St. Pölten University of Applied Sciences

/medien & digitale technologien

/fh/// st.pölten

Bringing Your Own Device into Multi-Device Ecologies

Technical Concept, Implementation and Challenges

Diploma Thesis

For attainment of the academic degree of

Dipl.-Ing. für technisch-wissenschaftliche Berufe

in the Masters Course Digital Media Technologies at St. Pölten University of Applied Sciences, **specialized area Mobile Internet**

Submitted by:

Niklas Thür, BSc

dm1710262558

Advisor and First Assessor:	FH-Prof. Dipl.-Ing. Dr. Markus Seidl, Bakk.
Second Assessor:	Dipl.-Ing. Kerstin Blumenstein, BSc

Biedermannsdorf, 11.09.2019

Declaration

- The attached research paper is my own, original work undertaken in partial fulfillment of my degree.

- I have made no use of sources, materials or assistance other than those which have been openly and fully acknowledged in the text. If any part of another person's work has been quoted, this either appears in inverted commas or (if beyond a few lines) is indented.

- Any direct quotation or source of ideas has been identified in the text by author, date, and page number(s) immediately after such an item, and full details are provided in a reference list at the end of the text.

- I understand that any breach of the fair practice regulations may result in a mark of zero for this research paper and that it could also involve other repercussions.

Date: 11.Sept 2019 Signature: 

Abstract

The field of smartphone integration in multi-device ecologies (MDE) has recently attracted much research interest. In multi-device ecologies various types of devices can communicate with each other and are aware of their mutual presence. In this context, many concepts focus on the integration of the user's own device. This approach is called "bring your own device" (BYOD). Nowadays, multiple museums offer their visitors interactive exhibitions. These might include a guide app or interactive surfaces like a multi-touch table. However, very few applications which combine the concepts MDE and BYOD exist in museums or similar cultural events. When visiting a museum nearly every visitor brings their own mobile device such as a smartphone or tablet. Therefore, using the visitor's own device in an interactive exhibition is very practical. Unfortunately, there is currently no suitable infrastructure to seamlessly link different types of devices in interactive exhibitions. This work proposes a technical concept and key elements for developing a mutli-device ecology with the focus on bring your own device. The technical concept and the key elements were developed in the MEETeUX project. In addition, an own multi-device ecology was developed for the annual exhibition of the monastery of Klosterneuburg. The proof of concept includes various key elements which are a subset of the key elements of the MEETeUX project and represent an essential part of an MDE for a museum. The proof of concept was tested in a five-month evaluation phase in the monastery. The evaluation focuses on the technical functioning of the MDE and showed that the system still had some vulnerabilities. Nevertheless, after fixing these, the MDE proofed that it is a suitable infrastructure to seamlessly link different interactive devices in museums. Still there are opportunities for improvement.

Kurzfassung

Das Gebiet der Smartphone-Integration in ”Multi-Device-Ecologies” (MDE) hat in letzter Zeit großes Forschungsinteresse auf sich gezogen. In Mehrgeräteökologien können verschiedene Arten von Geräten miteinander kommunizieren und sind sich ihrer gegenseitigen Präsenz bewusst. In diesem Bereich konzentrieren sich viele Konzepte auf die Integration eines eigenen Geräts der Benutzer. Dieser Ansatz wird als "Bring Your Own Device" (BYOD) bezeichnet. Heutzutage bieten viele Museen ihren Besuchern eine interaktive Ausstellung. Dies kann eine Guide-App oder ein interaktives Ausstellungsstück wie ein Multitouch-Tisch sein. Es gibt jedoch nur sehr wenige Anwendungen, die die Konzepte MDE und BYOD kombinieren in Museen oder ähnlichen kulturellen Umgebungen. Beim Museumsbesuch bringt fast jeder Besucher sein eigenes Mobilgerät, etwa ein Smartphone oder Tablet, mit. Daher wäre es sehr praktisch, das Gerät des Besuchers in einer interaktiven Ausstellung zu verwenden. Leider gibt es derzeit keine geeignete Infrastruktur, um verschiedene Gerätetypen in interaktiven Ausstellungen nahtlos miteinander zu verknüpfen. Diese Arbeit schlägt ein technisches Konzept und Schlüsselelemente für ein MDE mit Fokus auf BYOD vor, die auf dem MEETeUX-Projekt basieren. Zusätzlich wurde für die Jahresausstellung des Stift Klosterneuburgs eine eigene Mehrgeräteökologie mit dem Schwerpunkt BYOD entwickelt. Der Proof of Concept umfasst verschiedene Schlüsselelemente welche ein Subset der Schlüsselelemente des MEETeUX Projektes sind und die ein wesentlicher Bestandteil einer MDE für ein Museum darstellen. Der Wirksamkeitsnachweis wurde in einer fünfmonatigen Evaluierungsphase im Kloster durchgeführt. Die Evaluierung konzentriert sich auf die technische Funktionsweise des MDE und hat gezeigt, dass das System noch einige Schwachstellen aufweist. Trotzdem hat das MDE nach Behebung der Schwachstellen bewiesen, dass es eine geeignete Infrastruktur ist, um verschiedene interaktive Geräte in Museen nahtlos zu verbinden. Dennoch sind Verbesserungsmöglichkeiten vorhanden.

Contents

1 Introduction

The field of smartphone integration in multi-device ecologies has recently attracted much research interest. Bellucci et al. (2014) presented a survey, which provided a detailed overview of the state of the art in the form of a classification of the interaction possibilities between humans and the surrounding computational environment. Many concepts focus on the integration of the user's own device, for example (Shirazi et al. 2009) have linked a poker game on a touch table with the players' smartphones. Other researchers propose methods to control touch tables and / or big wall displays with the own device (e.g. Echtler et al. (2009), Kray et al. (2010), Seyed, Burns, et al. (2012), Winkler et al. (2014)). Many multi-device applications are used in a variety of areas. However, there are very few applications that can be used for museums or similar cultural events. Examples of the few works dealing with this area are the works of Koukopoulos and Koukoulis (2018), Othman et al. (2018) or Petrelli et al. (2017). A very promising work, is the survey of Kosmopoulos and Styliaras (2018). In their work they surveyed the latest advancements in the fields of indoor location, recommendation systems, guidance systems, content storage and presentation for museums.

The subject of this thesis are technical concepts for multi-device ecologies in museums where the user can bring their own device (BYOD) (see chapter 2). Many museums offer interactive exhibits such as touch displays for which various applications can be developed, e.g., a quiz. For example, the Museum of Lower Austria offers its visitors a quiz on two multi-touch tables. There is a version for children, and one for adults with a total number of ten questions. In the end, the user sees how many of the questions were answered correctly.

Unfortunately, most of these exhibits are not communicating with each other and therefore only provide a limited experience for the visitors. Even if the exhibits are linked together, the visitors' own devices are rarely used as a part of a device ecology. For example, the Jewish Museum in Vienna offers its visitors multiple exhibits like touch displays or interactive video walls, but the interaction is only possible with the museum's preconfigured tablets which can be borrowed at the reception.

In addition, many museums only use these active exhibits to enhance the experience of the visitors. An active exhibit could be a multi-touch display or touch displays in general. They are exhibits with which the user can directly interact. However, hardly any museum

uses the possibility of using exhibits as a passive exhibit. A passive exhibit could be e.g., a painting for which the user can get additional information in the form of an augmented reality overlay. The Celtic Museum Hallein offers an app which can be installed on the user's own device and then uses visual markers to display a 3D character in the app. The character tells stories of his life, but there is no other interaction possibility. The examples show that there are currently no suitable infrastructures to seamlessly link different devices in museums, especially with the focus on BYOD.

Therefore this thesis deals first with the questions, "Which interactive exhibits currently exist in museums?" and "Which technical concepts are used?". To answer these questions, several museums which include interactive exhibits were visited as a part of the MEETeUX project. Furthermore, for some museums, it was possible to research the functionality of the exhibition through their website.

In the next step, this thesis discusses a technical concept for a multi-device ecology with a focus on BYOD which will be presented in chapter 3. The concept includes multiple technologies which could be used to e.g., authenticate the user on active exhibits. The authentication could be implemented with Bluetooth beacons, RFID readers, or optical markers. These possibilities will be presented and discussed. Furthermore, chapter 3 will propose the different parts of the MDE and how these parts could be developed.

The central part of this work will focus on the implementation of an own MDE with the focus on BYOD. The MDE was developed in a follow-up project of the MEETeUX project and will be used in the annual exhibition of 2019 in Klosterneuburg. The proof of concept includes an app which can be installed on the visitors' own devices. It includes a server which manages the whole exhibition and furthermore it includes various active, interactive, and passive exhibits. For these parts various key elements were defined, each with its corresponding research question. Each key element will be described in detail, and moreover, the interaction between the various components will be presented in chapter 4.

Finally, the MDE was evaluated in an evaluation period starting in March and ending in July 2019 (see chapter 5). The thesis will present the used setup for the evaluation as well as the monitoring approach. In addition, the found bugs will be discussed, and some statistics of the exhibition will be presented. The last chapter presents the challenges of the developed infrastructure and open challenges which should be addressed in future work (see chapter 6).

1.1 MEETeUX Project

The MEETeUX project discusses future-relevant issues in the fields of interaction design and user experience design for the integrated use of media technology devices (mobile

devices, multi-touch tabletops, large areas) in multi-device ecologies (MEETeUX 2019).

In semi-public spaces, there is a lack of tested interaction designs and user experience concepts that make it challenging to apply the technologies to the potential user and the beneficial use of those devices. MEETeUX will facilitate this by designing usage scenarios for the integration of various endpoints demonstrating the use of multi-device ecologies in semi-public spaces. Therefore, MEETeUX focuses on the integration of the user's devices ("bring your own device" (BYOD), usual smartphones) into existing device ecologies. The use of their own equipment allows adaptation to the specific needs of the user and thus improves accessibility. In MEETeUX, selected scenarios, e.g., for the usage of knowledge transfer in a museum, a user-centered design approach is implemented and evaluated (MEETeUX 2019).

1.2 Method

In order to develop an MDE with the focus on BYOD, a multi-day workshop was held at the beginning. In this workshop, we first developed a basic concept and defined what functionality the MDE should be able to offer. After that, we researched multiple possibilities for inter-device communication and methods for location awareness. The technologies that were finally selected are described in sections 3.3 and 3.4. Afterwards, we tested which possibilities are best suited for the development of our app. Furthermore, we started developing prototypes for all parts of the MDE to test if the concept - which is described in chapter 3 - works. During the prototype development, several workshops were held to improve the concept and to find a partner who would allow us to implement the MDE in a real exhibition. We found this partner in the museum management of the Klosterneuburg Abbey. We were asked to implement the MDE in the annual exhibition "The Emperor's New Saints". Until March 2019, the MDE was further developed for the exhibition. The exhibition was opened at the beginning of March with which also the evaluation period started. During the evaluation period, the technical functionality of the MDE was tested. Furthermore, all occurring bugs were recorded and then solved. Finally, the evaluation phase ended on July 31st. At this point, some final statistics of the exhibition were collected, which will be presented in this thesis.

2 Related Work

The last chapter offered a brief introduction of the different topics and research questions of this work. Also the MEETeUX project was introduced on which the technical concept (see chapter 3) is based. This chapter now deals with related work. The chapter is divided into two sections. The first section explains various terms such as multi-device ecologies (MDE) or bringing your own device (BYOD). Furthermore, the section presents various related publications in these fields and as one will see there are only a limited number of papers which deal with MDE and BYOD for museums. In addition, some technical concepts are presented that deal with one of the most important part of an MDE in museums, the location awareness. The second chapter presents various possible applications which are already implemented in museums. Again the section will show that there are hardly any interactive museum exhibitions which include MDE and BYOD.

2.1 Terminology & Technology

The term "ecology" refers to the relationship between an organism and its environment, which may include other organisms (Loke 2003). A device ecology refers to a collection of devices which have a relation among each other, meaning that these devices can communicate with each other and are aware of their mutual presence. Therefore, in a multi-device ecology (MDE), various types of devices have to interact with each other. Due to the rising interest in the field of smartphone integration in multi-device ecologies, recent research papers cover this topic. Bellucci et al. (2014) presented a survey, which provided a detailed overview of the state of the art in the form of a classification of the interaction possibilities between humans and the surrounding computational environment. One challenge that arises today, however, is that a user often has more than one device with them, such as a laptop, smartphone or wearable. Therefore, it can often be an advantage if a User Interface (UI) is split across multiple devices. In addition, there are always situations where it is desirable to work together in collaborative settings using multiple devices. One approach to solve problems which can occur in these setups, is the work of Park et al. (2018). They propose an approach that automatically adapts multi-user interfaces for collaborative environments in real-time, namely AdaM. AdaM uses a given graphical user interface and then automatically decides which UI elements should be displayed on each device in real-time.

The system also offers an optimization approach for multi-user scenarios and considers user roles and preferences, device access restrictions and device characteristics.

Furthermore, Sánchez-Adame et al. (2019) propose a set of design guidelines that serve as a means for the construction of multi-device applications. In their work they explain why Graphical User Interface (GUI) consistency is a very important part of multi-device experiences. ”Consistency not only provides users with a robust framework in similar contexts but is an essential learning element and a lever to ensure the GUI efficient usage” (Sánchez-Adame et al. 2019). Also Paternò (2019) deals with concepts and design spaces for multi-device user interfaces. In his paper he discusses the motivations and characteristics of multi-device user interfaces based on the main design issues addressed and the various proposed solutions. In addition, the work includes a comparative discussion of relevant systems and frameworks and their main features. Also Grubert et al. (2016) deal with challenges in mobile multi-device environments. In their work they focus on mobile and wearable devices such as smartphones, tablets, wearables and head-mounted displays. For this purpose, they conducted a literature research and an expert survey to identify technical, design, social, perceptual and physiological challenges of mobile multi-device ecosystems. Additionally, the work of Cecchinato et al. (2017) focuses on smartwatches and their role in MDEs. To that end, they conducted a qualitative study on users' everyday use of smartwatches. In their work they aim to understand the added value of smartwatches as well as the challenges when a user is constantly connected at the wrist. In their findings they propose various design recommendations to improve the user experience of smartwatches. One of the challenges facing mobile devices is testing mobile applications. Due to the large number of mobile devices and the variations in their characteristics, it is not guaranteed that an application will work as intended on all devices (Vilkomir 2018). Vilkomir (2018) chose to address this issue in his latest research. The goal of his research is to find out how many devices are needed and which methods for mobile device selection are best to find specific device deficiencies.

