=default
eureka.client.service-url.defaultZone=http://localhost:8761/
    eureka/
eureka.instance.metadata-map.zone=default
```

Listing 6.40: Eureka Server zone configuration

Eureka Client Configuration For Eureka Clients, which are all microservice except the Eureka and Config Service, the Maven dependency from Listing 6.41 is required. Additionally, the Eureka Client configuration, which can be found in the appendix under Listing 9.10, is needed. This configuration mainly contains timeouts, polling intervals and zone configurations (which can also be found in Listing 6.42).

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</
     artifactId>
</dependency>
```

Listing 6.41: Eureka Client Maven dependency

```
# zone configuration
eureka.client.prefer-same-zone-eureka=true
eureka.client.region=at-vie-1
eureka.client.availability-zones.at-vie-1=default
eureka.client.service-url.default=http://194.182.175.254:8761/
    eureka
eureka.client.service-url.defaultZone=http://localhost:8761/
    eureka
```

Listing 6.42: Eureka Client zone configuration

6.7 Payment Technologies and Services

To understand the possibilities of payment and how payments are implemented, an overview of payment in general and associated technologies is presented.

6.7.1 Payment Requirements

The payment microservices are one of the most important parts of this project. They are responsible for communicating with outside payment services, execute transactions, communicate errors and retrieve information about the transactions. A payment service needs to handle and save the payment information securely. Payment itself is a sensitive topic and comes with great legal responsibility. Thus the service should outsource this responsibility and the associated risks as much as possible. Popular payment methods are favoured and should be implemented first to maximize the possible userbase. Lower transaction fees are preferred, since this reduces the fees for organizations using this project. The payment-flow from the customer to the organizations must be as smooth as possible and not pose a barrier. The payment system should support as many different currencies as possible, to ensure that many people can use it. Finally, the services have to be able to execute transactions autonomously and without human interaction.

6.7.2 Value Transfer

The aim of this project is to enable payment in restricted environments. In other words, the two user groups *customers* and *organizations* need to be able to exchange value assets. To implement automatic payments, both parties have to use assets that can be managed without user interaction. Whatever assets will be used for the exchange, in the end the value has to be convertible to a *real* currency on a bank account, which represents both users point-of-view and an axiom for this payment system. Payment cards like credit or debit card represent an information link to a bank account through the associated numbers on them. The exchange of value in this context means using this information to move assets from one bank account to another.

The electrical exchange of money is done via Electronic Funds Transfer (EFT). EFT is defined as “a funds transfer initiated through an electronic terminal, telephone, computer (including on-line banking) or magnetic tape for the purpose of ordering, instructing, or authorizing a financial institution to debit or credit a consumer’s account” [30].

Wire Transfer
Wire transfer where first launched by the Western Union in 1872, using the telegraph network to send information. The operations required for a Wire transfer are shown in Figure 6.8. Transfer of the money does not happen instantaneous and can take from multiple hours to a few days. The fees for such a transaction are different for each bank and country, but are usually around 25$ per transaction. Domestic transactions are often times cheaper.

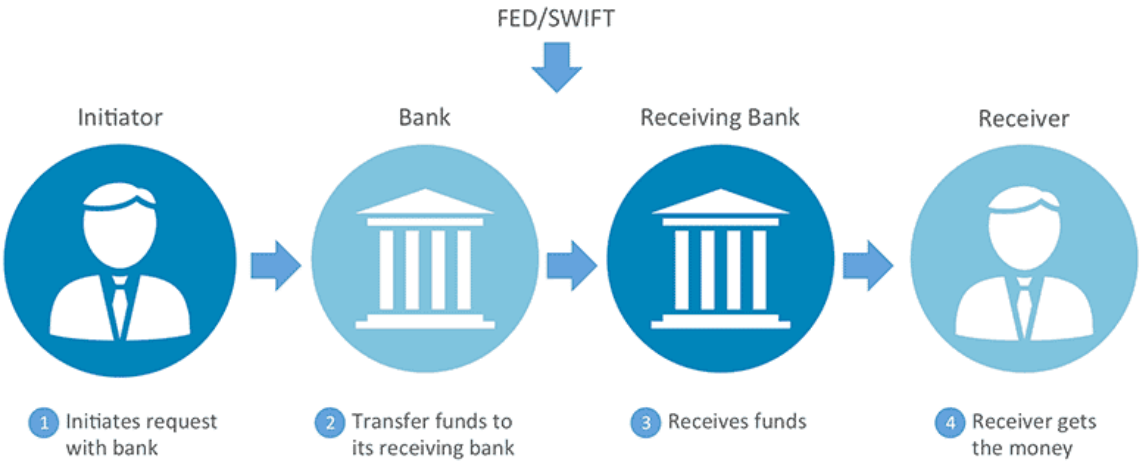


Figure 6.8: Diagram of a Wire transfer.
1. The initiator make a transaction request at his bank (electronically) and supplies the amount he wants to transfer and the target bank account, identified by International Bank Account Number (IBAN) and Business Identifier Code (BIC) codes. 2. The bank forwards the request to the responsible bank. 3. The reviving bank executes the transactions supplied. 4. The receiver receives money on his bank account.

Automated Clearing House (ACH)
An ACH is a "electronic network for financial transactions" [133]. It is a computer-based version of a Clearing House and is specialized on payments. The operations required to execute a transaction via a ACH are show in Figure 6.9. As mentioned by Pritchard, since ACH are electronic they require few resources and can executed completely autonomously [127]. Furthermore, they make it easy to keep track of income and expenses and thus reduce accounting efforts. Also a ACH offers a unified way of making transactions across banks and countries. ACH operators charge a fee per transaction either based on the transaction amount or a fixed price. Generally those fees are higher for small business, because they are volume based, and thus ACH systems are use mainly by big companies. There are different requirements for the settlement time. The most common duration for a transaction is 3 days, but there are also single day transactions. To reference a bank account the numbers on credit or debit cards are used.

Nearly every country has specific ACH operators, in Austria the Geld Service Austria (GSA) is one of the main ACHs. In the United States of America (USA) there are multiple big system and most of them are based on the ACH Network managed by the National Automated Clearing House Association (NACHA). The European Union (EU) also has its own ACH for the Single Euro Payments Area (SEPA) called Pan-European automated clearing house (PE-ACH).

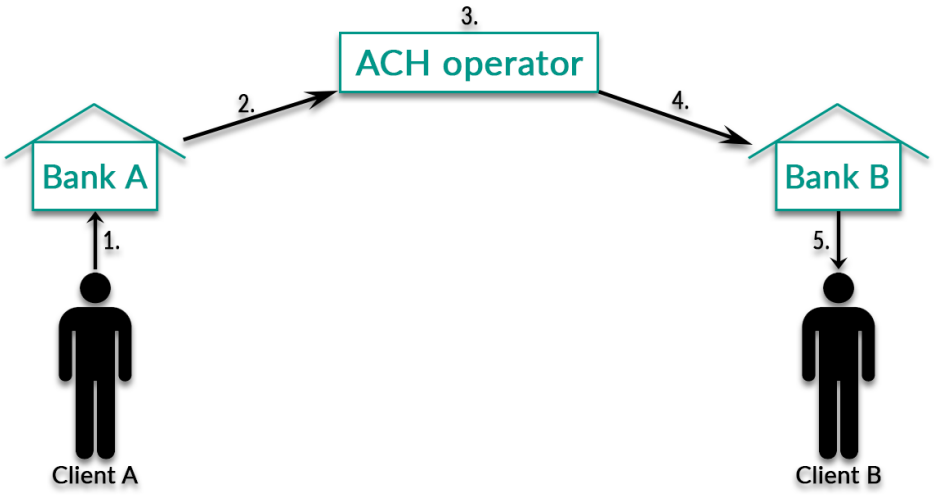


Figure 6.9: A diagram of the typical ACH operations [108].

1. The customer sends a transaction initiation request to a bank (A) electronically. 2. The bank gathers all transaction request for all users per ACH. Periodically a request is send to the ACH with all gathered transaction requests combined and sorted. This is done periodically, at least once a day. 3. The ACH operator combines all information from all banks within a cycle, occurring multiple times a day. In this cycle the received request are checked and verified. 4. All banks get informed about the net settlement amount the are responsible for in this cycle. Once all banks (B) sends the settlement amount to the ACH and cover all transactions, the cycle is complete. 5. A bank account (A) get credited, while another one get debited (B).

Payment Service Provider

PSPs hide underlying EFT and offer a single point-of-contact solution for payments. The payment flow of a PSP transaction can be seen in Figure 6.10. Most PSPs offer a so called *payment gateway* to handle the requests. "A Payment Gateway authenticates and routes payment details in an extremely secure environment between various parties and related banks" [125].The benefits of such a service include:

- Available at all times
- Real time authorization and rapid transaction processing
- Reporting and statics generation
- Merchants have to care less about payment and take little responsibility
- Single point-of-contact and abstraction
- Less (legal) responsibility; Trying to minimize for more customers.
- Potentially lower fees than when using EFT, since huge amount of transactions are processed.

The time required to transfer funds is completely dependent on the underlying payment method and thus can take from multiple hours to a few days.

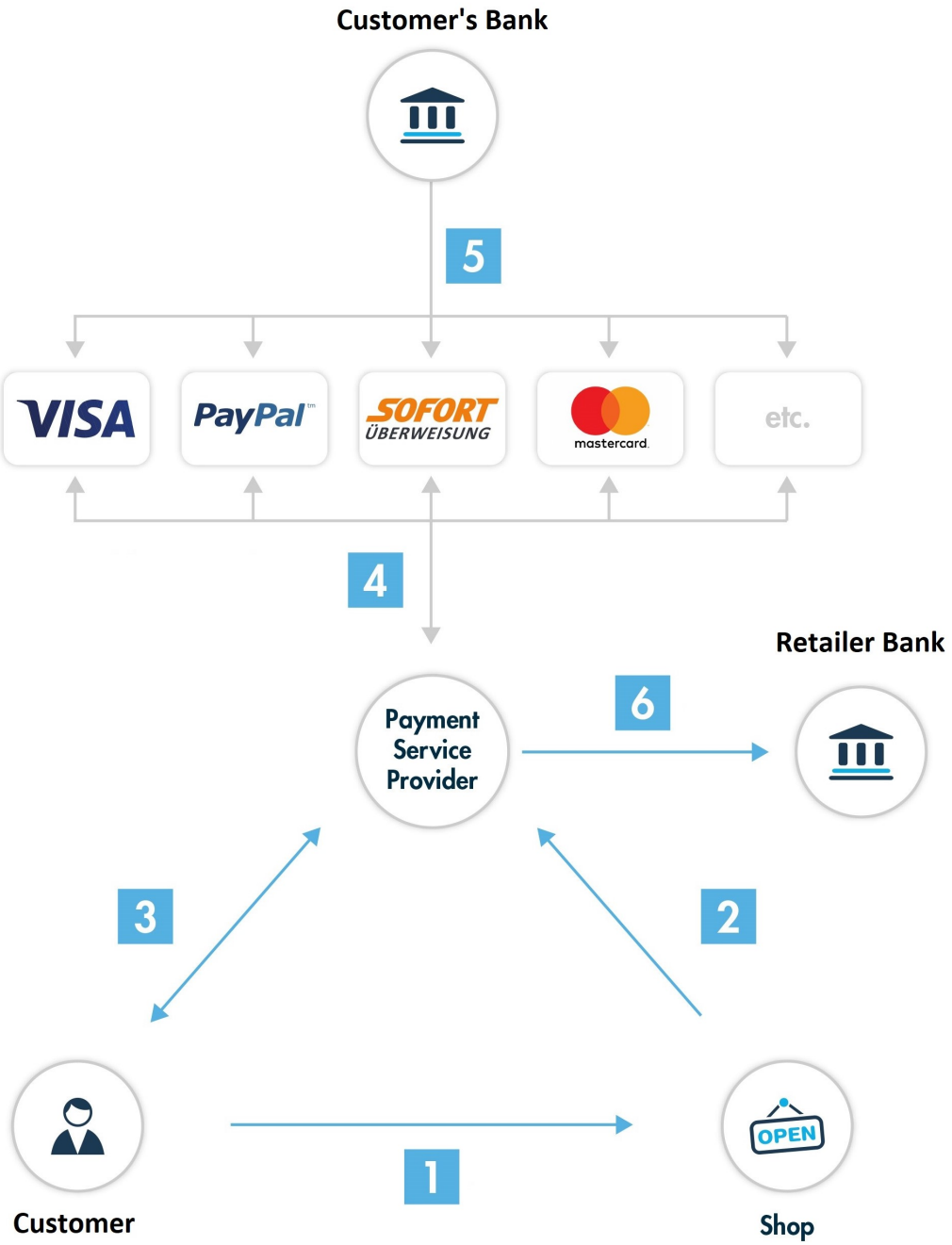


Figure 6.10: Typical payment flow of a PSP [70].
1. A customer wants to buy something in an online shop. 2. The customer gets forwarded to the PSP where he enters his credentials or bank data, and payment information and agrees to purchase the selected products. 3. The PSP sends a confirmation notice and the purchase is completed from their point of view. 4. A request to money transfer service is sent. 5. The transfer service requests the necessary funds from the responsible bank. 6. The retailer receives the money (minus processing fees) on his deposited bank account.

6.7.3 Storing Payment Information

To automatically execute payments, the information entered by the customers has to be stored securely and be available at the Point-Of-Sale.

6.7.3.1 Raw Data

The easiest approach is simply storing the card numbers in a database. The big problem with this solution is that saving this information comes with great legal responsibility and compliance requirements. The Payment Card Industry Data Security Council (PCISSD) is an international security standard for handling payment card information. The violation of this standard results in penalties, often in the form a fines reaching from 5000$ to 100,000$ per month. [28]

6.7.3.2 Digital Wallets

According to a 2016 paper, a digital wallet offers at least the following base functionality [123]:

- "It offers secure enrollment of the user (application download, identity check) and secure provisioning of credentials (e.g., user ID and password for wallet access)"
- Stores credentials, payment information, addresses and other information securely.
- The wallet is funded by a physical store of value, such as payment cards, bank accounts or virtual currencies.
- Tokenization: The payment information is converted into a data string that can be used by third parties for transactions, without actually supplying the real credentials.

The secure data storage and anonymization through tokenization are the biggest advantages of digital wallets and correlate with the requirements for this project. A user only has to register his account once via a wallet provider and can submit the tokenized account via the website. The payment service uses this token to execute a transaction via an appropriate PSP. All mayor smartphone operating systems have their own digital wallets. There are many more digital wallet issuers, which also offer wallets for different platforms. Almost all PSPs store their customer's payment information in a form of digital wallet. Platforms like PayPal and Amazon also offer interfaces to access them. [123]

6.7.3.3 Vaulting Payment Methods

Some PSP offer a feature called vaulting, which means that they store the payment information for you as a digital wallet. This wallet is referenced by an id that can be stored in any database without security concerns and used to make transactions at any time. This puts the legal and security responsibility into the hand of the PSPs. How the payment information is send to the PSP in the first place depends on the PSP. This step is also crucial for whether VIPER had the possibility of seeing the payment information at any point in time determines the PCISSD compliance and thus it is better to have as little payment information as possible at any point in time.

Vaulting is the preferred way to store the payment information. Digital wallets may also be used to support PSPs that do not offer vaulting.

6.7.4 Payment Flow

When it comes to executing transactions, VIPER shares many similarities to PSPs. VIPER wants to offer a single point-of-contact for customers and developers, and also be able to support many different payment methods. Because of these similarities, VIPER could become a PSP and execute transactions via ACH or Wire networks. This would have the advantage of having absolute control over the payments and being independent of any other services but the payment networks. Only the fees for the EFT and no additional charges of any third party apply. The decision fell strongly against this idea, since these advantages are negligible considering the legal responsibility and knowledge required. The implementation of a PSP requires banking knowledge and experience and strong collaboration with payment network and would mean a great administrative overhead. Instead of reinventing the wheel, VIPER represents an abstraction of existing PSPs, which are legally responsible for the transactions and already have a great infrastructure. VIPER thus offers many payment methods and combines the strengths of the used PSPs, without the customer noticing any difference compared to a typical PSP. Furthermore, VIPER does not carry the responsibility of a financial institution, but instead has the same level of responsibility as any retailer using a PSP. The implementation of this abstraction can be done in different ways which are explained in this section.

6.7.4.1 Centralized Flow

In a centralized payment flow, all the transactions are made to a VIPER owned bank account and are forwarded to the organization account. For details see Figure 6.11. The advantage of this payment flow is that the Organization only needs to supply a bank connection to receive money and do nothing else. On the other hand, this again creates the problem of transferring the money without a PSP. In this case a ACH transfer could be used, but this would again introduce the aforementioned problems and contradicts the single-point-of-contact advantage. Furthermore, this would again create legal responsibilities for VIPER.

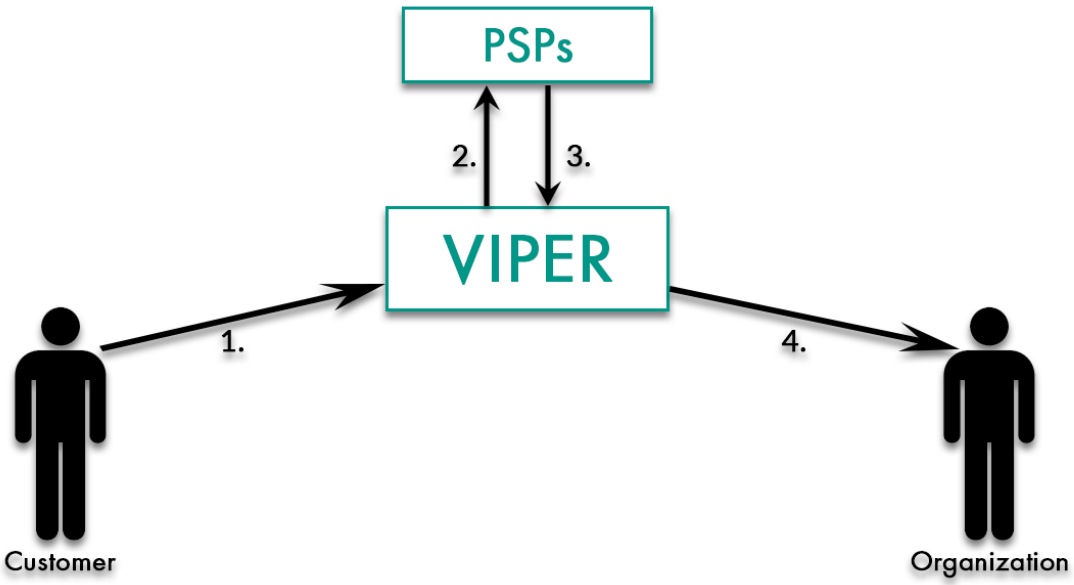


Figure 6.11: Diagram of a centralized payment flow.
1. The customer sends a transaction request to the VIPER service. This is done by the client library in the application he is using. 2. The VIPER service sends an appropriate request to the responsible PSP. 3. The PSP transfers the money to the VIPER bank account. 4. The organization receives the transaction amount, minus transaction fees.

6.7.4.2 Indirect Flow

The indirect approach focuses on reducing the legal responsibility by only indirectly taking part in the transaction. The core idea is that instead of using a VIPER owned PSP account, a Organization owned PSP account is used. For further details see Figure 6.12. With this solution the transaction happens between the Customer and the Organization directly and VIPER only receives the fees directly from the Organization. To simplify the fee payment process and automate it, this solution can be reused by replacing the Customer with the Organization and the Organization with VIPER. This approach drastically reduces the legal responsibility since the Organization has to comply with PSP rules. A downside of this is that the Organization now has to create PSP accounts for them selves. A possible solution is to create the PSP accounts automatically for them, but this is currently not possible, since no PSP provides any features for doing so. In the future possible partnerships with PSPs could make this possible. Despite the disadvantages, the reduction in legal responsibility has led us to the conclusion that this payment flow is best suited for VIPER.

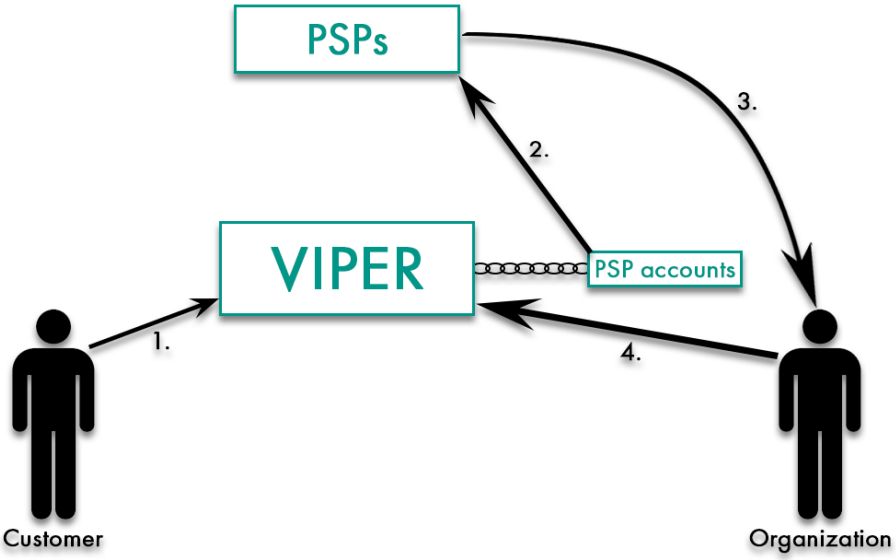


Figure 6.12: Diagram of a indirect payment flow.
1. The customer sends a transaction request to the VIPER service. This is done by the client library in the application he is using. 2. A PSP account provided by the organization is used to send a request to the responsible PSP. 3. The PSP transfers the money the the Organization bank account. 4. The organization pays fees to VIPER.

6.7.5 Payment Service Provider Comparison

In this section most important PSPs are compared and an overview is presented. An overview of the market-share per PSP can be seen in Figure 6.13

PayPal
PayPal is one of the first PSPs in the world and one of the biggest platforms for online transactions today. This fact is represented by the huge market-share of nearly 50%, as seen in Figure 6.13. According to PayPal, the platform is actively used by 1 million users in Austria alone and by 267 million users worldwide [118]. Like most , PayPal uses digital wallets to store the payment information. A reason for PayPals popularity among many customers is their support service, which offers to manage conflicts between customers and retailers for them. PayPal offers Vaulting for credit cards and supports nearly 30 different currencies. Furthermore, all mayor credit card networks are supported along with debit cards. The API is accessible via a ReST interface and thus supported by nearly all technologies, some of which also have a wrapper library for the requests and data, for example python [117]. The PayPal website offers extensive documentation of the API and guides to learn the fundamentals. Paying with PayPal is free of charge and only retailers have to pay fees, which only apply for each transaction and thus the PayPal account itself is free to use. For transactions with value lower than 2,500€ the fees are $0.35 + 3.4\%$ of the value. The variable fees change with the transaction value range and are 2.9% above 2.500€, 2.7% above 10.000€, 2.4% above 50.000€ and 1.9% above 100.000€. PayPal supports more than 200 markets worldwide, more than any other payment service.

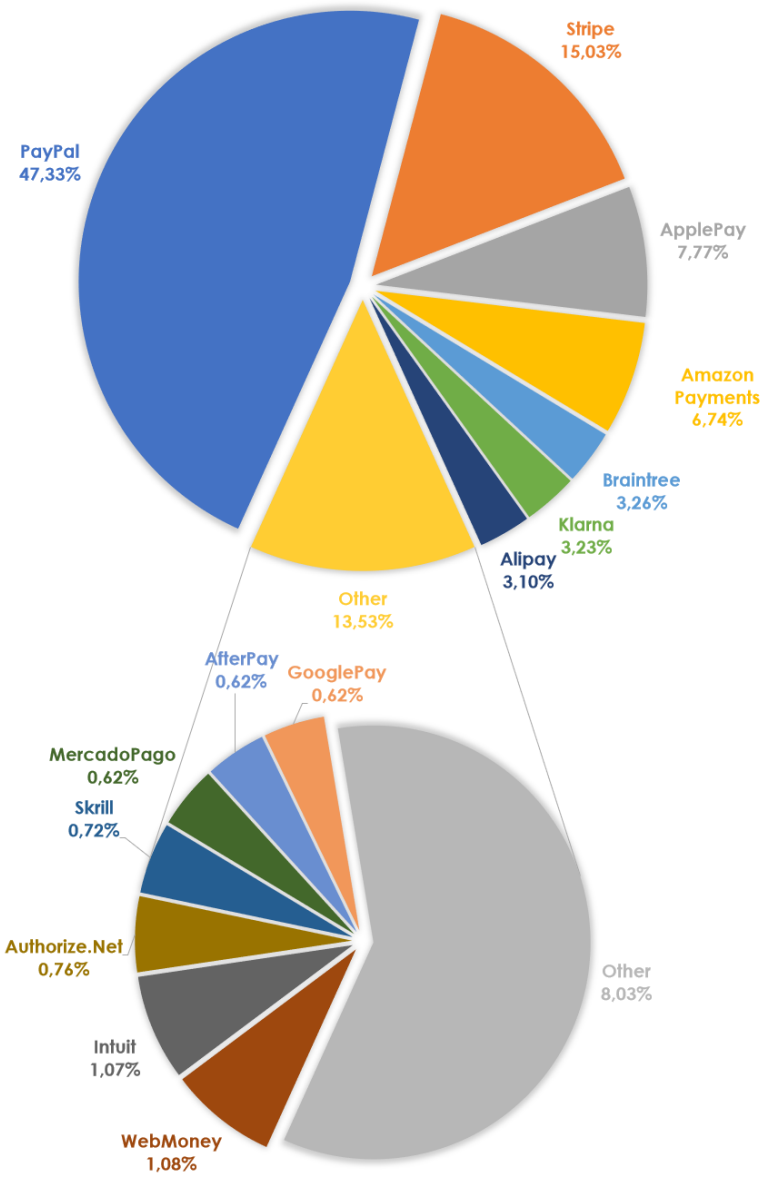


Figure 6.13: Diagram of the top 15 PSPs by market share.
The percentage represents the share of websites that use this particular payment provider from the Alexa Top 1M website ranking [4, 37, 141].

Stripe
After PayPal, Stripe is the largest PSP with a market share of 15%. Strip offers a variety of payment options, including all major card types, wallets from 3rd parties, including Apple Pay, Google Pay, WeChat Pay and Alipay. Furthermore, Stripe also supports ACH transaction as-well as Klarna. European transaction are charged $1.4\% + 0.3\$$ and other transaction $2.9\% + 0.3\$$. Strip offers an SDK for multiple platforms and a great documentation. Processing times are in fixed intervals for different countries and financial conditions. For most purposes a 7-business-day interval applies. Vaulting of payment card is supported. [154]

Apple Pay
Apple Pay is also one of the most popular services. It is only available on Apple devices and the SDK is only available in Swift, Apples own programming language, and low-level C. Apple Pay only charges the fees of the underlying EFT. The transaction time lasts from 1 to 3 business days. Apple pay also offers instant transactions, with transaction times as low as 30 minutes, but charge 1% (minimum 0.25$, maximum 10$) for it. Vaulting is not supported by Apple Pay. [13]

Alipay Alipay is one of the largest online payment platforms, mainly focused on mobile payment. This platform is interesting, since it is very popular in Asian countries and they claim having 520 million registered users. It is not possible to implement Alipay without registering first by supplying documents for a legal entity. The SDK is solely ReST-based and little documentation is available. Fees range from 2.2% below a monthly transaction volume of 1 million RMB down to 1.6% above 10 million RMB volume per month. Processing times are around 1 business day. AliPay manly supports Chines bank cards, but also some international credit cards, aswell as ACH methods and Western Union. Alipay does not support vaulting. [3]

Amazon Pay Amazon pay is a subsidiary of Amazon, an e-commerce company. The payment service uses the companies consumer base and uses their already submitted payment methods to execute payments. The main focus of Amazon pay is to make these accounts available outside their platform. Amazon has nearly 1.5 billion users worldwide, this means nearly 1.5 billion potential Amazon pay users, who already own an account. In reality Amazon pay has a 8% market share and is gaining popularity. Amazon pay only supports credit and debit cards, as well as Amazon gift cards. Amazon pay charges 2.9% for US domestic transaction and 3.9% for cross-border transactions. The authorization of each transaction costs about 0.30$ for either transaction type. For integration purposes, Amazon pay offers a variety of SDKs for different languages, but the documentation is of medium quality. Amazon pay holds all orders for 30 minutes, in order to enable cancellations from both side. Processing times should not be longer than a few business days, but can reach up to 21 business days. [106, 5]

Braintree Braintree is a subsidiary of PayPal and is thus tightly coupled with its features. Braintree offers complete support for PayPal payments and all major card types. Digital wallets like PayPal, Venmo, Apple Pay and Google Pay are supported. Also ACH methods are supported, including Klarna. Fees for all transaction fees are $2.9\% + 0.3\$$, except for PayPal transaction, where PayPal's fees apply. All payouts are either done via credit cards, taking 2-5 business days, or via PayPal. Vaulting is supported for payment cards, PayPal accounts and ACH methods. Braintree offers a variety of SDKs for multiple platforms and offers a great documentation. [24]

Cryptocurrency Cryptocurrencies are a relatively new method of payment and experienced wide popularity in the last years. Its main advantage is that it is independent of banks and completely anonymous. There are a variety of cryptocurrencies available, with Bitcoin and Etherium being most popular ones. Because of the large amount of possible cryptocurrencies, many implementations may be required. Fees can range from a few cents to tens of dollars, depending on the traffic load. Processing times similarly can range from a few minutes up to hours, but are almost always completed within a day. There are currently no PSP like services

available for cryptocurrencies, thus requiring deep knowledge of blockchain technology for the implementation.

Technologie	Popularity	Versatility	Fees	API quality	Processing time	Vaulting supported	Result
Braintree	2	1	2	1	1	1	8
PayPal	1	2	3	2	1	1	10
Stripe	2	1	2	4	3	1	13
Amazon Pay	2	4	3	3	2	1	15
Apple Pay	3	4	1	5	2	2	17
AliPay	3	3	2	6	2	2	18
Cryptocurrencies	4	5	3	7	0	2	21

Table 6.1: The table above shows that Braintree is clearly the best choice as a PSP.

6.7.6 Payment Broker Service

The payment broker service plays a central role in the payment process and is used as an additional abstraction level (see Figure 6.14). Instead of sending the payment request directly to the payment services, the request is send to the broker who forwards them accordingly and thus hides the different payment services. Furthermore, the payment broker is responsible for storing the transaction objects.