Multi-device applications are being tested in a wide variety of fields. However, there are few multi-device applications for museums or similar cultural offers. An existing system is "I-m-Cave" (Huang et al. 2014), an interactive tabletop system that makes it possible to explore the Mogao Caves in Dunhuang, China virtually. Another example is the multi-device, location-aware museum guide ”UbiCicero” (Ghiani et al. 2009). The guide exploits large screens when users are nearby and includes various types of games in addition to the museum and artwork descriptions. The framework Environs (Dang and André 2014) focuses on high-quality screencasts between the devices. Vepsäläinen et al. (2015) conducted a usability study in which they studied how users experienced four different methods of initiating web-based interaction between a smartphone and a large display surface. Furthermore, Dini et al. (2007) ”use a combination of PDAs and public displays to enhance the learning experience in a museum setting by using game-playing interactions”. Although

many museums have multi-touch tables, they rarely use them as part of a device ecology. They are used more for the presentation of existing information than to collaboratively work out new information.

Several approaches use the user's own device as an additional display in an existing multi-user tabletop setting. Mostly the device used is a smartphone. The concept that the user uses his/her device is called "bring your own device" (BYOD). For example, Shirazi et al. (2009) have linked a poker game on a touch table with the players' smartphones. Some articles deal with the control of touch tables and / or big wall displays with the users' own device (OD) (e.g. Echtler et al. (2009), Kray et al. (2010), Seyed, Burns, et al. (2012)). Terrenghi et al. (2009) have contributed important foundations in the form of an investigation of multi-person display ecosystems and a derived taxonomy. A forward-looking contribution uses a mini-projector docked to the smartphone to create custom multi-user multi-touch surfaces (Winkler et al. 2014). One of the few examples in which the connection of a large multi-touch wall with smartphones in the semi-public space (at a university) is permanently installed, is the system CubIT (Rittenbruch 2013). Boring and Baur (2013) do not use a multi-touch table, but instead present an overarching interaction concept that employs mobile devices to interact with large public displays. Almost all approaches have in common that they were built and evaluated only in the laboratory and very few applications can be used for museums or similar cultural events.

An example for related work which also focuses on BYOD in museums is the work of Koukoulis and Koukopoulos (2016). They propose a methodology for the design and evaluation of mobile systems for museums with the focus on enhancing the visit experience for end users. In a later work, Koukopoulos and Koukoulis (2018) have developed a prototype that does not only enhance the visit experience, but all the basic interactions among the users of a museum and the museum content e.g., preparing a new exhibition. To that end, they implemented specific services which are designed for the museum personnel which help them to perform common everyday actions and other services which are designed to improve visitors' experience. Some museums also offer visitors the opportunity to download audio guides or additional information on their smartphone (see section 2.2). Other museums offer their visitors a mobile guide which can be downloaded as an app. Othman et al. (2018) conducted an empirical study of visitors' experience at Kuching Orchid Garden. They designed, developed and evaluated a mobile guide application by comparing visitors'experience with and without the aid of the mobile guide application.
So far, this chapter only presented research dealing with the actual visit of a museum. However, as mentioned in the work of Falk et al. (2016), the museum visit begins long before arriving at the museum and continues long after the visit. For this reason Petrelli et al. (2017) have developed a system that creates a tangible data souvenir for a museum's visitors so that they stay engaged with the museum even after the visit. The approach of Petrelli et al. (2017) here is that the personalized, tangible data souvenir acts as a kind

of bridge between the physical, personal experience of the museum visit and the digital online experience of staying engaged with the museum. For this purpose, their system dynamically records the visit by logging information such as where the visitor is at particular points in time and what exhibits the visitor engages with. This data is then processed to create the personalized souvenir in form of a card on which the museum visit is represented.
In addition to the already mentioned approaches for museums, there are also some other approaches that can improve a visitor's experience. Kosmopoulos and Styliaras (2018) conducted "A Survey on Developing Personalized Content Services in Museums". In their work, they surveyed the latest advancements in the fields of indoor location, recommendation systems, guidance systems, content storage and presentation. All these areas are nowadays important for further personalizing and improving the visitor experience in a museum.

As the work of Kosmopoulos and Styliaras (2018) shows, a critical part of an MDE in museums is the location awareness. There are some works dealing with this area. For example, the SoD ToolKit (Seyed, Azazi, et al. 2015) allows users to be localized in a multi-device setup by using a Microsoft Kinect. Especially when integrating the visitors' OD, the device needs a possibility to know when a new exhibit is in range. An already tested approach to identify the position of users in museums is the work of Tesoriero et al. (2008). In their article, they propose a conceptual model for art museums that supports an automatic positioning system based on the technology RFID. Hardy et al. (2010) propose a way for users to directly interact with their phones and static (e.g., posters) or dynamic (e.g., projections) displays by combining these displays with NFC. Broll et al. (2011) also use the NFC technology to explore the design of interaction techniques for dynamic NFC-displays that go beyond the common touch select interaction.
Urano et al. (2017) developed an indoor location estimation method using mobile Bluetooth Low Energy (BLE) tags, which are worn by people and BLE scanners, which are attached to a building. Lim et al. (2007) use smart WIFI antennas to localize mobile targets. To that end, the antennas receive the signal strength from a mobile target and send the signal strength information to a data processing station. Nishiyama et al. (2017) developed an advanced floor recognition method by using WIFI access points and the barometer hardware in smartphones. To provide a more reliable and robust detection method Wang et al. (2011) developed an indoor localization algorithm with WIFI and Bluetooth. Finally, Lazik et al. (2015) and Murata et al. (2014) use ultrasonic sound to provide an accurate indoor positioning system.

2.2 Possible Applications

This section presents various interactive exhibitions which were or are an integrated part of museums. The focus here is not only on exhibitions that implement an MDE or apply the BYOD concept. All museums were considered that implemented some form of an interactive exhibition. Some museums use the BYOD concept to e.g., offer the user a guided tour in form of an downloadable app. Others use Augmented Reality (AR) in their app to show the visitor a storyteller. Still other museums have no app at all but implemented a form of an interactive exhibit. In the following the interactive exhibitions of the museums and which technologies and concepts they used are described.

The first example for an interactive exhibition is part of the Museum Albrechtsburg Meissen in Germany (Blumenstein, Breban, et al. 2019). For this museum, the developers called Fluxguide created an iOS and Android app for usage on the user's own devices. So the exhibition emplifys the BYOD concept. The app provides a guided tour for deaf visitors using videos in German sign language. The visitors can mark their favorite exhibits and an description will be then sent to their email account. The localization works through manual entry of a number, and the app can also be used if you are not in the exhibition.

Another museum using the BYOD concept is located in the United Kingdom and is called Bletchley Park (Blumenstein, Breban, et al. 2019). The museum offered a multimedia guide for the family exhibition "Bletchley Park". The exhibition included an adult tour and a family tour. The adult tour included an audio guide with Historian Jonathan Foyle. The family tour included code-cracking challenges, puzzles, and storyteller parts. In addition, the exhibition offered numerous video contributions from people who worked at Bletchley Park and saved many lives during wartime. The multimedia guide is part of an iOS app which the users can install on their own device. Furthermore, the museum offers multiple rental devices.

The London's Natural History Museum app is another example of an interactive museum tour using the BYOD approach (Museum Tour Guides 2019). In addition to room plan information, it offers information about all exhibits, in the form of images and text. The app also provides information about current museum events and includes interactive quizzes. Furthermore, the app is fully functional even outside of the museum.

The Canadian Museum of Human Rights does not offer its visitors interactive exhibits but an app with self-guided tours in English and French (Blumenstein, Breban, et al. 2019). The contents are presented as text or as an audio guide. It also includes a translation into sign language. Bluetooth beacons trigger the localization. The user can also enter the exhibit number manually. In addition, the app offers an interactive map that shows the location of the visitor and includes a navigation system (text-based). The exhibition includes so-called "Hot Spots" which are blue points on the floor and trigger an AR panorama view.

The Celtic Museum Hallein in Austria also offers an app with the BYOD concept (Hallein 2019). Visitors can use the app AR markers, which are distributed in the museum (see Figure 2.1). If you hold your smartphone over the mark, the display shows a celt that tells different stories of his life.

Figure 2.1. Picture of the AR marker in the Celtic Museum Hallein (Hallein 2019).

Furthermore, the app "Wiener Wasserweg" provides a virtual tour guide for the Old Danube (Wiener Gewässer 2018). The tour is a total of 13 km long and offers 22 stations. The visitor can either complete the tour on foot or by bike and each station offers information on the history, wildlife and use of the Old Danube. It also provides the user with some information on beach baths, lawns, restaurants and boat rentals at their current location. In addition, the app offers a quiz mode for playful learning.

The Gotlands Museum app offers its visitors a guide which displays a map of the museum (gotlandsmuseum.se 2019). On the map, the user will find various locations for which images, text, audio, 360 content or a quiz is available. The app also offers predefined tours of the museum from which the user can choose.

The Maritime Museum in Rotterdam also offers its visitors a tour guide in the form of an app with the BYOD principle (Maritime Museum 2019). The app provides visitors with two tours which include additional information in the form of films, animations and digital souvenirs. One tour is a tour of the museum harbor. Here, the app displays a map and is uses GPS to track the current position of the visitor. The visitor receives information about an exhibit

by scanning the respective QR code in the app. The second tour is an audio tour in which the museum staff tells the visitor something about the individual exhibits.

The Kennedy Space Center also has an official app for their museum (Kennedy Space Center 2019). The app does not offer a tour guide directly, but acts as a kind of navigation aid. The visitor finds all offered exhibits in the app and can receive directions to them. In a map view, they see where they are currently located and where the exhibit is. Here, the app uses GPS for position finding. In addition, the app provides information about events, shows and opening hours of the exhibits. Furthermore, it offers a way to save the exhibits that one likes the most in a favorites list. Consequently, visitors can read the information about the exhibits at home.

The next app is not directly linked to a museum but to the city of Wels (wels.at 2019). The topic of the Roman period of the city was playfully adapted with the app "Heroes of the Roman Age". Emperor Hadrian arrives in Ovilava to put together a delegation of heroes to help Rome out of the crisis, as he makes his way through the city finding his allies at nine different locations. The heroes of Ovilava, in addition to historic short explanations, offer animated augmented reality stagings, such as the bronze equestrian statue of a Roman emperor on the occasion of the departure for the Traun. In order for the visitor to find the individual locations, the app contains a map that shows the locations of the allies. In addition, the app includes various quizzes for each station with which the visitor can earn points.

The "Deutsches Technikmuseum" in Berlin also has an app that can be installed on the visitors' own devices (Deutsches Technikmuseum 2019). The app offers three different tours that can be experienced as a quiz tour or as an audio tour. The quiz tour includes quirky quiz questions and selfie tasks, and in the end, visitors are even rewarded with a virtual certificate. Each exhibit has its own number which the visitor must enter in the app. For every correctly answered question the visitor receives points. In addition, the app is available in three different languages and even offers a school class mode. Following the quiz tour, the teacher will receive a summary of the results of the class in the form of a link via e-mail.

The "Deutsches Museum" in Munich also has an app with various functions (Deutsches Museum 2019). On the one hand, the app offers an interactive map for all levels of the museum. On the other hand, the app offers various tours that visitors can do in the museum. Each exhibit contains text, images and audio. In addition, the visitor can add the exhibits to a favorites list and these list can be used for a personalized tour through the museum. Furthermore, the app provides information about current events and performances in the museum and if the visitor wants, the app even sends a push message as a reminder.

The BMW Museum also offers similar functions in its app (bmw-welt.com 2019). With

the help of the BMW Museum app, visitors can experience selected aspects, themes and epochs of the brand's history in all its details. In addition, they can be guided through the individual houses in a personalized order. The respective explanations are presented on the visitor's own smartphone as texts or audio. Alternatively, content can also be livened up by image motifs of the exhibit. An interactive map display helps visitors better navigate their way through the departments.

Until now we saw several museums offering their visitors an app with the BYOD approach (Blumenstein, Breban, et al. 2019). Now we focus on the integration of interactive exhibits in museums. The first example can be found at the Fort Worth Museum of Science and History in Texas. The exhibit includes a video wall on which dinosaurs run animatedly. Children can paint the dinosaurs on a sheet of paper, then scan the paper which will then present the dinosaur on the video wall in paper style. The Dinos can then be scanned to display on the Video Wall.

The National Park exhibition of Hunsrück-Hochwald offers its visitors a multimedia exhibition of the national park (Hunsrück-Hochwald 2019). The exhibition includes four core themes which are presented in an unusual and effective way. These themes reflect the National Park with its diverse tasks, its networked forests, its unique moors and the different landscapes and habitats. Projections and experimental stations provide insights and views into nature and its inhabitants. Unfortunately, this interactive exhibit does not support BYOD and, moreover, the devices are not interconnected.

A museum that includes several interactive exhibits is the House of Science in Vienna (Blumenstein, Breban, et al. 2019). One exhibit is called the Mars Explorer, where two Lego-built Arduino cars can be controlled with a tablet. The cars drive on a stretch of cardboard and have the goal to grab a ball and bring it to the other end of the track. Another example is the info table. There are two areas on a table which are marked with a frame. Around the table are several animals. If you place an animal in one of these frames, the living space of the animal appears in a virtual space on three walls in the room via a projector. By turning the animals, also the virtual space turns. Furthermore, different videos are displayed on the walls. If the user moves an animal closer to the wall, the corresponding video will start to play. The last example from this museum is a quiz. The quiz question is displayed on a screen which is in front of the visitor. On the floor in front of the visitor are three plates with numbers. The answer options can be selected by the visitor stepping on one of these plates. The faster they are, the more questions they can answer in a given time.

The art-historical museum in Vienna also offers an iOS and Android app that can be installed on the visitor's OD (Blumenstein, Breban, et al. 2019). The app offers visitors six different tours of the museum. Pictures and written instructions indicate the path of the tour. The app also includes a museum map, but without a current location. During the tour, the visitor is guided to a specific object in the room and receives information about

it. Figure 2.2 shows how such a tour is displayed in the app. The app is also usable if the visitor is not in the exhibition.

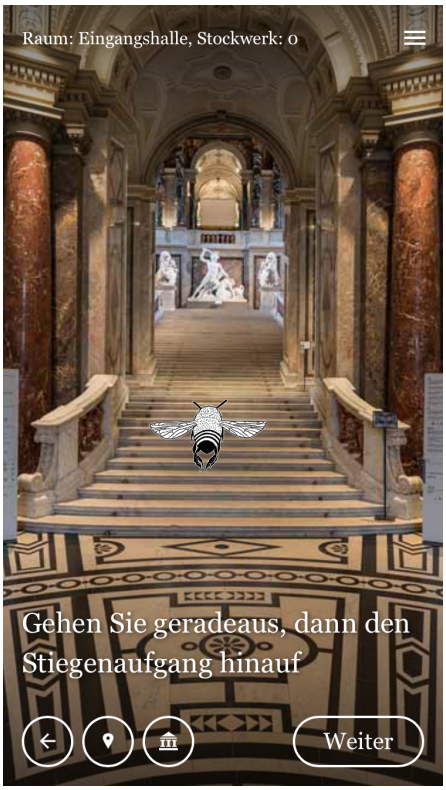


Figure 2.2. Tour of the art-historical museum in Vienna (Blumenstein, Breban, et al. 2019).

At the museum of Literature in Vienna, users can borrow a loaner to explore the exhibition (Blumenstein, Breban, et al. 2019). The exhibition offers a multimedia guide that includes video, audio, and text information. The exhibits are triggered by the Near Field Communication (NFC) technology.

An example of an active exhibit using a multi-touch table can be found at the Hamline University's Center for Global Environmental Education, St. Paul Minnesota (Blumenstein, Breban, et al. 2019). This is a multi-touch table application designed for 1-4 users. The desktop adapts to the number of users. Each user sees a geographical map of the Mississippi and can navigate on it (see Figuer 2.3). The application includes predefined categories. If the user selects one, then the points of interest for that category are displayed on the map. The dots offer multimedia content such as pictures, 360° panorama pictures, or videos. Sometimes there are also quizzes for different topics.

Finally, the Space Guard Academy of the Lowell Observatory in Flagstaff Arizona will be presented (Blumenstein, Breban, et al. 2019). This is a gamified exhibition. In the exhibition, visitors are cadets of a fictional Space Academy. The goal is that visitors learn as much as possible about asteroids. The exhibition offers six different stations and at the end

Figure 2.3. The multi-touch application of the Mississippi river (Ideum 2019).

a hall of fame. At each station, visitors can collect points and badges to increase their rank. To identify the users, the exhibition uses ID cards. This is a card on which a barcode is printed. In the beginning, the visitor can create a new character and then receives his/her ID card. Each station contains a reader with which the ID card must first be read before the station can be used. The interactive stations always use a combination of screens and touchscreens. An example of a station is the asteroid race. Here, visitors must select one of five asteroids, all of which have a different orbit. Players should choose the asteroid which will hit Jupiter the fastest (selection on a tablet / small touchscreen). Every five minutes a race starts and the visitor sees if he/she has interpreted the orbital data correctly (race on a 75 ” screen). At the end of the exhibition the visitors can see the hall of fame where visitors can see their profile and take a picture of themselves.

These museums were just some examples which were found during the research of the MEETeUX project and this thesis. The website of the MEETeUX project offers a visualization which gives an overview of the researched interactive installations in museums[1]. As one can see, there are many museums which offer their visitors an app to guide them through the exhibition. The app offers additional content, and the museums use different approaches to trigger the different exhibits. In addition, there are various types of active exhibits that work with a variety of technologies. Whether it is a projection that can be interacted with using a Kinect camera or a multi-touch table that can be used to implement a variety of applications. However, no museum really combines the use of an app installed on the visitor’s OD with the active exhibits of the exhibition.

[1] http://meeteux.fhstp.ac.at/assets/matrix/

3 Technical Concepts & Possibilities

The previous chapters presented different interactive exhibitions which are already implemented in museums. Additionally, the terms multi-device ecology and BYOD was defined. As the previous chapters show there a lot of different applications which can be included in a multi-device ecology. Therefore, it depends on the needs of the museums which possibilities should be included in their multi-device ecology.

This section now focuses on a basic concept of a multi-device ecology (MDE), which was presented in the paper of Blumenstein, Kaltenbrunner, et al. (2017). We tried to develop a concept which offers a basic set of possibilities but which can easily be extended. Therefore, this section will also present different possibilities which such a system could include and also technical concepts which can be used for the different parts of the multi-device ecology.

3.1 Basic Concept

By design, a basic concept for a multi-device ecology should include the possibility for different devices to communicate with each other. In addition, such a concept should include an instance that keeps track of the status of all exhibits and visitors. Blumenstein, Kaltenbrunner, et al. (2017) called this instance Guide of Devices (GoD). The devices include the visitors' own devices and different groups of exhibits. For visitors who want to enter the multi-device ecology with their own device they need to install an app or open it in the browser at the entrance of the museum.

The exhibits in a museum can be categorized into three different groups, namely passive exhibits, interactive exhibits, and active exhibits. Passive exhibits only include some additional information for the visitor, e.g., text and images for a display item in the museum. For this kind of exhibit, no further computers are needed. The visitor's own device (OD) displays further information when the OD receives a trigger from a display item. What these triggers could be will be described further in section 3.4.
The next group is the interactive exhibits. This group includes exhibits where the visitor can interact in some kind with the display items in the museum without the need for an additional computer. An example could be an augmented reality overlay of an old text.

When the visitor holds their smartphone over the exhibit, an overlay appears that displays the text, which could be challenging to read or perhaps written in another language, in a readable computer font.
The last group is active exhibits. These exhibits require an additional computer and offer the visitors the possibility to do something actively. An example could be a quiz. The visitor comes to a display, and the OD receives a trigger for the active exhibit. The visitor now connects to the quiz server and can then participate in the quiz by answering the questions, which are displayed on the screen, on their own ODs. Another example could be a multi-touch table where the visitor could explore a digital copy of a display item, or he/she could participate in a collaborative game. In this scenario, the visitor puts his/her phone on a designated position on the touch table. The OD again receives a trigger from the touch table an sends a message to GoD. GoD then checks if the multi-touch table is free and if so it sends a message to the multi-touch table to notify the table that there is a new visitor joining.

As one can see, there are a lot of different possibilities for the different groups of exhibits, but mainly for the active exhibits it is crucial that there is an instance which tracks the status of these exhibits. If an active exhibit has a problem or is for whatever reason crashing, GoD should know about it. Only then can GoD send a message to the visitors' devices to notify them that the exhibit is currently unavailable. Furthermore, GoD can tell visitors if the active exhibit is currently free or if it is occupied.

Apart from the exhibits, GoD also offers a lot of possibilities to track and manage the status of the visitors. With GoD, we could track the visitor's current position, which would give developers further location-based possibilities. The system could then, for example, include a navigation system with which the visitor could find more exciting exhibits or specific places he/she needs, e.g., the toilet. Furthermore, the MDE could offer various visitor tours for the museum. If a visitor is interested in a tour, he/she starts the tour on their OD and the OD, with the help of GoD, navigates the visitor through the different rooms of the museum. This could also include some crowd control. If GoD has the possibility to track the visitors, it could notify a visitor if there are many people in the next room. Then the OD could offer an alternative route. Of course, in this scenario, GoD could only track the visitors who are using the museum's app.

Another option would be tracking group members. Museums are often visited by groups, e.g., families, tourists, or school classes. If all the members of such a group would had a smartphone, we could also include these groups into the multi-device ecology. Blumenstein, Kaltenbrunner, et al. (2017) proposed a method to use the museum's own group tickets to form a group. The group ticket could include a QR-Code which the group members have to scan with their ODs and GoD adds everyone to a group for the MDE. Of course, there are also other possibilities to form groups, but the important thing is that

these groups offer new types of interactions for the visitors. For example, the MDE could include exercises for pupils which the teacher as the group leader could manage. If the teacher starts a new exercise, the pupils have to find different parts of the exercises in the museum and are therefore roam around. To keep an overview of the pupils, the MDE could provide the teacher with the possibility to track the visitors while they are doing their exercise. So the teacher always knows where his/her pupils are, or GoD could even send the teacher an alert if one of the pupils leaves the museum.
Of course, this concept is not limited to schools. Also families could just split apart to see different parts of the museum, and in the meanwhile, they can keep track of the other family members' location.

The concept of the groups would also offer the possibility for different collaborative games or exercises. An example could be a game where the members of a group start a game at a multi-touch table and then have to gather different parts to accomplish the game. When all parts are gathered, they meet again at the multi-touch table and put together the different parts they found. As a reward, they could then unlock a particular area in the app or maybe get a special badge.

Another way how GoD could support the visitors is a recommendation system. A lot of online shops suggest their users additional items which they may also like. Usually, these websites call it 'Other users also liked' or something similar. Just like these online shops, the app could offer the possibility to like the various display items in the museum and then GoD could send the visitor suggestions for exhibits he/she could also like.

3.2 Key Elements

This chapter will present technical possibilities for an MDE with the focus on BYOD. For this purpose, the basic concept of Blumenstein, Kaltenbrunner, et al. (2017) was presented which includes various concepts for an MDE. This section now presents the key elements for the MDE of the MEETeUX project. After introducing the key elements, this chapter will explore the different ways to implement some key elements.

Server The server should know the status of all exhibits and the visitors' OD. Furthermore, it decides if a user can interact with an exhibit or not. Respectively, the server handles the communication between the exhibit and the OD. It also stores all relevant data for the exhibition e.g., the contents of the exhibits.

App The app should be avaible for Android and iOS. The app should provide all functionalities which are necessary for the exhibition. Therefore, the app should provide a general user interface, the functionality to interact with the various exhibits, a way to

register as a user, access to the device's hardware and additional features like the option to show Augmented Reality content.

Guest Support The MDE should provide the user with an option to register as a guest user. Guest users could get a limited experience of the exhibition. Therefore, a guest user should always have an option to register as a real user afterwards.

Exhibits The exhibition should include different types of exhibits. All exhibits will provide some kind of functionality in the app. Some only offer additional information and others provide the visitor some kind of interaction. The visitor is uses his/her OD to interact with the exhibits. For this purpose, the OD uses the protocol TUIO. "TUIO is an open framework that defines a common protocol and API for tangible multitouch interfaces. The TUIO protocol allows the transmission of an abstract description of interactive surfaces, including touch events and material object states" (Kaltenbrunner 2019).

Device Communication The app needs the possibility to communicate with the server and the server with the exhibits. Therefore, this chapter will introduce different technologies for device communication. Here, the focus is on technologies that also allow the server to send a message to the visitor if necessary. So the chapter focuses on bidirectional communication approaches.

Location Awareness The app needs a possibility to know when there is a new exhibit in range. Therefore, this chapter presents various active and passive location triggering approaches which are a feasible location awareness approach. Furthermore, each time the app discovers a new exhibit it notifies the server of its new position. The app should also provide an option to always show the visitor his/her nearest location.

Navigation If the server is aware of the visitors current location the app can provide a navigation system. This could be only a map of the museum where the user sees his/her current location or a full navigation system where the visitor is guided to the designated location.

Groups The MDE should support at least one possibility to generate groups of users. The groups should provide the visitors additional possibilities for collaborative interactions or localization of the other members.

Recommendation System The app provides a possibility for the user to like exhibits in which he/she is interested. The MDE uses this information to suggest further exhibits of interest to the user.

Multi-Language Support The MDE should be able to provide all contents in multiple languages. This includes the app as well as all exhibits.

3.3 Device Communication

As one can see a multi-device ecology could include a lot of different features, but one essential part is the communication between the devices. The OD must be able to communicate with GoD, but at the same time, it also has to communicate with the active exhibits. However, it is not enough that only the OD can communicate with the other devices. GoD also needs a way to tell the users if the status of an exhibit changed or when it comes to group exercises GoD needs a way to send messages to a group.
In addition, the active exhibits need a way to send messages to GoD but also to communicate directly with the users. That is because the exhibition could have an active exhibit where the visitor has to do something on his/her smartphone, which has an effect on e.g., a multi-touch table. An example could be a drawing game where the users have to draw something on their smartphone, which should immediately show on the table. In this example, it would produce an additional delay if the drawing data is send to GoD and GoD then transmits the data to the table. This could be crucial for real-time games or exercises.

In typical web applications, the client sends a message to a server (e.g., REST API), and the server responds with resources or data. This means that in the traditional client-server model, the server has no way to send data to the user without a request of the client. This approach wastes time and ensures longer load times, which is excellent for standard websites and various apps, but in our approach, it could be fatal.
Therefore, there are some approaches to overcome the overhead of the traditional procedure. The first one is the 'traditional' polling approach. This is more like an easy workaround where the client sends requests in specific time intervals, and the server sends a response for each request (see Figure 3.1). However, based on the interval, it again takes some time until the client gets a response and furthermore, multiple requests and responses produce much overhead.

A more advanced approach is the long polling concept. When using long-polling, the browser continually sends requests to the server, but the server only responds if there is new data to be sent. This approach is visualized in Figure 3.2. This approach is already better in terms of all requirements, but still causes some overhead.

The third approach is called Websockets. Websockets use their own protocol to provide bidirectional communication between client and server. The client sends a request to the server to upgrade a standard HTTP(S) connection to a WebSocket connection. If the server supports Websockets, it replys with a success message, and then the connection is upgraded to a WebSocket connection (see Figure 3.3). After that, the client and the browser can send data frames in both directions over a TCP socket. This protocol is especially useful for scenarios where messages need to be exchanged at a high frequency. The downside of this approach is that not all browsers already support Websockets.