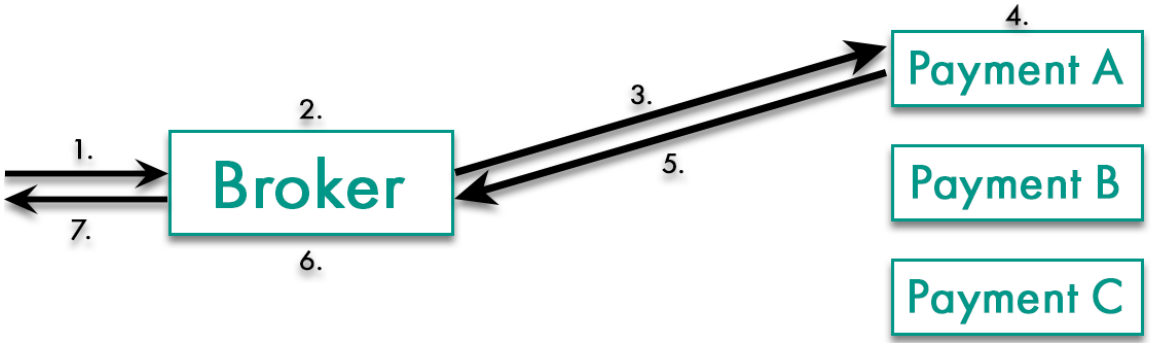


Figure 6.14: Flow of a payment request via the payment broker.
1. A transaction request is send to the payment broker via the API service. 2. The payment broker creates a temporary transaction object. 3. The transaction object is send to the responsible payment service. 4. The payment service execute the transaction (for details see subsection 6.7.7) 5. The transaction object is returned with a reference to the PSP attached. 6. The transaction object is saved in the database. 7. A result message is returned to the requesting service.

The payment-broker service is responsible for forwarding payment requests to the appropriate payment-service and storing the resulting transaction object. When receiving a payment request, the payment-broker sends a request to the client-web-service to verify the sent authentication string for the given payment account. If the verification fails, the payment request is aborted and an error message is returned (as shown in Listing 6.43).

```
1 @PostMapping("{id}")
2 public Response pay(@PathVariable("id") ObjectId id,
    @RequestBody TransactionRequest request) {
3   Response customerResponse = customerService.authorize(id,
      request.getAccount(), request.getAuthentication());
4   if(customerResponse.getCode() != 0)
5     return new Error(1, customerResponse.getMessage());
```

Listing 6.43: Authorizing user-initiated transaction

Once the user is verified, the Organization account owning the Application, which sent the request, is queried and passed to the payment service associated with the given payment account. Currently only the Braintree service is implemented and thus all request are sent to it, as shown in Listing 6.44. In the future, the responsible payment service will be queried from the customer-web-service and the requests adjusted.

```
1 ObjectId appId = new ObjectId(request.getAppId());
2 Response orgResponse = developerService.getAppOrg(appId);
3 BraintreeTransaction braintreeTransaction = new
    BraintreeTransaction((String)orgResponse.getContent(),
    request.getOrders(),request.getCurrency(),accountName);
4 Response braintreeResponse = braintreeService.pay(id, request.
    getAccount(), braintreeTransaction);
```

Listing 6.44: Sending a transaction to a payment service

If the transaction is successfully executed, a transaction object is created, as shown in Listing 6.45. Otherwise an error message is returned.

```
1 Transaction transaction = new Transaction(id, new Date(),
    currencyRepository.findByAbbr(request.getCurrency()).get(),
     appId, request.getInfo(), (String)braintreeResponse.
    getContent(), orders);
2 transactionRepository.save(transaction);
```

Listing 6.45: Creating and saving a transaction object

6.7.7 Braintree Service

The Braintree service is an implementation of a payment service that uses Braintree as a PSP. The steps required to execute a transaction are shown in Figure 6.15 and are applicable to all possible payment service implementations with minor changes. A payment service is responsible for storing the payment information and executing payments.

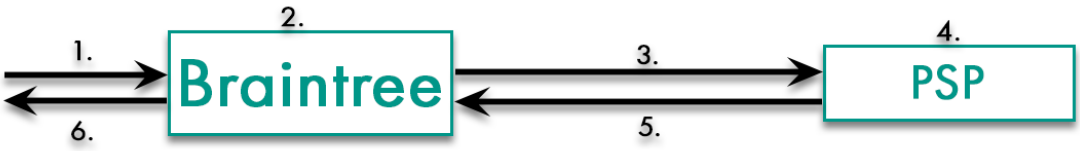


Figure 6.15: Flow of a transaction request to the Braintree service.
1. A transaction object is sent from the payment broker to the service. 2. The transaction object is converted to a format required by the PSP. Additional information such as the payment information, in this case the account token, is gathered. 3. The converted request and the payment information is send to the PSP. 4. The PSP verifies all information and executes the transaction accordingly. 5. A Braintree transaction object is returned to the Braintree service. 6. The transaction object is extended by the required information from the Braintree transaction object and send to the payment broker.

6.7.7.1 Account storage

The Braintree service has to store the information for the organization to execute payments via Braintree and the tokens for the customers.

Organization account
Braintree requires 3 strings to be passed to their API to authenticate the retailer. These strings can be retrieved from the Braintree dashboard in the API settings. The required strings are: Merchant ID, an immutable string identifying the Braintree account, the public key, a string used for encryption and the private key, a string used for decryption. These Strings are handled in a Java object that can be seen in Listing 6.46 and store it as a JSON document in a MonogDB database. The object is identified by the organization id, which connects it to the entities stored in other services. A basic CRUD ReST interface is provided to manage the accounts. The requests to do so are sent from the Developer-web service.

```
1 public class OrganizationAccount {
2   private ObjectId organizationId;
3   private String publicKey;
4   private String privateKey;
5   private String merchantKey;
6 }
```

Listing 6.46: Organization account object

Customer account

The customer account is used to store the reference to the vaulted payment method managed by Braintree and the associated name given by the user. The object can be seen in Listing 6.47. A basic CRUD ReST interface is provided to manage the accounts. The requests to do so are sent from the Customer-web service.

```
1 public class CustomerPaymentAccount {
2
3   private ObjectId userId;
4
5   private String name;
6
7   private String token;
8 }
```

Listing 6.47: Customer account object

Braintree currently offers no functionality to share vaulted payment accounts between merchants. This is problematic, since VIPER would have to vault the Customer's payment account for every organization they want to interact with. This means an extreme overhead when creating an account and if the customer wants to remove or alter his payment information, it has to be change in a lot of different places, which may lead to inconsistency. Furthermore, the vaulting is only possible via the website and thus the customers would have to re-enter their payment accounts for each organization. This means VIPER is currently forced to use a centralized payment flow. Braintree is currently developing a so called *Granting* API, which allows the service to store all Customer payment account on the VIPER Braintree account and thus store the information in one place only. As of March 2019, this API is in a closed-Beta phase and cannot be used by everyone. The current implementation is based on the Granting functionality, which will be shown in the next section. For presentation purposes, VIPER currently relies on the centralized approach, which would require manual money transfer to the organizations as of now.

6.7.7.2 Braintree Vaulting

Vaulting in a payment account means storing the payment information at Braintree without having the raw information in the VIPER servers at any time. This is important, because it greatly reduces VIPERs legal responsibility. All the steps associated with vaulting and executing a payment can be seen in Figure 6.16. The first request is done by the website as described in subsubsection 7.4.2.2 and is forwarded through client-web-service to the Braintree service. The appropriate client token is returned by the code shown in Listing 6.48. Here the `defaultContext` is used, which refers to the VIPER Braintree account, since the service has to vault all data on this account for the reasons mentioned in the previous section.

```
1 @GetMapping("{id}/token")
2
3 public Response clientToken(@PathVariable("id") ObjectId id) {
4
5   String token = defaultContext.clientToken(id.toHexString());
6
7   return new Success<>("collected token", token);
8 }
```

Listing 6.48: Client token creation

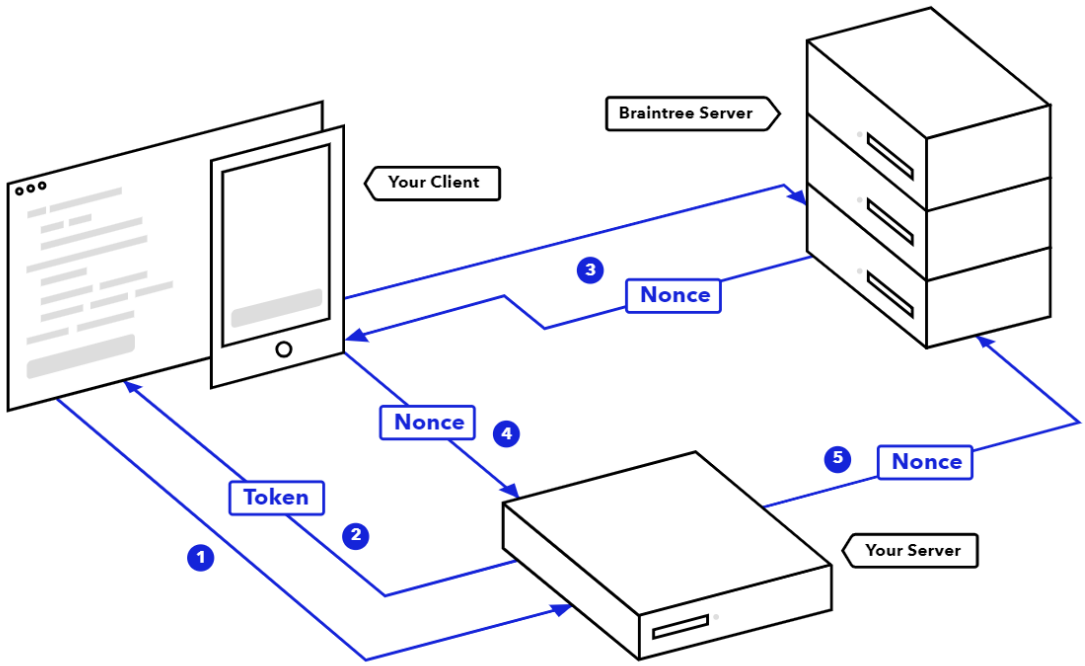


Figure 6.16: Diagram of the steps involving the Braintree Server [24].
1. A request is sent to the Braintree service. 2. The service returns the generated client-token to the website. This token is used for the Braintree Web-SDK to be able to communicate directly with Braintree. 3. The Customer enters his payment information via the website or a pop-up window provided by the SDK. The website sends the information to the Braintree Server, which saves the information, and returns a Nonce. 4. The Nonce is sent to the Web-server and is used to reference the now vaulted payment method. 5. To execute a transaction, a transaction request and a Nonce are sent to the Braintree Server.

This token is used by the website as shown in subsubsection 7.4.2.2. Once the user selects an appropriate payment method, either a new window is opened where they enter their informations and the data is directly sent to Braintree or the information is entered through forms on the website and a request is created to send the data to Braintree. Either way, once Braintree receives the data and validates it, a nonce string is returned to the Website. The Braintree service could now query the payment methods for a user, but without knowing any identifying information about them. This is problematic, since the service has to associate the vaulted account with a name given by the user and thus need a reference to it. Braintree identifies the accounts via the token, but since only the Nonce is received when vaulting this information is useless. This problem was solved by sending the received nonce to the Braintree service and matching it to the right account by creating a Nonce for each vaulted method and comparing them. This can be seen Listing 6.49. The created Nonces have to be the same, since a nonce string is valid for 24 hours and single-use thus receiving the same string on the website and the service.

```
1 PaymentMethod getAccountFromNonce(String nonce,String id){
2   List<? extends PaymentMethod> methods = getCustomer(id).
       getPaymentMethods();
3
4   for(PaymentMethod method: methods){
5     String cnonce = getNonce(method.getToken());
6
7     if(nonce.equals(cnonce))
8       return method;
9   }
10
11  return null;
12 }
```

Listing 6.49: Matching nonces to assoicated payment methods

The token of this payment method is saved in a `CustomerPaymentAccount` object and now references the vaulted method.

6.7.7.3 Payment Execution

The execution of a payment is done via request to the payment service. To payment is base on an Transaction object, shown in Listing 6.50, which needs to be provided to the service.

```
1 public class Transaction {
2
3   private ObjectId organizationId;
4
5   private List<Order> orders;
6
7   private String currency;
8
9   private Name name;
10 }
```

Listing 6.50: Transaction object

This object tells the service which Organization receives the payment, what orders where made, in which currency this transaction is to be executed and the name of the payment account the user wants to use. Listing 6.51 shows an Order object, which consist of a description of the item, provided by the organization, the amount of items the customer wants to buy and the price.

```
public class Order {

  private String desc;

  private int amount;

  private long price;

}
```

Listing 6.51: Order object

Once the transaction is received, the appropriate *OrganizationAccount* object is loaded from the database. This object is used to create a *BraintreeContext* object, which provides basic functionality for interacting with the underlying Braintree Java library.The *CustomerPaymentAccount* is loaded from the database by the name included in the Transaction object and the id provided by the payment-broker.

To create a transaction request for Braintree, a Nonce has to be created. The Grant API is used to give different Braintree accounts to the vaulted payment methods. In Listing 6.52 a grant request is created, which specifies that the consumer of the Nonce, in this case the Organization involved in the transaction, is not allowed to store the payment account. The created Nonce is then sent to Braintree via the SDK. If the request is successful a Nonce object will be returned.

```
PaymentMethodGrantRequest grantRequest =
  new PaymentMethodGrantRequest().allowVaulting(false);

Result<PaymentMethodNonce> resultNonce =
  defaultContext.gateway.paymentMethod()
  .grant(customerPaymentAccount.getToken(), grantRequest);

String nonce = resultNonce.getTarget().toString();
```

Listing 6.52: Creation of a shareable nonce

The created Nonce can be used to setup a transaction request, as shown in Listing 6.53. The total value if the transaction is calculated by iterating all order objects and summing up all prices times the wanted amount. The `submitForSettlement` option is used to actually execute the settlement step, instead of just preparing the request and executing it manually in the Braintree dashboard.oc

```
TransactionRequest transactRequest = new TransactionRequest()
  .amount(request.getPrice())
  .paymentMethodNonce(nonce)
  .options()
    .submitForSettlement(true)
    .done();
```

Listing 6.53: Setup of a transaction request

Line-item entries are created for the request, in order to communicate the exact details of the purchase to the customer, the organization and VIPER for verification and legal reasons. Each Order object for this transaction gets called with the `toItem` method shown in Listing 6.54 and appends a line-item to the transaction. Each line-item requires a name, which corresponds to the description property, a quantity number analogous to the amount field, the unit amount, meaning price per item, and the total amount for all units. The service needs to tell Braintree whether this is a credit item, like a refund or a discount, or debit item, a normal transaction item. Since price is stored as an integer in cents, to prevent rounding errors, and the Braintree SDK expects a `BigDecimal` object in hundred cent units with 2 decimal places, it needs to be converted by dividing by 100 and rounding down to 2 decimal places. The rounding mode *down*, in contrast to the *floor* mode, corresponds to truncating the number, instead of always rounding down, thus avoiding rounding errors.

```
public void toItem(TransactionRequest transaction){
  transaction.lineItem()
    .name(desc)
    .quantity(new BigDecimal(amount))
    .unitAmount(new BigDecimal(price).
      divide(new BigDecimal(100), 2, RoundingMode.DOWN))
    .totalAmount(new BigDecimal(price * amount)
      .divide(new BigDecimal(100), 2, RoundingMode.DOWN))
    .kind(TransactionLineItem.Kind.DEBIT)
    .done();
}
```

Listing 6.54: Adding line items

Finally, to execute the transaction the request is send to Braintree, which verifies all information and performs the necessary steps to transfer the money. This can be seen in Listing 6.55.

```
Result<com.braintreegateway.Transaction> result =
 context.execute(transactionRequest)
```

Listing 6.55: Execution of a transaction

6.8 Web Services

6.8.1 Service Design

The Customer and Developer Web Services are the back ends of the VIPER web application (see chapter 7). The basic schema, used to implement these services can be seen in Figure 6.17. This digram shows the basic architectural schema used in both Customer and Developer Web Service. The *RestController* contains multiple endpoints over which it gets requests. The business logic for handling these requests sits in the *ServiceClass* of the *RestController*. The *ServiceClass* uses *Repositories* to retrieve *Entities* from the database and *FeignClients* to make requests to other services. The results of handling a request are then returned back to the *RestController* where *Data Transfer Objects (DTOs)* are used for the HTTP response. This structure is expanded to multiple *RestController* classes in the actual services.

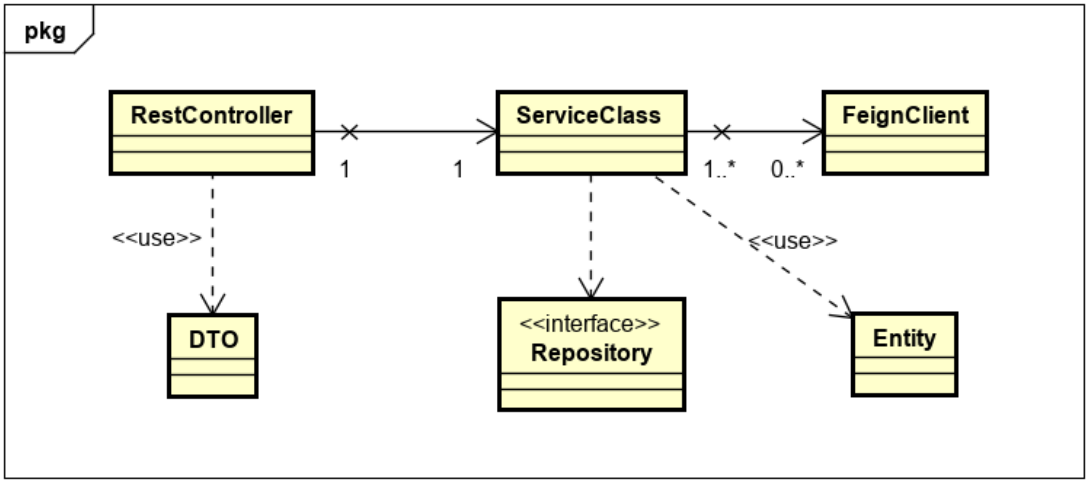


Figure 6.17: Basic schema of the customer and developer web service

6.8.2 Customer Web Service

The Customer Web Service is the back end of the customer part of the VIPER website. It provides all functionality necessary for managing customer accounts, their settings and payment methods.

Customer Database As detailed in subsection 6.3.3, the Customer Web Service uses a MongoDB database. The architecture of this database is shown in Figure 6.18. The *User* is the main entity of this database design. It has *User Settings* and *Payment Methods*. Only the name, type, authentication method and authentication code of a user's payment method are stored at the Customer Web Service. A reference to the payment account, which is stored at Braintree, is kept at the Braintree Service (see subsection 6.7.7).

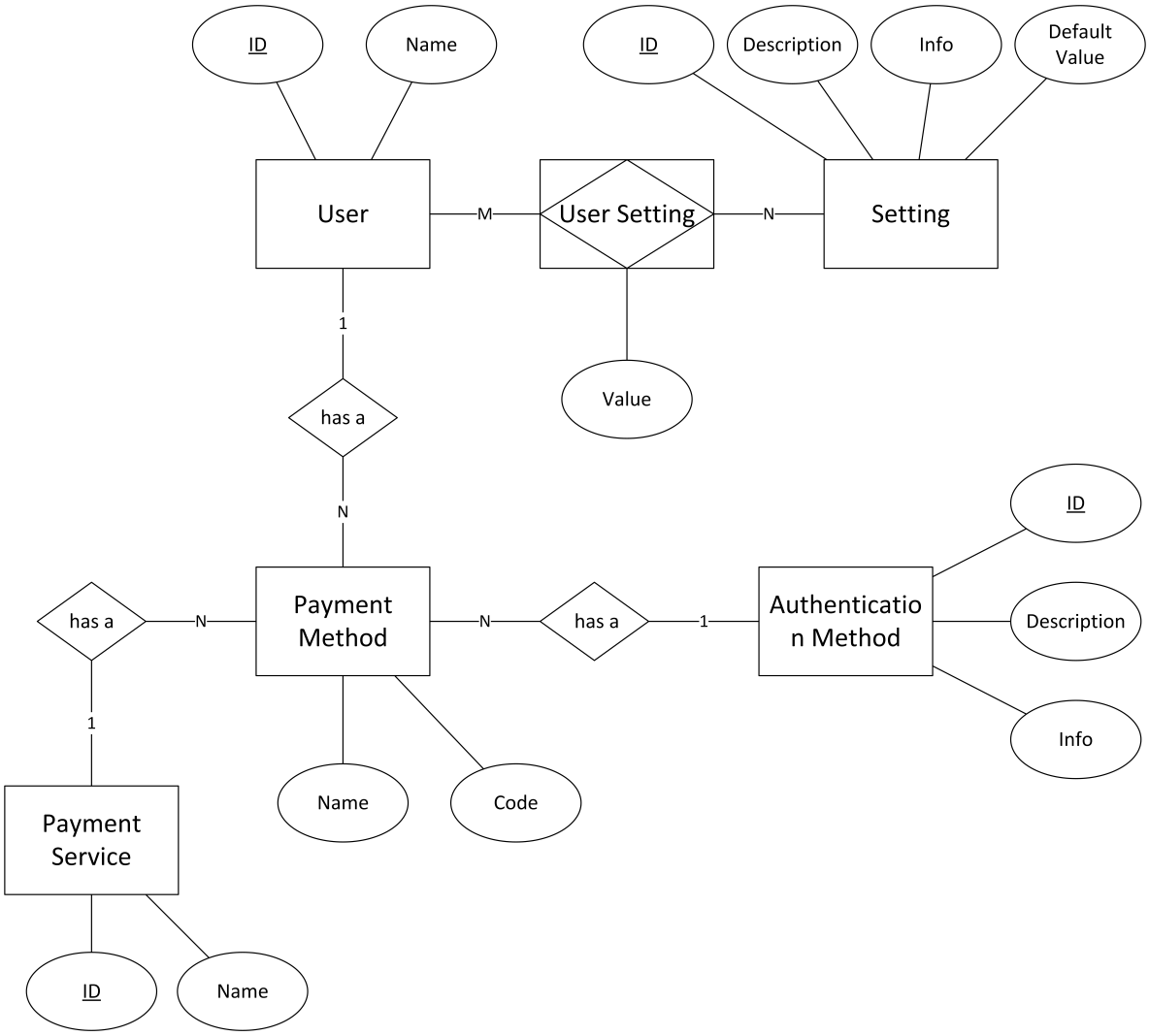


Figure 6.18: ERD of the customer web service's database

Using HTTP Status Codes With Feign For requests to the Authentication and Braintree Service, HTTP status codes are used to determine the success of the request. To use status codes the return type of the Feign Client request has to be *feign.Response*, as already mentioned in 6.4.2.3. The status code of a *feign.Response* can be retrieved using the method *feign.Response::status*. To convert the *feign.Response* object to an usable *Response* object the code from Listing 6.56 was written. It takes the response body of the *feign.Response*, converts it to a string and then uses a *Jackson ObjectMapper* to map the values of this string to a *Response* object.

```
1 public Response convertFeignResponse(feign.Response response)
      throws IOException {
2   String responseBody;
3
4   try (InputStream inputStream = response.body().asInputStream
        ()) {
5     responseBody = IOUtils.toString(inputStream,
          StandardCharsets.UTF_8);
6   }
7
8   ObjectMapper objectMapper = new ObjectMapper();
9   return objectMapper.readValue(responseBody, Response.class);
10 }
```

Listing 6.56: Conversion of a feign.Response to a Response object

Payment Method Security Every payment method of a user is protected by an authentication code. How a user enters this authentication code is not governed by the back end. Both the website and VR or AR applications can decide on the way the user has to enter the code (PIN, pattern, etc.), but both have to be equal. Before sending it to the back end the code has to be converted to a string. The code is then hashed using the Scrypt hashing algorithm, as detailed in subsubsection 6.5.3.2. The code snippet in Listing 6.57 shows how the code is encrypted and how it is verified. It is important to never compare the provided code with the one stored in the database using *String::equals*, because the Scrypt hashing algorithm produces a different output string for the same input, every time it is executed. Instead, the *SCryptPasswordEncoder::matches* method has to be used.

```
1 // Hash customer authentication code
2 passwordEncoder.encode(code);
3
4 // Verify customer authentication code
5 passwordEncoder.matches(code, user.getPaymentMethods().get(
      account).getCode());
```

Listing 6.57: Hashing and verification of a payment method authentication code

6.8.3 Developer Web Service

The Developer Web Service is the back end of the developer part of the VIPER website. It provides all functionality necessary for managing organizations and their developers and applications.

Developer Database The Developer Web Service uses MongoDB as a DBMS (see subsection 6.3.3). The service's database architecture is shown in Figure 6.19. *Developer* and *Organization* are the main entities of this database. Every *Developer* belongs to one *Organization*

and has *Developer Settings*. For *Organizations* the name, email, website and address is stored. *Organizations* have *Applications*, which are actual applications of the organization that are registered to use the VIPER payment system.

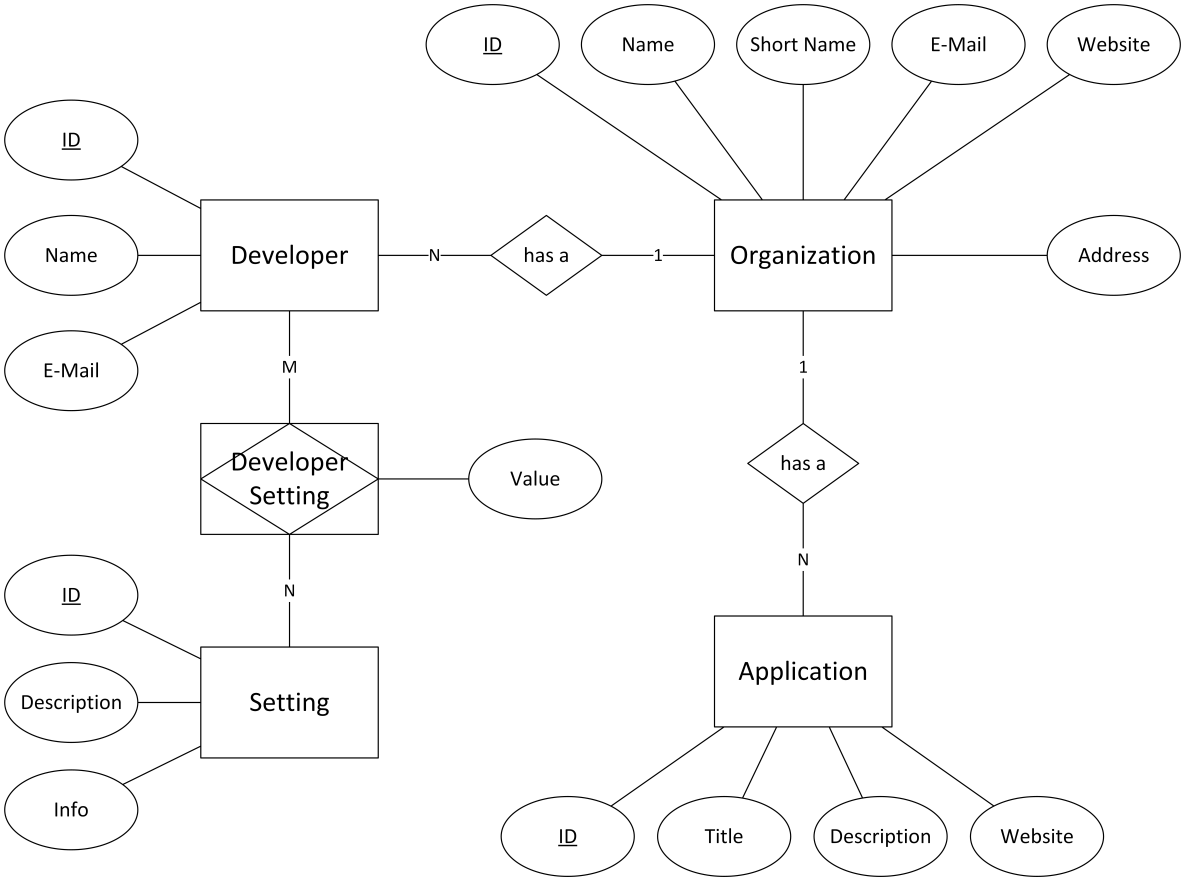


Figure 6.19: ERD of the developer web service's database

Developer Web Service API Design As the there are multiple roles within an organization, the API of the Developer Web Service is different from the Customer Web Service. The Developer Web Service has many endpoints that can only be accessed by developers with admin privileges. These include the endpoints for changing organization and payment information as well as endpoints for managing the developers of an organization. Different from customers, developers who are not admins cannot change their own information. As the main task of normal developers is to develop applications and not to manage an organization, the VIPER back end does not allow them to do so.

6.9 Cloud Infrastructure

“Cloud computing is the on-demand delivery of compute power, database storage, applications, and other IT resources through a cloud services platform via the internet with pay-as-you-go pricing” [6]. In other words, a cloud provider owns hardware, which is accessible via the internet, and offers the purchase of usage rights. The big advantages of using a cloud service over a self hosted infrastructure are [6]:

- You only pay for what you use and do not have to maintain the infrastructure.
- It is cheaper, since the infrastructure is built at a large scale and thus the economic principals of mass production apply.
- The infrastructure scales to needs of the services.
- Processes and software is standardized and preconfigured to work easy and fast. Also support is available.

There are also a few downsides when using a cloud-provider:

- Big target for cyber-attacks; if the provider is attacked, the whole infrastructure may suffer.
- Dependence on outside services, that cannot be fully controlled.
- Data security and eavesdropping concerns.

6.9.1 Cloud Service Providers

6.9.1.1 Requirements

A cloud-provider has to supply machines accessible in Austria and preferably with a Linux OS, for performance and licensing reasons. Furthermore, dynamic scaling and data storage has to be available. Deploying containerized applications also must be supported, because containers are used to deploy our applications, as described in section 6.10. The micro-services require little resources, since they are meant to be as small as possible and be duplicated on demand for more throughput. In Table 6.2 estimated cloud requirements are listed.

Service type	RAM (Gb)	CPU (cores)	Storage (Gb)	Network (per day)	Required instances
core-service	1	1	10	<1 mb	1-3
composed/infrastructure-service	1	1	10	1 mb	4-10
edge-service	2	2	10	2 mb	2-4
other (e.g Website, Gitlab)	4	2	100	<10 mb	2-5

Table 6.2: Estimated required resources for the services.

The cloud infrastructure is required from October 2018 until the end of the project in March. This means that the services have to run approximately 6 months.