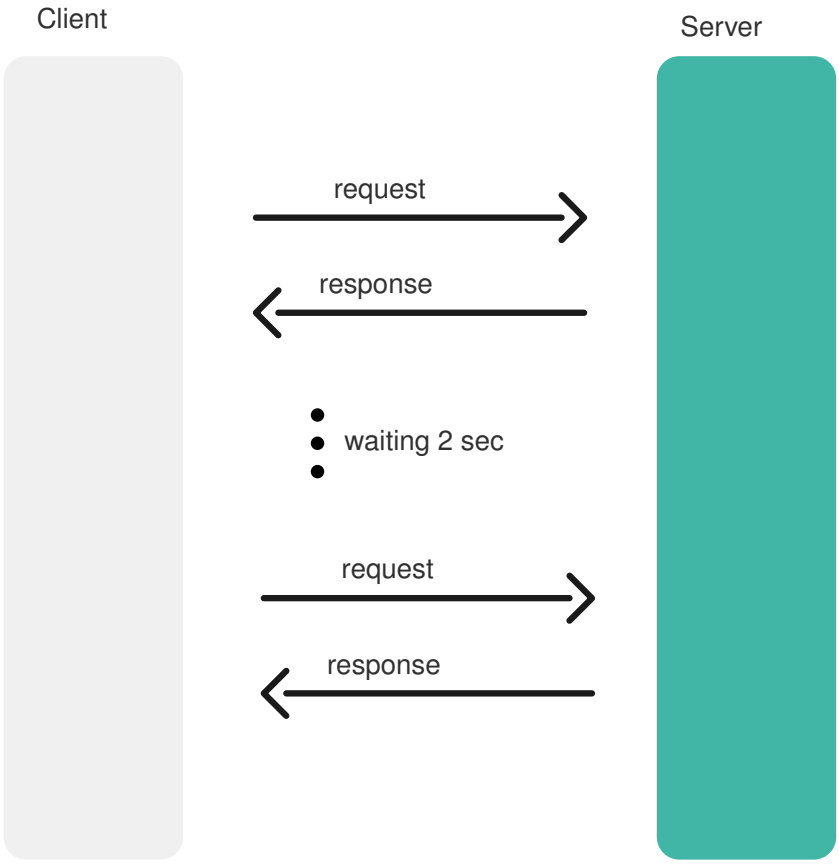


Figure 3.1. Schematic representation of how 'traditional' polling works.

3.4 Location Awareness

As mentioned before, the app needs a location awareness method. The app is used to display the content for the exhibits. Therefore, it needs a method to know when a new exhibit is nearby. This section introduces different location awareness technologies found in the literature review and analysis of the various museums. The technologies were split into two groups. The first group deals with active methods of position determination. This means that the user has to trigger the location by himself/herself actively. The second group deals with passive technologies. For passive technologies, the user does not have to take action. Instead, the app uses a technology in which it independently and automatically determines the position in the exhibition.

3.4.1 Active Location Triggering

The first approach and also a straightforward one, is the manual input of a code. For this approach, each exhibit has a unique code which can be e.g., simply a number like 125. The user finds the number on the exhibits and enters it in the app. The app then displays

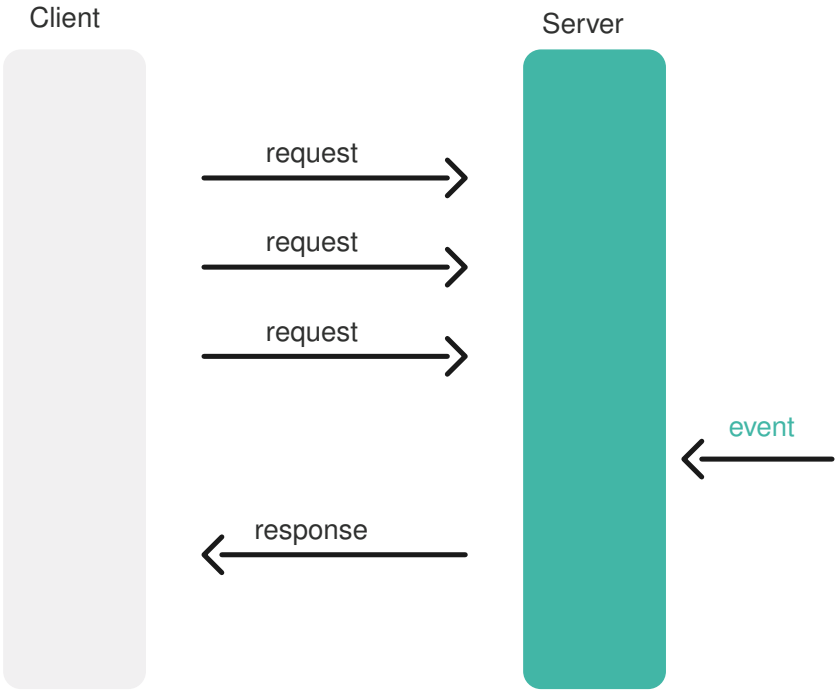


Figure 3.2. Schematic representation of how long polling works.

the content of the exhibit. Some museums use this approach as a backup if the primary technology is not completely reliable or inaccurate. A museum which uses this approach as a backup is the Canadian Museum of Human Rights (see figure 3.4).

The next active method is the use of optical markers. Optical markers can be anything, such as a barcode or a QR code. Another possibility is the use of augmented reality markers such as those used in the Celtic Museum Hallein. For their exhibition, they developed their markers to make a Celt appear above of the marker and tell a story (see Figure 2.1). Optical markers offer the possibility to store various data in the marker, which are then interpreted by the app. However, in any case, the user must always actively scan these markers. Therefore, the app needs access to the camera of the OD.

Finally, there is the Radio Frequency IDentification (RFID) protocol, which is an automatic identification method. It is a contactless communication technology that provides information for identifying objects. An RFID system consists on the one hand of a data carrier (called transponder or tag) and on the other side of a read/write device with an antenna. RFID works with weak electromagnetic waves that are emitted by a reader. If you bring a transponder into the range of this antenna, you can read information without contact from the memory of the transponder or store data on it (tagnology.com 2019).
One implementation of this protocol is the Near field communication (NFC) standard. NFC is an international standard for the wireless exchange of data over short distances (10-20 cm). Nowadays, many smartphones are equipped with the technology ex-factory, others can be retrofitted with an NFC sticker, which is attached to the back of the device and re-

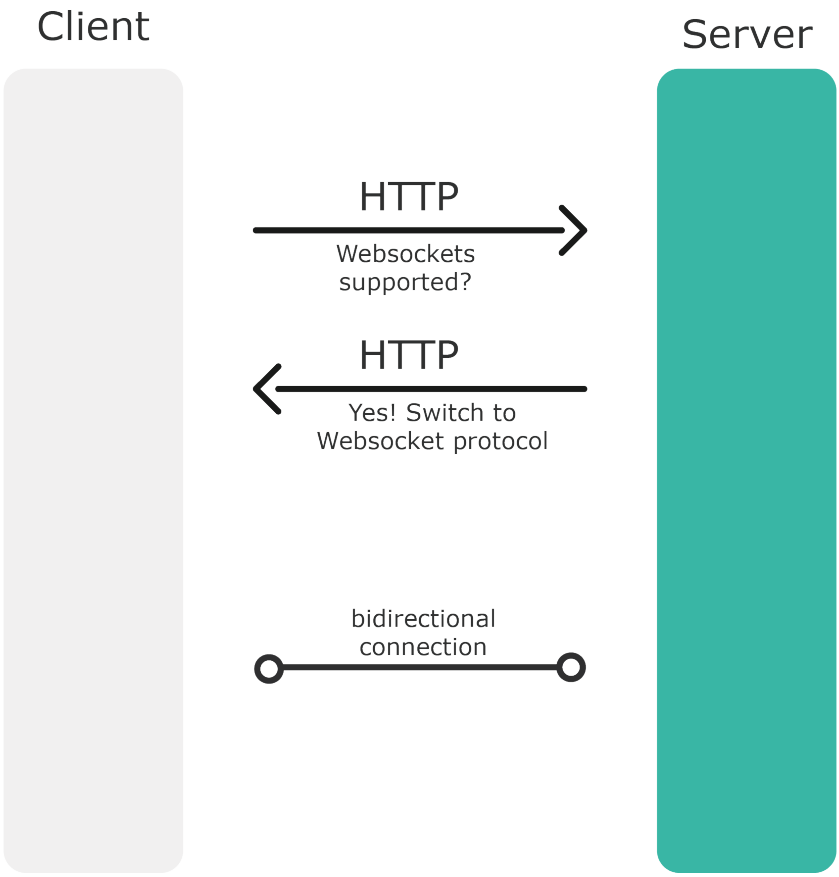


Figure 3.3. Schematic representation of how Websockets work.

places the built-in NFC chip (congstar.de 2019). It should also be mentioned that iOS apps can only access the NFC chip from version 12 onwards. For data transfer via NFC, the smartphone or tablet is held to a preprogrammed NFC tag that triggers a specific action on the OD. Even if NFC works contactless, it is natural for most users to put the smartphone directly on an NFC field. Therefore, and because of the short-range, this technology has also been classified in the active methods.

3.4.2 Passive Location Triggering

The first passive location triggering approach is Bluetooth. Most of today's smartphones have a Bluetooth chip with which they can receive data from Bluetooth transmitters (beacons). A beacon is a small Bluetooth transmitter which repeatedly transmits a single signal which the OD can receive if it is in range. The signal of the beacon is a combination of letters and numbers which is transmitted in a regular interval of approximately 1/10th of a second (Kontakt.io 2019). A manufacturer of such beacons is the company Kontakt.io, which also offers an SDK for iOS and Android to receive the beacon signals in a native app. The beacon also has the ability to wake up apps even if the app is not actively running. To

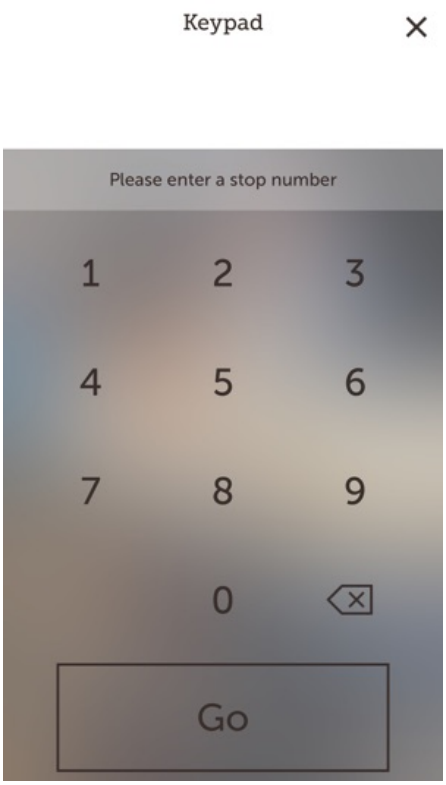


Figure 3.4. Manual code input in the app of the Canadian Museum of Human Rights (Tristan Interactive 2019).

sum up, the beacons are simply broadcasting an ID number that is then used in the OD's app to trigger the exhibits. Beacon scanning requires Bluetooth and the location services to be enabled on the smartphone.

Another but also more complex approach is the triangulation of a device with WIFI hotspots. As already mentioned in chapter 2 Lim et al. (2007) developed a WiFi localization system for an indoor environment like a museum. They use smart antennas to receive signal strength from a mobile target. Afterward,d the signal strength information is sent to a data processing station which combines the received data to find the direction of arrival of the signal and triangulate the mobile target position (Lim et al. 2007). Therefore, in this approach, it is not the OD which is aware of its location. Instead, it is necessary to forward the information to the server, which then forwards the information to the ODs.

The final approach is the usage of ultrasonic sound. As mentioned before Murata et al. (2014) and Lazik et al. (2015) use this approach for an indoor positioning system, but currently, there is no basic setup which could be easily integrated into an exhibition. Therefore, it would be necessary to develop ultrasound transmitters and also to program a library that can receive and interpret these signals. In addition, not all smartphone microphones can receive ultrasound signals. Therefore, this approach is not very practical for older devices.

3.5 OD/App

When one is talking about app development, there are three different approaches, namely web development, hybrid development, and native development. As the name suggests, web app development deals with apps which are developed for the web and are loaded in a web browser. There are a lot of different frameworks like Angular.js, Vue.js, React, and many more, but all have in common that they have only minimal access to the hardware of the phone. This is because they are loaded in a web browser and therefore always have limited access to the PC's or phone's hardware.
In the last section, various location tracking approaches were presented, and as we could see, all of them had to have access to a part of the hardware. While access to the camera is now also possible through the browser (with the user's consent), access to other hardware such as Bluetooth or the microphone is still not possible. Therefore, web applications are not the best choice for the development of an MDE.

Unlike web applications, hybrid apps are able to access the hardware of smartphones. Although hybrid apps are often developed using web technologies, they are then compiled into a native app and therefore have access to the hardware. So, hybrid apps have the advantage that the app has to be developed just once and can then be compiled to both iOS and Android. Therefore, hybrid apps would be quite suitable for use in an MDE.

Native apps refer to applications on mobile devices that have been specially designed and developed for the operating system of the respective device. So they are developed to run on a device with an operating system like iOS or Android. Currently, iOS applications are usually developed with the programming language Swift and Android applications with the language Java or Kotlin. Since these apps are developed in the native programming language of the operating system, the apps have access to the hardware of smartphones. The significant disadvantage of the development of a native app is that the app must be developed for iOS and Android separately. It should also be considered that iOS apps can only be compiled on a Mac computer.

In addition to the access to the hardware, one should always keep in mind what the app should be able to do. Earlier it was already mentioned that there are interactive exhibits that include, e.g., augmented reality. Additionally, the app might also be required to include a small game? In both cases, the use of a game development environment such as Unity could be useful. Blumenstein, Kaltenbrunner, et al. (2017) proposed a setup where they developed a mobile project in Unity, which can then be compiled to a native iOS or Android app. It is then possible to extend the exported native app with additional content. Unfortunately, Unity exports two native apps, which would mean that we would have to develop additional content for both iOS and Android. To avoid this Blumenstein, Kaltenbrunner, et al. (2017) used a native web view in which e.g., a website could be shown. However, the

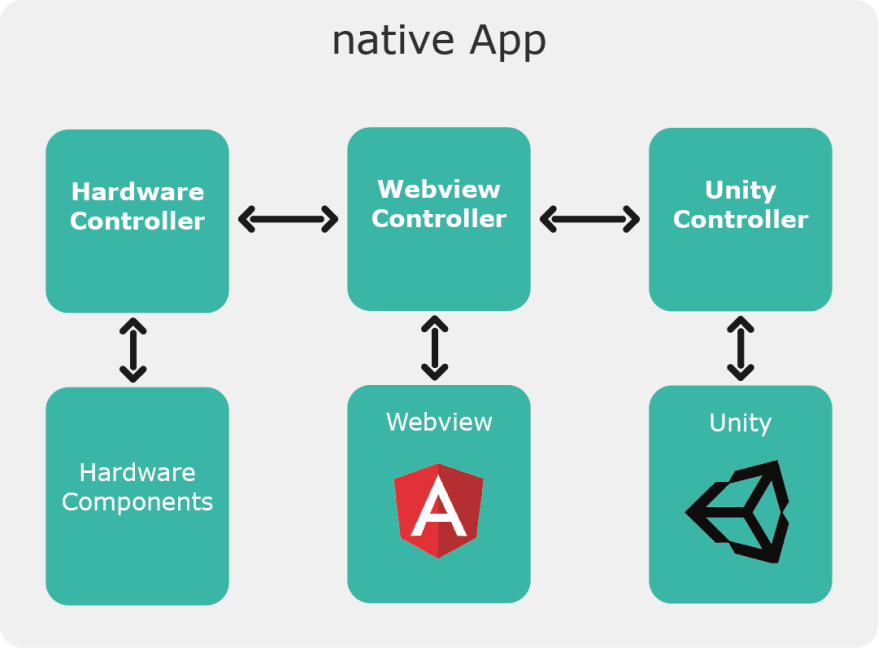


Figure 3.5. Proposed app structure of Blumenstein, Kaltenbrunner, et al. (2017).

webview can also be used to display a web application. So Blumenstein, Kaltenbrunner, et al. (2017) developed all parts, which do not include the access of hardware or something which has to be developed in the Unity, with the web framework Angular. So the bulk of the app had to be developed only once. The Angular project was compiled and then added to the native Unity project. The Unity app was extended so that the Unity part is not started immediately. Instead, the Angular project is the starting point, which is displayed in the web view. The added part of the native app now functions as an interface between the Anulgar project and the Unity part as figure 3.5 shows.
If the angular project needs to access the hardware of the smartphone, it sends a message to the native interface. The native part accesses the hardware and then sends the needed data back to the webview. Furthermore, if the user reaches an interactive exhibit, the Angular project sends a message to the native part, and the native part then loads the corresponding unity scene were the game or augmented reality part is displayed.