6.9.1.2 Choice

The VIPER team was able to securer a sponsorship from A1 digital, an Austrian consulting company [43]. A1 digital is the parent company of Exoscale, a Swiss cloud provider, that fits all requirements stated above [54]. The sponsorship of A1 digital was in the form of 5000$ worth of Exoscale credits. Exoscale offers multiple instance sizes, three of which are aligned with requirements for our services. The *Tiny* instance offer 1 Gb of RAM and 1 CPU Core, which fits the core-service and composed-service requirements and cost 11$ per month per instance. *Small* instances fit the edge- and infrastructure-service needs with 2 Gb of RAM and 2 CPU cores, costing 21$ per month per instance. *Medium* instance should fit all other purposes, like the website-server and Gitlab, with 4 Gb of RAM and 2 CPU Cores, costing 42 $ per month per instance. Since the internal network communication is free-of-charge, only the edge-services and non-service instances need to considered in network concerns. Exoscale charges 0.02$/Gb for outbound Internet usage and 1$/Gb-month for Disk-space. All of this amount to an approximate monthly cost of 300$ for running all services during the project runtime. For the whole project span this is an expected cost of 1800$. Since the sponsored amount of 5000$ is more than double the expected cost, Exoscale was chosen as the cloud provider for this project. As of April 2019 the project consumed 1390$, which is below the expected usage cost.

6.9.2 Exoscale

Exoscale has server infrastructure deployed in Switzerland, Germany and Austria. Server instance can be created in any of the above countries, but only instance deployed in Austria where used for this project. The instances that have been deployed primarily ran *Ubuntu 18.04 64-bit*, because all team members are familiar with Ubuntu and no OS specific functionality is required. *S3* object storage is also provided by Exoscale and was used to store website assets, cache CI related data and store docker-machine keys (see subsubsection 6.10.1.2). The Domain Name Service (DNS) service Exoscale provides was used for the website, the edge services and the gitlab instance (see subsection 6.9.3). [54]

6.9.3 Version Control

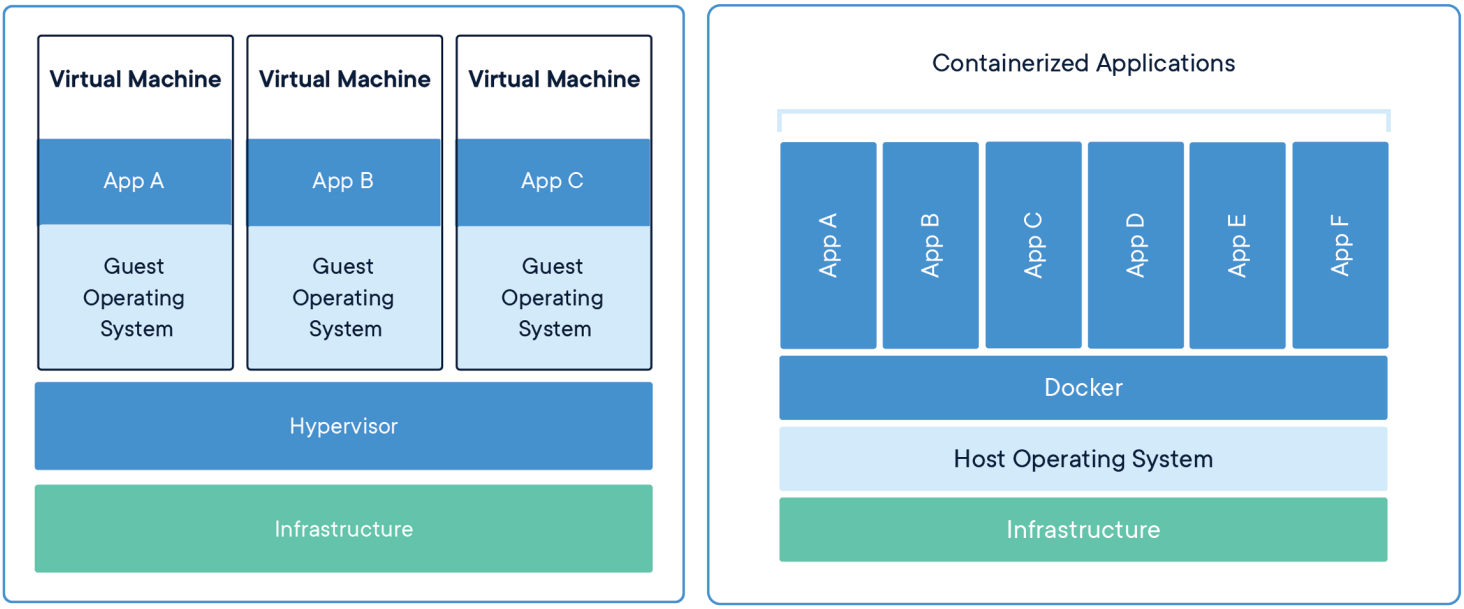
For version control a private instance of Gitlab, a free tool for managing Git-repositories, was setup on Exoscale. Individual repository for each microservice, demo and the website were created. The microservice repositories all include gitlab-ci and docker configuration files and a *service* folder, which includes the maven files and the service source code.

6.10 Service Integration and Deployment

To actually be able use the VIPER services, they have to be deployed on the cloud infrastructure. This means that the programs have to be executed on a server and be assigned to an IP address. Because every service and other program that needs to be deployed, for example the website, depends on different libraries this can become difficult to do and may require specific procedures for every service. Furthermore, the services also have their own databases, which also need to be deployed. In order to avoid these problems a new level of abstraction for the application is used, which is explained in the following section.

6.10.1 Containerization

The term containerization comes from the freight transport world and refers to the usage of containers to handle cargo objects. Loading ships, trains or other vehicles with cargo was a difficult process in the past, since the items had very different shapes and sizes and thus required special handling. In the early 1900s containers where used to transport freight to avoid these problems. Containers provide a fixed size framework to work with and thus all items can be handled the same. In this way all operations associated with the transport of freight can be optimized to handle predefined containers and thus improve performance. In recent years software developers have adapted this concept to handle software. Instead of handling software and dependencies on their own, the software is packed inside a container, in this case a type of virtual machine. Instead of specific deployment procedures for every software, the same procedure can be used for all deployments [105].



(a) Virtual machine diagram.

(b) Containerized VM diagram.

Figure 6.20: Comparison of Virtual machines and containers [45].

As already noted by Mouat, containers are different from traditional virtual machines [105]. Virtual machines run on a Hypervisor and each machine has its own operating system and software installed (see Figure 6.20a). In contrast, a container system shares the virtual operating system and only the software for each machine has to be installed, thus reducing the required size significantly (see Figure 6.20b). This reduces the overhead for the underlying OS and increases the overall performance and workload capabilities.

6.10.1.1 Docker

Docker is a containerization framework that is based on existing Linux virtualization technologies [47] . The definition for a docker container is written in a [Dockerfile]. an example can be seen in Listing 6.58. It follows a simple *command*, *arguments* schema. At the beginning the operating system required is defined with the *FROM* command. Not only operating systems can be used here, but also existing containers on which this image is to be based. The name of a container is always succeeded by a tag, identifying a version or in the special case of [latest] the most recent version. To build and mange containers, the Docker Command Line Interface (CLI) is used. For further command and CLI references see the official docker documentation [46].

```
1 FROM ubuntu:15.04
2 COPY . /app
3 RUN make /app
4 CMD python /app/app.py
```

Listing 6.58: Example docker file

Microservice Containers
All micro-services are similar, since they are all based on Java Spring, which is deployed via maven, and most of them also need a corresponding database.

```
1 FROM maven:3.6-jdk-8-alpine
2
3 EXPOSE 8000
4
5 ADD ./service /tmp/service
6
7 ENTRYPOINT ["mvn","spring-boot:run","-f","/tmp/service"]
```

Listing 6.59: Typical docker file for a microservice

Most configurations look similar to the ones show in Listing 6.59. The base container *maven:3.6-jdk-8-alpine* was chosen, because it already includes maven, which is required

for running the service. The services are exposed to different ports, depending on their requirements. This Dockerfile is located inside each microservice repository and thus copies the *service* folder into the container. When executing the container, the command `mvn spring-boot:run -f /tmp/service` is executed to start the service inside it.

Almost all services use MongoDB as a database. The database could be installed directly inside the services and thus avoiding having multiple services, but this would entangle these two components. This means that these two parts always have to be deployed simultaneously, which is unnecessary, if only one of them has changed. Furthermore, this would introduce problems that micro services are meant to prevent and thus it was decided to have the database as a separate container. Having two or even more containers again introduces the problem that every service cannot be treated the same and thus nullifies the benefits of containers. Docker-compose was created to overcome this situation. With it multiple service can be defined one file, share resources between them, have virtual networks and most importantly handle the service as a single predefined unit. The definition is written in YAML and stored in a *docker-compose.yml* file, which is required inside every microservice repository. An example config can be seen in Listing 6.60. To deploy the service, `docker-compose up` has to be called on a server. For more details see the official docker-compose documentation [44].

```
1 services:
2   braintree:
3     build: .
4     restart: always
5     ports:
6       - "8000:8000"
7
8   mongo:
9     image: mongo:4.0.3
10    restart: always
11    ports:
12      - "27017:27017"
```

Listing 6.60: Typical docker-compose file for a microservice

6.10.1.2 Docker Machine

To deploy a service one last step remains, namely getting the service on a server in the infrastructure. Either the whole repository could be sent to a server and the container built there or build the repository somewhere else and just send the container to the server. Since the latter option requires less bandwidth, is better suited for scaling, since no rebuilding is required, and only includes the necessary parts of the service it was chosen by us. Docker provides a tool called *docker-machine*, which install the Docker Engine on virtual hosts and thus allows execution of docker containers inside them. Also many cloud providers are supported, including Exoscale.

Docker-machine enables the creation and management of instances inside Exoscale. To create a instance, the Exoscale API keys are required. Furthermore, it is possible to set the preferred availability zone, disk size and instance size. When executing the command returned by `docker-machine env name`, the local docker Engine is bound to the docker-machine and thus all docker or docker-compose commands are executed there. When creating a machine, docker-machine saves the meta data and ssh keys inside the */.docker/machine* directory. Unfortunately, docker-machine does not provide a mechanism to synchronise this configuration across machines. Thus it was decided to synchronize the folder via an Exoscale S3 bucket. This is done via a tool called *s3cmd*, which connects to a S3 bucket and can perform file and synchronization operations. All these steps were put together inside a bash script that can be seen in Listing 6.61. This script takes two arguments, the name of the service, which will also be assigned to the docker-machine, and the wanted instance size in Exoscale. By default the instance size is *micro* (line 2). Before creating a docker-machine, the existing config is gathered from the S3 bucket (line 4). If a docker-machine with the name already exists, there is no need in creating a new one, since the services will be overwritten anyway by docker if they have the same name. If no machine with the given name exists, a new one is created and afterwards the newly created docker-machine configuration is synchronized. All of this is done in line 6. Afterwards the environment is bound to the now existing machine and docker-compose is used to deploy the service.

```
1 #!/bin/sh
2 SIZE=$2 || micro
3
4 s3cmd sync s3://viper-ci/machine/ ~/.docker/machine/
5
6 docker-machine ls | grep $1 || docker-machine create ... $1 &&
      s3cmd sync ~/.docker/machine/ s3://viper-ci/machine/
7
8 eval $(docker-machine env $1 --shell bash)
9 docker-compose build
10 docker-compose up -d --force-recreate --no-build
```

Listing 6.61: Deploy script

Alternatively other tools could be used, for example Kubernetes, which is better suited for automatic scaling and microservice management. It was decided to use docker-machine, because all team members were already familiar with this tool and implementing Kubernets or other tools within the infrastructure would have taken too long for this project.

6.10.2 CI/Continous Delivery (CD)

CI/CD refers to the process of merging the code to one central location and deploying it. This is done to allow fast updates to the production environment and automatic deployment. In the CI phase the code is tested and checked to ensure no bugs are introduced to the production

system. The actual deployment is associated with the CD phase [22].

In this context CI is responsible for building the services and testing them. After successful testing, the deployment script can be called to cover the CD phase. These steps could be implemented on any machine, preferably a server, by simply executing the containerization and deployment steps listed before. This process could either be triggered by an event, like a push to a repository, or be executed periodically, for example every night or hour. There are many CI/CD system available, which offer different possibilities. Jenkins is one of the most popular open-source tools. It runs on a on server and executes all steps there, allowing for great customization and execution management [82] . TravisCI is also a very popular tool, which runs on its infrastructure. It mainly supports Github repositories and has to be paid for, in order to use it for commercial products [161]. It was decided to use GitlabCI, because it is already available inside the existing Gitlab infrastructure and free to use.

6.10.2.1 Gitlab-CI

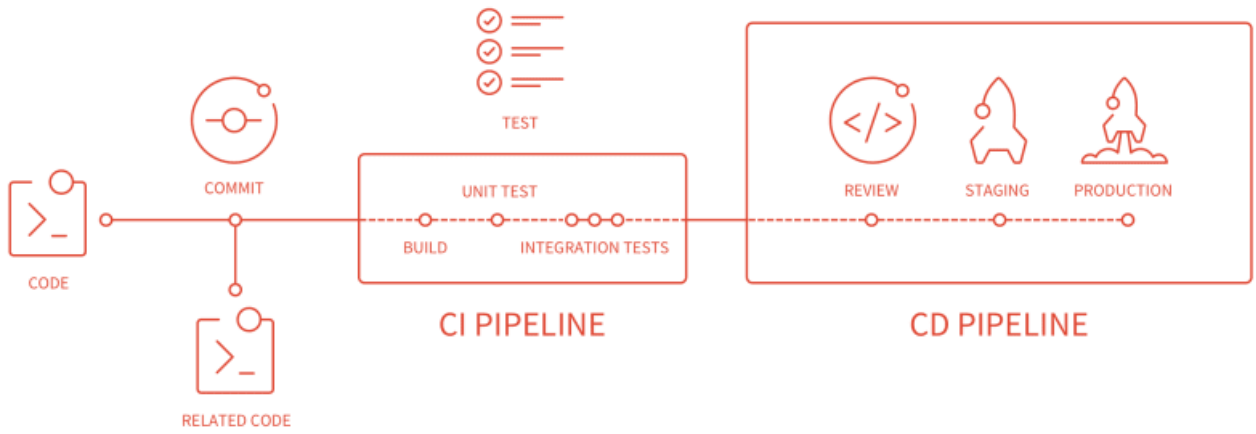


Figure 6.21: Diagram of Gitlab-CI [64].

Gitlab-CI is a CI/CD system provided by Gitlab. To set it up, the instructions on the official Gitlab website were followed. Each repository that wants to use GitlabCI needs a *.gitlab-ci.yml* file. The file defines in which docker-image the CI process will take place and which steps are required to completed the CI process. The steps are defined by commands to be executed inside the container and if any command does not return 0, the CI process fails. The instructions for CD are also the be represented as a step in this file. The whole flow from commit to deployment can be seen in Figure 6.21. For more details view the offical Gitlab-CI documentation [65].

Since most service have similar setups, a custom docker-image on which the CI configuration will be based was created. A new repository containing the deployment script, a configuration file for the S3 connection and an additional docker-compose file for the moni-

toring services discussed in section 6.11 was created. The Dockerfile, based on a standard docker image, for the machine copies all of these file inside the container, installs necessary software like docker-machine, docker-compose and the *s3cmd*. Gitlab-CI configuration was also included, which builds the image and stores it in the Gitlab docker registry to be accessible everywhere. A typical microservice Gitlab-CI configuration can be seen in Listing 6.62. It uses the described docker image from the registry, starts the docker daemon and only needs to call the deploy script to execute all further steps. This reduces the required configuration for each service significantly and is independent of the actual CI/CD implementation.

```
1 image: docker.viperpayment.com/viper/ci-docker:latest
2
3 services:
4   - docker:dind
5
6 build:
7   stage: deploy
8   script:
9     - deploy-service SERVICE-NAME small
```

Listing 6.62: Microservice .gitlab-ci.yml config

6.11 Monitoring and Logging

Working with microservices adds an additional layer of complexity. The distributed and self-contained nature of microservices makes it hard to trace errors and monitor the whole system. Monitoring microservice architectures is a great challenge by itself and thus will not be covered in detail here.

Logging
Logging is a crucial aspect of motoring, since it shows what each services does and what causes an error. The Elastic-stack offers solutions like Logstash that store all the logs centralized and offers analytic tools to gain insights [50]. Since it is time consuming to setup such a system, a logging infrastructure was not actually implement. Instead the the docker logging functionality was used to view the live logs of the services. This is currently feasible, since we only have a fair amount of micro-services. For future purposes a logging infrastructure and analytic tools are needed.

Service
Monitoring also includes recording the system performance and detecting hardware failures to ensure high-availability. To implement this, the existing Github repository *dockprom* was used, which uses different technologies for monitoring [128]. It uses *Prometheus*, an open-source monitoring solution, as the main framework [129]. To analyse the performance and resource usage of docker-containers *cadvisor* is used [66]. The information is displayed in a *Grafana* dashboard and shows the metrics of individual containers [68].

The *dockprom* repository unifies all these technologies using docker-compose and thus can be used without further configuration. To actually show any data, the docker containers need to send data to the *dockprom* service. This is done by defining the docker-compose services *node-exporter* and *cadvisor*, as described in the *README.md*, in a file called *monitor-compose.yml* inside the *ci-docker* repository on Gitlab, described in subsubsection 6.10.1.2. That way, it is possible to simply use the complex definitions inside any microservice that needs to be monitored by including the definitions shown in Listing 6.63 as docker-compose services.

```
1  nodeexporter:
2    extends:
3      file: ~/monitor-compose.yml
4      service: nodeexporter
5    ports:
6      - "9100:9100"
7  cadvisor:
8    extends:
9      file: ~/monitor-compose.yml
10     service: cadvisor
11   ports:
12     - "8080:8080"
```

Listing 6.63: docker-compose services for monitoring

6.12 Testing

Testing the back end is crucial to ensure it works as expected. The testing is divided into different foci, namely database testing and integration testing. Unit testing is not useful for microservices, since this only creates a new kind of test that need to maintained and nearly no actual functionality is implemented inside a non core microservice, which is the majority. Thus instead of unit testing the Integration testing is applied for each service on its own interfaces.

6.12.1 Integration Testing

Testing the ReST interfaces and the business logic of the services is crucial to ensure they work as expected. Testing the business logic within a service and functionality that requires the collaboration of multiple services is indistinguishable on a interface level and thus does not make a difference testing-wise. The Spring Framework offers many different testing functionalities [145]. An example test implementation can be seen in Listing 9.1, where a *GET* request is sent an the response is tested.

Testing a microservice infrastructure is difficult since a single service can depend on multiple other services, which also have dependencies. The only possibility to test a microservice and ensure all dependencies are available is to make an exact copy of the live system. This testing step is called staging and requires the same infrastructure as the live system. To test a service, it is deployed in the staging environment and the test case are executed and results recorded. Since this setup is more complicated and requires way more infrastructure which also costs money, staging was not implemented in this project. The testing was mostly done in the production environment, which is a very bad practice, but since this project is only a prototype and the time constraints did not allow for a sophisticated setup it was applied.

Integration testing for payment is difficult, since many parts are involved that are outside the VIPER infrastructure. Furthermore, it takes a few days for a transaction to be settled and thus testing it would require asynchronous testing which may take multiple day and thus reduce the agility of the system. For this project it is necessary to assume that the payment services work correctly and the transactions are settle correctly. What can be tested is the response of payment API calls. Braintree offers testing functionality for their services and thus the error handling and different unforeseen payment scenarios can be tested. [23]. This mostly applies to authentication errors or insufficient fund exceptions.

6.12.2 Database Testing

All database tests are designed to be regression tests, ensuring that recent changes to the database did not break any existing code. As the databases are only accessible from within the microservice they belong to, no external database testing tools can be used to run the database tests. Thus, the *Spring Boot Testing Framework* and *JUnit* were used [146]. The basic setup of a Spring Boot Testing class is shown in Listing 6.64.

```
1 @ContextConfiguration
2 @RunWith(SpringRunner.class)
3 @SpringBootTest(classes = AuthenticationApplication.class)
4 public class DatabaseTest { ... }
```

Listing 6.64: Spring Boot Testing class

All test cases are implemented inside this class and have to be annotated with *@Test*. The basic structure of these tests is to create a new entity, write it to the database, retrieve it again and/or modify it in some way, to check if the final entity looks as expected. To ensure that all tests are run under the same conditions and results are repeatable the database is cleared before every test case, using an *init* method annotated with *@Before*. This means that the database tests must not be run in a production environment as all data will be lost. If these test are run automatically during the testing stage of a CI deployment, it has to be ensured that a separate testing environment is created for performing the tests.

6.13 Client Library

The client library is an integral part of the VIPER payment system as it allows developers to use VIPER with minimal development effort.

6.13.1 Programming Languages and Technologies

Requirements The technologies used to implement the client library have to allow for the development of a library, which can be included in a wide variety of applications and platforms. As the vast majority of VR and AR applications are developed using 3D graphics engines, special focus has to lie on them.

The Programming Language Many of the most popular 3D engines support the use of either C++ or C#, which both support C libraries (list of the most popular game engines [155]). C libraries are also supported by other popular programming languages like Java (with JNA [81]) and Python (with ctypes [35]). C libraries, which can be implemented using C++, are not only support by a wide variety of popular programming languages but also on all popular platforms. They can be compiled to a *.dll* for Windows and to a *.so* for Linux and Android. Thus, C libraries and the C++ programming language fulfill all requirements stated above and were chosen for implementing the client library.

C++ HTTP-Client Libraries The client library has to make HTTP request to the VIPER back end. As C++ has no built-in HTTP library a third party library had to be used. Two popular C++ HTTP libraries are POCO [124] and the Microsoft cpprestsdk [98]. Both libraries provide all needed functionality. The cpprestsdk library has two advantages over POCO: it is more lightweight and has a NuGet package for Android. Using this NuGet package with Visual Studio significantly simplifies the development process of the Android application as no dependencies have to be managed manually. For these reasons Microsoft's cpprestsdk was used for the client library.

6.13.2 Client Library Implementation

The core part of the client library containing all the business logic was written in C++ and can be used for any platform. The interface, over which the client library function are called, had to be written twice. Once for the Windows library (.dll) and once for the Android library (.so).

Client Library Design The digram in Figure 6.22 shows the design of the client library's main part. The *Client* class contains all business logic required to make payments. It has methods for login and logout, for retrieving a list of available payment accounts and for making a payment. All of these methods have a synchronous and asynchronous version. As this client library is intended to be used in AR and VR it is important to provide asynchronous, non-blocking functions. With synchronous functions the UI or virtual environment would freeze for the execution time of the function, which can be multiple seconds in the case of the *make_payment* method. This would significantly impair the user experience of these applications. The developers of these applications could call the functions inside a new thread to resolve the issue, but this would contradict the basic idea of providing a client library, as it should reduce development efforts. Thus, asynchronous, non-blocking versions of the

synchronous methods were created in the client library. In the end it was actually easier to do it the other way around. The asynchronous methods were created first and the synchronous versions were built using them.

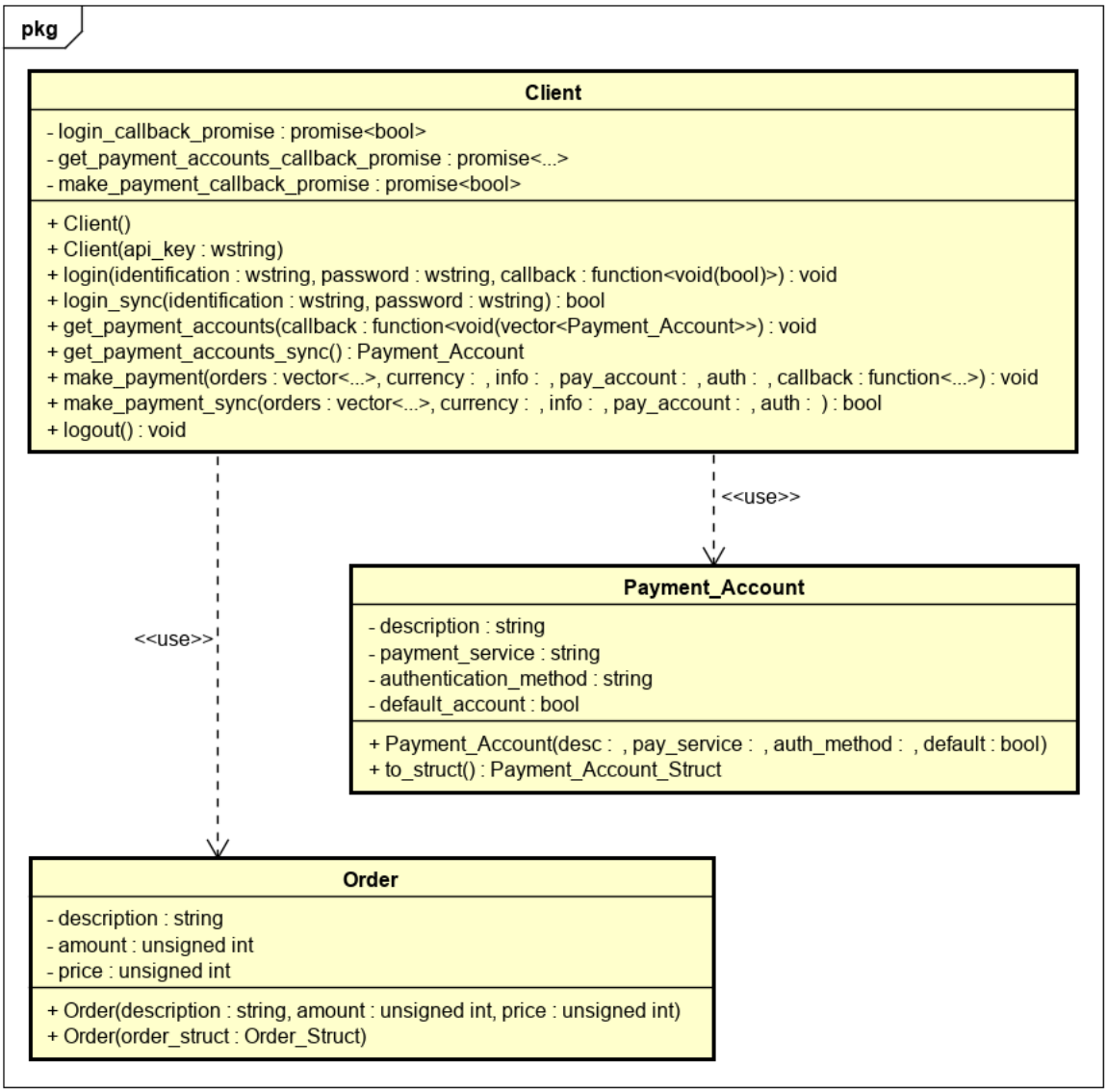


Figure 6.22: UML class diagram of the client library. Some data types are shorted or left out to improve readability

Using the cpprestsdk Library All HTTP requests with the cpprestsdk are built using asynchronous tasks. The basic structure of any HTTP request is shown in Listing 9.11, which can be found in the appendix. The most important parts of this structure are shown in Listing 6.65.

The call to the *request* function initiates the request. The following lambda function is executed when a response is received. Here, the success of the request is determined. Part of this is always to check the HTTP status code of the response. After doing so the received response is returned to the following lambda function. In this function the response

is processed and, in the case of an asynchronous method, passed to a callback function.

```
1 //Make request
2 http_client(L"https://api.viperpayment.com/path/to/endpoint").
     request(request)
3   .then([=](http_response response) -> pplx::task<
       http_response> {
4   ...
5   return pplx::task_from_result(response);
6 })
7 //Process response
8   .then([=](pplx::task<http_response> previous_task) {
9   ...
10  if (callback)
11    callback(...);
12 });
```

Listing 6.65: The most important parts of making a HTTP request with the cpprestsdk

Building Requests The *http_request* that is passed to the *request* function as shown in Listing 6.65, is built out of three parts: the request method, request headers and a request body. An example of this can be found in Listing 6.66. First a request body in JSON format is build. Then the *http_request* is created with a HTTP method (in this library only *method::GET* and *method::POST* were used). An *Authorization* header is added with the JWT received in the login request. At the end the previously created JSON request body is added.

```
1 //Building the JSON request body
2 json::value body;
3 body[L"property1"] = json::value(42);
4 body[L"property2"] = json::value(L"some value");
5
6 //Build HTTP requst
7 http_request request(methods::POST);
8 request.headers().add(L"Authorization", this->jwt_token);
9 request.set_body(body);
```

Listing 6.66: Build HTTP requests for the cpprestsdk

Synchronous HTTP Requests To create a synchronous version of an asynchronous method the whole method body could be copied into a new method and *.wait()* could be added to the end of the task making the whole request synchronous and blocking. But this code duplication would make the client library harder to maintain. Thus, C++ *promises* [153] and *futures* [152] were used to wait for the result of the asynchronous method, as shown in Listing 6.67.

First, a *promise* and *future* pair is create. The *promise* is the part that a value is written to and the *future* is the part from which this value is read. Then, an asynchronous method is

called without passing a callback. This tells the method to pass its return value to the *promise* instead of the callback, as shown in Listing 6.68. The *future::get* method blocks until a value is written to the *promise* and then returns this value.

```
1 //Create promise
2 this->get_payment_accounts_callback_promise = promise<vector<
    Payment_Account>>();
3
4 //Create future of this promise
5 future<vector<Payment_Account>> get_payment_accounts_future =
    this->get_payment_accounts_callback_promise.get_future();
6
7 //Call method without callback
8 this->get_payment_accounts(NULL);
9
10 //Wait for result
11 return get_payment_accounts_future.get();
```

Listing 6.67: Using C++ promises and futures to make an asynchronous method synchronous

```
1 if (callback)
2   callback(payment_accounts);
3 else
4   this->get_payment_accounts_callback_promise.set_value(
      payment_accounts);
```

Listing 6.68: Setting the promise value if no callback is provided

Chapter 7

Web Application

7.1 Introduction

A website is a very powerful tool to provide and distribute content. The International Telecommunications Union (ITU) estimated that the internet usage worldwide was about 51.2 % at the end of 2018.[79] That means that over half of the global population is currently using the internet and the market is still growing. A website can be accessed by almost every device. No matter if its a PC, a mobile device, a smart TV or a gaming console. Since not everyone who has access to the internet also has access to a PC, a website is the most promising tool for content distribution. Also, a website has the huge advantage, that it is not bound to any type of device and therefore after being created once, it can be accessed by all the mentioned devices. Considering this fact, a website is more affordable in development, because it only has to be created once and not dependently for each different platform.
The latest trend shows, that more and more companies publish their applications on the web, instead of creating platform dependent desktop applications or mobile apps. This has many advantages for companies. First of all, they save a lot of money since they do not have to buy any licenses to publish their applications in different stores. They also get effortless and affordable marketing that helps to rapidly grow their community. Secondly, it also saves a lot of time and resources because of the simple design flow and platform independence. Lastly, a website has the advantage, that it gets picked up and gets shown by search engines like Google or Bing.
In the case of VIPER an interface that communicates with the back end was needed, to enable companies to create their applications and add their payout account and also enable users to add their payment info. A big requirement was to also make it accessible for gaming consoles like the PlayStation or the X-Box because those are heavily used devices in the VR market. This made a website the only reasonable choice for creating a management tool for our services. Time was additionally a major concern since the tool had extensive requirements and was planned to be created by only one developer. Due to the extensive knowledge and prior experience in web development, it was the preferred way for creating the tool in the planned period.