In this concept, the build process is more complicated, as usual. The reason is that the Unity project always deletes the whole app folder while exporting. Therefore, the first step is always the export of the Unity app. Then the app has to be modified with the interface, the webview, and the hardware features. Then the Angular project has to be compiled and added to the native app. This process has to be done each time there is a change in the unity project. If there is only a change in the Angular project, then only this part needs to be recompiled and re-added to the native app.

4 Proof of Concept

The previous chapters presented similar work dealing with the field of MDE and BYOD. This work also discussed multiple different interactive exhibitions which are already implemented in a museum. Furthermore, the last chapter showed a lot of different possibilities and a technical concept for a multi-device ecology. This chapter now presents the actual implementation of an MDE. As a follow-up project of the MEETeUX project, we developed an interactive exhibition including an MDE for the monastery of Klosterneuburg[1]. The interactive exhibition is part of the annual exhibition of the year 2019 named 'The Emperor's New Saint'. The exhibition enables visitors to experience history, ranging from local conflicts to political decisions concerning the emperor's realm. The app allows visitors to take on the role of a person living in emperor Maximilian's time. Three selectable storytellers, namely Emperor Maximilian, Ladislaus Sunthaym and Till Eulenspiegel give insights into different aspects of the same story.

4.1 Key Elements

In the previous chapter the key elements of the MEETeUX project were described. For the exhibition of Klosterneuburg only a subset of these elements were chosen and implemented. Part of the research focus of this thesis is the question of how to implement these elements and integrate them into an MDE. Therefore, the following listing shows for each key element the corresponding research question.

Server How can one implement a server which always knows the status of the exhibition including all exhibits and the visitors ODs? More specifically, how can the server handle the communication between all devices and furthermore, decide if a user can currently interact with an exhibit or not?

App How does an app have to be designed so that it offers the possibility to access the hardware of the device, offers a location awareness approach, provides a general user interface and includes the functionality to interact with the various exhibits?

Guest support How can the MDE provide an option for visitors to use the app as a guest

[1] https://www.stift-klosterneuburg.at

user and still provide the visitor with the same functionality as registered user?

Exhibits How can the MDE provide different types of exhibits and also include a possibility for the communication between the exhibit and the server or the app? Therefore, the communication protocol will not be limited to the TUIO protocol.

Device Communication Is the WebSocket technology a sufficient approach for bidirectional communication between all devices in a MDE?

Location Awareness Is the usage of Bluetooth and Bluetooth Low Energy beacons a sufficient approach for the location awareness?

Multi-Language Support How must the MDE be designed to give visitors the opportunity to experience the exhibition in different languages?

The following sections will show how these key elements were implemented in the proof of concept. First, the concept of the MDE for Klosterneuburg is described and then each part of the MDE is described in detail. Chapter 5 contains the evaluation of the MDE and shows potential problems found in the key elements.

4.2 Concept of the Multi-Device Ecology

The multi-device ecology consists of three parts. The first part is the server which we call Guide of Devices (GoD). The second part is the own device (OD) of the users, and the last part is the different exhibits. Figure 4.1 shows the schematic structure of the MDE, which was implemented at the monastery of Klosterneuburg. As mentioned in the previous chapter, the server is the instance which knows about the status of all users and exhibits. GoD stores all the necessary data in a SQL database and also manages the content which will be loaded in the app. As the figure shows, GoD is connected with all the visitors' ODs and exhibits via a WebSocket connection. Also, the ODs are connected via a WebSocket connection if they join an active exhibit. This allows bidirectional communication between all active elements in the multi-device ecology.

GoD is connected to the Internet, and therefore, it is possible to use the app even at home. The only exception is active exhibits. The active exhibits have their own Websocket Server running so that the ODs can directly connect with them. Though, the active exhibits are only accessible within the internal network. Therefore, external access to active exhibits is not possible. It follows, of course, that the ODs must be connected to a WLAN which is connected to the same local network as one of the active exhibits.

To trigger the different exhibits, we used Bluetooth low energy beacons (BLE beacons). These beacons broadcast their identifier to all nearby devices whereby the device is able to

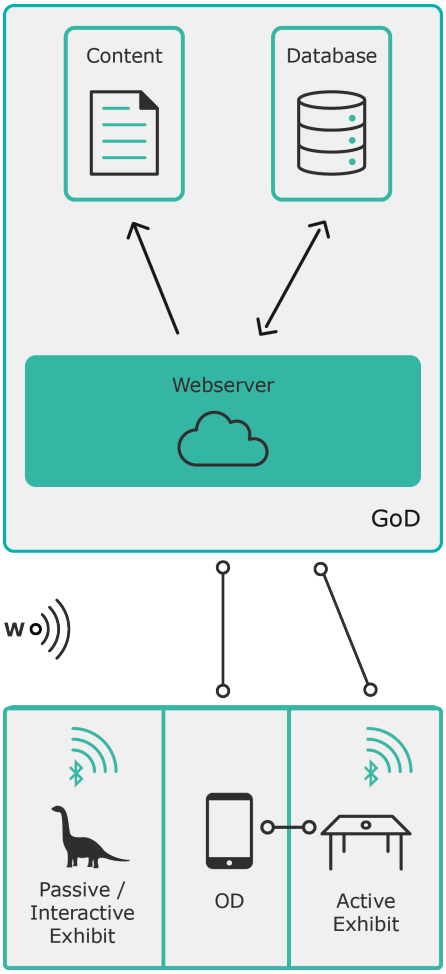


Figure 4.1. MDE structure of the monastery of Klosterneuburg.

perform actions when in close proximity to a beacon. We used the beacons of the company Kontakt.io[2] (see figure 4.2). Kontakt.io offers a library for iOS and Android to receive the encrypted data sent by the beacons. The beacons transmit quite a lot of different data, e.g., a universally unique identifier (UUID) or a minor and major value. It is not possible to change the UUID, but it is possible to change the minor and major values. We used these two values to identify all of our locations. To do that we used the minor value as the id of the location and the major value as the parent id of the location. The different locations are ordered in a hierarchical tree structure. At the top is the museum itself, next are the different sections and finally the exhibits themselves. The assignment of the IDs for the different locations will be explained in more detail in section 4.5.

In total, the exhibition includes 35 different locations. Figure 4.3 shows a plan of the museum and the exhibition. Each color represents a section, and each number (e.g., 1.1 or 2.7) represents an exhibit. The visitors start at the entrance, which has the color red and start with the exhibit ”1.1”. The museum and exhibition are designed in a way that visitors

[2] https://kontakt.io/

Figure 4.2. The used BLE beacons of Kontakt.io (Kontakt.io 2019)

just have to follow a path. The path continues from section one through sections two and three until four. Finally, in section five, the visitors have to go to the first floor. On the first floor, there are two more exhibits before the visitors leave the museum by the exit on this floor.

In total, the MDE includes three active exhibits which are connected to GoD. The first is a quiz where an unlimited number of visitors can join, answer questions, and collect points for a high score list. The second one is a legend game where the visitors can build the legend of the monastery of Klosterneuburg. And the last one is an interactive and explorable version of the genealogical tree of the Babenberger family. The technical part of these exhibits is described in more detail in section 4.6.

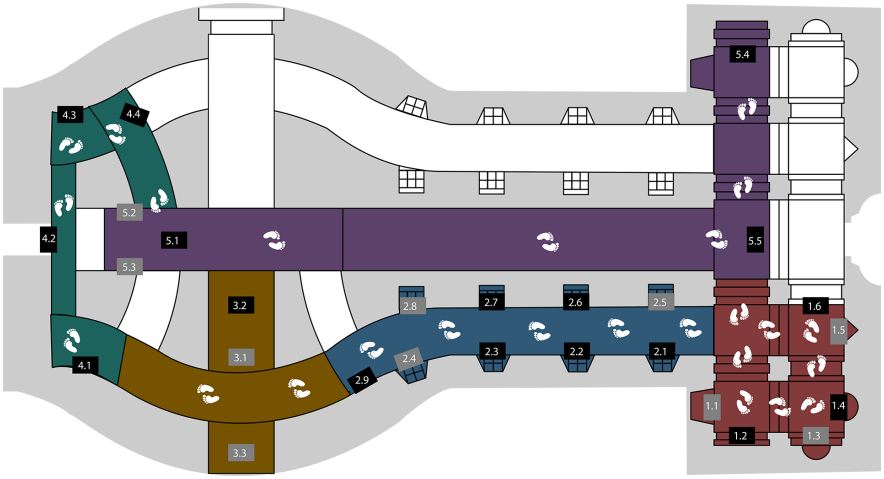


Figure 4.3. Plan of the museum and exhibitions of Klosterneuburg.

4.3 OD/App

Section 3.5 presented the app structure of Blumenstein, Kaltenbrunner, et al. (2017) where a native Unity app is adopted to access necessary hardware components and add a Webview to display an Angular project. For the app of Klosterneuburg, we changed this structure by removing the Unity part. We did that because we do not need a Unity part for the exhibition in Klosterneuburg. For the interactive exhibits, the app only needs a way to display augmented reality (AR) elements. For this purpose, Vuforia is sufficient, because it offers a native library[3] for iOS and Android. The rest of the structure stays the same as one can see in Figure 4.4. In the next sections, the different elements of the app are described in more detail.

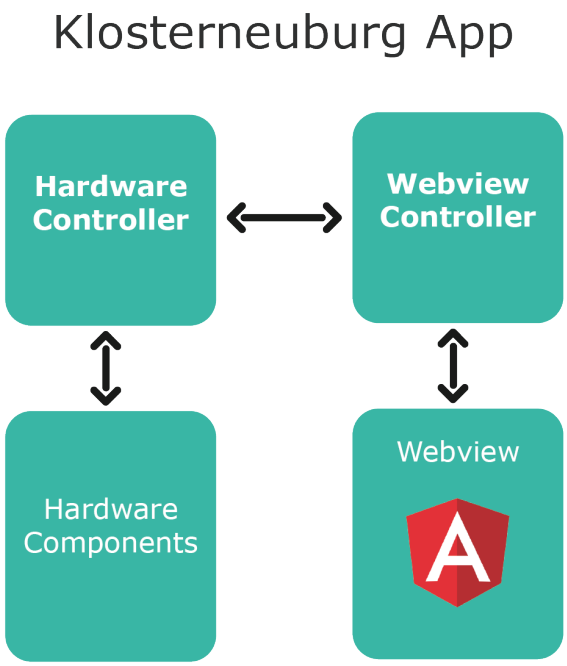


Figure 4.4. Structure of the Klosterneuburg app.

4.3.1 The Emperor's New Saint

This section presents the final app for Klosterneuburg. The screenshots of the application show the functionality of the app. This thesis only focuses on the technical structure, not on the user interface itself. If a visitor opens the app, he/she first sees the start screen (see figure 4.5). In this first screen, the visitor can either register as a new user or as a new guest user. They can also log in with their user credentials. If the visitor registers as a

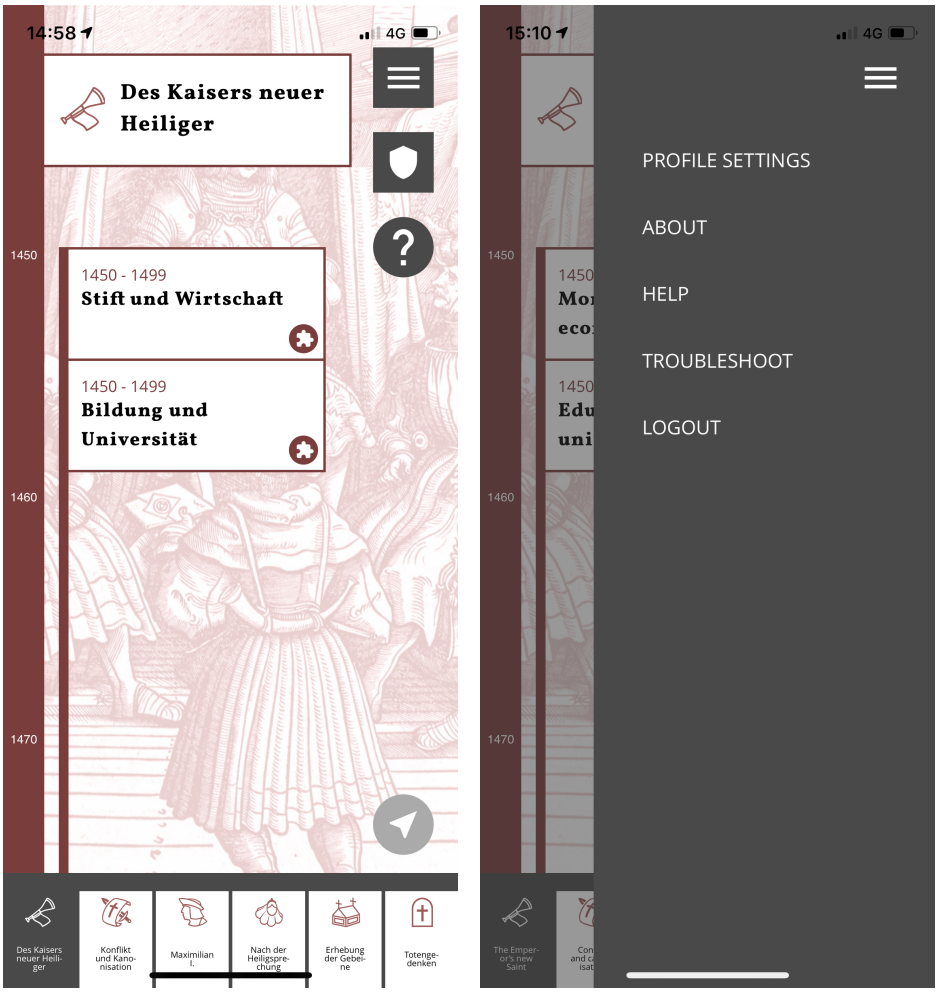
[3] https://library.vuforia.com/getting-started/overview.html

new user, the app receives a token with which all further events will be authenticated (more information in section 4.4.4). The token is stored in the storage of the smartphone. The next time the visitor opens the app, it checks if a token exists and if so the token is used to log the user in again automatically.

Figure 4.5. Start screen of the Klosterneuburg app.

When the visitor is logged in he/she sees the timeline screen (see figure 4.6:a). The exhibition is structured in six different sections, which can be seen at the bottom of the screen. The exhibition starts with the section ”The Emperor’s new Saint” and ends with the section ”Commemoration”. Above the sections, the visitor can see a timeline in which the user can scroll and which includes all the exhibits of this section. By clicking on one of these elements, the detail screen of this exhibit can be opened. The arrow button at the bottom helps the user to find the exhibit closest to them. By clicking on it, the app switches to the exhibit’s section and is scrolling in the timeline to the designated element.

The first button at the top opens the side menu, which includes four elements (see figure 4.6:b). In the profile settings, the user can change their information, or when they registered as a guest user, they can change their account to a real user account. The ”About”



(a) Timeline screen (b) Side menu

Figure 4.6. Timeline screen and side menu of the Klosterneuburg app.

section contains information about the exhibition and the research project MEETeUX. In the ”Help” section, the visitor can find useful information and a guide of the different parts of the app. The ”Troubleshooting” section helps the visitor if the app does not work as intended. It shows the visitor if the app found some kind of problem, e.g., if the user is not connected to the correct WIFI. The last menu element is logging the user out of the app. Guest users should be careful with the ”Logout” button because if they log out the stored token will be deleted and then it will not be possible to log in again with the same guest account.

The second button which has a shield icon (see figure 4.7) opens the ”Code of Arms” (COA) screen of the app. In this screen, the visitor can build their own code of arms which is constructed of four elements, namely shield, charge, helmet and mantling, and two colors. While all shields are already unlocked when the user registers the other parts must be unlocked while exploring the exhibition. By clicking on the desired COA part, the

app shows the necessary task to unlock the part. For example, in order to unlock the eagle charge, the visitor has to switch to the Sunthaym's perspective on any exhibit. In order to save the changes locally and at the server, the visitor must click on the save changes button at the top.

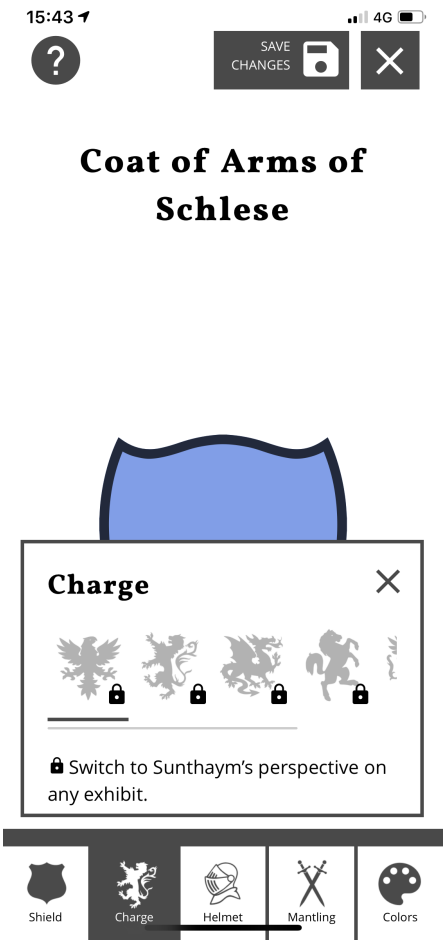
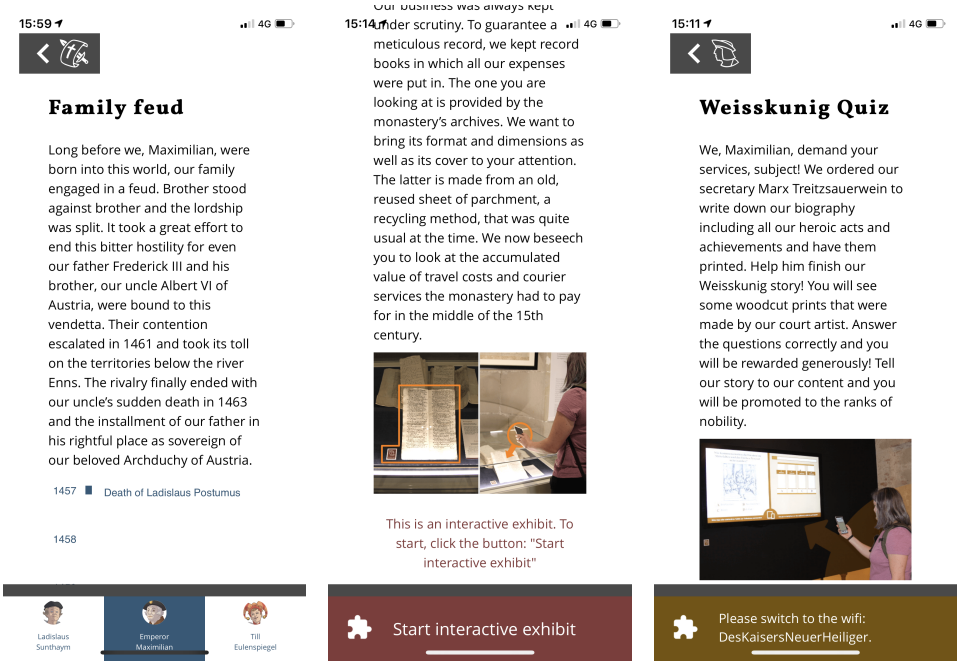


Figure 4.7. Code of arms screen of the Klosterneuburg app.

To show the content of the exhibits, the visitor must click on a location in the timeline. In the exhibition of Klosterneuburg there are three different types of exhibits, namely passive, interactive, and active exhibits. Figure 4.8:a shows a passive exhibit. These exhibits contain text, images, and a timeline with important events of the past. The passive exhibits have three different storytellers, namely Ladislaus Sunthaym, Emperor Maximilian and Till Eulenspiegel. Every storyteller tells the story from his own point of view.
Figure 4.8:b shows an interactive exhibit. In the Klosterneuburg exhibition, all interactive exhibits use augmented reality (AR) to display additional content. In the detail view of the interactive exhibit, the app shows the user how they can use their OD to see the AR content. To start the AR content, the visitor has to click on the button "Start interactive exhibit" at the bottom of the detail view. The AR view shows the visitor a translation or readable version of the displayed records.



(a) passive exhibit (b) interactive exhibit (c) active exhibit

Figure 4.8. The three different types of exhibits.

The last part of figure 4.8:c shows the detail screen of an active exhibit. The figure shows the ”Weisskunig Quiz” of the exhibition. Here the user can connect with the exhibit to participate in a quiz (for more information see section 4.6). At the end of the detailed view, there is another button to connect to the active exhibit. If the user is not in the right WIFI, then the app will inform the user as figure 4.8:c shows. The user is also informed if the exhibit is currently offline or occupied. If none of this is true, then the user can establish a connection.

4.3.2 The Native Part

For the exhibition in Klosterneuburg a native app for Android[4] and iOS[5] was developed. The purpose of the native part is only to access the smartphone's hardware. As mentioned before we use the Software Developer Kit (SDK) of Kontakt.io to detect the Bluetooth beacons. In order for the SDK to fully function, the app needs access to the location services and Bluetooth of the smartphone. Therefore, the user has to approve that the app is allowed to use these hardware functions. When the app starts the beacon scanning, it always receives a list of available beacons. The first item in the list represents the nearest received beacon. If the last beacon received does not match the latest nearest beacon, then the

[4]https://github.com/fhstp/meeteux-android
[5]https://github.com/fhstp/meeteux-ios

new beacon will be forwarded to the web part. The web part then notifies the user that there is a new exhibit nearby and automatically scrolls to it in the timeline. It is possible that a beacon is received only once as the strongest beacon. The beacon information would then be forwarded to the web part immediately. To prevent this from happening, a CircularBuffer was programmed. The app is now does not use the strongest beacon in the list. Instead, it adds the new values of the list to the Buffer of the beacons. Then the median value is calculated for each beacon to determine which is currently the closest one.
It should also be mentioned that in Android, the app itself can configure how often it scanns for beacons. In iOS, the operating system controlls the beacon scanning rate, which could sometimes cause the iOS app to take longer to recognize a new exhibit.

Furthermore, the app needs access to the WIFI functionality of the smartphone to check if the user is connected to the exhibition WIFI. If the user is not connected to the right WIFI, he/she will not be able to connect to the active exhibits. Therefore, the web part asks the native part for the WIFI SSID at specific points e.g., when the user wants to join an active exhibit.

Finally, the native part is responsible for storing the authentication token which the app receives after the registration or login. The iOS app solves this problem ”by giving your app a mechanism to store small bits of user data in an encrypted database called a keychain” (Apple 2019). In the Android app, we use the Shared Preferences storage. ”The SharedPreferences APIs allow you to read and write persistent key-value pairs of primitive data types: booleans, floats, ints, longs, and strings” (Android 2019). If the web part needs the token, it sends a notification to the native part. The native part is accessing the corresponding data storage and sends back the found token or `NULL` if no token was found.

4.3.3 The Web Part

The web part mostly consists of an Angular project. As mentioned before the compiled Angular project is displayed in a native Webview. The Webview also enables communication between the web and the native part. This thesis does not cover all parts of the app in detail but only the the most important ones.

Figure 4.9 shows the concept of the Angular project. As in any Angular project, the view consists of Angular components. A component can be an entire view or just a pop-up. Angular is responsible for creating and rendering these components as well as for the creation and rendering of its children. Angular checks if the component’s data-bound properties change, and also destroys the component before removing it from the DOM. ”Angular offers lifecycle hooks that provide visibility into these key life moments and the ability to act when they occur” (Angular.io 2019). Therefore, the component can e.g., establish a socket

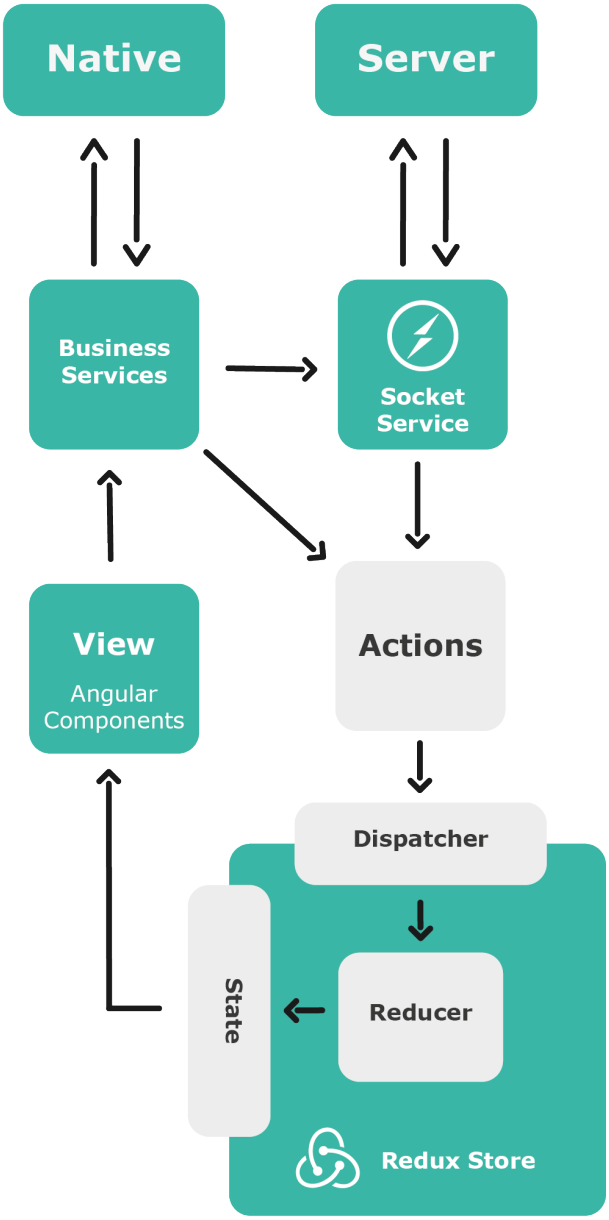


Figure 4.9. Project concept of the Angular project.

connection to the active exhibit when the app routes to the corresponding view by listening to the `ngOnInit()` hook.
A component always consists of four different files. The first one is the `component.ts` file, which contains the complete source code of the component. The source code of the project is always written in Typescript. The second file is the `component.spec.ts` file, which includes unit tests for the source files. Each source file should have a spec file. The third file is the `component.html` file which contains the actual GUI in the form of HTML

syntax. In the HTML file, we can access the data of the source file. Furthermore, Angular uses data-bound properties to update the displayed DOM of the app automatically. The last file is the `component.css` file, which includes all styling elements for the component. The stylesheet is programmed with the standard Cascading Style Sheets (CSS) format.

Since this app was programmed for a museum, a requirement for the app was to support multiple languages. In order to be able to display UI components of the app in several languages, the library "ngx-translate[6]" is used. The library offers Angular a translation service with which the different languages can be defined and easily switched. The texts for the UI elements are stored in JSON files. For each language, there is one JSON file. Therefore, there is one file for the German texts and one for the English texts.
The translation service is only responsible for the text in UI components like buttons or menu items. However, the contents of the exhibits are stored on the server. This has the advantage that content can be adapted at any time or entire exhibits can be added without the need for an update. Therefore, it is necessary to download all information about the exhibits and their contents right after the registration or login. The default language of the app is German, and therefore the contents are by default downloaded in German. The user has the possibility to change their desired language. The app then loads all contents in the new language from the server.