7.2 Design

Design is an important part for a website. The structure has to be thought through carefully to ensure every part is align independently of the screen resolution. VIPER also has very specific requirements, since it targets two different audiences. On one hand the clients and on the other hand the organizations.

7.2.1 Material Design

“Material Design is a visual language that synthesizes the classic principles of good design with the innovation of technology and science.”[91]

As quoted above by Google, they created a way to construct a good design for even highly technical applications. This visual language was invented by Google in 2014 under the code-name "Quantum Paper". They tried to establish a common ground for all their application so that they would have a unique touch in every single one of their applications. Due to this, it would be possible for every user to immediately identify that this website is made by Google. Material Design is conceptualized, to provide a unified resource for all types of devices like smartphones, tablets, laptops or desktop PCs.
The name Material Design comes from a metaphor. The *Material* is designed to create a digital fabric in which the material responds naturally to the users actions. It is inspired by paper and ink to make it look more natural. This makes it more tactile and more connected with sense and touch. The main difference to real paper is, that it can be split, rearranged and moved when needed. This is primarily used to create responsive and screen independent designs. Through techniques like transitions, padding, depth or shadows it gets a way more natural feeling for the users and behaves like the user would expect it to. This creates a very unique, easy and at the same time natural flow.
To give the user visual hints on what he can do, shadows, edges, and dimensions are used. By providing each object with familiar tactile features, the user quickly gets cues on how to use this object. For example to show, that a Floating Action Button (FAB) is static throughout the pages, it gets placed independent from the current layout and a depth effect through a shadow, to show that it is floating above the current page and does not get changed. An example of a FAB can be found in Figure 7.1. Through these kinds of visual cues, the user instantly knows what which part of the application is doing.

Figure 7.1: Example of a FAB button

To even further improve the meaning of the design, animations are added. While animations do not interrupt the user experience, they strengthen the natural feeling the user has as the animations mimic nature. Additional motion cascades from touch points and visual feedback really help to connect the user even more. A good example of motion cascades would be

ripples. These are used to give the user feedback that he has touched an object. They can also identify where the user has touched the object. In the Figure 7.2 it starts a circle, which is propagating outwards at the point, where the user clicked the object. It is represented by the slightly transparent blue circle.

Figure 7.2: Example of ripples in a container

Material Design also has a lot of advantages compared to other commonly used designs like Flat-Design or Skeuomorphism. First of all, due to the fact, that this design concept was created by Google, there is clear guidance for developing Android, web and IOS apps. This fact also removes the guesswork when designing apps. So a unique design throughout all created applications could be achieved and all development steps, which are sensitive to errors, can be omitted. Material Design also brings a lot of animations into the app. It provides many in-built animations, which helps to cut development time and also removes troubles with animating.

7.2.2 Colors, Fonts and Icons

The color-scheme of a website is one of the most unique selling point. It leaves a permanent impression on the user so that these colors will always be associated with the website. The color choice is one of the most powerful tools in design. They can be used to attract attention, express meaning and earn a customers loyalty. Good color choices take careful planning to optimally influence the clients, alongside the layout and the many information the website has to offer. Colors help us to store and process the information, at which we are currently looking, more easily. This effect is essential when you want to increase brand recognition and help users to remember your service.

Deciding on colors for VIPER
After multiple color suggestions, the team decided to use teal as the primary color. Since teal is a mix of the colors blue and green, it represents the feelings and suggestions of both colors. Blue stands for trust, security, and calmness. It is important to trigger these feelings in the customers to show them that our platform is trustworthy and that we are handling their sensitive data with care. The color green on the other hand stands for excitement and

nature. Excitement is an equally important feeling to excite the users when they are using VIPERs services, to bind them even more and enhance loyalty. The nature aspect was also a perfect match for our logo, the snakehead, and our project name, VIPER. The secondary or accent color is pink. It was chosen since it is a little used color on the web. This is essential so we can create a uniquely identifiable web service and are deeply remembered by any customer. Furthermore pink is a refreshing color, which allows us to create a lightweight experience with maximized usability. The last color which was essential for our conceptualization of the material design is the warn color. Since it is normally defined and should not be changed, we chose a standardized deep orange. Despite the fact, that we already had every necessary color for the Material Design, the team decided to add 2 more colors to the palette (Figure 7.3). Two slight variants of our secondary and warn colors were chosen. From this palette, we derived every design and layout for all our applications.[94][95][96]

Figure 7.3: Color palette of VIPER

Material.io Color Tool

Googles website *material.io* provides many tools to create designs. One of them is called the color tool[92]. In this tool, the primary and secondary colors are entered and the tool gives you a general idea of how the designs will look like. Much more important is, that this tool also shows you how to use text on these colors and shows you a lighter and darker version of the provided colors. For example, if teal is entered as the primary color, it says that white is only ok if you use it in a large text because the text gets unreadable if it is too small. The example is contained in Figure 7.4. The size of the texts are defined by Google as follows: Large text is defined as 14 point (typically 18.66px) and bold or larger, or 18 point (typically 24px) or larger. Normal text is below 18 points or below 14 points and bold.

Figure 7.4: Colors in the *Material.io Color Tool*[67]

Typography and Iconography
In the designs, two main fonts were used. The first one is *Lato* and the second one is *Roboto*. Both are sans serif fonts, which means, that they do not have any extra strokes at the end of the main vertical and horizontal lines. Mostly these extra strokes are subtle but some are also clear and very pronounced. An example of a serif font can be found in Figure 7.5 and an example for a sans serif font can be found in Figure 7.6. Sans serif fonts are easier to read in web applications. Due to their almost uniform stroke width, the font stays readable when the resolution of the screen is reduced or the font size gets smaller. Whereas serif fonts are better when the user has to read large blocks of text since the serifs keep the eye in the line and help to stay focused.

Figure 7.5: Merriweather Font - Serif

Figure 7.6: Roboto Font - Sans Serif

To further increase the users experience on our applications, we used icons to help interpret the content we provide. This is because our brains are used to, and better trained to attain knowledge about images rather than texts or numbers. Icons help the users to quickly find the tab, content or information they are looking for and therefore greatly increase the usability and induce a natural flow into the users experience. Some icons like the 'I' for information, the house for a home button or the shopping cart for checkout are universally understandable images (Figure 7.7). So every user independent from culture or language is able to understand the proper meaning of this icon. This universality is great to minimize the information, which the users have to process to accomplish their specific goals.

Figure 7.7: Information, Home, Shopping Cart Icons

7.3 Technologies and Frameworks

To successfully create a website, a lot of different technologies are needed to accomplish the goal. Depending on the needs and goals of the website, there are multiple factors to consider and compare to decide what is needed. First of all, the main category are the controllers and

services, which help to create the logic and interactions of the website. They provide the main logic of the application and manage everything from displaying dynamic information to animating content or retrieving data from an API. Without any controllers and services, the website could just display static content and would be very pale. This makes them an essential part and therefore the right framework has to be chosen wisely.

7.3.0.1 Framework Comparison

There are many frameworks, libraries, and design principles, which help in creating web applications. Since there is so much to choose from, the selection should only be done of a detailed comparison of the main options. The goal was explicit from the start. The main management tool of VIPER should be a single page web application. This narrowed the selection down to three main ways of creating the application. The first one would be to write the complete page from scratch in native JS and the other two would be different JS frameworks called React and Angular. Every option has its advantages and disadvantages and therefore they have to be extensively compared and checked with the requirements of the project.

Native JavaScript
The first and most obvious reason on why to use native JS is performance. Since it is not dependant on any libraries or other unnecessary code it can compute most things way faster. This might be a good reason to use native JS, but the performance by far is not the biggest problem, it is keeping the UI in sync with the internal structure of your application. To compare native JS to the mentioned frameworks a small application which just creates a simple list of email addresses, the user provides, will be used. This example can be found in Figure 7.8.

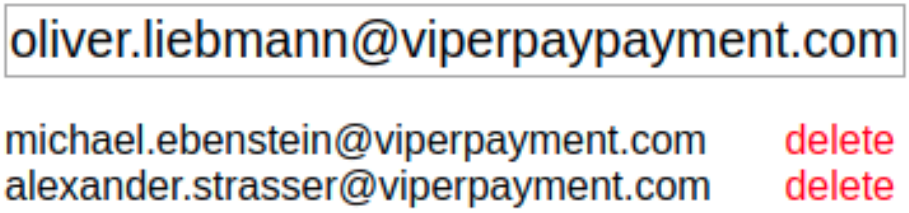


Figure 7.8: Snippet to compare frameworks

If one would try to maintain the example and just wants to render the list of items one would have to write it like in Listing 7.1.

```
const items = [
    'alexander.strasser@viperpayment.com',
    'michael.ebenstein@viperpayment.com'
];
const emailList = document.getElementById('emailList');
const ul = emailList.querySelector('ul');
for (let i = 0; i < items.length; i++){
    const email = items[i];
```

```
9       const li = document.createElement('li');
10      const span = document.createElement('span');
11      const del = document.createElement('a');
12      span.innerText = email;
13      del.innerText = 'delete';
14      del.setAttribute('data-delete-email', email);
15      li.appendChild(del)
16      li.appendChild(span)
17      ul.appendChild(li)
18  }
```

Listing 7.1: JavaScript example to render a simple list

Listing 7.1 shows how complicated it is to just render a simple list of items onto the UI. To now implement the functionality to create and delete single items it would be very complex to maintain the state of the UI every time an event occurs. Another problem with this approach is, that the code is fragile. In the case that another source of information, like an external server, is needed and an update occurs every 5 seconds. Every single item has to be compared to existing ones and the UI has to be updated at the same time, a significant performance dropout would emerge. In our case, when maintaining sensitive information like credit cards or pay pal accounts, it would under no circumstance be acceptable to have an UI which is out of sync with the information our application has. All these reasons make using native JS very unpractical and not applicable to our problem.

React

React is mainly a library but can also be used as a framework. It was created by Facebook in 2013 and gets maintained by Facebook, Instagram, and their community. The background of this framework was to create a dynamic UI with high performance, to enable users to use the Facebook chat and simultaneously receive news feed updates.
React uses a technology called Virtual Document Object Model (DOM). The structure of the DOM is explained in the subsection 7.5.1. With Virtual DOM an abstract copy of the Real DOM is created in memory so that every change is only created in memory until the UI gets updated. This abstraction layer makes updates fast and reliable and allows to build a highly dynamic UI. Due to the fact, that React was created by Facebook, a lot of their components can be reused to create your own applications. This takes away a lot of development time. Also because of Facebook it is open-source and gets constantly developed further and is open for the community.
“In case you didn’t notice we’re driving a car here with two flat tyres, the hood just flew up in front of the windshield, and we have no clue what’s going on anymore!”[80]
This quote by Michael Jackson and Ryan Florence, perfectly describes the development of React. Due to the swift development of React the environment changes frequently and rapidly. Developers have to regularly relearn the new ways of developing with React, which can be a hugh drawback. Furthermore, due to the fast development, the documentation is poor. Things often change and that makes it hard to precisely document the whole ecosystem. For example tools like Redux and Reflux should help developers to develop apps faster but in the and most developers struggle to integrate them into their apps. The poor documentation can lead to massive set backs in the development process.

To use React one has to learn JSX. It combines HTML with JS to help protect the code from injections, which is a huge advantage. Though most developers complain that JSX is a big disadvantage. It has a very steep learning curve and it needs a lot of time to get into the complexity of JSX. This again consumes a lot of time.
All these facts also make React a bad choice as the controller of VIPER.

7.3.0.2 Angular

Angular is another front-end framework, which was created by Google LLC. in 2016. It is the chosen framework for the website of VIPER. Angular helps to build interactive and dynamic single page application. It has easy ways to handle RESTful APIs and create HTML templates. Due to its big support through Google, it is always up-to-date and according to them it is under Long-Term Support (LTS). Angular also uses Typescript (subsubsection 7.3.0.3), which is another huge advantage.
The UI of Angular is defined by HTML as in any other website. All the logic is in separate classes and therefore independent of the websites structure. This helps when developing big applications since one does not have to make the decision on what has to be loaded first. After everything gets defined in Angular, the framework makes all the decision and optimizes the load time. This is essential since according to Amazon, every 100-millisecond improvement in page loading speed led to a 1% increase in revenue.[7]
Angular also has a modular structure. It organizes its code into buckets or modules. Those modules can be components, directives, pipes or services. This layer helps to organize the code and makes it easy to reuse chunks of code. Through these modules, Angular knows what is necessary to display the current page and can, therefore, lazy load the modules. Lazy loading is a technique to only load what is necessary and loads the rest on demand or in the background. Modules also help when developing in a team or just updating the application, since they can be exchanged without changing any other code of the application.
A major concern for the website was the communication to the back end API. Angular provides an very easy to use HTTP client. This made it effortless to develop the communication to the API and reduced the development time drastically.
Angular also provides an animation tool which makes it easy to create high-performance and complex animation timelines. Since animations make a website look very natural and interactive it is important that easy access to animations is provided. Most of the time animating takes up a large chunk of the development time and is therefore often omitted by many developers but with Angular, it is very intuitive and quick.
The framework is also very concerned about accessibility. Therefore they made it easy to create Accessible Rich Internet Applications (ARIA) enabled components and guides, developers, to create accessible applications. It also provides a built-in A11Y testing infrastructure. A11Y is a community-driven project to make web accessibility easier. They provide a lot of information on how to create accessible applications and Angular follows all these guidelines.[1]

Furthermore Angular provides the Angular-CLI. This tool helps to generate Angular projects, which are already pre-configured and can be used to start developing an Angular application. It also can generate the previously mentioned modules, so that they always follow best practices and also automatically creates tests for these modules. During the development, the Angular-CLI helps with linting and serving the app so that it automatically refreshes the page in the browser to immediately show the code changes live. Lastly, it provides a lot of

commands and tools to run extensive tests against the application.[9]

7.3.0.3 Typescript

TypeScript (TS) is an open-source object-oriented programming language, which gets developed and maintained by Microsoft. It is a superset of JS and it primarily adds static typing to the language, but it also adds classes and interfaces. All these additions help the many Integrated Development Environment (IDE)s to verify the code and prevent a lot of common mistakes. Through the static typing, for example, the IDE is able to recognize type errors during compiling and is, therefore, able to prevent runtime errors. TS can be used to develop both for client-side and server-side applications and is designed to create large applications. It transcompiles the TS code to JS code, since no browser can interprete TS code. Transcompiling is the process of translating one source code to the source code of another language. TS also helps to reduce the code. Since it is a wrapper for JS it has many helper classes and some code bits can, therefore, be reduced and be reused. Also through all the addition, the code gets easier to read, to maintain and to understand. This helps to cut the code because code bits are not rewritten when they can not be found in the overflow of unreadable code.

```
1 //TypeScript code
2 class Snake {
3     private name: string;
4     constructor (private name: string) {
5         this.name = name;
6     }
7     introduce() {
8         return "Hello, I'm a" + this.name;
9     }
10 }
11
12 //JavaScript code
13 var Snake = (function () {
14     function Snake(name) {
15         this.name = name;
16     }
17     Snake.prototype.introduce = function () {
18         return "Hello, I'm a" + this.name;
19     };
20     return Snake;
21 })();
```

Listing 7.2: Comparison of TypeScript classes to JavaScript classes

As one can see in Listing 7.2, the code of TS in contrast to JS follows a more standardized pattern of object-oriented programming languages. It defines class properties inside the class, whereas javascript only assigns the variable inside the constructor function. This can easily lead to properties that are forgotten, overseen or even misused. Also, the circumstance, that one has to work with the prototype of an object to add functions, makes the code very complex and unreadable. Furthermore, through the support of modules, it is very easy to

split the code into multiple files and make the code effortless to organize. This is an essential feature when building complex and voluminous applications. Since the compiler translates TS to JS, it can perform many code optimizations and speed it up, it also can follow best practices to create optimal code performance.

7.3.0.4 Libraries

For the implementation of the website, a number of libraries were essential. These libraries are further explained in this section.
The first library is *ngx-braintree* and it is used to add the payment options via Braintree. This was a crucial part of the project because it was the only library that allowed us to accomplish one of our main goal (the payments). A more detailed usage and the problems which arose when using it can be found in subsubsection 7.4.2.2.[113]
Next, *Moment.js* is an extremely powerful tool when working with times or dates. It can parse, validate, manipulate and display dates and times in JS. This library was essential to parse, analyze, manipulate and display all the statistics of the page.[101]
ng2-charts is an Angular wrapper for the *Chart.js* library. *Chart.js* is a simple and very flexible library to create all kinds of charts. It was essential to display all the statistics on the page. A more detailed explanation can be found in Figure 7.4.1.2.[112]
The library *angular2-image-upload* implemented a component which provides an interface to upload images to the server. It was used to enable the image uploads for profile and product pictures. The interface of this component is highly customizable and can show a preview of the uploaded picture.[11]
To parse the JWT, the *angular-jwt* library was used. JWT is a token, that stores all user information inside an object and is used to authenticate the users requests on the server. It can check if a token is already expired and extract the data from the token.[10]
The most essential library was *rxjs*. RxJS stands for Reactive Extensions for JavaScript. It is used to optimize the utilization of callbacks and asynchronous tasks. In every part of the application, where no immediate response was expected, RxJS was used to handle the callbacks. It also enabled the functionality to send HTTP request in one part of the application and evaluate the data in another part. RxJS works by providing an observable in one part, which can then be resolved in another part of the application. This made the code more readable and easier to handle.[132]

7.4 Architecture

In this section, the whole architecture of the application is explained. Particularly the setup of the website, the essential processes and the layout gets explained.

7.4.1 Interface

The interface of the website is split into three separate parts. The first one is the front page, which is just used to display information about the product and explain how it works for a customer and for an organization.

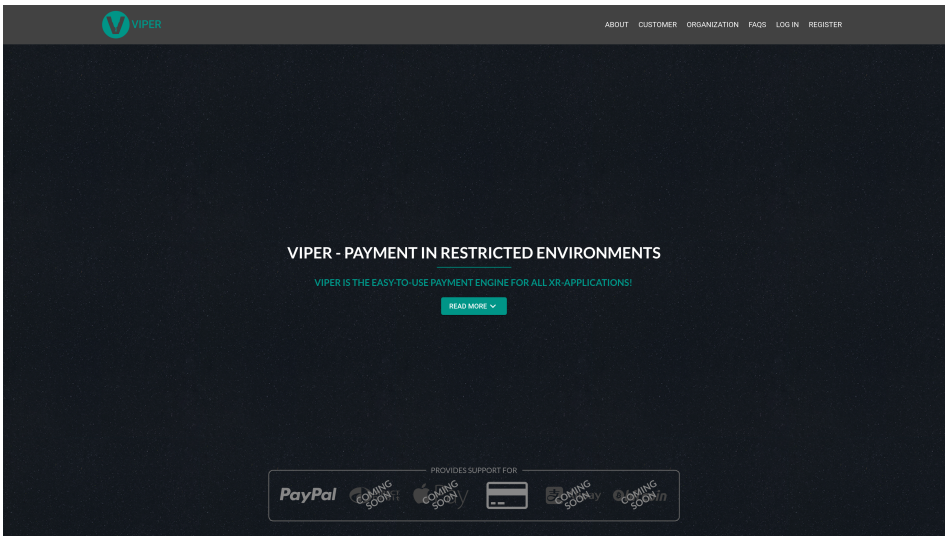


Figure 7.9: Image of the frontpage

The second part is the register/login form and the third part is the main application.

7.4.1.1 Register and Login Form

This part was again split into the developer and the client section. The concept was to create a small window in front of a background image with the login or register details. Above the form, the logo and the project name was displayed, to identify that the user is on the VIPER page as seen in Figure 7.10.

Login Form

This page is identical for both clients and developers, the only thing that changes is the heading. For developers it says 'Developer Login' and for clients, it says 'Client Login'. As one can see in the Figure 7.10, it automatically validates the inputs, so the user can only log in when all inputs are provided. When the user clicks the login button, the component method *startLoginProcess* is called. The code for the login process can be found in Listing 7.3.

```
1 startLoginProcess(values, valid) {
2   if (valid) {
3     this.userManager.loginUser(values.email, values.password,
          values.remember, 'ROLE_DEV').subscribe(() => {
4       this.router.navigateByUrl('developer/dashboard');
5         }, error => {
6           this.snackBar.open(error.message, 'OK', {duration:
              5000});
7         });
8   }
9 }
```

Listing 7.3: Code for the login process

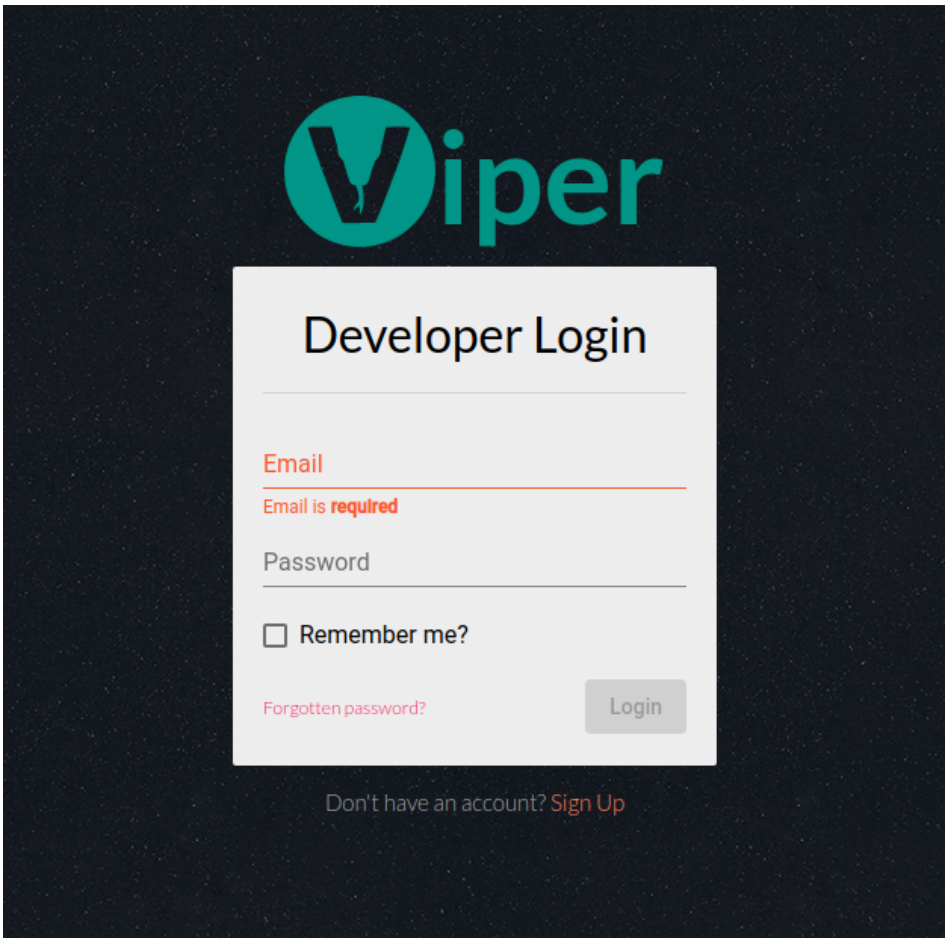


Figure 7.10: Image of the login page

This method first of all checks if the input the user provided is valid. Then the *userManager* is called and tries to login the user. The *userManager* is explained in subsubsection 7.4.2.1. The *loginUser* method of the *userManager* takes the email, password, if the user wants to save his email address and the role of the user. For the client, the role is *ROLE_USER* and for the developer, it us *ROLE_DEV*. This role is used to determine if the user has logged in on the right page. For example, if a developer tries to log in on the user page, he gets an error, which states that he is not authorized to access this page. If it was successful, the user gets forwarded to either the developer or the client page. If the request was denied, the credentials were wrong or the server could not be reached, a snack bar appears and states the error. The snack bar helps to keep the design of the page clean but still get the attention of the user and inform him, that an error occurred. This type of notification was used all over the application to notify the user.

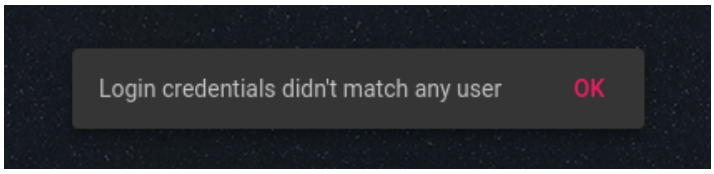


Figure 7.11: Snackbar to notify user

Register Form
The registration form for the client was a simple form with the name, username, email, and password of the client. After pressing the *Create Account* button, the same process as the one for the login form is carried out (subsubsection 7.4.1.1). The only thing that changes is the method of the user manager that gets called. Instead of *loginUser* it is *registerClient*.

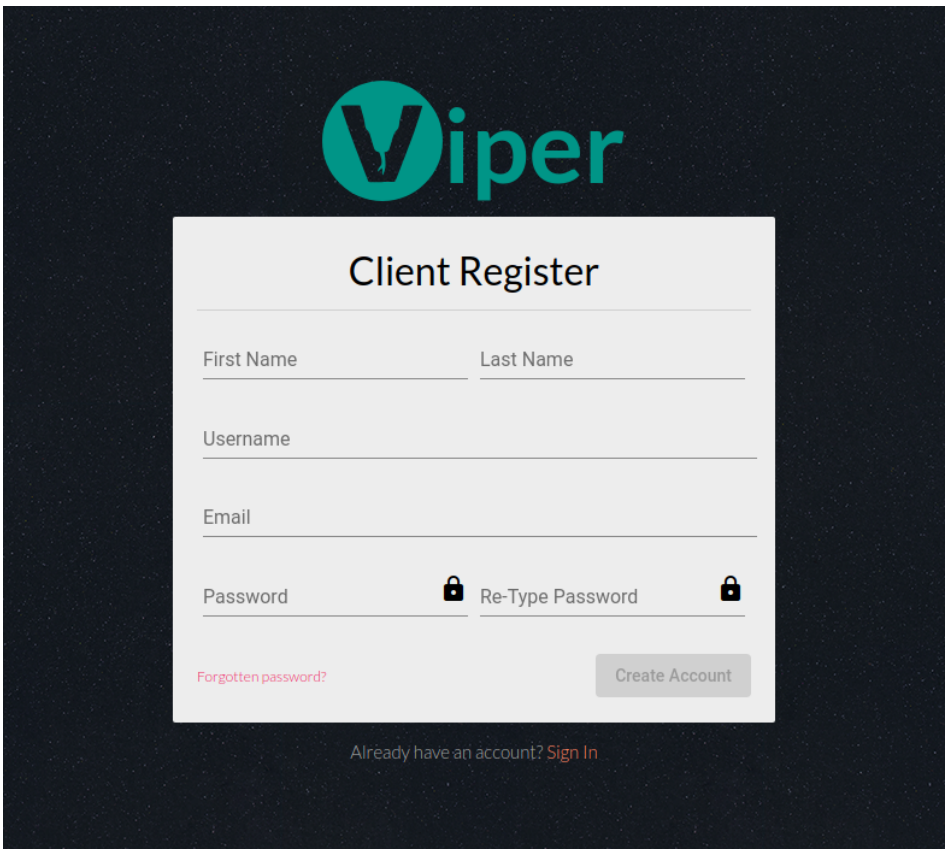


Figure 7.12: Client register form

For the organizations, the registration form was a lot more complex. Since an organization has to provide more information to register themselves, it was not possible to fit all input fields onto one page. The concept then was to create a stepper to lead the user through all the different steps and makes a seamless transition between the different categories of information. This helps to not repel the user due to the large amount of information he has to provide.

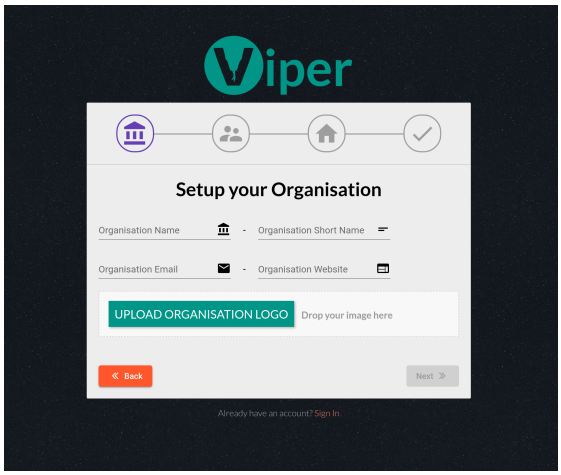


Figure 7.13: Developer register - Organization setup step

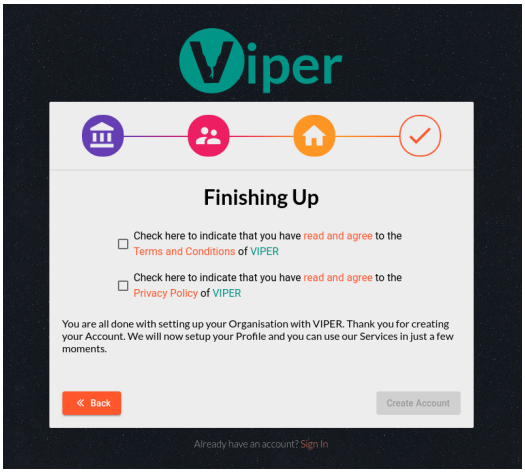


Figure 7.14: Developer register - finish step

To realize the stepper a register component and for each category, another component was created. The register component handles the transition from one category to the next. Furthermore, it processes the provided information and registers the organization. It also shows the progress through the icons on the top of the window. One icon and one color per category were picked to create the progress display. If the step is active the icon gets outlined with the color (Figure 7.13) and if the step is finished, the icon gets filled with the color (Figure 7.14). To track the step the organization is currently at, a variable called *flowPoint* was introduced. This variable determines, which parts of the stepper have to be displayed and further changes the progress display. To change the design of the progress display, 2 classes for every step were defined. The first one is the active class and the second one is the success class. For the second step these classes would be *.activeSecond* and *.successSecond*. The layout of these classes can be found in Listing 7.4. These 8 classes are then used to determine the current design of the progress.

```
%second-outline {
    border-color: $second-color;
    color: $second-color;
}
%second-fill {
    border-color: $second-color;
    color: #eeeeee;
    background-color: $second-color;
}
.activeSecond > {
    .icon:nth-child(2) {
        @extend %second-outline
        > mat-icon {
          @extend %second-outline
        }
    }
    .lineToNext:nth-child(1)::after {
```

```
18          background-image: linear-gradient(to right, $first-
                color , $second-color);
19      }
20  }
21  .successSecond > {
22      .icon:nth-child(2) {
23          @extend %second-fill
24          > mat-icon {
25              @extend %second-fill
26          }
27      }
28      .lineToNext:nth-child(1)::after {
29          @extend %second-fill
30      }
31  }
```

Listing 7.4: SCSS classes for the progress display

The *.active* class outlines the icon with the specific color. This is achieved by changing the *color* and the *border-color* properties, which switch the icon color and adjusts the color of the circle. The *.success* class fills the circle around the icon, makes the icon grey and colors the circle. These classes and variables are created for every component. After this has been implemented, the *flowPoint* decides, which classes are currently active and the progress display adjusts itself.
Every step has one input and two outputs and one setting. These Input/Output (IO) parts are used to communicate with the component. The setting **ngIf* is there to determine if the step should be displayed. In the example of Listing 7.5, the *developer-register-organisation-info* component is only visible when the flowPoint is currently at 1. The *(action)* output is used for navigation. If the user presses *Next* it outputs 1 and increases the flowPoint. If the user presses *Back* it outputs -1 and decreases the flowPoint. These changes of the flowPoint automatically update the displayed step. The input *[validateInfo]* is used to trigger the validation of all steps. If the validate variable is set to *true*, all components validate their data and output it via the *(validated)* output. This then adds the information to an object and tells the register component that it has successfully validated all data. If every component has responded with the validated data, the user gets registered with the same process as the user.

```
1 <app-developer-register-organisation-info
2     *ngIf="flowPoint===1"
3     (action)="navigate($event)"
4     [validateInfo]="validate"
5     (validated)="addInfo($event,'company')">
6 </app-developer-register-organisation-info>
```

Listing 7.5: Example of a step component call

7.4.1.2 Developer Pages

The main developer page consists of 5 pages. The dashboard, statistics, application, developer-accounts, and payment-accounts page. The main layout of the page consists out of three parts. The first part is the top bar. It is used to display the account information and the profile picture. When clicked a small menu is opened. It can be found in Figure 7.15 at the top right. This small menu provides some quick links for the user. The second part is the navigation bar on the left. This shows all the pages of the application, that the current user can access. It can be either viewed in the text view (Figure 7.20) or the icon view (Figure 7.21). The view can be changed via the collapse button at the bottom. This setting is only for user preference and to provide a clearer overview of the page. The third part is the main window. Here all the different pages are displayed. How this works is explained in subsection 7.4.3.
Due to time issues, the statistics endpoints are not yet created and are planned for the future. At the moment, only mock data is used.

Dashboard
The dashboard shows the statistics for the current month. It shows the revenue chart of the sum of all applications and a chart of the top 5 applications with the most revenue. How these charts work are described in paragraph 7.4.1.2 Furthermore it shows all applications and link to their specific statistics page.

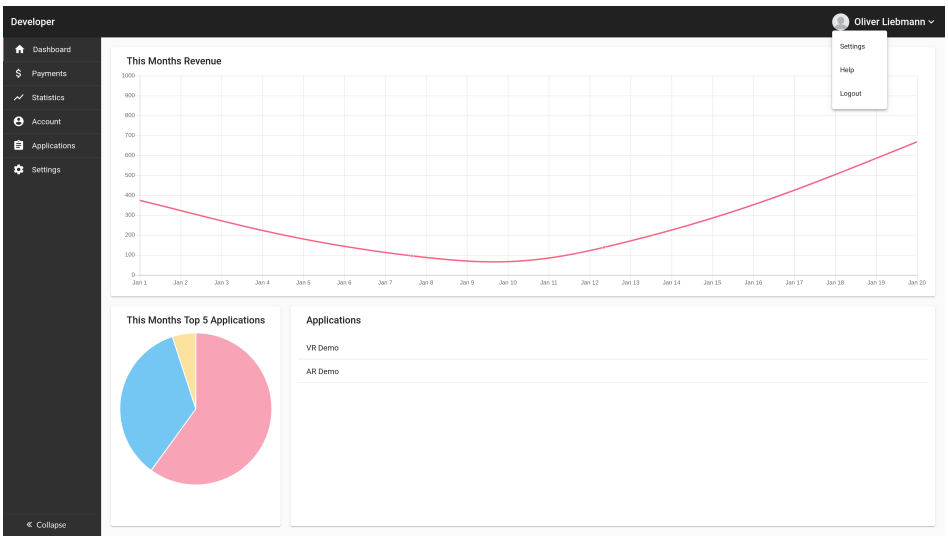


Figure 7.15: Developer Dashboard

Statistics
The statistics page is a tool for the developers to extract information about their profit and what their best selling products are. Like the dashboard, it shows the total revenue, the top 5 profitable apps and the links to the statistics page of the specific applications. The statistics page of the application looks exactly like the default statistics page except it shows all the products of the application instead of the applications.
To display all the charts, the *ng2-charts* library was used. The first thing to do to show a chart is to create a canvas element with the configuration needed by the library. An example canvas can be found in Listing 7.6.

```
<canvas
    baseChart
    [datasets]="revenueChartData"
    [options]="revenueChartOptions"
    [chartType]="'line'">
</canvas>
```

Listing 7.6: Example of a canvas for ng2-charts

The *baseChart* attribute tells the library, that this canvas is used for a chart. The *[datasets]* attribute feeds the statistics data to the library as an array of datasets. Out of these datasets, the library generates the chart. To configure the chart, a config object is fed into the *[options]* attribute. An example configuration can be found in Listing 7.7. The last attribute *[chartType]* defines what kind of chart it is. On the website, only two types were used. The first one was *line* for the revenue chart and the second one was *pie* for the top 5 chart.

```
{
    scaleShowVerticalLines: false,
    responsive: true,
    maintainAspectRatio: false,
    scales: {
        xAxes: [{
            type: 'time',
            distribution: 'linear',
            time: {
                min: moment('01-01-2018'),
                max: moment('12-31-2018'),
                unit: detail
            }
        }],
    },
    tooltips: {
        callbacks: {
            title: tooltipItems => {
                return moment(tooltipItems.xLabel).format(
                    dateFormat);},
            label: tooltipItems => {
                return tooltipItems.yLabel + ' Euro';}
        }
    }
}
```

Listing 7.7: Example configuration for ng2-charts

To now feed the data into the chart, the statistics provider service was used. This service was created to gather the data from the VIPER servers and process them to be then displayed.

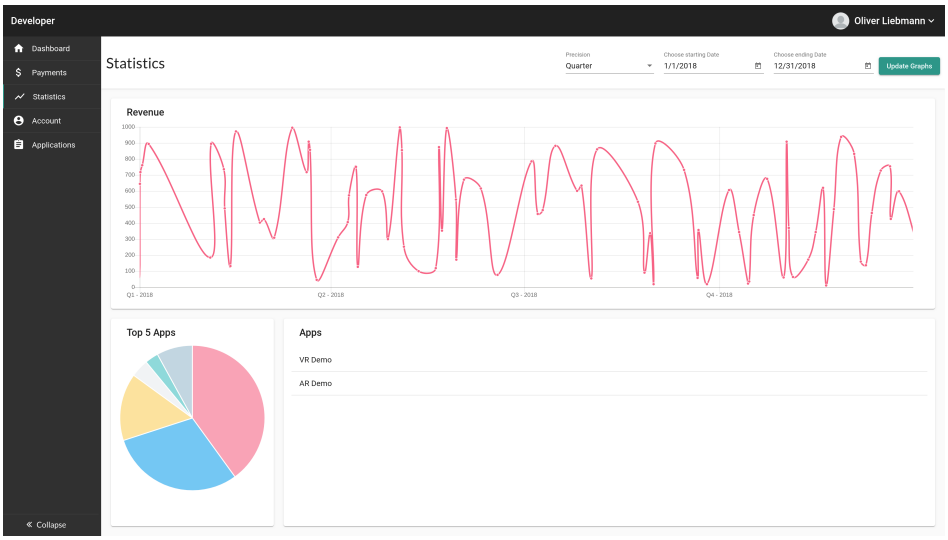


Figure 7.16: Developer statistics page

Applications

To create the applications, which are then displayed in the VR- and AR-apps, the application page was created. Here the developer can add all of his applications. When the +-FAB is pressed a dialog (Figure 7.17) opens. Here the developer can enter the necessary information about the application and upload an image. After clicking the *Create* button, another page opens (Figure 7.19). This page allows the developer to make changes to the applications and create products for the application. When the user presses the *Add Product* button, another dialog appears, which is used to create a product (Figure 7.18).

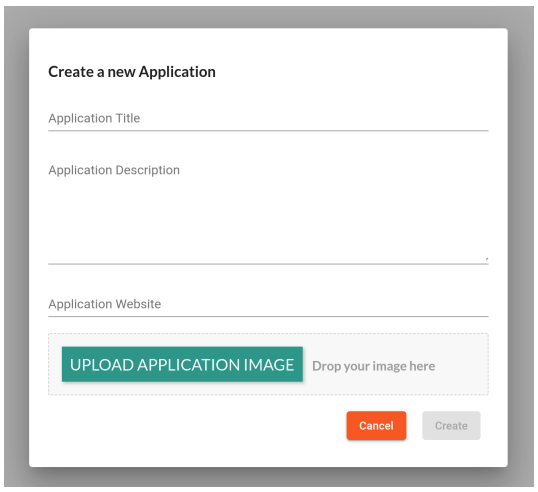


Figure 7.17: Create application dialog

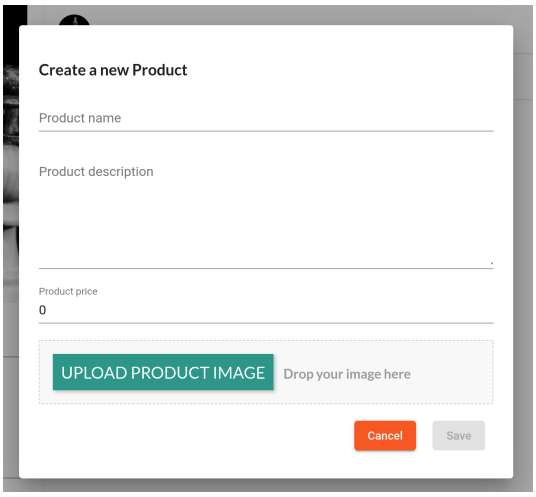


Figure 7.18: Create product dialog

The individual products are displayed as cards. This card design makes the application responsive. When there is not enough space in one row, the card simple gets placed in the next row.

Developer Accounts

The accounts page is used to manage the developer accounts of an organization (Figure 7.21). In future changes, all developers will have individual rights to manage applications. On the page all developers get displayed with their rank (admin- or user-icon), full name and email address. Everything of an account can be changed and the developer can be deleted, through the more options button on the right.

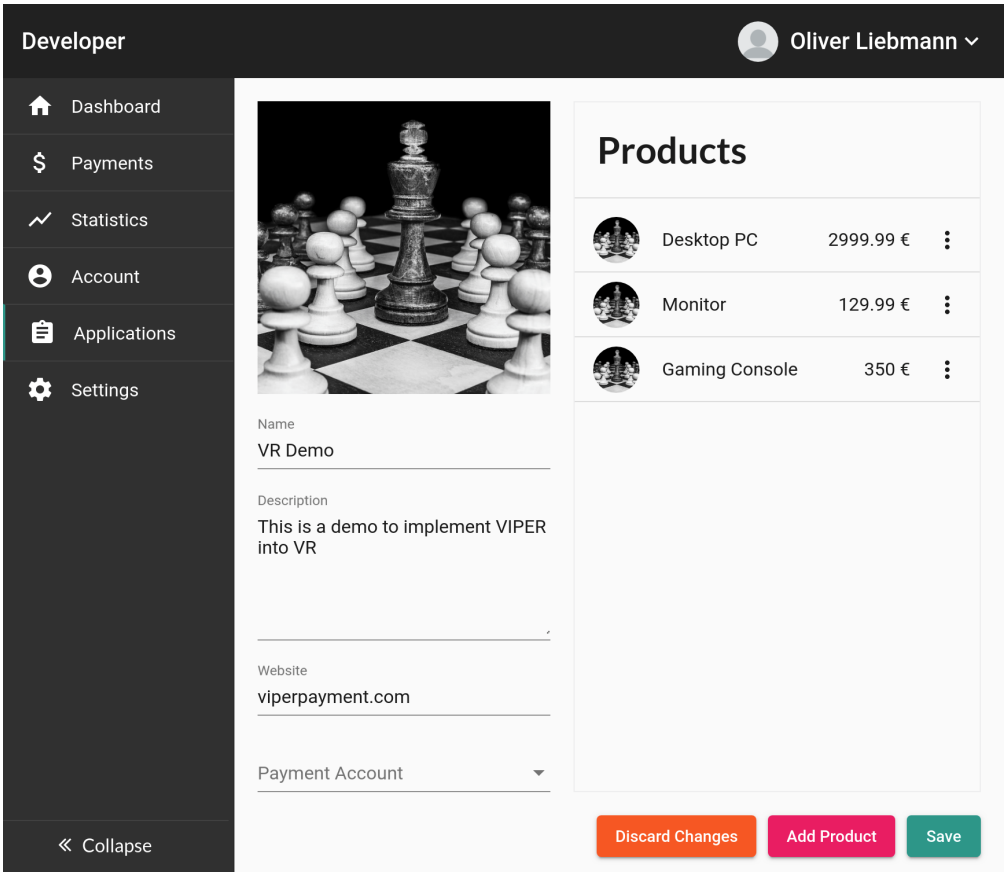


Figure 7.19: Product page

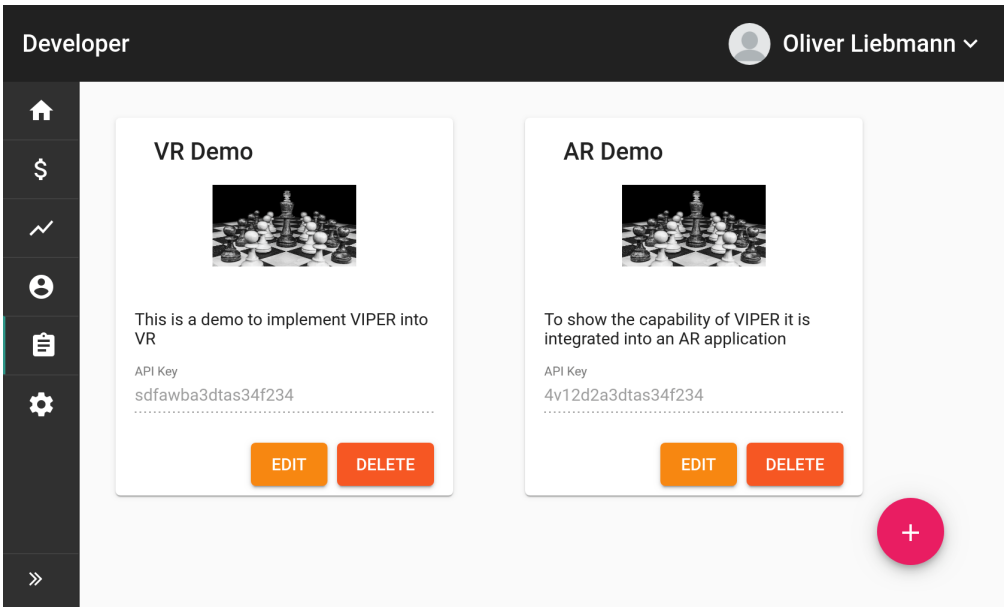


Figure 7.20: Developer applications page

Payment Accounts

The last page for the developer, is the payment accounts page (Figure 7.22). This page is for the organization to add their Braintree account to our services. This account is then used to transfer the money of the bought items to the organization. If the user clicks the +-FAB, a simple dialog is opened and the user can input the information for the Braintree account. After successfully creating the account it is immediately displayed on the page.

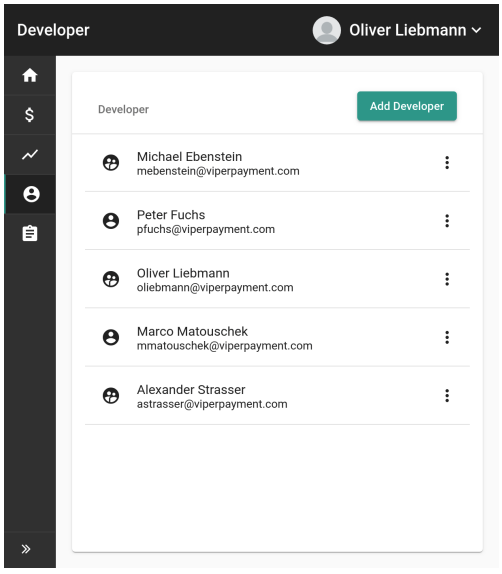


Figure 7.21: Developer accounts page

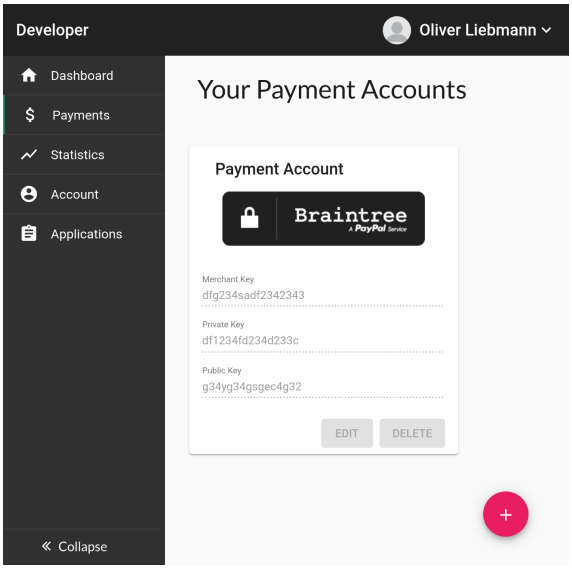


Figure 7.22: Developer payment accounts page

7.4.2 Services

The services are an essential part of the application since they are implementing the main logic. How to implement all services with Angular is explained in this section.

7.4.2.1 User Manager

The *userManager* service was important since it manages the sessions and the user login and registration. When a user uses a register or login form and presses the according button, the *userManager* starts the process of registering/logging in the user.

```
1 loginUser(identification: string, password: string, remember:
      boolean, role: string) {
2     const data = {'identification': identification, 'password'
          : password};
3     return Observable.create(observer => {
4         this.httpClient.post(this.config.getLoginPath(), data,
              {observe: 'response'})
5         .subscribe(resp => {
6             const token = resp.headers.get('Authorization');
7             // [...] - simple error handling
8             this.startLoginProcess(token, observer, role);
9         }, error => observer.error(error.error));
10    });
11 }
```

Listing 7.8: User manger login method

Listing 7.8 is the code for the login method. It is used to log in both clients and developers. First of all, it builds a data object from the provided information. Then it creates an observable, that can be resolved in another part of the application. To finish the observable, the application first tries to login the user on our servers and then saves the data inside the application. If everything was successful, the observable is resolved. If an error occurred, the error is returned and an error message is displayed. The request is sent via Angulars *HttpClient*. To send the request the *HttpClient* needs the URL that is provided by the config service, the data, and some options. The config service is a simple server that supplies all HTTP request with the right URL. In this example the *observe response* option is set. This enables the *HttpClient* to read the headers of the response and therefore the ability to extract the JWT. When the request was successful, the application starts the login process. This process is the same for registering and logging in. It simply saves the JWT and gathers information about the account.
After the token has been saved, the application is able to tell if a user is currently logged in. The method for checking if the user is logged in can be found in Listing 7.9. It checks if the token is expired with the help of the *jwtHelper* class. If the token is expired, the user gets logged out and the method returns false to signify that no user is currently logged in. Otherwise, the method returns true and therefore says that a user is currently logged in.

```
isUserLoggedIn() {
    const token = this.getToken();
    if (this.jwtHelper.isTokenExpired(token)) {
        this.removeToken();
        return false;
    }
    return true;
}
```

Listing 7.9: User manager *isUserLoggedIn* method

7.4.2.2 Payment Service Integration with Braintree

As described in section 6.7, our payment provider of choice is Braintree. This constrains us to use their provided API to communicate with their services. This means that the website has to integrate its *Javascript v3* SDK. Normally this library is easy to integrate since it is just a script, which has to be loaded. However, Angular is written in TS and therefore needs all it is libraries also in TS or with a type definition for every method. Node Package Manager (NPM) provides a Braintree package called *Braintree-web*[26], which integrates all the library calls into an easy to use package, but it is not written in TS and is therefore unusable for us. The package can be found here[26]. To try to use this package although it is not TS, we tried to change the configuration of Angulars TS compiler to ignore everything that has to do with the Braintree package. But still, the compiler refused to integrate this JS library and consequently, another solution was needed.

DefinitelyTyped
DefinitelyTyped is a repository for high quality TS type definitions. More information about DefinitelyTyped can be found on their website[39]. Type definitions are files for the TS compiler that identify the types of every variable and method of a JS library so it can run type checks with the library. DefinitelyTyped provides many type definitions for optional Angular components, hence they are well known and can be trusted to always have up-to-date definitions. They also maintain type definitions for the *braintree-web*[163] package. After the *braintree-web* package and the type definitions for it had been installed, another problem occurred. The Braintree SDK also heavily relies on third-party libraries. For example, to integrate PayPal payments into the application, the library created by PayPal is needed. The problem which arises now is that there are no type definitions for the PayPal library, which again causes the compiler to throw exceptions. This problem makes the approach to integrate the type definitions for the *braintree-web* package again unusable.

Braintree Dropin UI
The Braintree Dropin UI is a component created by Braintree. More information about the Drop-in UI can be found in their documentation[25]. This component gives the developer a fully working flow for every payment option he wants to integrate. The only drawback with this approach is that you can not customize it. The design, the flow, and the requests are unmodifiable. This makes the component a hassle to integrate into an existing application

with predefined designs and already existing flows. Nevertheless, it is the only way at the time to integrate the Braintree service into the website. NPM again provides a package called *ngx-braintree*. This package has type definitions and is also optimized for Angular because the package has been componentized. Therefore it is effortless to integrate into an existing Angular application. This package can be found on npm[113]. To use this package, it has to be installed via NPM, the command is shown in Listing 7.10.

```
npm install ngx-braintree --save
```

Listing 7.10: *ngx-braintree* installation command

After the successful installation, the component has to be called in the HTML file and finally has to be configured through the attributes of the tag. How it is called and configured can be found in Listing 7.11

```
<ngx-braintree
    [getClientToken]="getClientToken"
    [createPurchase]="createNonce"
    [enablePaypalVault]="true"
    (paymentStatus)="success()"
    (dropinLoaded)="onDropinLoaded()">
```

Listing 7.11: Configuration of the *ngx-braintree* tag

This setup is enough to integrate the Braintree Drop-in UI component as seen in Figure 7.23. However, it does not do anything until all the required methods are actually implemented. The first method, which gets passed to the *[getClientToken]* attribute, queries the client token from our servers. This happens through a simple HTTP request to our infrastructure, which then returns the client token. The token is mandatory for Braintree so it knows, which user is adding a new payment method. The *createPurchase* attribute requires a method that sends a request to the server and carries out the purchase. After that, it returns the purchase if it was successful. Since we do not want the user to purchase something, but rather save his payment account, we change the createPurchase to fit our objectives. The new purpose of this method is to retrieve the nonce, which is then used to identify the newly created account. Additionally, the interface changes and shows a form, where the user can enter a name, an authentication method and the pin/pattern for this account.

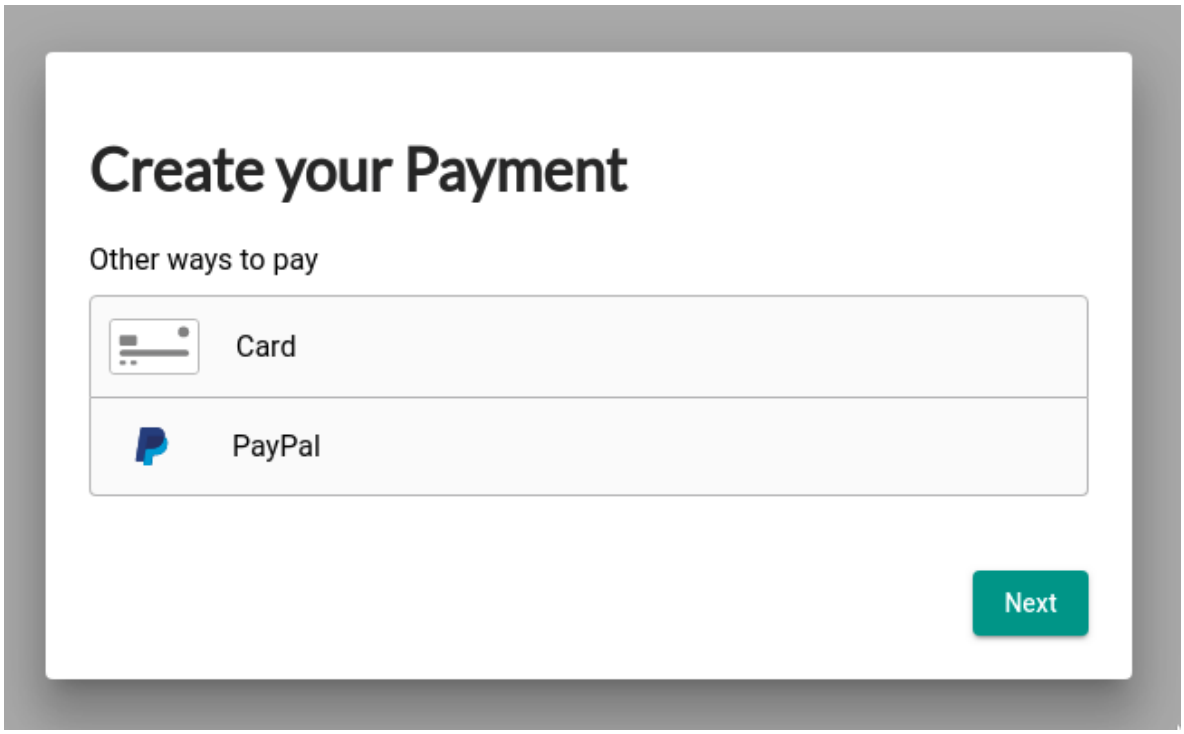


Figure 7.23: Braintree Drop-in UI

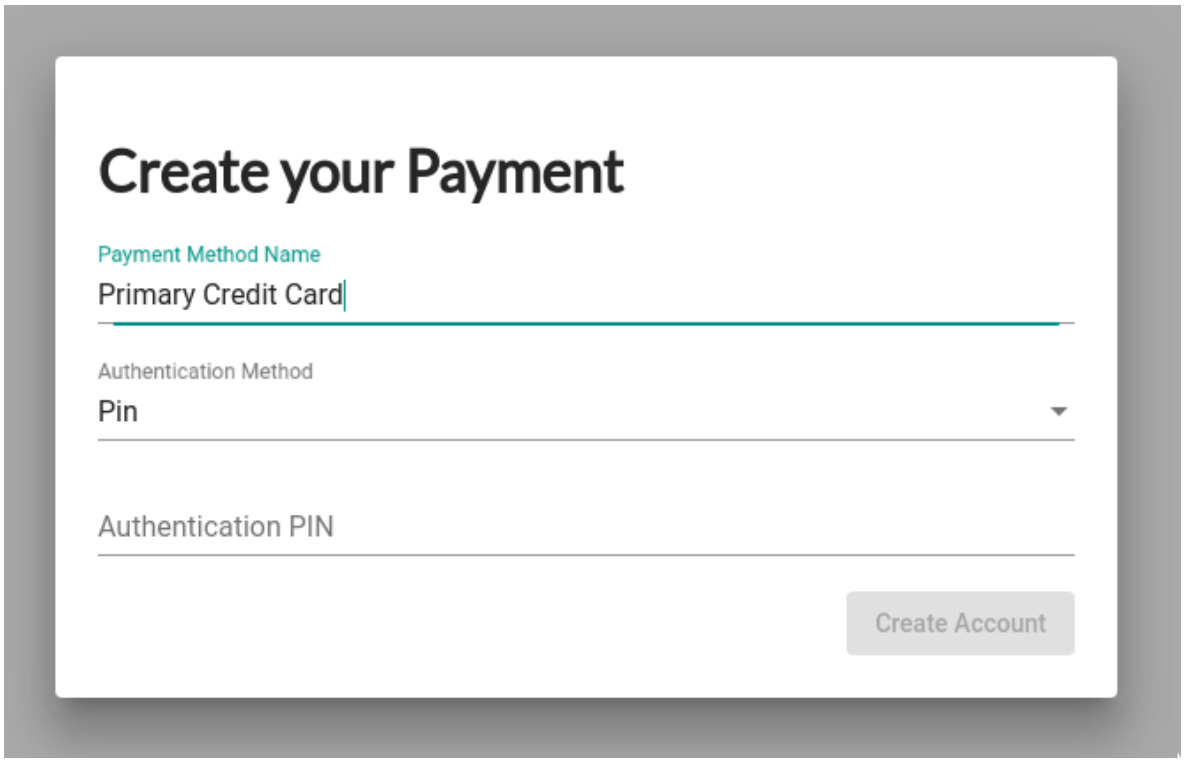


Figure 7.24: Create Payment Form

If the user now presses the *Create Account* button, the application sends a HTTP request to the server and tries to save the reference to the payment account in our database. Was this request successful, the *createPurchase* attribute gets resolved via a success message and the payment status is set to successful. However, should the request fail, the Braintree Drop-in UI gets loaded again and tries the flow from the beginning. The *[enablePaypalVault]* attribute is a configuration point to decide if Braintree should show an option for the user to enter his PayPal account.
When the payment status gets set to successful, the component automatically executes method that was assigned to the attribute *(paymentStatus)*. This method then closes the dialog and adds the new payment account to the UI.
The last attribute called *(dropinLoaded)* executes the provided method after all necessary scripts have been loaded, Braintree has identified the client token and the HTML tree has been built. It is important to catch this event, since the Drop-in UI always shows the previously saved accounts, but this is unnecessary for our application since it is only used to create a new account. At startup Braintree provides a button that says "Choose another way to pay". So before the user can enter anything, the application automatically clicks this button and therefore skips the step where the user sees his saved accounts.

7.4.3 Routing

The routing module, which is beeing directly supplied by Angular, enables navigation from one view to another view without refreshing the page. To enable routing inside an Angular application, the *RouterModule* and the *Routes* interface have to be imported, like in Listing 7.12.

```
1 import {RouterModule, Routes} from '@angular/router';
```

Listing 7.12: Importing the router module

First, a *Routes* object has to be created, which then gets imported in the *AppModule* of the application via the *RouterModule*. For example a *Routes* object could look like Listing 7.13.

```
1 const appRoutes: Routes = [
2     {path: '', component: FrontPageComponent},
3     {path: '**', component: Error404PageComponent}
4 ];
5
6 @NgModule({
7     imports: [
8         RouterModule.forRoot(appRoutes)
9     ],
10     [....]
11 })
12 export class AppModule {
13 }
```

Listing 7.13: Example routes configuration

The routes get defined as simple objects. The first parameter is the path, which has to be called in the browser so that the component, which is the second parameter, gets displayed.