In order to manage the data received by the server, the app stores it in one centralized state. For this, the app uses Redux[7]. Redux enables the app to manage the entire application state in one object, namely the Store. If the store gets updated, it will trigger re-renders of the view which observe the state. If a service wants to update the state, it dispatches an action. The dispatched actions call a function which handles these actions, modifies the state and returns the updated Store. These functions are named Reducers.

The app uses RxJS to observe the state changes and to modify component properties. "RxJS is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code" (*RxJS* 2019). The listing 4.1 shows an example of an Observable which is used in a component to keep track of the changes in the Store. In this example, the app observes the location status to show the user if the active exhibit is available, occupied, or offline.

Listing 4.1. Source code of a state observable.

```
1 this._unsubscribe = this.appStore.subscribe(() =>
2 {
3     const state = this.appStore.getState();
4     this.updateLocationStatus(state.locationStatus);
5     if(state.closestExhibit)
```

[6] https://github.com/ngx-translate/core
[7] https://redux.js.org

```
6     {
7         this.updateJoinButtonStatus(state.closestExhibit);
8     }
9     this.locationSocketStatus = state.locationSocketStatus;
10 });
```

To access the hardware, the app uses business services. In more detail it uses the `NativeCommunicationService` class. Listing 4.2 shows the communication implementation for iOS. For iOS, it is quite simple because the app just sends a JSON object which contains the message name and the data which should be sent to the native part.

Listing 4.2. Source code of the communication with the iOS native part.

```
1 const message =
2 {
3     'name' : messageName,
4     'data' : messageBody
5 };
6
7 this.winRef.nativeWindow.webkit.messageHandlers.
8 observe.postMessage(message);
```

The Android implementation is slightly different because it does not use one `postMessage` method. Instead Android calls specific methods which are defined in the native Android part like one can see in figure 4.3

Listing 4.3. Source code of the communication with the Android native part.

```
1 this.winRef.nativeWindow.MEETeUXAndroidAppRoot.getDeviceInfos();
```

To receive the response of the native part, the app uses the native window. For this purpose, we add a function to the window which can be called from the native part. For example to receive the response for the ”getDeviceInfos” request we define the method of listing 4.4.

Listing 4.4. Source code of the ”index.html” file to receive the device information.

```
1 window.send_device_infos = function(deviceinfos)
2 {
3     window.angularComponentRef.
4         componentFn("send_device_infos",deviceinfos);
5 };
```

The function in the ”index.html” file redirects the function to Angular (see listing 4.5) where the app calls the `callFromOutside` function. As parameters this function receives the message name and the data.

Listing 4.5. Source code of the ”app.module.ts” file to receive messages from the native part.

```
winRef.nativeWindow.angularComponentRef = {
    zone: this.zone,
    componentFn: (message, value) =>
        this.callFromOutside(message, value),
    component: this
};
```

Based on the message name which is processed in a switch construct the app then calls a method in the `NativeResponseService` which handles the data and updating the Store. An example of the device information can be found in listing 4.6.

Listing 4.6. Source code for the ”send_device_infos” case.

```
case 'send_device_infos':
{
    this.nativeResponseService.odRegister(value);
    break;
}
```

If needed, the business services use the socket service to send an event to the server. Listing 4.6 shows that the `odRegister()` method of the `NativeResponseService` was called. This method uses the socket service to register a new user. The Angular project uses the ”ngx-socket-io[8]” library to establish a socket connection with the server and the active exhibits. The library also allows the app to have multiple socket connections at the same time.
Listing 4.7 shows the register method of the socket service. As one can see, the method consists of two main parts. First, the socket connection is used to emit an event to the server. Next, the app registers an event listener to receive the response. For most events, the app only registers an event listener when it emitted an event to the server itself and is waiting for a response.

Listing 4.7. Source code of the ”registerOD” method in the socket layer.

```
public registerOD(data: any): any
{
    this.socket.emit('registerOD', data);
```

[8] https://www.npmjs.com/package/ngx-socket-io

```
4
5       this.socket.on('registerODResult', result =>
6       {
7         const res = result.data;
8         const message = result.message;
9
10        if (message.code > 299)
11        {
12          this.store.dispatch(this.statusActions.
13              changeErrorMessage(message));
14          return;
15        }
16
17        this.store.dispatch(this.userActions.
18          changeUser(res.user));
19        this.store.dispatch(this.userActions.
20          changeLookupTable(res.locations));
21        this.store.dispatch(this.userActions.
22          changeToken(res.token));
23        this.store.dispatch(this.statusActions.
24          changeLoggedIn(true));
25
26        this.locationService.setToStartPoint();
27
28        this.router.navigate(['/mainview']).then( () => {...});
29
30        this.socket.removeAllListeners('registerODResult');
31      });
32  }
```

As described in section 4.4.3, the result of the server always contains a data and a passage part. If an error occurs at the server, the data part is `NULL`, and the message code is bigger than 299. Therefore, the app dispatches an error if the code is bigger than 299 and exits the method. If the request was a success, the received data is used to update the Store by dispatching various actions. Like figure 4.7 shows sometimes the app uses the Angular router to navigate to another view of the app. Finally, all socket service methods remove the registered event listeners. This is required because the app would otherwise receive the response twice at the next request of the same event, which would cause a problem.

4.4 The Server - GoD

In this section, the server will be described in more detail. The first subsection focuses on the programmed software part of GoD. Secondly, the used operating system and server configuration will be described in more detail. Also, continuous deployment and monitoring tools will be described in this section. The last section will focus on the hardware we used for the Klosterneuburg server.

4.4.1 GoD

GoD was programmed with Node.js[9] which is a JavaScript runtime and an open-source server environment. The default programming language is JavaScript, but you can extend a Node.js project with Typescript. So what is Typescript? TypeScript is a superset of the JavaScript programming language and was developed by Microsoft based on the ECMAScript 6 (ES6) standard of JavaScript. TypeScript language constructs such as classes, inheritance, modules, anonymous functions, and generics were also adopted in ES6. The problem is that ES6 is not runnable on all devices or browsers. Therefore, applications can be developed in Typescript and then compiled to ECMA Script 3 (ES3) or to ECMA Script 5 (ES5). Each JavaScript code is also valid TypeScript code, so that common JavaScript libraries (such as jQuery or AngularJS) can also be used in TypeScript.
Typescript enables IDEs to provide a richer environment for finding common errors while programming software. So, for larger JavaScript projects, adopting TypeScript might result in more robust software, while still being deployable where a regular JavaScript application would run. Important is that the language constructs of TypeScript only exist until compilation and not at runtime. The reason is that the TypeScript code is compiled to ES3 or ES5 code which do not know about constructs like classes or types.

Due to the growing awareness of Node.js, there are now a large number of modules with which servers can be developed a lot quicker and easier. The following list shows the most important modules of the Node.js server:

Express Express is a simple and flexible Node.js framework that provides many powerful features and capabilities for web applications and mobile applications. With countless HTTP utility methods and middleware features, creating a powerful API is quick and easy (StrongLoop 2019).

Sequelize Sequelize is a promise-based Object Relation Mapping (ORM) for Node.js. ORM is a process of mapping between objects and relational databases. An ORM acts like an interface between two systems and therefore, provide advantages for

[9] https://nodejs.org/en/

developers like saving time and effort and rather focusing on business logic. ORM helps in managing queries for multiple tables in an effective manner. Lastly, an ORM is capable of connecting with different databases. Sequelize supports the dialects PostgreSQL, MySQL, MariaDB, SQLite and MSSQL (Depold 2019).

Socket.io "Socket.IO is a library that enables real-time, bidirectional and event-based communication between the browser and the server. It consists of a Node.js server and a JavaScript client library" (Socket.IO 2019). It offers many different features like auto-reconnect, disconnection detection for both server and client, multiplexing, or rooms, which allows sending specific messages only to a group of clients. But Socket.IO is not an ordinary WebSocket implementation because it adds additional metadata to the socket. Furthermore, Socket.IO is using WebSockets if possible, but if the client is not supporting WebSockets, the server switches to long polling.

JWT To authenticate users to the server, GoD uses JSON Web Tokens. "JSON Web Tokens are an open, industry-standard RFC 7519 method for representing claims securely between two parties" (auth0 2019). The module allows the server to decode, verify, and generate JWTs.

dotenv "Dotenv is a zero-dependency module that loads environment variables from a .env file into process.env." (Beatty 2019). The file must be stored in the root directory of the project, and the content consists of a key, value pairs.

winston God uses Winston as a simple and universal logging library with support for multiple transports. A transport could be the console, a log file, or a database. Each Winston logger can have multiple transports configured at different levels, e.g., error, debug, information (Robbins 2019).

Nodemon Nodemon is a development server that will monitor for any changes in your source code and automatically restarts the server.

Mocha GoD uses mocha to test the Socket.io interface. "Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple and fun. Mocha tests run serially, allowing for flexible and accurate reporting while mapping uncaught exceptions to the correct test cases" (mochajs.org 2019).

Chai Chai is a behavior-driven development (BDD) / test-driven development (TTD) assertion library for node and the browser that can be delightfully paired with any JavaScript testing framework (chaijs.com 2019). God uses chai in combination with mocha for testing.

4.4.2 Environment Variables

Of course the server needs config to e.g. access the database or to know where the log files are stored. The config of a server is everything that is likely to vary between deploys (production, developer environments, etc). These configuration variables should not be stored in the code as constants. That would be a violation of the 'Twelve-Factor App' rules of Wiggins (2019). He proposes that config should always be strictly separated from the code. Instead the config should be stored in environment variables. As mentioned before GoD uses the Node.js module Dotenv to load environment variables from a .env file into process.env. Since the .env file contains specific server configuration the file will not be added to the repository. Instead the repository includes a dotenv_example file which can be renamed to .env and filled out. The listing 4.8 shows the content of the dotenv_example file.

Listing 4.8. The content of the dotenv_example file.

```
1  NODE_PATH =
2  SERVER_PORT =
3  CERT_PATH =
4  KEY_PATH =
5  CA_PATH =
6  SECRET =
7  HTTPS = 0/1
8
9  # Database
10 DB_USER =
11 DB_PASSWORD =
12 DB_NAME =
13 GENERATE_DATA = true/false
14
15 # Logging
16 ERROR_LOGGER_FILE =
17 COMBINED_LOGGER_FILE =
18 LOGGER_LEVEL =
```

In the following list all variables which are necessary for GoD will be briefly described:

NODE_PATH The path should lead to the root directory of the project. GoD includes an asset HTML file which can be used to test the socket interface without the need of the app. The path is necessary to load the file when entering the server address in the browser.

SERVER_PORT This variable determines the port on which the server should be running. It should be considered that superuser privileges are needed when running the server on a critical port like 80 (HTTP) or 443 (HTTPS).

CERT_PATH This variable represents the file path to the SSL certificate for the Node.js server (this file is needed to start the HTTPS server).

KEY_PATH The file path to the private key for SSL certificate which was specified in ”CERT_PATH”.

CA_PATH The file path to the certificate of the certificate authority which signed the server certificate. This file is needed in to verify the authenticity of the certificate.

SECRET The secret is used to encode, decode, and verify the JSON Web Tokens (JWT). Default JWT is using the H256 (HMAC with SHA-256) algorithm. Therefore, the secret should be 256-bit long.

HTTPS This variable indicates if the server should be running as a normal HTTP server or as a more secure HTTPS server. It is recommended to use an HTTPS server when running the server in production. Nevertheless, it is not necessary to do so if one is developing locally. If the variable has the value 1, the server will start as an HTTPS server.

DB_USER The database user is required for Sequelize to access the database. Therefore, the specified username must be created in the desired database, and it needs full access to this database.

DB_PASSWORD The password of the specified database user must be entered here.

DB_NAME The database name which Sequelize should use for the server.

GENERATE_DATA If the value is set to true, the server will always delete all database tables and data and then recreate them. The server will also add example data to the database like locations, content, or an example user.

ERROR_LOGGER_FILE Like mentioned before GoD is using the tool Winston as a logger. This variable, therefore, contains the file path to the file in which the errors are logged/stored (it must be a path to a file, e.g. ”/srv/logging/error.log”).

COMBINED_LOGGER_FILE All log events except the errors are stored in this file. Like the ”ERROR_LOGGER_FILE” the path must be a file-like ”/srv/logging/combinedLogs.log”.

LOGGER_LEVEL Winston supports eight different logging levels[10]. The logging level in-

[10] https://www.npmjs.com/package/winston#logging-levels

dicates what kind of messages will be logged in the application. Each level has a dedicated integer level, e.g., errors have the value of 3. The variable should contain the integer value of the logging level you want the application to have. For example, one could use the value six, which includes all messages up to the info messages.

4.4.3 Server Structure

As already mentioned, GoD offers the app a Socket.io application programming interface (API), which is event-based. This means that once a client is connected to the server, various event listeners are registered for the new socket connection. The app can now send an event to the server to receive data or trigger an action. With each the event, the app can send a payload to the server in the Javascript Object Notation (JSON) format. As one can see in figure 4.10 the Socket.io interface is added to the Express server in order to be available for the clients. When the server receives an event, a method of a controller in the business layer is called. The controller is processing the received data, accessing the database, and returning the result to the socket layer. The socket layer is then sending back the result to the client.

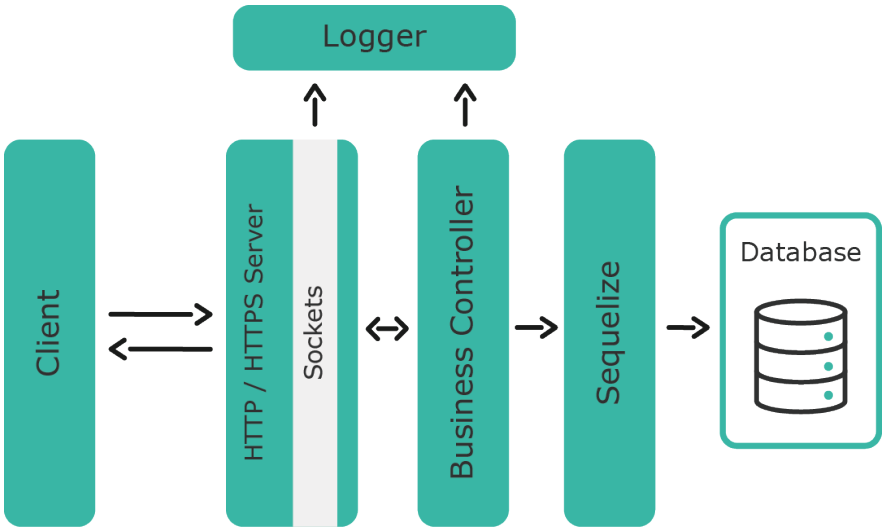


Figure 4.10. Schematic structure of the server/GoD.