The path can also be supplied with data. That functionality will later be used to determine the current page the user is looking at. In the example Listing 7.13 we have two routes. The first route with the empty path displays our front page if no URL path gets called. The second route has two asterisks in the path. This string stands for a path wildcard. This means that every URL, which is not defined in the *RouterModule*, will display the *Error 404 Page*. Since the *AppModule* can get quite large and unreadable when the application grows in complexity, the routes got outsourced into their own module. It is the exact same setup, but the *RouterModule* gets exported in the new module, so it then can be imported in the *AppModule*. How to import the *RouterModule* is described in Listing 7.14.

```
1 @NgModule({
2     imports: [
3         RouterModule.forRoot(appRoutes)
4     ],
5     exports: [RouterModule],
6 })
7 export class RoutesModule {
8 }
```

Listing 7.14: *RouterModule* import in *AppModule*

The newly created *RoutesModule* gets then imported into the *AppModule* like mentioned above. The Angular routing is now fully configured. However, the router does not know where to put the routed content yet. To enable the router to display the content, a router outlet has to be integrated into the HTML file of the app. The syntax of the router module can be found in Listing 7.15.

```
1 <router-outlet></router-outlet>
```

Listing 7.15: *router-outlet* tag

To implement a very readable URL style, children were also used inside the router. Listing 7.16 is an example of the routes of our statistics and dashboard pages for the developers.

```
1 const appRoutes: Routes = [{
2     path: 'developer',
3     component: DeveloperPageComponent,
4     children: [
5         {path: '', pathMatch: 'full', redirectTo: 'dashboard'
              },
6         {path: 'dashboard', component:
              DeveloperDashboardComponent, data: {active: '
              dashboard'}},
7         {path: 'statistics', component:
              DeveloperStatisticsComponent, data: {active: '
              statistics'}},
8         {path: 'statistics/:appTitle', component:
              ApplicationStatisticsComponent, data: {active: '
              statistics'}},
```

```
9           {path: 'statistics/:appTitle/:productTitle', component
                : ProductStatisticsComponent, data: {active: '
                statistics'}}
10      ]
11  }];
```

Listing 7.16: Example of routes with child routes

When the path */developer* is called the *DeveloperPageComponent* gets displayed. Inside this component, a router outlet, like before, has to be added to the HTML file. The children of a path also consist of the same parameters as all other routes. In the example Listing 7.16, there is one route that catches an empty addition after the */developer* path, which then redirects the URL to the dashboard of the developer. Two out of the three statistics paths have a route parameter inside of it. A route parameter is defined via a variable name and a colon in front of it. This parameter can then be accessed inside of the application through the *ActivatedRoute* module provided by Angular. The used method is shown in Listing 7.17 For this to work the mentioned module has to be injected into the constructor to then be used.

```
1 constructor(private activatedRoute: ActivatedRoute, [....]) {
2 }
3
4 ngOnInit(): void {
5     this.activatedRoute.params.subscribe(params => {
6         let appTitle = params['appTitle'];
7         //Dispatch action to load the details of the app.
8     });
9 }
```

Listing 7.17: Example method to gather route parameters

As just explained, the main application consists of a parent and a child. The parent handles the navigation and provides a predefined space where the different pages can be displayed. Consequently, the child gets loaded and now can display its content. For the parent, it is important, which page is currently beeing displayed. The information of the active child are supplied by the router itself. Every child path has a data attribute, as seen in Listing 7.16. Listing 7.18 shows the method, that is used to access the data. Again the *ActivatedRouter* module has to be injected and then the data can be requested by the parent.

```
1 constructor(private activatedRoute: ActivatedRoute, [....]) {
2 }
3
4 ngOnInit() {
5     this.activatedRoute.data.subscribe(data => {
6         this.activeChild = data['active'];
7         //Dispatch action to activate the right navigation
              point
8     });
9 }
```

Listing 7.18: Method to access route data

7.5 Testing, CI and CD

Testing is an important part of the development process. It is used to avoid common mistakes, but also mistakes that are no predictable during the development process. They can also prevent errors when changing the code, for the reason that the tests would throw errors if an essential parts of the application was changed. This leads to a more robust coding environment.

7.5.1 Unit Testing

Jasmine
Jasmine is a testing framework for javascript. It is capable of testing any kind of Javascript application without depending on other Javascript frameworks or a DOM. The DOM is a tree like structure which defines the internal structure of the website. It gets created by the browser while it reads the HTML file and interprets the tags. Those tags then get loaded into the memory which can then be further used by Javascript to read, change or add certain elements of the structure. This leads to a change in the UI.

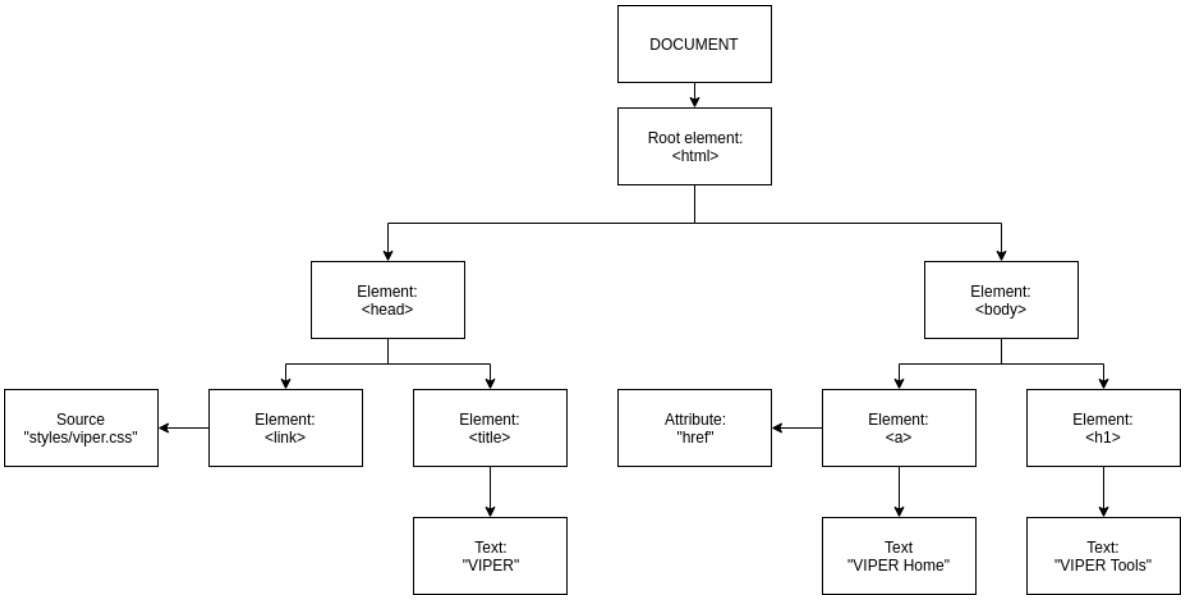


Figure 7.25: Example DOM tree

Jasmine follows a Behavior Driven Development (BDD) procedure to ensure that each line of the program is unit tested. BDD is a software development process that helps to create tests, which are more user-focused and based on the systems behaviour. It mainly focuses on where to start the process, what to test and what not to test and understanding why a test failed. At its core, BDD rethinks how to approach unit and acceptance testing and how to

avoid common issues. Furthermore, Jasmine has a very clean syntax which makes it very easy to read and maintain a large number of tests.

```
describe('CalculatorComponent', () => {
    it('should expect 1 to be equal to 1', function () {
        expect(1).toEqual(1);
    });
});
```

Listing 7.19: Example of the simple Jasmine syntax

As seen in Listing 7.19, the code is very similar to natural language. Every describe block is equivalent to one test case and the *it* just describes which behaviour the test is verifying. In this example, the *CalculatorComponent* gets tested to ensure, that 1 equals 1. If the Angular-CLI is beeing used to run Jasmine, it takes care of the configuration and therefore does not require a config file

Karma

The drawback with Jasmine alone is that after every code change, the Jasmine runner in the browser has to be refreshed. To support multiple browsers it would be even more troublesome, since every single one of them would have to be manually refreshed.
Karma helps at this point. It is not a testing framework, it just launches a HTTP server, and starts the test runner inside the spawned browsers. Karma supports real browsers, real devices such as phones and tablets and can even run on a terminal instance. A terminal instance is just an environment with only text IO and without any UI. Through the broad spectrum of support, it is easy to develop and test on multiple instances. Furthermore, Karma automatically reruns every test after it detects that some part of the code has changed. For our web app, we configured two browsers. The first one is Chrome and the other one is Firefox. Both browsers are configured are configured as real and as headless. The headless part is just for the testing in our CI environment.

```
browsers: [
    'Chrome', 'ChromeHeadless',
    'FirefoxHeadless', 'Firefox'
],
customLaunchers: {
    ChromeHeadless: {
        base: 'ChromeHeadless',
        flags: ['--no-sandbox']
    },
    FirefoxHeadless: {
        base: 'Firefox',
        flags: ['-headless'],
    }
}
```

Listing 7.20: Configuration of headless browsers

Listing 7.20 is the configuration, which is needed for Karma to spawn Firefox and Chrome and run the unit tests in them. It is also quite easy to configure real devices like an Android browser. One just has to install the Karma plugin which is then able to launch an Android browser.

```
npm install karma-android-device-browser-launcher --save-dev
```

Listing 7.21: Command to install Android support for Karma

After the successful installation of the launcher, the Android launcher has to be configured. By adding a *RealAndroidBrowser* to the browsers and create another custom launcher as in Listing 7.22, this functionality is enabled.

```
browsers: ['RealAndroidBrowser'],
customLaunchers: {
    RealAndroidBrowser: {
        base: 'AndroidDevice',
        deviceUuid: 'android-30015gea6dcc910b',
        sdkHome: '/opt/Android/sdk',
        deviceBrowser: 'chrome'
    }
}
```

Listing 7.22: Configuration for a real Android browser

The *deviceUuid* has to be changed to the actual ID of the real device, which can be found with the Android Debugging Bridge (ADB). The *sdkHome* has to be changed to the actual path of the SDK which is installed on the local machine. The *deviceBrowser* can be one of three browsers: chrome, internet or firefox. If one wished for all three, it can be configured by creating multiple customLaunchers. After the configuration is finished, Karma automatically starts a browser on the real device and will test it.
To now run the Jasmine tests with Karma, the Angular-CLI has to be called, which then starts up all the necessary tools to run the tests.

```
ng test
```

Listing 7.23: Command to start testing

It is a very simple process and therefore Karma is very helpful when trying to test various devices as quickly and reliably as possible.

7.5.2 End-to-End Testing - Protractor

Protractor is an end-to-end testing framework for Angular. It executes the tests just like a user would act in a real browser. Protractor is built on top of WebDriverJS, and is therefore able to use native events and browser dependent drivers to interact with the tested application, just like a normal user would. Protractor is a Node.js program, which means that it is necessary for Node.js to be installed to run any tests. It also uses Jasmine for its testing interface and consequently uses the same style for tests like Jasmine.
Typically Protractor runs its tests on a publicly available application. However, in our case, it

is more sensible to host the application on a local instance via a Selenium Server. Selenium is a framework for automated software tests against web applications. Protractor also comes with webdriver-manager. This is a simple helper tool which helps with creating an instance of a Selenium server. To now start the Selenium server, it is necessary to first update the webdriver-manager. The Angular-CLI also has a built-in tool which handles the creation of the Selenium server and automatically starts Protractor to run all the test against the application.

```
webdriver-manager update
ng e2e
```

Listing 7.24: Command to start end-to-end testing

The configuration of Protractor is also straightforward. Most options are pre-configured, when the framework gets installed. As seen with Karma, Protractor needs to know, which Browsers it should launch and run the tests in. Also, the baseUrl which it has to call needs to be specified. Lastly to support the already mentioned Jasmine syntax it is important to declare it as the used framework.

```
capabilities: {
    browserName: 'chrome',
},
directConnect: true,
baseUrl: 'http://localhost:4200/',
framework: 'jasmine',
```

Listing 7.25: Karma configuration

To support multiple Browsers, the capabilities have to be changed to multiCapabilities. That configuration then allows Protractor to spawn multiple Browsers and run all tests in parallel.

```
multiCapabilities: [{
    'browserName': 'chrome'
}, {
    'browserName': 'firefox'
}]
```

Listing 7.26: Protractor multiple browser support

The *baseUrl* specified in this particular configuration points to the Angular development server which gets started by the Angular-CLI when starting the end-to-end tests.

7.5.3 Continuous Integration and Continuous Deployment

For collaboration, Continuous Integration (CI) and Continous Delivery (CD) we use Gitlab and the Gitlab testing environment.

Countinous Integration

To test the website, Node.js has to be installed. After that, the required libraries are installed and the testing is started. This procedure worked fine with Jasmine and Karma. Anyhow, the

setup was not compatible with Protractor, since it needs a real Desktop Environment (DE) to simulate the users actions. However, the Gitlab environment does only provide a command line environment, which means that a DE has to be manually simulated. There is a tool called Xvfb, which stands for "X Window Virtual Framebuffer". For a client application Xvfb looks like a normal DE, but in reality it simulates all graphical operations in virtual memory and does not show any output to the screen. To run Xvfb does not need a graphics adapters, screen or input device. It just needs access to a network layer to properly work. All of this makes Xvfb a perfect environment to execute the Protractor tests in. The problem which arose during the configuration is that, it could not properly start in the Gitlab testing environment.
The easier solution was to use a preconfigured docker container. On the Docker Hub a user called "trion" created a container called "ng-cli-karma", which comes with Xvfb pre-installed an pre-configured.[162] Using this container has the advantage, that the container is always up-to-date and working. Through using this container the process simplified a lot. Karma and Jasmine worked normally, but Protractor did not need any special configurations or tweaks. As soon as Protractor got started, it detected the DE and successfully executed the tests.

Continous Deployment
To deploy the website, a docker container with a nginx configuration was created. Nginx is the webserver, which is used to serve the files for the website. More detailed information about the deployment can be found in section 6.10. To build the website the following command has to be executed:

```
1 ng build --progress false --prod
```

Listing 7.27: Command to build the Angular application

Listing 7.27 starts the building process. After finishing it outputs the main file for the website and all it is compiled sources. The *–progress false* flag says, that the tool should not show the current progress, since it is not needed. The *–prod* flag tells the Angular-CLI that it should build the application for production mode. This has many advantages when serving the website to the public. First of all it makes the size of the website a lot smaller through minification, uglification and dead code elimination. During minification, the tool removes all whitespaces and comments. This reduces the size of the application drastically. While uglification takes place, the code gets rewritten to use short and cryptic variable and function names. This helps to make the code unreadable and therefore protect it from unauthorized people. Dead code elimination, as the name already suggests, removes every code bits, which do not get used. Furthermore, it pre-compiles the Angular component templates, so the users browser does not have to interpret everything, which would be very time consuming and the user would experience a higher latency when viewing the page. This process is called Ahead-of-Time Compilation. Additionally the tool concatenates all files and libraries into a few bundles, which makes it easier to decide what has to be loaded when the page gets displayed. Lastly Angular enables the production mode and suppresses all debug outputs. After the building has finished. All files are loaded into the docker container, then the container gets deployed and started.
To secure the connection between our services and the user HTTPS has to be enabled on the nginx server. To be able to activate HTTPS a Secure Sockets Layer (SSL) certificate is needed. This certificate proves that you are the righteous owner of the domain under which the website is served. To always have an updated certificate we use a tool called *certbot*. This

tool verifies that you are the owner of the domain and creates the certificate for you.
To then enable HTTPS on nginx the config in Listing 7.28 had to be created.

```
server {
    listen              443 ssl;
    server_name         viperpayment.com;
    ssl_certificate     /etc/letsencrypt/live/viperpayment.com
       /fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/viperpayment.com
       /privkey.pem;

    [...]

    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

Listing 7.28: Command to build the Angular application

This configuration tells nginx that it should only listen to any requests, that are on port 443 and are under the domain *viperpayment.com*. Port 443 is the default port for HTTPS and gets automatically called by all browsers. The *ssl_certificate* and *ssl_certificate_key* options, specify the paths where the files from SSL certificate are located. The location option tells nginx that everytime a file gets requested that does not exist, the *index.html* gets served. This is important since Angular handles the routing on it is own and only if another component is request, the webserver should send it back.
When the container successfully started, the website is publicly accessible through a secure channel.

Chapter 8

Project Management

8.1 Different Project Management Models

When starting the project the team decided on a project management method to work as efficient and define the goals as simple as possible. The available management methods can be separated into agile and waterfall models.

8.1.1 Waterfall Model

The waterfall model has one of the most basic software development models as it is linear and sequential.[83] Thus after finishing one step in the waterfall-model, this step gets marked as completed and not touched again for the remaining project. Also, before being able to move on to the next phase, the one before it needs to be completed (as visualized in Figure 8.1).

The waterfall model can be generally split into four different phases:

Requirement Analysis and Specification Phase In the first phase, the project team gathers and documents the requirements of the product. This results in a document, the Software Requirement Specification (SRS)-document, which may act as a contract between the developer and the customer.[83] The SRS is written in a natural language as it must be understood by both the customers and the developers.

Design Phase The design phase translates the SRS into a structure suitable for designing and programming. The process of this phase gets documented in the Software Design Description (SDD)-document. It may contain technical jargon[83] as it contains the information needed to implement the software.

Implementation and Testing Phase In the implementation and testing phase the product gets developed and tested as defined in the SDD. The testing phase can further be split into two parts: Unit testing, where single modules of the software are tested, and integration and system testing, where the functionality of several processes is tested. In the example of VIPER these tests could include testing the whole payment process. After finishing the implementation and testing phase, the product gets shipped out to the customer.

Maintenance Phase The maintenance phase starts with the release of the software. The main focus of this phase is to keep the product functional by correcting errors and enhance some of its functions and capabilities.[83]

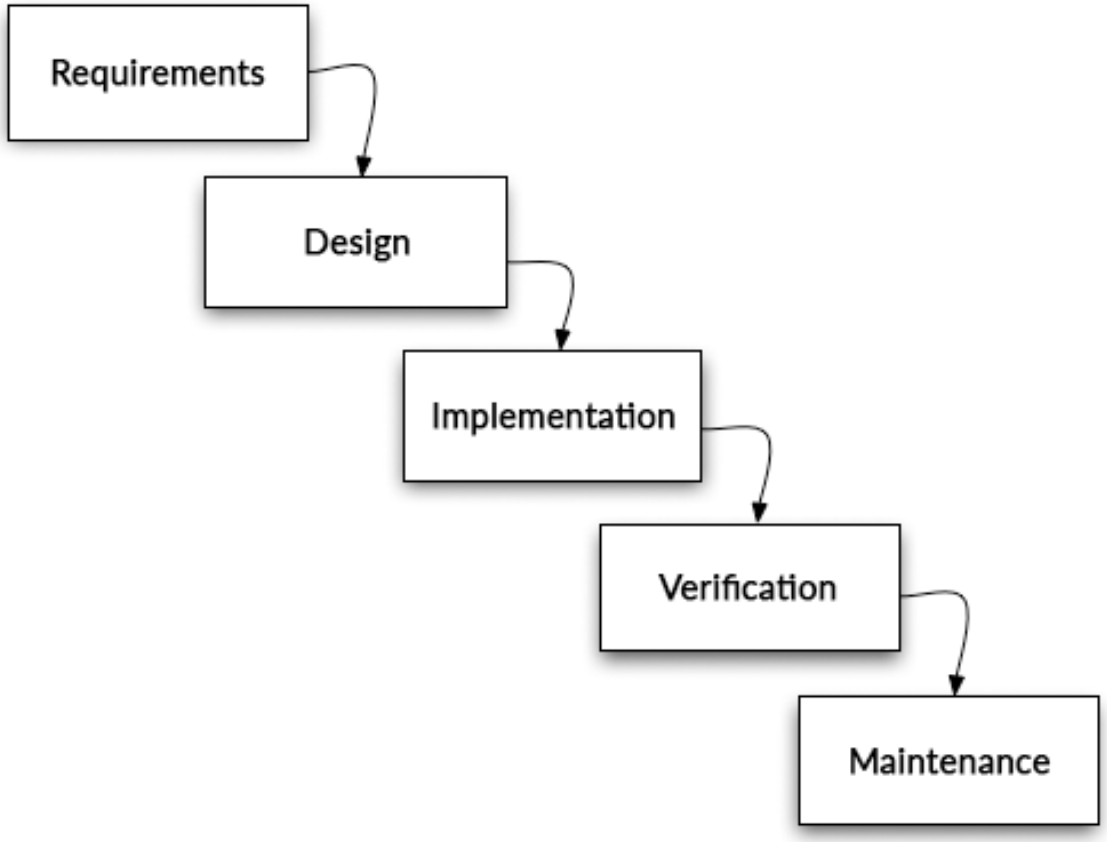


Figure 8.1: Graphic illustration of the waterfall model

8.1.2 Agile Model

The agile model mostly follows iterative and incremental practices of developing a software or product.[89] The life cycle of agile project management (which is visualized in Figure 8.2) is focused on being adaptive and people-based rather than predictive and process-oriented.[60] Agile project management methods provide several smaller software releases to increase the relationship between client and developers which leads to a very cooperative development phase.[89]

As shown in Figure 8.2, agile project management methods are not defined into several phases as these phases restart repeatedly. The agile model contains phases of planning, developing, testing and releasing. These four steps are handled circularly: A feature gets planned, then developed, then tested and then released. This repeats with every feature. The result is a very adaptable software development model.

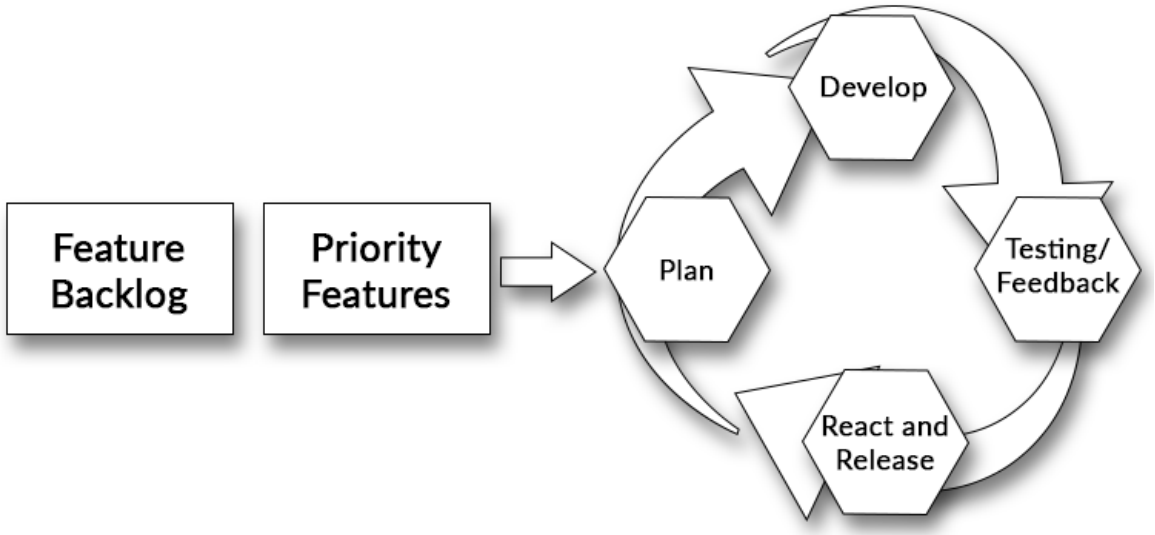


Figure 8.2: Life cycle of an agile project management method

8.2 Agile Project Management

For this project, the agile model was chosen. The main reasons for this decision were the customer involvement and promotion of team work.[83] Since the requirements where very dependent on the available time for this project, a very adaptive model where the ultimate goal was not completely defined was needed. As [83] states, an agile model should be chosen in these cases.

Furthermore, the team had to decide on an agile method. The decision fell on Scrumban which is a combination of Scrum and Kanban. This model was chosen due to the team's experience with this methodology. Scrumban takes the advantages of both models and combines them leaving behind a model that sets the timer for a process cycle ("Sprints") to one to two weeks and includes daily (in the team's case weekly) sprint meetings where the current status is discussed. Also, the team included Kanban's work board to visualize the tasks as well as Scrum's user stories for defining the requirements.

8.2.1 Sprints

The team decided to split the project into six sprints with a runtime of about 2 weeks each, where holidays were excluded:

Sprint 1 The first sprint started on 29th September and ended on 13th October. In this sprint, the team mainly created mockups and designed the interfaces and database. Also, the members made themselves comfortable with the development frameworks, mainly for the AR- and VR-applications.

Sprint 2 The second sprint, which lasted from 14th October to 27th October, included the first implementations of the user stories. This sprint was very focused on setting up the environments and creating basic functionalities such as moving around in the room (VR) or displaying virtual objects (AR). Also, the team enhanced some of the API-definitions and implemented the database.

Sprint 3 In the third sprint, the team managed to find a sponsor in A1. This allowed VIPER to be hosted on more stable and faster service. With this sponsorship, the team moved from Github to Gitlab, which made the implementation of CI easier. There were some advancements in the development of the demonstrations as well as designing the website for developers and clients.

Sprint 4 The fourth sprint began on 11th November and lasted until 24th November. The plan for this sprint was to finish the main applications as well as making communication with the service possible. Also the developer website should have been finished. Unfortunately, most of these tasks, namely all except the AR-demonstration, were not finished in time. They thus had to be moved to the sprints 5 and 6.

Sprint 5 and 6 The last two sprints included developing the final functionalities for the client website and the demonstration applications as well as deploying all services. Also, the team finally implemented the client library into the application making experiencing a payment process available. Originally, we planned to complete these sprints from the 25th November to the 22nd December. Due to some delays especially with the communication between back end and applications, the sixth sprint was extended to late January. Also, the team decided to only develop one application for VR since multiple demonstrations could not be finished in time. On 30th January, the project was finished.

8.2.2 Documentation

To document and keep track of the progress made during the project, weekly meetings were scheduled, discussing tasks, achievements, issues and possible solutions to them. In addition, roughly every month a meeting with the client was held, in order to inform them about the project as a whole and to receive feedback. These exchanges had to be documented, not only as a tool to keep a consistent overview about the project, but also to be used as a retrospective. Meetings with the client were scheduled a week prior per e-mail, informing about the date and time of the meeting as well as the planned agenda. This proved to be important, as the client received an overview of the following meetings contents and could add important points to the agenda if necessary. This way, every member was informed of the meeting subjects beforehand and could prepare accordingly.

Furthermore, the communication between the client and the project team in the form of periodical meetings was crucial in order to gain feedback and to fulfill the requirements they presented, in order to create the product they bare in mind. Close to every meeting included status updates on the current project progress, in which every team member discussed their task. The client gave feedback, asked questions that might not have been considered yet and gave their input. Afterwards the remaining agenda points were discussed.

The meeting protocols were distributed to every team associate via mail and were uploaded to an accessible Google Drive folder, in order to inform every member regarding the discussed topics, especially absentees.

VIPER

,

Time	-
Attendees	
Absent	
Author	

Agenda	
1	

Reports	**Stated by**
1.1	

To-Do	**Responsible**	**Date**

Follow-Ups

Figure 8.3: Viper Meeting Protocol Layout

Every meeting is documented using this specific layout. With it, all necessary information is presented in a clear and concise way. The header includes general information about the meeting, regarding attendees, date, time and duration as well as location. What follows are the planned agenda points to be discussed. Each report states the crucial information and exchanges presented during the meeting and are separated by agenda point. The reports are then further divided into sub-reports if necessary. Every entry must be assigned to an attendee in order to keep track of all statements correctly. The protocol then continues by listing current 'To-Do' tasks, e.g. tasks to be done in the near future as a way to create an overview of all assignments and responsibilities. Every 'To-Do' is assigned to one or more team members and receives a deadline. The protocol is concluded by listing 'Follow-Ups' which describe crucial topics to be discussed in future meetings.

8.3 Project Management Tools

Selection of the right tools for project management is crucial as these tools can greatly affect how easy it is to manage the team, project work and time.

8.3.1 ZenHub

For managing all user stories of the chosen project method Scrumban ZenHub [174], which integrates well with Github, was chosen. This tools provides a Scrumban Board integrated into Github. The board has separate pipelines to manage the current status of all user stories. User stories that have not been touched are in the initial *New Issues* pipeline. User stories of the current sprint are put into the *Backlog* pipeline. From there on the development process is started and the user stories is taken through all of the development stages and the corresponding pipeline: *Design*, *In Progress*, *Testing*, *Documentation*, *Review/QA* and finally *Closed*. The user stories of ZenHub are actually just Github Issues and all features of ZenHub, like pipelines and tags, are realized using standard Github features. For creating overviews and custom diagrams with a program like Microsoft Excel or Google Sheets, all user stories can be exported to a CSV file.

8.3.2 Toggl

As a time management tool Toggl [158] was chosen. This tool is used for real-time time tracking and has therefore the potential to deliver the most accurate results. Every time work on a user story is started a timer entry is started. It is stopped again when the work is done or interrupted. For clarity all Toggl timer entries were given the same name as the user story that was worked on during that time. Together with the list of all timer entries, this allows for detailed controlling of the project's progress. Additionally, Toggl provides diagrams to visualize the work done on the project. The timer entries used in these diagrams can be filtered by team member, project and task to allow for a detailed view of individual aspects of the project. The diagram in Figure 8.4 shows all work done for this project.

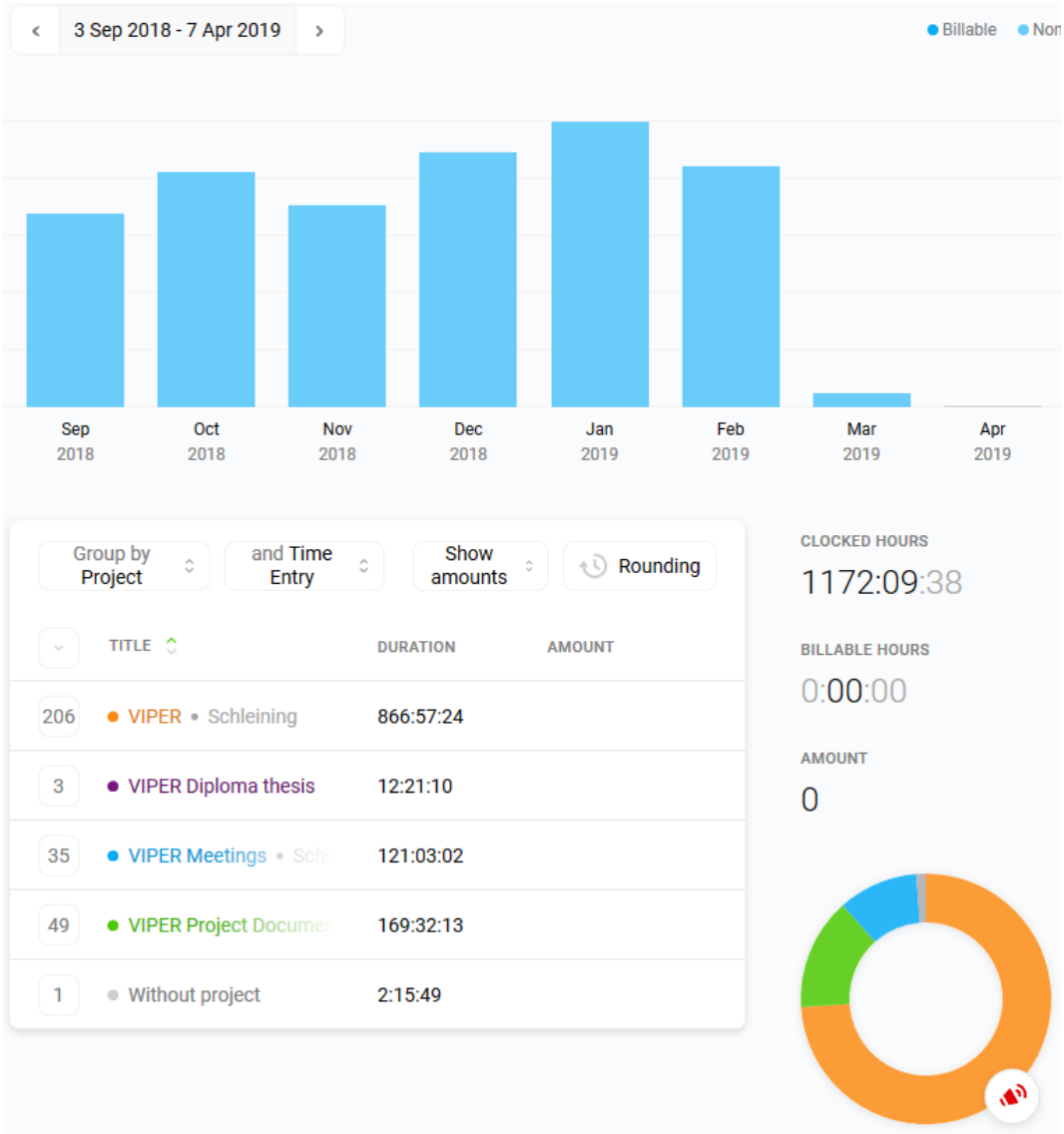


Figure 8.4: Toggl diagram of all work that was done for this project

Chapter 9

Conclusion and future work

This project showed a new way of paying inside virtual restricted environments by moving the payment setup outside and only executing minimal authentication steps inside of it. Different system architectures and their strengths and weaknesses were compared. Possible business applications and monetization strategies were shown. The implementation of the demonstration applications was shown and the challenges of authentication inside them discussed. Back end technologies that make the payment possible were compared and the differences of payment technologies highlighted. The web application for the payment account management was explained and the design choices elaborated. Finally the project management techniques used to manage and coordinate this project were explained.

9.1 Conclusion

This project showed a new way of paying inside virtual restricted environments by moving the payment setup outside and only executing minimal authentication steps inside of it. Different system architectures and their strengths and weaknesses were compared. Possible business applications and monetization strategies were shown. The implementation of the demonstration applications was shown and the challenges of authentication inside them discussed. Back end technologies that make the payment possible were compared and the differences of payment technologies highlighted. The web application for the payment account management was explained and the design choices elaborated. Finally the project management techniques used to manage and coordinate this project were explained.

9.2 Future work

Virtual Reality: While the Virtual Reality (VR) demo solidly stands on its own, there is still room for improvement. Some functionalities have yet to be implemented, as a lack of time resulted in the focus on the essential components of the application. Features such as being able to log in manually, a hover effect displaying information of objects and additional stylistic elements could add to the overall impression of the application. In addition, different methods of authentication may be provided, rather than just pin input.

Currently the demo only supports Oculus devices. Future advancements can be made by not only implementing support for various alternative VR devices, but also by creating applications in other popular development environments, since the developed application

can only act as a guideline project for Unity developers. This would result in a great increase in targeted developers that might make use of Viper.

Furthermore preexisting demo applications can be extended by integrating Viper and thereby showing the versatility and ease of implementation that the service provides.

Augmented Reality: In the future, more Software Development Kits (SDKs) like ARCore, Vuforia or Apple's ARKit could be supported. Also, a valid version of Wikitude could be implemented. These features would allow more developers to reference to VIPER Interactive Payment Engine Reification (VIPER)'s demonstration applications and increase the popularity of the service. Additionally, the Augmented Reality (AR)-demo could support more AR-technologies like marker-based AR or complex augmentations.

There could be even more applications, specifically designed for the Operating Systems (OSs). These applications would be developed in Android Studio and Xcode and would be a reference for the developers of these development environments. Also, applications could get developed for AR-headsets like Microsoft's HoloLens 2. Development on these devices would allow VIPER to increase their market potential even further and make payment available in every restricted environment.

With developing on more devices, VIPER could even implement new versions of identity verification. These could be biometric authentications such as fingerprint or iris scans.

Back end To use the microservice system in a production environment multiple security aspects have to be improved and all components have to be tested further. The security of the services can be increased by implementing database encryption and using Transport Layer Security (TLS) for inter-service communication. Additionally, the most recent software releases of all frameworks should be used. To support more payment methods, more payment services can be implemented. The Braintree service needs to be update once the *grant-API* is released by Braintree. In the European Union new payment related legal changes are planned and the service needs to be adjusted accordingly. The deployment of the infrastructure may be moved to different cloud providers once all Exoscale credits are used. Switching cloud providers requires a reevaluation of available cloud providers, more detailed funding plans and changes in the deployment procedures. To increase the manageability of microservices a Kubernetes infrastructure could be implemented.

Web Architecture For future improvements, a more customizable web application would give the user a better feeling. The dashboard could be adjusted so that the user can pick the windows he wants to see when the application opens. This would also allow the user to rearrange the windows to his personal preferences. Another improvement from which the website could gain a lot would be more and customizable statistics. These statistics should additionally be customizable and dynamic. Additionally type definitions for the PayPal library could be created to make a more seamless integration of the Braintree SDK. Thus removing the liabilty to third party integrations of the Braintree SDK. Finally, a usability study should be carried out to see how users react to the application and how it could be enhanced according to this information.

Glossary

Angular-CLI “The Angular-CLI is a command line tool, which makes it easy to create an application that already works, right out of the box and already follows best practices. It provides tools, which help with development, linting and testing. Furthermore it helps maintaining best practices, since it can generate components, routes, services, etc. and automatially create simple test shells for all of these.”[9]. 163, 184–187

application An application that is registered to an organization and has an Application Programming Interface (API) key with which it gets access to the VIPER payment API. 12, 18, 84, 100, 109, 137, 138, 151

assets “media or data that is used in Unity, such as 3D models, audio files and images”. 34

binocular vision the vision of both eyes overlapping. 24

Clearing House A financial instituation that manages the exchange of value assets between to parties. It is used to reduce the risk of a partie failing to meet its trade settlement obligations, centralizes and unifies the Clearing process and is cost efficent. 116

client library A dynamic C++ library that connects applications to the VIPER back end services. 18, 58, 71, 151–153

Cloud Anchors “Cloud Anchors lets you make ARKit and/or ARCore anchors available to multiple devices in the same environment. Users in the same environment can add Cloud Anchors to the AR scene that they see on their device. Your app can render 3D objects attached to the Cloud Anchors, letting users see and interact with the objects simultaneously”[138]. 49, 51

customer The person who uses the VIPER service to pay for item in applications which support it. 12, 83, 84, 93, 98, 106–109, 135, 138

DBMS A system software for creating and managing databases. 83–88, 137

developer The member of an organization who develops applications. 12, 18, 83, 84, 100, 106, 137, 138, 151

EMV “a specification for payment cards utilizing built in micro chips”. 29

Extended Tracking "Extended Tracking is the concept that a target's pose information will be available even when the Target is no longer in the field of view of the camera or cannot directly be tracked for other reasons. Extended Tracking utilizes the Device Tracker to improve tracking performance and sustain tracking even when the target is no longer in view"[55]. 49, 51

Face Detection The process of detecting a face on the image captured by a camera. 53

Feature Points Visually distinct features in a captured camera image. 51

Flat-Design Flat-Design is a design principle, where two-dimensional elements and bright colors are used. real world elements are used for reference but they are reduced to minimalistic shapes. This minimalistic design also dispenses of gradients and textures.. 157

head mounted display "A Head-Mounted Display is a worn device that has a small display in front of one or each eye. Its main use is for Virtual Reality applications, though other uses include aviation and engineering.". 24

Hypervisor A software that emulates the hardware of a machine. From the old word for OS, supervisor. [122] . 142

Image Recognition The process of recognising an image which is stored in a database. 53

immersive the feeling of being completely involved, present in something (e.g. a virtual environment). 24, 44

Instant Tracking "Instant tracking is an algorithm that does not aim to recognize a predefined target and start the tracking procedure thereafter, but immediately start tracking in an arbitrary environment. This enables very specific use cases to be implemented"[75]. 45

maven "Apache Maven is a software project management and comprehension tool."[93] . 140, 142

NuGet NuGet is a .NET package manager for Visual Studio [8]. 151

organization A group or legal entity, registered at VIPER, which can receive payments via their registered applications. 12, 22, 83, 84, 109, 137, 138

Point-Of-Sale Sometime called point-of-purchase; The time and place where a transaction is completed. 119

prefab "an object in Unity with the intent to be reused or distributed". 38

security group An Exoscale security group is a set of firewall rules that apply to all Exoscale instances in this security group. 109

Simultaneous Localization And Mapping "'SLAM' is not a particular algorithm or piece of software, but rather it refers to the problem of trying to simultaneously localise (i.e. find the position/orientation of) some sensor with respect to its surroundings, while at the same time mapping the structure of that environment" [40]. 45, 48, 51, 53, 62

Skeuomorphism "Skeuomorphism refers to a design principle in which design cues are taken from the physical world. This term is most frequently applied to user interfaces (UIs), where much of the design has traditionally aimed to recall the real world - such as the use of folder and files images for computer filing systems, or a letter symbol for email - probably to make computers feel more familiar to users."[142]. 157

teleoperation the remote control of a device or machine. 45

user A person registered at VIPER. This can be either a customer or a developer. 12, 83, 84, 87, 89, 98, 100–106, 108, 109, 135, 137, 210, 212

Visual Studio Microsoft Visual Studio is an Integrated Development Environment (IDE) from Microsoft. It is used to develop programs, mainly with C# and C++. 151

Acronyms

ACH Automated Clearing House. 116, 117, 120, 123, 124

ADB Android Debugging Bridge. 185

API Application Programming Interface. 12, 71, 78, 81–83, 97–100, 109, 110, 122, 138, 149, 161, 163, 177, 192

AR Augmented Reality. 5, 11, 12, 15, 18, 19, 30, 43–56, 65, 70, 72, 98, 100, 137, 151, 173, 191, 192, 198

ARIA Accessible Rich Internet Applications. 163

BDD Behavior Driven Development. 183

BI Business Intelligence. 78

BIC Business Identifier Code. 116

CD Continous Delivery. 144–146, 186

CI Continuous Integration. 55, 76, 140, 144–146, 150, 184, 186, 192

CL Client Library. 71

CLI Command Line Interface. 142

CORS Cross Origin Resource Sharing. 110

CRUD Create, Read, Update, Delete. 91, 128, 129

CT Computed Tomography. 46

DE Desktop Environment. 186, 187

DHCP Dynamic Host Configuration Protocol. 97

DNS Domain Name Service. 140

DOM Document Object Model. 162, 183

DTO Data Transfer Object. 135

EFT Electronic Funds Transfer. 115, 117, 120, 124

ERD Entity Relationship Diagram. 100, 101, 136, 138

EU European Union. 116

FAB Floating Action Button. 156, 173, 175

FOV Field of view. 24, 26, 27

GPS Global Positioning System. 49

GPU Graphics Processing Unit. 101

GSA Geld Service Austria. 116

HMD head mounted display. 24, 26, 27, 30–32, 41, 42

HTML Hypertext Markup Language. 52, 162, 163, 178, 180–183

HTTP Hypertext Transfer Protocol. 82, 94–98, 103–105, 135, 136, 151–153, 163, 165, 176, 178, 180, 184, 220

HTTPS Hypertext Transfer Protocol Secure. 97, 110, 187, 188

IBAN International Bank Account Number. 116

IDE Integrated Development Environment. 164

IO Input/Output. 170, 184

ITU International Telecommunications Union. 155

JEE Java Enterprise Edition. 80

JS JavaScript. 52, 161, 162, 164, 165, 177

JSON JavaScript Object Notation. 82, 94, 96, 98, 153

JWT JSON Web Token. 102–105, 108, 153, 165, 176, 210

LTS Long-Term Support. 163

MR Mixed Reality. 43, 44, 50–52

MRI Magnetic Resonance Imaging. 46

NACHA National Automated Clearing House Association. 116

Netflix OSS Netflix Open Source Software. 81

NPM Node Package Manager. 177, 178

OS Operating System. 16, 49, 51–54, 78, 139, 198

OWASP Open Web Application Security Project. 101

PC Personal Computer. 11, 155

PCISSD Payment Card Industry Data Security Council. 119

PE-ACH Pan-European automated clearing house. 116

PSP Payment Service Provider. 21, 117–123, 125–127

QR-code Quick Response-code. 19, 46, 47, 55, 67, 68

ReST Representational State Transfer. 16, 77, 78, 94, 95, 97, 98, 109, 122, 124, 128, 129, 149

SDD Software Design Description. 189

SDK Software Development Kit. 27, 32, 45, 49–56, 60, 62, 177, 185, 198

SEPA Single Euro Payments Area. 116

SLAM Simultaneous Localization And Mapping. 45, 48, 62

SOAP Simple Object Access Protocol. 94, 98

SRS Software Requirement Specification. 189

SSL Secure Sockets Layer. 187, 188

TGM Technologisches Gewerbemuseum. 7

TLS Transport Layer Security. 97, 110, 198

TS TypeScript. 164, 177

TV Television. 54

UI User Interface. 54, 58, 59, 151, 161–163, 180, 183, 184

URL Uniform Resource Locator. 94, 95, 99, 110, 111, 176, 181, 182

USA United States of America. 116

USP Unique Selling Point. 50

UX User Experience. 18, 58, 64, 71

VIPER VIPER Interactive Payment Engine Reification. 5, 11, 12, 17–19, 21, 55, 56, 59, 60, 66–68, 75, 79–82, 86, 87, 93, 97, 98, 102, 104, 109, 110, 112, 113, 135, 137, 138, 141, 151, 155–158, 161, 163, 166, 172, 189, 192, 198

VR Virtual Reality. 5, 11, 12, 15, 18, 19, 23–28, 30, 32, 41, 43, 44, 50, 51, 54, 98, 100, 137, 151, 155, 173, 191, 192, 197

VS Visual Studio. 52, 73

XML eXtensible Markup Language. 98, 99

XR Cross Reality. 5, 11, 15, 55, 78

Bibliography

[1] *A11Y Project*. 07.04.2019. The Accessibility Project. URL: https://a11yproject.com/.

[2] Abolfazl Aleahmad, Hadi Amiri, and Masoud Rahgozar. "Main Memory Databases vs. Disk-Resident Databases". In: (Jan. 2006).

[3] AliPay. *AliPay Website*. 03.04.2019. URL: https://global.alipay.com/products/online.

[4] Amazon. *Alexa Index top 1M*. 03.04.2019. URL: http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[5] Amazon. *Amazon Pay website*. 03.04.2019. URL: https://pay.amazon.com/.

[6] Amazon. *Amazon Web Services cloud computing*. 03.04.2019. URL: https://aws.amazon.com/what-is-cloud-computing/.

[7] *Amazon Page Load Revenue Increase*. 01.04.2019. Amazon. URL: https://blog.qburst.com/2017/05/make-your-apps-load-faster-with-angular-2/.

[8] *An introduction to NuGet*. 2019/04/01. Microsoft. URL: https://docs.microsoft.com/en-us/nuget/what-is-nuget.

[9] *Angular-CLI*. 24.03.2019. Google LLC. URL: https://cli.angular.io/.

[10] *angular-jwt package*. 05.04.2019. npm, Inc. URL: https://www.npmjs.com/package/angular-jwt.

[11] *angular2-image-upload package*. 05.04.2019. npm, Inc. URL: https://www.npmjs.com/package/angular2-image-upload.

[12] *AppGameKit - VR*. 7.04.2019. The Game Creators Ltd. URL: https://www.appgamekit.com/dlc/vr.

[13] Apple. *ApplePay Website*. 03.04.2019. URL: https://developer.apple.com/documentation/passkit/apple_pay.

[14] *ArangoDB*. 2019/04/01. AragoDB Inc. URL: https://www.arangodb.com/.

[15] *ARCore Fundamental Concepts*. 26.02.2019. Google. URL: https://developers.google.com/ar/discover/concepts.

[16] *ARCore Supported Devices*. 26.02.2019. Google. URL: https://developers.google.com/ar/discover/supported-devices.

[17] *ARKit Apple Developer Documentation*. 13.03.2019. Apple. URL: https://developer.apple.com/documentation/arkit.

[18] *ARKit2*. 26.02.2019. Apple. URL: https://developer.apple.com/arkit/.

[19] Ronald T. Azuma. "A Survey of Augmented Reality". In: *Presence: Teleoper. Virtual Environ.* 6.4 (Aug. 1997). 20.02.2019, pp. 355–385. ISSN: 1054-7460. DOI: 10.1162/pres.1997.6.4.355. URL: http://dx.doi.org/10.1162/pres.1997.6.4.355.

[20] *Beatsaber Footage*. 7.4.2019. Playstation Europe. URL: https://www.flickr.com/photos/playstationblogeurope/27771434737.

[21] Ph.D Ben Ramsbottom B.Sc. *How Virtual Reality is Changing Healthcare for the Better*. 7.04.2019. The Doctor Weighs In. URL: https://thedoctorweighsin.com/virtual-reality-improving-healthcare/.

[22] G. Booch. *Object Oriented Design: With Applications*. The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings Pub., 1991. ISBN: 9780805300918. URL: https://books.google.at/books?id=w5VQAAAAMAAJ.

[23] Braintree. *Braintree testing documentation*. 05.04.2019. URL: https://developers.braintreepayments.com/reference/general/testing/java.

[24] Braintree. *Braintree Website*. 03.04.2019. URL: https://developers.braintreepayments.com/start/overview.

[25] *Braintree Dropin UI*. 07.04.2019. Braintree. URL: https://developers.braintreepayments.com/guides/drop-in/overview/javascript/v3.

[26] *Braintree-web package*. 05.04.2019. npm, Inc. URL: https://www.npmjs.com/package/braintree-web.

[27] Green Buzz. *How To Attract An Audience With Augmented Reality*. 24.02.2019. Green Buzz Agency. URL: http://greenbuzzagency.com/attract-audiences-with-augmented-reality/.

[28] Alan Calder and Geraint Williams. *PCI DSS: A Pocket Guide*. 4th. It Governance Ltd, 2015. ISBN: 1849287813, 9781849287814.

[29] *Client Side Load Balancer: Ribbon*. 2019/04/01. Pivotal Software, Inc. URL: https://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#spring-cloud-ribbon.

[30] U.S. Congress. *Electronic Fund Transfer Act (EFTA) (15 USC 1693 et seq.)*
An Act to extend the authority for the flexible regulation of interest rates on deposits and accounts in depository institutions.
https://www.federalreserve.gov/boarddocs/caletters/2008/0807/08-07_attachment.pdf. 1978.

[31] Sharon Coone. *Riot Games wins Sports Emmy for 2017 World Championship Broadcast*. 20.02.2019. Blitz Esports. URL: https://blitzesports.com/lol/article/3384/riot-games-wins-sports-emmy-2017-world-championship-broadcas.

[32] *Couchbase*. 2019/04/01. Couchbase. URL: https://www.couchbase.com/.

[33] Alan B. Craig. "Chapter 7 - Mobile Augmented Reality". In: *Understanding Augmented Reality*. Ed. by Alan B. Craig. Boston: Morgan Kaufmann, 2013, pp. 209 – 220. ISBN: 978-0-240-82408-6. DOI: https://doi.org/10.1016/B978-0-240-82408-6.00007-2. URL: http://www.sciencedirect.com/science/article/pii/B9780240824086000072.

[34] *Create a .pfx/.p12 certificate file using OpenSSL*. 2019/04/01. SSL.com. URL: https://www.ssl.com/how-to/create-a-pfx-p12-certificate-file-using-openssl/.

[35] *ctypes - A foreign function library for Python*. 2019/04/01. Python Software Foundation. URL: https://docs.python.org/3/library/ctypes.html.

[36] Meenu Dave. "SQL and NoSQL Databases". In: *International Journal of Advanced Research in Computer Science and Software Engineering* (Aug. 2012). URL: https://www.researchgate.net/publication/303856633_SQL_and_NoSQL_Databases.

[37] daytanyze. *Payment Service Provider marketshare data (Alexa Top 1M)*. 03.04.2019. URL: https://www.datanyze.com/market-share/payment-processing/Alexa%20top%201M/.

[38] *Declarative REST Client: Feign*. 2019/04/01. Pivotal Software, Inc. URL: https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-feign.html.

[39] *DefinitelyTyped website*. 05.04.2019. DefinitelyTyped. URL: http://definitelytyped.org/.

[40] *Definition of SLAM*. URL: https://www.kudan.eu/kudan-news/an-introduction-to-slam/.

[41] Swati Dhingra. *REST vs. SOAP: Choosing the best web service*. 2019/04/01. TechTarget. URL: https://searchmicroservices.techtarget.com/tip/REST-vs-SOAP-Choosing-the-best-web-service.

[42] *Diagram of a monolithic application*. 03.04.2019. URL: https://cdn-images-1.medium.com/max/1000/1*DH0QraG0ojgeDCtg5M-tmQ.png.

[43] A1 Digital. *A1 Digital Website*. 03.04.2019. URL: https://www.a1.digital/.

[44] Docker. *Docker Compose Website*. 03.04.2019. URL: https://docs.docker.com/compose/.

[45] Docker. *Docker Container overview*. 03.04.2019. URL: https://www.docker.com/resources/what-container.

[46] Docker. *Docker documentation*. 03.04.2019. URL: https://docs.docker.com/.

[47] Docker. *Docker Website*. 03.04.2019. URL: https://www.docker.com.

[48] L. Dusseault and J. Snell. *PATCH Method for HTTP*. RFC 5789. http://www.rfc-editor.org/rfc/rfc5789.txt. RFC Editor, 2010. URL: http://www.rfc-editor.org/rfc/rfc5789.txt.

[49] Greg Edwards-Stewart Amanda; Hoyt Tim; Reger. "Classifying different types of augmented reality technology". In: *Annual Review of CyberTherapy and Telemedicine* 14 (Jan. 2016). 22.02.2019, p. 200.

[50] Elastic. *Logstash Website*. 03.04.2019. URL: https://www.elastic.co/products/logstash.

[51] *Encryption at Rest*. 2019/04/01. MongoDB, Inc. URL: https://docs.mongodb.com/manual/core/security-encryption-at-rest/.

[52] *Erstellen von Virtual-Reality-Spielen in Lumberyard*. 7.04.2019. Amazon Web Services Inc. URL: https://docs.aws.amazon.com/de_de/lumberyard/latest/userguide/virtual-reality.html.

[53] *Everything you need to build on Android*. 27.02.2019. Google. URL: https://developer.android.com/studio/features.

[54] Exoscale. *Exoscale Website*. 03.04.2019. URL: https://www.exoscale.com/.

[55] *Extended Tracking*. 24.02.2019. PTC. URL: https://library.vuforia.com/articles/Training/Extended-Tracking.

[56] *Feign Apache HttpClient*. 2019/04/01. MvnRepository. URL: https://mvnrepository.com/artifact/io.github.openfeign/feign-httpclient.

[57] *Field of View for Virtual Reality Headsets Explained*. 7.04.2019. VR-Lens-Lab. URL: https://vr-lens-lab.com/field-of-view-for-virtual-reality-headsets/.

[58] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. http://www.rfc-editor.org/rfc/rfc7231.txt. RFC Editor, 2014. URL: http://www.rfc-editor.org/rfc/rfc7231.txt.

[59] Roy T. Fielding et al. “Reflections on the REST Architectural Style and "Principled Design of the Modern Web Architecture" (Impact Paper Award)”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 4–14. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3121282. URL: http://doi.acm.org/10.1145/3106237.3121282.

[60] M. Fowler. *Agile Software Development*. 17.03.2019. URL: http://martinfowler.com/agile.html.

[61] Spencer Gibb. *Spring Cloud Greenwich.RELEASE is now available*. 2019/04/01. Pivotal Software, Inc. URL: https://spring.io/blog/2019/01/23/spring-cloud-greenwich-release-is-now-available.

[62] Henri Gilbert and Helena Handschuh. “Security Analysis of SHA-256 and Sisters”. In: *Selected Areas in Cryptography*. Ed. by Mitsuru Matsui and Robert J. Zuccherato. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 175–193. ISBN: 978-3-540-24654-1.

[63] R. Silva; J. C. Oliveira; G. A. Giraldi. *Introduction to Augmented Reality*. 20.02.2019. National Laboratory for Scientific Computation. URL: http://lncc.br/~jauvane/papers/RelatorioTecnicoLNCC-2503.pdf.

[64] Gitlab. *GitlabCI Diagram*. 03.04.2019. URL: https://about.gitlab.com/images/blogimages/cicd_pipeline_infograph.png.

[65] Gitlab. *GitlabCI Overview*. 03.04.2019. URL: https://docs.gitlab.com/ee/ci/.

[66] Google. *Cadvisor Website*. 03.04.2019. URL: https://github.com/google/cadvisor.

[67] *Google’s Material.io color tool with the VIPER colors*. 20.02.2019. Google, material.io. URL: https://material.io/tools/color/#!/?view.left=1&view.right=0&primary.color=009688&secondary.color=E91E63.

[68] Grafana. *Grafan Website*. 03.04.2019. URL: https://grafana.com/.

[69] Said Hayani. *Here are the most popular ways to make an HTTP request in JavaScript*. 2019/04/01. Medium. URL: https://medium.freecodecamp.org/here-is-the-most-popular-ways-to-make-an-http-request-in-javascript-954ce8c95aaa.

[70] Heidelpay. *Payment Service Provider diagram*. 03.04.2019. URL: https://www.heidelpay.com/fileadmin/user_upload/bilder_grafiken/was_ist_ein_psp/schaubild_psp.jpg.

[71] Rolland Jannick; L. Holloway Richard; Fuchs Henry. “Comparison of optical and video see-through, head-mounted displays”. In: *Proceedings of SPIE - The International Society for Optical Engineering* (Jan. 1994). DOI: 10.1117/12.197322.

[72] *HTTP request methods*. 2019/04/01. Mozilla. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

[73] Ian Hughes. *It’s only rock n roll but I like it. Snapchat Bitmoji*. 24.02.2019. flickr. URL: https://www.flickr.com/photos/epredator/37087332821.

[74] J. Ingeno. *Software Architect’s Handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing, 2018. ISBN: 9781788627672. URL: https://books.google.at/books?id=6EZsDwAAQBAJ.

[75] *Instant Tracking*. 21.02.2019. Wikitude GmbH. URL: https://www.wikitude.com/external/doc/documentation/7.0/unity/instanttrackingnative.html.

[76] *Interface MongoRepository<T,ID>*. 2019/04/01. Pivotal Software, Inc. URL: https://docs.spring.io/spring-data/data-mongodb/docs/current/api/org/springframework/data/mongodb/repository/MongoRepository.html.

[77] *Introducing managed private networks*. 2019/04/01. Exoscale. URL: https://www.exoscale.com/syslog/introducing-managed-private-networks/.

[78] *Introduction to JSON Web Tokens*. 2019/04/01. Auth0. URL: https://jwt.io/introduction/.

[79] *ITU internet statistcs*. 07.04.2019. ITU. URL: https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx.

[80] Michael Jackson and Ryan Florence. *Modern Web Quote*. 26.03.2019. This Dot Media. URL: https://www.youtube.com/watch?v=Vur2dAFZ4GE&t=913.

[81] *Java Native Access (JNA)*. 2019/04/01. URL: https://github.com/java-native-access/jna.

[82] Jenkins. *Jenkins Website*. 03.04.2019. URL: https://jenkins.io/.

[83] Smita Jhajharia, vaishnavi kannan, and Seema Verma. “Agile vs waterfall: A Comparative Analysis”. In: *ijsetr* 3 (Oct. 2014). 17.03.2019, pp. 2681,2683.

[84] Dominique Righetto John Steven Jim Manico. *Password Storage Cheat Sheet*. 2019/04/01. OWASP. URL: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Password_Storage_Cheat_Sheet.md#ref5.

[85] *JPA Repositories*. 2019/04/01. Pivotal Software, Inc. URL: https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html.

[86] Tech. Sgt. Daryl Knee. *Virtual, augmented reality may hold key to future Air Force training*. 20.02.2019. Air Combat Command Public Affairs. URL: https://www.afmc.af.mil/News/Article-Display/Article/1735608/virtual-augmented-reality-may-hold-key-to-future-air-force-training/.

[87] Dr. Joseph J. LaViola et al. “Analyzing SLAM Algorithm Performance for Tracking in Augmented Reality Systems”. In: 2017.

[88] *Let’s Encrypt is a free, automated, and open Certificate Authority*. 2019/04/01. Internet Security Research Group (ISRG). URL: https://letsencrypt.org/.

[89] Kashumi Madampe. "Successful Adoption of Agile Project Management in Software Development Industry". In: *International Journal of Computer Science and Information Technology Research* 5 (Nov. 2017), pp. 27–33.

[90] Katja Malvoni, Solar Designer, and Josip Knezovic. "Are Your Passwords Safe: Energy-Efficient Bcrypt Cracking with Low-Cost Parallel Hardware". In: *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, 2014. URL: https://www.usenix.org/conference/woot14/workshop-program/presentation/malvani.

[91] Material.io. *Material.io font size definitions*. 27.02.2019. Google. URL: https://material.io/tools/color/.

[92] *Material.io color tool*. 07.04.2019. Google, material.io. URL: https://material.io/tools/color/.

[93] Maven. *Maven Website*. 03.04.2019. URL: https://maven.apache.org/.

[94] *Meaning of The Color Blue*. 07.04.2019. Bourn Creative, LLC. URL: https://www.bourncreative.com/meaning-of-the-color-blue/.

[95] *Meaning of The Color Green*. 07.04.2019. Bourn Creative, LLC. URL: https://www.bourncreative.com/meaning-of-the-color-green/.

[96] *Meaning of The Color Turquoise*. 07.04.2019. Bourn Creative, LLC. URL: https://www.bourncreative.com/meaning-of-the-color-turquoise/.

[97] microservices.io. *Diagram of a microservice application*. 03.04.2019. URL: https://microservices.io/i/Microservice_Architecture.png.

[98] *Microsoft C++ REST SDK*. 2019/04/01. Microsoft. URL: https://github.com/Microsoft/cpprestsdk.

[99] *Mobile and Tablet Android Version Market Share Worldwide*. 26.02.2019. statcounter. URL: http://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide.

[100] *Mobile Operating System Market Share Worldwide*. 26.02.2019. statcounter. URL: http://gs.statcounter.com/os-market-share/mobile/worldwide.

[101] *Moment.js*. 05.04.2019. Moment.js. URL: http://momentjs.com/.

[102] *MongoDB*. 2019/04/01. MongoDB, Inc. URL: https://www.mongodb.com/.

[103] *MongoDB - Auditing*. 2019/04/01. MongoDB, Inc. URL: https://docs.mongodb.com/manual/core/auditing/.

[104] Pierluigi Montagna. *Random Guy Developer Blog*. 06.03.2019. Blogger. URL: http://pievisdev.blogspot.com/2015/05/survival-shooter-in-unity.html.

[105] A. Mouat. *Using Docker*. O'Reilly, 2015. ISBN: 9781491915769. URL: https://books.google.at/books?id=zw2zrQEACAAJ.

[106] muchneeded.com. *Amazon usage statistics*. 03.04.2019. URL: https://muchneeded.com/amazon-stats/.

[107] Snehal Mumbaikar, Puja Padiya, and Department Of Computer Engineering. "Web Services Based On SOAP and REST Principles". In: *International Journal of Scientific and Research Publications (IJSRP)* 3 (2013). ISSN: 2250-3153. URL: http://www.ijsrp.org/research-paper-0513.php?rp=P171217.

[108] NACHA. *2013 NACHA Operating Guidelines*. https://www.firstmid.com/wp-content/uploads/2014/02/2013-Corporate-Rules-and-Guidelines.pdf. 2013.

[109] DI (FH) Philipp Nagele. *Wikitude SDK 8.1: Plane detection, support for iOS 12, Android 9 and more*. 26.02.2019. Wikitude. URL: https://www.wikitude.com/blog-plane-detection-ios12-android9/.

[110] DI (FH) Philipp Nagele. *Wikitude Supported Devices*. 26.02.2019. Wikitude. URL: https://www.wikitude.com/documentation/latest/android/supporteddevices.html.

[111] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015. ISBN: 9781491950333. URL: https://books.google.at/books?id=jjl4BgAAQBAJ.

[112] *ng2-charts package*. 05.04.2019. npm, Inc. URL: https://www.npmjs.com/package/ng2-charts.

[113] *ngx-braintree package*. 05.04.2019. npm, Inc. URL: https://www.npmjs.com/package/ngx-braintree.

[114] Anton Nikolov. *Design principle: Consistency*. 27.02.2019. UX Collective. URL: https://uxdesign.cc/design-principle-consistency-6b0cf7e7339f.

[115] *OpenCV introduction*. 26.02.2019. OpenCV. URL: https://opencv.org/.

[116] Janne Paavilainen et al. "The Pokémon GO Experience: A Location-Based Augmented Reality Mobile Game Goes Mainstream". In: 20.02.2019. May 2017. DOI: 10.1145/3025453.3025871.

[117] PayPal. *PayPal API PyPI install package*. 03.04.2019. URL: https://pypi.org/project/paypal/.

[118] PayPal. *PayPal Website*. 03.04.2019. URL: https://www.paypal.com.

[119] C. Percival and S. Josefsson. *The scrypt Password-Based Key Derivation Function*. RFC 7914. RFC Editor, 2016.

[120] COLIN PERCIVAL. "Stronger key derivation via sequential memory-hard functions". In: (Jan. 2009).

[121] *Perfect Password - GRC's Ultra High Security Password Generator*. 2019/04/01. Gibson Research Corporation. URL: https://www.grc.com/passwords.htm.

[122] Thad Peterson and Ron van Wezel. *IBM Systems Virtualization: Servers, Storage, and Software*. 1st ed. Redbooks, 2008, p. 15. URL: http://www.redbooks.ibm.com/redpapers/pdfs/redp4396.pdf.

[123] Thad Peterson and Ron van Wezel. *The Evolution of Digital and Mobile Wallets*. 1st ed. MAHINDRA COMVIVA, 2016, p. 4. URL: https://www.paymentscardsandmobile.com/wp-content/uploads/2016/10/The-Evolution-of-Digital-and-Mobile-Wallets.pdf.

[124] *POCO C++ Libraries - Simplify C++ Development*. 2019/04/01. Applied Informatics Software Engineering GmbH. URL: https://pocoproject.org/index.html.

[125] Ved Prakash Gulati and Shilpa Srivastava. "The Empowered Internet Payment Gateway". In: (Mar. 2019).

[126] Michele Preziuso. *Password Hashing: Scrypt, Bcrypt and ARGON2*. 2019/04/01. Medium. URL: https://medium.com/@mpreziuso/password-hashing-pbkdf2-scrypt-bcrypt-and-argon2-e25aaf41598e.

[127] Justin Pritchard. *ACH Payments Can Benefit Everybody. See How They Work*. URL: https://www.thebalance.com/how-ach-payments-work-315441.

[128] Stefan Prodan. *Dockprom Github Repository*. 03.04.2019. URL: https://github.com/stefanprodan/dockprom.

[129] Prometheus. *Prometheus Website*. 03.04.2019. URL: https://prometheus.io/.

[130] Christian Szegedy; Wei Liu; Yangqing Jia; Pierre Sermanet; Scott E. Reed; Dragomir Anguelov; Dumitru Erhan; Vincent Vanhoucke; Andrew Rabinovich. "Going Deeper with Convolutions". In: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842.

[131] *Router and Filter: Zuul*. 2019/04/01. Pivotal Software, Inc. URL: https://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#_router_and_filter_zuul.

[132] *RxJS library*. 05.04.2019. ReactiveX. URL: https://rxjs-dev.firebaseapp.com/.

[133] Subhrangshu S. Sarkar. "Technological Innovations in Indian Banking Sector-A Trend Analysis". In: *Journal of Commerce and Management Thought* 7 (Jan. 2016), p. 171. DOI: 10.5958/0976-478X.2016.00012.4.

[134] *Salted Password Hashing - Doing it Right*. 2019/04/01. Defuse Security. URL: https://crackstation.net/hashing-security.htm.

[135] *Samsung Gear 360*. 13.3.2019. Samsung Newsroom. URL: https://www.flickr.com/photos/samsungtomorrow/24806330129.

[136] *Service Discovery: Eureka Clients*. 2019/04/01. Pivotal Software, Inc. URL: https://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#_service_discovery_eureka_clients.

[137] *Service Discovery: Eureka Server*. 2019/04/01. Pivotal Software, Inc. URL: https://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#spring-cloud-eureka-server.

[138] *Share AR Experiences with Cloud Anchors*. 24.02.2019. Google. URL: https://developers.google.com/ar/develop/java/cloud-anchors/overview-android.

[139] Scott Stein; Ian Sherr. *Why AR is going to give you 'superpowers' in the future*. 13.03.2019. CNET. Feb. 2019. URL: https://www.cnet.com/news/the-future-of-ar-according-to-microsoft/.

[140] Sanni Siltanen. "Theory and applications of marker based augmented reality". PhD thesis. Jan. 2012, pp. 38, 40, 54. URL: https://www.vtt.fi/inf/pdf/science/2012/S3.pdf.

[141] similartech. *Payment Service Provider marketshare data*. 03.04.2019. URL: https://www.similartech.com/categories/payment.

[142] *Skeuomorphism*. 30.03.2019. Techopedia Inc. URL: https://www.techopedia.com/definition/28955/skeuomorphism.

[143] *SOAP vs. REST: The Differences and Benefits Between the Two Widely-Used Web Service Communication Protocols*. 2019/04/01. Stackify. URL: https://stackify.com/soap-vs-rest/.

[144] Spencer Gibb. *How is Spring Cloud Gateway different from Zuul?* 2019/04/01. Stackoverflow. URL: https://stackoverflow.com/questions/47092048/how-is-spring-cloud-gateway-different-from-zuul.

[145] Spring. *Spring testing documentation*. 05.04.2019. URL: https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html.

[146] *Spring Boot features - Testing*. 2019/04/01. Pivotal Software, Inc. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html.

[147] *Spring Cloud*. 2019/04/01. Pivotal Software, Inc. URL: https://spring.io/projects/spring-cloud.

[148] *Spring Cloud Config*. 2019/04/01. Pivotal Software, Inc. URL: https://spring.io/projects/spring-cloud-config.

[149] *Spring Data*. 2019/04/01. Pivotal Software, Inc. URL: https://spring.io/projects/spring-data.

[150] *Spring Data JPA - Reference Documentation*. 2019/04/01. Pivotal Software, Inc. URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/.

[151] statcounter. *Mobile and Tablet iOS Version Market Share Worldwide*. 26.02.2019. statcounter. URL: http://gs.statcounter.com/ios-version-market-share/mobile-tablet/worldwide.

[152] *std::future*. 2019/04/01. cppreference.com. URL: https://en.cppreference.com/w/cpp/thread/future.

[153] *std::promise*. 2019/04/01. cppreference.com. URL: https://en.cppreference.com/w/cpp/thread/promise.

[154] Stripe. *Stripe Website*. 03.04.2019. URL: https://stripe.com.

[155] *The Top 10 video Game Engines*. 2019/04/01. GameDesign. URL: https://www.gamedesigning.org/career/video-game-engines/.

[156] *The Ultimate VR Headset Comparison Table: Every VR Headset Compared*. 7.04.2019. ThreeSixtyCameras. URL: http://www.threesixtycameras.com/vr-headset-comparison-table/.

[157] *The world's leading real-time creation platform*. 27.02.2019. Unity Technologies. URL: https://unity3d.com/unity.

[158] *Toggl*. 2019/04/01. Toggl. URL: https://toggl.com/.

[159] *Travel VR: Explore the world on your couch*. 31.1.2018. OmniVirt. URL: https://www.omnivirt.com/blog/top-travel-tourism-virtual-reality-vr-examples.

[160] *Travel VR: Explore the world on your couch*. 31.1.2018. OmniVirt. URL: https://www.omnivirt.com/blog/top-travel-tourism-virtual-reality-vr-examples.

[161] TravisCI. *TravisCI Website*. 03.04.2019. URL: https://travis-ci.com/.

[162] *Trion - ng-cli-karma docker container*. 07.04.2019. DockerHub. URL: https://hub.docker.com/r/trion/ng-cli-karma/.

[163] *Type definitions for braintree-web package*. 07.04.2019. Braintree. URL: https://www.npmjs.com/package/@types/braintree-web.

[164] *Unity Lincense Pricing*. 7.04.2019. Unity Technologies. URL: https://store.unity.com/.

[165] *Unity Test Runner*. 07.03.2019. Unity. Mar. 2018. URL: https://docs.unity3d.com/Manual/testing-editortestsrunner.html.

[166] *Unreal Engine EULA*. 24.04.2019. Epic Games Inc. URL: https://www.unrealengine.com/en-US/eula.

[167] *Using HTTP Methods for RESTful Services*. 2019/04/01. RestApiTutorial.com. URL: https://www.restapitutorial.com/lessons/httpmethods.html.

[168] *Virtual Reality in Education*. 7.04.2019. Avantis Systems Ltd. URL: http://www.classvr.com/virtual-reality-in-education/.

[169] *Virtual Reality in Military*. 7.04.2019. ThinkMobiles. URL: https://thinkmobiles.com/blog/virtual-reality-military/.

[170] A. Visconti and F. Gorla. "Exploiting an HMAC-SHA-1 optimization to speed up PBKDF2". In: *IEEE Transactions on Dependable and Secure Computing* (2018), pp. 1–1. ISSN: 1545-5971. DOI: 10.1109/TDSC.2018.2878697.

[171] *Vuforia AR Features*. 13.03.2019. PTC. URL: https://vuforia.com/features.

[172] *Wikitude SDK Full Features Overview*. 13.03.2019. Wikitude. URL: https://www.wikitude.com/products/wikitude-sdk-features/.

[173] *Working with Spring Data Repositories*. 2019/04/01. Pivotal Software, Inc. URL: https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html.

[174] *ZenHub*. 2019/04/01. ZenHub. URL: https://www.zenhub.com/.

List of Figures

List of Tables

Listings

Appendix

A Augmented Reality

A.1 Protocol for manual testing

	Input	**Expected Output**	**Actual Output**	
1	Start the application	See the login screen	Login screen visible	Y
2	Click the "Skip login and try demo-version"-button	Start menu + welcome message	Start menu seen and short information about active user	Y
3	Click "logout"-button	Return to login screen	Login screen visible	Y
4	Login with "user1" (username) and "1234" (password)	Start menu + welcome message	Start menu seen and same welcome message as with skipping	Y
5	Click on settings	See settings screen	Settings-screen visible	Y
6	Click the "Back"-button	Return to start menu	Start menu visible	Y
7	Click on "Buy virtual items"	First demo starts (with tutorial)	Shows screen that says "AR-Demo 1: Buy virtual items"	Y
8	Click on "Start" / "Next" / "Let's go"	Shows the actual demo (with grid)	Shows the demo with a "Trial"-overlay	Y
9	Move the phone around	The grid turns green	The grid turns green	Y
10	Click the initialize button	The grid locks in at a specific point + buttons appear at the bottom	Grid does not move around anymore but stands still; also, buttons appear on the bottom	Y
11	Drag a button onto the grid	A virtual object appears and is placed on the grid	Virtual object appears and is placed on the grid	Y

12	Click on the object	Frame appears around the object	Frame appears around the object (sometimes a bit buggy)	Y
13	Drag the object around	Object moves around	Object moves relative to the finger	Y
14	Scale the object with two fingers	Object changes size	Object changes size	Y
15	Rotate the object with two fingers	Object rotates	Object rotates (very fast)	Y
16	Drag object to the trash can	Object disappears	Object disappears	Y
17	*Click the clear all items button	All objects disappear	All objects disappear	Y
18	Click the reset button	Grid resets from locked state	Grid unlocks and is now freely movable again	Y
19	**Click "Purchase Items"-button	Screen with selected item(s) appears	Background darkens and screen with selected item appears	Y
20	Click on "Purchase"-button	Displays select-payments screen	Displays select-payments screen with one available payment-option	Y
21	Click on "Select"-button	Displays verification screen	Displays verification-screen	Y
22	Insert password "1234"	Verification finishes successfully	Verification finishes successfully	Y
23	***Insert password "9876"	Verification finishes with an error	Verification finishes with an error-message	Y
24	Close the application and reopen it	Start menu with login message	Start menu with login message for "user1"	Y
25	Click on "Buy real items"-button	Second application starts	Blank screen with "trial"-messages starts	Y
26	Scan QR-code	See 19.****	Payment process finishes	Y

Table 9.1: Manual testing protocol

* Insert multiple objects before input
** Insert object and click on it before input
*** Repeat steps 19-21 before input
**** Also do steps 19-22 + 23 afterwards

“Y” ... test succeeds
“N” ... test fails

B Back End and System Design

B.1 Testing

```
1 @RunWith(SpringRunner.class)
2 public class TestController {
3 TestRestTemplate restTemplate = new TestRestTemplate();
4 HttpHeaders headers = new HttpHeaders();
5 @Test
6 public void testGetPaymentAccounts() throws Exception {
7 HttpEntity<String> entity = new HttpEntity<String>(null,
    headers);
8 ResponseEntity<String> response = restTemplate.exchange(
9 createURLWithPort("/accounts"), HttpMethod.GET, entity, String
    .class);
10 String expected = "{\"id\":1,\"description\":\"My Account\"}";
11 JSONAssert.assertEquals(expected, response.getBody(), false);
12 }
13 }
```

Listing 9.1: Testing a ReST interface in Spring

B.2 Database

```
@Document
public class User {

  @Id
  private ObjectId _id;

  @NotNull
  private Name name;

  private boolean verified;

  @DBRef(lazy = true)
  private Map<String, UserSetting> userSettings;

  private Map<String, PaymentMethod> paymentMethods;


  public User() {
    this.verified = false;
    this.paymentLock = false;
    this.userSettings = new HashMap<>();
    this.paymentMethods = new HashMap<>();
  }

  // Getter and setter methods were removed
}
```

Listing 9.2: Example of a document class

B.3 Feign Client

```
1 @FeignClient("authentication")
2 public interface AuthenticationService {
3
4   @PostMapping("user")
5   feign.Response createUser(@RequestBody UserCredentials
       credentials);
6
7   @DeleteMapping("user/{id}")
8   Response deleteUser(@PathVariable("id") ObjectId id);
9
10  @PatchMapping("user/{id}")
11  Response updateUser(@PathVariable("id") ObjectId id,
       @RequestBody Map<String, Object> changes);
12
13  @PatchMapping("user/{id}/pw")
14  Response changeUserPassword(@PathVariable("id") ObjectId id,
        ChangePasswordRequest request);
15
16  @GetMapping("test-con")
17  Response testConnection();
18
19  @GetMapping("user/{id}")
20  Response<AuthenticationUserDetails> getUserDetails(
       @PathVariable("id") ObjectId id);
21 }
```

Listing 9.3: Example of a Feign client

B.4 Authentication Service

```
@EnableWebSecurity
public class SecurityConfig extends
    WebSecurityConfigurerAdapter {

  @Autowired
  @Qualifier("UserDetailsServiceImpl")
  private UserDetailsService userDetailsService;

  @Autowired
  private JwtConfig jwtConfig;

  @Override
  protected void configure(HttpSecurity http) throws Exception
      {
    http
    .csrf().disable()
    // make sure we use stateless session
    .sessionManagement().sessionCreationPolicy(
       SessionCreationPolicy.STATELESS)
    .and()
    // handle unauthorized attempts
    .exceptionHandling().authenticationEntryPoint((req, rsp, e
       ) -> rsp.sendError(HttpServletResponse.SC_UNAUTHORIZED)
       )
    .and()
    // Add a filter to validate user credentials
    .addFilter(new JwtUsernameAndPasswordAuthenticationFilter(
       authenticationManager(), jwtConfig))
    .authorizeRequests()
    .antMatchers(HttpMethod.POST, jwtConfig.getUri()).
       permitAll()
    .antMatchers("/user/**").permitAll()
    .antMatchers("/dev/**").permitAll()
    .antMatchers("/app/**").permitAll()
    .antMatchers(HttpMethod.GET, "/test-con").permitAll()
    .antMatchers(HttpMethod.GET, "/actuator/**").permitAll()
    // any other requests must be authenticated
    .anyRequest().authenticated();
  }

  // Spring has UserDetailsService interface, which can be
     overridden to provide our implementation for fetching
     user from database
  @Override
```

```
36   protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
37     auth.userDetailsService(userDetailsService).
          passwordEncoder(sCryptPasswordEncoder());
38   }
39
40   @Bean
41   public JwtConfig jwtConfig() {
42     return new JwtConfig();
43   }
44
45   @Bean
46   public SCryptPasswordEncoder sCryptPasswordEncoder() {
47     return new SCryptPasswordEncoder();
48   }
49 }
```

Listing 9.4: Full security configuration of the authentication service

```
1 public class JwtUsernameAndPasswordAuthenticationFilter
    extends UsernamePasswordAuthenticationFilter {
2
3   private AuthenticationManager authManager;
4
5   private final JwtConfig jwtConfig;
6
7   public JwtUsernameAndPasswordAuthenticationFilter(
      AuthenticationManager authManager, JwtConfig jwtConfig) {
8     this.authManager = authManager;
9     this.jwtConfig = jwtConfig;
10  }
11
12  @Override
13  public Authentication attemptAuthentication(
      HttpServletRequest request, HttpServletResponse response)
       throws AuthenticationException {
14    try {
15      // 1. Get credentials from request
16      UserCredentials userCredentials = new ObjectMapper().
          readValue(request.getInputStream(), UserCredentials.
          class);
17
18      // 2. Create auth object for auth manager
19      UsernamePasswordAuthenticationToken authToken = new
          UsernamePasswordAuthenticationToken(userCredentials.
          getIdentification(), userCredentials.getPassword(),
          Collections.emptyList());
20
21      // 3. Authentication manager authenticate the user
22      return authManager.authenticate(authToken);
23
24    } catch (IOException e) {
25      throw new RuntimeException(e);
26    }
27  }
28
29  // Upon successful authentication, generate a token
30  @Override
31  protected void successfulAuthentication(HttpServletRequest
      request, HttpServletResponse response, FilterChain chain,
       Authentication auth) throws IOException,
      ServletException {
32    String token = createJwt(auth.getName(),
33              auth.getAuthorities().stream().map(
```

```
                    GrantedAuthority::getAuthority).collect(
                    Collectors.toList()),
34                jwtConfig.getExpiration(),
35                jwtConfig.getSecret());
36
37      // Add token to header
38      response.addHeader(jwtConfig.getHeader(), jwtConfig.
           getPrefix() + token);
39    }
40
41    public static String createJwt(String sub, List<String> auth
         , int exp, String secret) {
42      long now = System.currentTimeMillis();
43
44      return Jwts.builder()
45          .setSubject(sub)
46          .claim("authorities", auth)
47          .setIssuedAt(new Date(now))
48          .setExpiration(new Date(now + exp * 1000))
49          .setIssuer("com.viper.service.authentication")
50          .signWith(SignatureAlgorithm.HS512, secret.getBytes())
51          .compact();
52    }
53  }
```

Listing 9.5: The authentication filter used to authentication users and create new JWTs

```java
@Service
public class UserDetailsServiceImpl implements
    UserDetailsService {

  @Autowired
  private DeveloperRepository developerRepository;

  @Autowired
  private UserRepository userRepository;

  @Override
  public UserDetails loadUserByUsername(String identification)
      throws UsernameNotFoundException {
    List<GrantedAuthority> grantedAuthorities = new ArrayList
        <>();

    Optional<User> optionalUser = userRepository.
        findByEmailOrUsername(identification, identification);

    if(optionalUser.isPresent()) {
      User user = optionalUser.get();

      grantedAuthorities.add(new SimpleGrantedAuthority("
          ROLE_USER"));

      return new org.springframework.security.core.userdetails
          .User(user.get_id().toHexString(), user.getPassword()
          , grantedAuthorities);
    } else {
      Optional<Developer> optionalDeveloper =
          developerRepository.findByEmail(identification);

      if (optionalDeveloper.isPresent()) {
        Developer developer = optionalDeveloper.get();

        grantedAuthorities.add(new SimpleGrantedAuthority("
            ROLE_DEV"));

        if (developer.isAdmin())
          grantedAuthorities.add(new SimpleGrantedAuthority("
              ROLE_ADMIN"));

        return new org.springframework.security.core.
            userdetails.User(developer.get_id().toHexString(),
            developer.getPassword(), grantedAuthorities);
```

```
34         } else {
35           throw new UsernameNotFoundException("Identification "
                + identification + " not found");
36         }
37       }
38     }
39 }
```

Listing 9.6: The user details service used to retrieve user infromation form the database

B.5 API Gateway Service

```
1  public class JwtTokenAuthenticationFilter extends
       OncePerRequestFilter {
2
3    private final JwtConfig jwtConfig;
4
5    public JwtTokenAuthenticationFilter(JwtConfig jwtConfig) {
6      this.jwtConfig = jwtConfig;
7    }
8
9    @Override
10   protected void doFilterInternal(HttpServletRequest request,
         HttpServletResponse response, FilterChain chain) throws
         ServletException, IOException {
11     // 1. get the authentication header
12     String header = request.getHeader(jwtConfig.getHeader());
13
14     // 2. validate the header and check the prefix
15     if(header == null || !header.startsWith(jwtConfig.
           getPrefix())) {
16       // If not valid, go to the next filter
17       chain.doFilter(request, response);
18       return;
19     }
20
21     // 3. Get the token
22     String token = header.replace(jwtConfig.getPrefix(), "");
23
24     // exceptions might be thrown in creating the claims if
           for example the token is expired
25     try {
26       // 4. Validate the token
27       Claims claims = Jwts.parser()
28           .setSigningKey(jwtConfig.getSecret().getBytes())
29           .parseClaimsJws(token)
30           .getBody();
31
32       String id = claims.getSubject();
33       if(id != null) {
34         @SuppressWarnings("unchecked")
35         List<String> authorities = (List<String>) claims.get("
               authorities");
36
37         // 5. Create auth object
38         UsernamePasswordAuthenticationToken auth = new
               UsernamePasswordAuthenticationToken(id, null,
```

```
                authorities.stream().map(SimpleGrantedAuthority::
                new).collect(Collectors.toList()));
39
40              // 6. Authenticate the user
41              SecurityContextHolder.getContext().setAuthentication(
                    auth);
42          }
43
44      } catch (Exception e) {
45          // In case of failure. Making sure it's clear
46          SecurityContextHolder.clearContext();
47      }
48
49      // go to the next filter in the filter chain
50      chain.doFilter(request, response);
51  }
52
53 }
```

Listing 9.7: The authentication filter used to authorize requests at the gateway service

```
1  public class AuthFilter extends ZuulFilter {
2
3    private static Logger log = LoggerFactory.getLogger(
       AuthFilter.class);
4
5    private static List<String> adminRestricted = new ArrayList<
       String>(){{
6      add("org");
7      add("dev");
8      add("payment");
9    }};
10
11
12   @Override
13   public String filterType() {
14     return "pre";
15   }
16
17   @Override
18   public int filterOrder() {
19     return 0;
20   }
21
22   @Override
23   public boolean shouldFilter() {
24     RequestContext ctx = RequestContext.getCurrentContext();
25     HttpServletRequest request = ctx.getRequest();
26
27     return request.getUserPrincipal() != null;
28   }
29
30   @Override
31   public Object run() {
32     RequestContext ctx = RequestContext.getCurrentContext();
33     HttpServletRequest request = ctx.getRequest();
34
35     if(!authorize(request)) {
36       ctx.setSendZuulResponse(false);
37       ctx.setResponseStatusCode(HttpStatus.SC_FORBIDDEN);
38     }
39
40     ctx.addZuulRequestHeader("ID", request.getUserPrincipal().
         getName());
41
42     return null;
```

```
43    }
44
45    private boolean authorize(HttpServletRequest request) {
46      @SuppressWarnings("unchecked")
47      List<GrantedAuthority> roles = (List<GrantedAuthority>)
          SecurityContextHolder.getContext().getAuthentication().
          getAuthorities();
48
49      String path = request.getServletPath();
50
51      String base = path.substring(1);
52      int si = base.indexOf('/');
53
54      String endpoint;
55
56      if(si >= 0) {
57        endpoint = base.substring(si + 1);
58        if(endpoint.indexOf('/') >= 0)
59          endpoint = endpoint.substring(0, endpoint.indexOf('/')
              );
60
61        base = base.substring(0, si);
62      } else {
63        endpoint = "/";
64      }
65
66      switch (base) {
67        case "api":
68        case "customer":
69          if(!request.isUserInRole("USER"))
70            return false;
71
72          break;
73        case "developer":
74          if(!request.isUserInRole("DEV"))
75            return false;
76
77          if(adminRestricted.contains(endpoint) && !request.
              isUserInRole("ADMIN"))
78            return false;
79      }
80
81      return true;
82    }
83  }
```

Listing 9.8: Zuul filter that blocks forbidden requests and adds an ID header to all request

B.6 Eureka service

```
# server port
server.port=8761

# logging
logging.level.root=info

# management endpoint configuration
management.endpoints.enabled-by-default=false
management.endpoint.health.enabled=true
management.endpoint.info.enabled=true
management.endpoints.web.exposure.include=health,info

# Eureka server configuration
# ---------------------------

# configuration for a single Eureka server
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false

# zone configuration
eureka.client.region=at-vie-1
eureka.client.availability-zones.at-vie-1=default
eureka.client.service-url.defaultZone=http://localhost:8761/
    eureka/
eureka.instance.metadata-map.zone=default

# cache configuration
eureka.server.retention-time-in-m-s-in-delta-queue=120000
eureka.server.delta-retention-timer-interval-in-ms=15000
eureka.server.response-cache-auto-expiration-in-seconds=120
eureka.server.disable-delta=false

# self preservation configuration
eureka.server.enable-self-preservation=false
eureka.server.renewal-percent-threshold=0.7
eureka.server.renewal-threshold-update-interval-ms=30000
```

Listing 9.9: Configuration of the Eureka Server

B.7 Eureka Client

```
# server port
server.port=8000

# management endpoint configuration
management.endpoints.enabled-by-default=false
management.endpoint.info.enabled=true
management.endpoint.health.enabled=true
management.endpoints.web.exposure.include=health,info

# Eureka client configuration
# ---------------------------

# Eureka server connection configuration
eureka.client.register-with-eureka=true
eureka.client.use-dns-for-fetching-service-urls=false
eureka.client.should-unregister-on-shutdown=true
eureka.client.allow-redirects=true
eureka.client.filter-only-up-instances=true

# zone configuration
eureka.client.prefer-same-zone-eureka=true
eureka.client.region=at-vie-1
eureka.client.availability-zones.at-vie-1=default
eureka.client.service-url.default=http://194.182.175.254:8761/
    eureka
eureka.client.service-url.defaultZone=http://localhost:8761/
    eureka

# healthcheck configuration
eureka.client.healthcheck.enabled=true
eureka.client.heartbeat-executor-exponential-back-off-bound=10
eureka.client.on-demand-update-status-change=true

# registry fetch configuration
eureka.client.fetch-registry=true
eureka.client.registry-fetch-interval-seconds=15
eureka.client.cache-refresh-executor-exponential-back-off-
    bound=10

# instance info replication configuration
eureka.client.instance-info-replication-interval-seconds=15
eureka.client.initial-instance-info-replication-interval-
    seconds=30

# timeouts configuration
```

```
42 eureka.client.eureka-server-read-timeout-seconds=4
43 eureka.client.eureka-server-connect-timeout-seconds=3
44 eureka.client.eureka-connection-idle-timeout-seconds=25
45
46 # Eureka instance configuration
47 # ----------------------------
48
49 # metadata configuration
50 eureka.instance.metadata-map.zone=default
51
52 # IP configuration
53 eureka.instance.prefer-ip-address=true
54 eureka.instance.ip-address=${HOST_IP}
55
56 # ports configuration
57 eureka.instance.non-secure-port=8000
58 eureka.instance.secure-port=8443
59 eureka.instance.non-secure-port-enabled=true
60 eureka.instance.secure-port-enabled=false
```

Listing 9.10: Eureka Client configuration

B.8 Client Library

```
1  json::value body_data;
2  body_data[L"property"] = json::value(L"value");
3
4  http_request request(methods::POST);
5  request.set_body(body_data);
6
7  bool *success = new bool[1];
8  success[0] = false;
9
10 http_client(L"https://api.viperpayment.com/path/to/endpoint").
       request(request)
11   .then([=](http_response response) -> pplx::task<
         http_response> {
12   if (response.status_code() == status_codes::OK)
13     success[0] = true;
14
15   return pplx::task_from_result(response);
16 })
17   .then([=](pplx::task<http_response> previous_task) {
18   if (success[0]) {
19     try {
20       ...
21     }
22     catch (const http_exception& e) {
23       wostringstream ss;
24       ss << e.what() << endl;
25       wcout << ss.str();
26
27       success[0] = false;
28     }
29   }
30
31   delete[] success;
32 });
```

Listing 9.11: Basic structure of any HTTP request made with the cpprestsdk