The result always has the JSON format and always contains a data and a message part. Listing 4.9 shows an example of a user registration. If everything was successful at the server, the data section would always contain some kind of data. The message section always contains a code, a date, and a string message. When everything was successful, the message code will be between 200 and 299. Everything above will indicate that an error occurred. If an error occurred, the data section would also be `NULL`.

In order to access the database more quickly and so that that SQL commands do not

have to be written directly, GoD uses Sequelize. For this purpose, a Sequelize object was created for each table in the database, and the respective table associations were defined. The business layer controllers are using these objects to interact with the database.
The logger can be accessed from each layer in order to log errors or other information for each layer.

Listing 4.9. The result of a guest user registration.

```
{
data: {
    locations: (37) [...],
    token: "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    user:
    {
        answeredQuestionnaire: false,
        appVersion: "1.0.0",
        contentLanguageId: 1,
        createdAt: "2019-07-29T13:07:42.479Z",
        deviceAddress: "deviceAddress",
        deviceModel: "deviceModel",
        deviceOS: "deviceOS",
        deviceVersion: "deviceVersion",
        id: "c876fd42-54ae-497b-9f6e-9e9cf0641297",
        ipAddress: "not set",
        isDeleted: false,
        isGuest: true,
        name: "Guest5",
        primaryColor: 1,
        secondaryColor: 1,
        socketId: "Lvx07Vsop6UOrmidAAAy",
        updatedAt: "2019-07-29T13:07:42.479Z"
    }
},
message: {
    code: 201,
    date: "2019-07-29T13:07:42.796Z",
    message: "User created successfully"
    }
}
```

4.4.4 Socket Layer

The code to create an HTTP or HTTPS server with Express is pretty simple. The source code in the listing 4.10 shows all the code needed to create an HTTP or HTTPS server with express. Basically, we just have to create a new instance of Express. After that, we are using the standard https or http module of Node.js to create a server. If we want to create an https server, the program must first load the SSL credentials and pass them to the `createServer` method. After that, we create a new instance of the WebSocket class and add it to the server. Finally, the application connects to the database (see listing 4.10:15). When the application successfully connected, the server starts listening on the port specified in the environment variable.

Listing 4.10. Source code of the server creation

```
1  this.app = new Express();
2
3  const runAsHttps:number = parseInt(process.env.https);
4  if(runAsHttps)
5  {
6      const cred = this.loadCredentials();
7      this.server = https.createServer(cred, this.app);
8  }
9  else {
10     this.server = http.createServer(this.app);
11 }
12
13 this.socket = new WebSocket(this.server);
14
15 this.socket.connectDatabase().then( () =>
16 {
17     this.server.listen(process.env.SERVER_PORT, () => {
18         this._logger.info('Server runs on Port ' +
19         process.env.SERVER_PORT);
20     });
21
22 });
```

To add the socket layer to the express server, we just create a new instance of Socket.io's IO class and pass the express server as a parameter `this.io = new IO(server)`. Now we can add an event listener to the new created instance which is listening for new connections: `this.io.on('connection', (socket) => {...});`. If a new client connects to

the server, the callback method will be called, and as a parameter, the new socket connection is passed to this method. This parameter represents the socket connection between the client and the server and can now be used to send events. On the server-side, the application adds all the necessary listeners to the new socket connection.
Listing 4.11 shows an example of a socket event listener. In the example, the listener is registered for the event ”registerODGuest” and passes the received data as a parameter to the callback method. Next, the application calls a method of a business layer controller. In this case it calls the `registerGuest` method of the `odController`. In order for the server to continue processing events, all business controller methods are asynchronous. Therefore, they are returning a promise[11] object. If the promise is resolved the result of the controller is passed to a callback method (see listing 4.11:3). In the case of the ”registerODGuest” functionality, the controller returns the newly generated user and all location information of the exhibition.
But before the server sends back the result to the client, a token will be generated with the JWT module. The token will embed the created user and is encrypted by the environment variable called ”SECRET”. In the next step, we add the token to the socket connection. The token is added to the connection so that we can now authenticate all events where necessary.

Listing 4.11. Socket event to register a new guest user at the server.

```
1  socket.on('registerODGuest', (data) => {
2      this.odController.registerGuest(data, socket.id)
3      .then( (result) => {
4          if(result.data && result.data.user)
5          {
6              const user = result.data.user;
7              const locations = result.data.locations;
8
9              // Generate token
10             const token = jwt.sign({user}, process.env.SECRET);
11
12             // Add token to result and to the socket connection
13             result.data = {token, user, locations};
14             socket.token = token;
15         }
16
17         socket.emit('registerODGuestResult', result);
18     });
19 });
```

[11] https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise

Not all socket events should be available to the client without a valid authentication token. Therefore, a middleware was added to the Socket.io layer as one can see in listing 4.12. In the first two lines of the middleware, we can first get the newly sent event from the packet variable and, secondly, access the token from the socket connection. The token was previously added to the socket connection when the client registers as a new user or when it logged in. The first check in the method `checkEventsTokenNeeded` now validates if the sent event needs a token or not. The following list shows which socket events do not need a token:

- 'registerOD'
- 'autoLoginOD'
- 'loginOD'
- 'disconnectUsers'
- 'registerODGuest'
- 'exhibitDisconnectedFromExhibit'
- 'checkUsernameExists'
- 'checkEmailExists'
- 'checkNameOrEmailExists'
- 'loginExhibit'
- 'updateSeat'
- 'addTokenToSocket'
- 'unlockCoaPartFromExhibit'
- 'getWifiSSID'

If the event needs an authentication token, the application now verifies if the token is valid. If so we can access the user object of the decoded token. If the user object exists, the server can check if the user is a guest user or not. If the user object is a guest user, the method `checkGuestAccess` validates if guest users have access to this event. This means that guest users may not have access to all socket events. This method evaluates whether the event should be accessible to guest users or not.

If everything is fine the middleware returns the `next()` method. If an error occurred, the application returns the `next(new Error('...')` method and passes an error to the method as a parameter.

Listing 4.12. Authentication middleware for the socket connections.

```
socket.use((packet, next) => {
    const event: String = packet[0];
    const token = socket.token;

    if(this.checkEventsTokenNeeded(event))
    {
        jwt.verify(token, process.env.SECRET, (err, decoded) => {
            if(err) {
                this._logger.error('JWT Error - Event: ' +
                event + ' Error: ' + err);
                return next(new Error('Invalid token Error'));
            }

            const user = decoded.user;

            if(user) {
                if(user.isGuest) {
                    if(this.checkGuestAccess(event)) {
                        return next();
                    }
                    else {
                        this._logger.error('...');
                        return next(new Error('...'));
                    }
                }
                else {
                    return next();
                }
            }

            return next();
        });
    }
    else {
        return next();
    }
});
```

Until now, we always talked about direct socket connections between a client and the

server. With the `socket` variable we get when a new connection is established, the server and client can send events to each other. But besides the direct connection, both the client and the server can also send an event to all connections. Here we speak of a broadcast that can look like this: `socket.broadcast.emit('some socket event')`. This event will be received by all participants, which is quite useful when we want to announce something globally.
Finally, the application needs another way to send direct messages to specific devices. An example could be that a visitor is not directly connected to an active exhibit, but the communication is via the server. Why should we want to do that after having previously mentioned that a direct connection allows for faster communication? The reason is that the application might just need some data from the client just at the beginning and at the end of the communication. The application then works without any further data from the client. An additional connection to the active exhibit would be superfluous. So that the data can now be sent to the exhibit, the server must be able to forward the data. This can be done when the server knows the socket id of the connection between the server and the active exhibit. Every socket connection has a universally unique identifier (UUID) with which the connection can be identified. We can use this UUID to send an event directly to this client. In order to do so, the application is storing all socket ids in the database. To send an event to a client, the application is not using the socket variable. Instead, we use the instance of the Socket.io class `IO` which we created in the constructor of the Websocket class of the server. The application can use the `to` method of the `IO` class to emit the event to the designated client: `this.io.to(socketId).emit('some socket event')`.

4.4.5 Business Controller

Like mentioned before the business controller is processing the data received by the sockets, accessing the database via the Sequelize objects and then prepare and return the result to the socket layer. GoD uses five different controllers, and each has its own specific purpose:

ConfigController This controller accesses the settings table of the database and compares if the app has the correct version number and if the WIFI with which the OD is connected is correct.

ExhibitController The purpose of this controller is only responsible for login and shutdown the active exhibits.

LocationController This controller is responsible for everything which is related to the exhibition's locations. For example, there is a method called `registerTimelineUpdate`, which is always called when the app on the OD receives a new beacon. Or the method `checkLocationStatus` which is used to check if an active exhibit is free,

occupied or even offline. The LocationController is the only one who needs a reference to the `IO` class of Socket.io in order to redirect messages for the exhibits or clients.

ODController The ODController offers methods which are related to the users. For example the controller contains the method `registerOD` or `registerGuest` which are used to create a new user or new guest user. It also contains the methods to log in users or to check if a username is already taken.

CoaController The code of arms is part of the app and is described in more detail in section 4.3. But basically, it offers the user the option to built their own code of arms. Therefore the user unlocks different parts while exploring the exhibition. The CoaController contains all methods to unlock the different parts or to change the selected items or color.

All the controllers are initialized in the constructor of the Websocket class as one can see in listing 4.13. Furthermore, all methods are asynchronous and return a promise. A listing of all controller methods and their corresponding purposes can be found in the wiki[12] of the GoD repository.

Listing 4.13. The constructor of the Websocket class.

```
1  constructor(server: any)
2  {
3      this.io = new IO(server);
4      this.odController = new OdController();
5      this.locationController = new LocationController(this.io);
6      this.exhibitController = new ExhibitController();
7      this.configController = new ConfigController();
8      this.coaController = new CoaController();
9      this.database = Connection.getInstance();
10
11     this._logger = Logger.getInstance();
12
13     this.attachListeners();
14 }
```

[12] https://github.com/MaximilianFHSTP/max-god/wiki

4.4.6 Database / Sequelize Layer

Most of the database logic is programmed in the Connection class. This class uses the Singleton design pattern, which means that all controllers are always using the same instance of this class. The first step to use Sequelize as an ORM mapper is to create a new instance of the Sequelize class. Listing 4.14 displays all the different parameters of the Sequelize constructor. Most of them are straightforward, but it is important to mention that in order to have correct timestamps at the server, the application must set the timezone option in the Sequelize constructor. By default, it is using the UTC timezone UTC±0.
The logging option indicates if Sequelize should log all executed SQL statements which are executed by the server. This can be pretty useful if one is debugging the server functionality, but it would generate a lot of log entries if it is used in production.

Listing 4.14. The constructor of the Sequelize class.

```
1 this._sequelize = new Sequelize(process.env.DB_NAME,
2 process.env.DB_USER, process.env.DB_PASSWORD,
3 {
4     host: 'localhost',
5     dialect: 'mysql',
6     logging: false,
7     timezone: '+02:00'
8 });
```

The three most important methods of the Connection class are `syncDatabase`, `initDatabaseTables` and `initDatabaseRelations`. The `syncDatabase` method is responsible to actually connect with the database. When a new instance of the Sequelize class is created, we pass all needed parameters to the class, but it does not automatically connect with the database. In order to do so the application has to call the `sync()` method as one can see in listing 4.15:18.

Listing 4.15. Sync method of the Conenction class.

```
1  public async syncDatabase()
2  {
3      const dataFactory = new DataFactory();
4      dataFactory.connection = this;
5
6      this._logger.info('Syncing␣to␣database!');
7
8      if(process.env.GENERATE_DATA === 'true')
9      {
10         await this._sequelize.sync({force: true});
```

```
11          this._logger.info('Creating data!');
12          await dataFactory.createData().catch(err => {
13              this._logger.error(err);
14                  this._logger.error("Could not create data!");
15          });
16      }
17      else
18          await this._sequelize.sync();
19
20      await this._settings.findByPk(1).then(result =>
21      this._currentSettings = result);
22  }
```

If the process variable ”GENERATE_DATA” is set to true the `sync()` method is called with the option `sync(force: true)`. If the force option is set to true Sequelize deletes all data and all tables in the defined database. After that it is regenerating the tables and adding the data which is defined in the `createdData()` method of the `dataFactory` class. Of course, this also means that you have to work with caution because setting the wrong value for this environment variable means that all collected data would be lost.

In the `initDatabaseTables` method, Sequelize defines all database tables and their corresponding attributes. Listing 4.16 shows the definition of the user table. As one can see, the definition has a JSON format. Each element represents an attribute of the database table, and all can be configured as you like. For example, the `id` is defined as the primary key and as UUID. Or the name is defined as a string, it cannot be null, and it must be a unique value. As type Sequelize always uses its own types which are then mapped to the corresponding database types. E.g. Sequelize has a `boolean` data type which represents as `TINYINT(1)` in the database.

Listing 4.16. Definition of the user table with Sequelize.

```
1  this._user = this._sequelize.define('user',
2  {
3      id: {
4          primaryKey: true,
5          type: Sequelize.UUID,
6          defaultValue: Sequelize.UUIDV4,
7      },
8      name: {
9          type: Sequelize.STRING,
10         allowNull: false,
11         unique: true
```

```
12      },
13      password: {
14          type: Sequelize.STRING,
15          allowNull: true
16      },
17      email: {
18          type: Sequelize.STRING,
19          unique: false
20      },
21      .
22      .
23      .
24      isDeleted: {
25          type: Sequelize.BOOLEAN,
26          defaultValue: false,
27          allowNull: false
28      }
29  });
```

All tables are defined like this and then stored in a variable to be accessible in the controller classes. The next step is to define the relations of the database tables. This is done in the `initDatabaseRelations` method. Listing 4.17 shows a 1:n relationship of the tables users and locations to store the user's current position. The definition of the relation can also be used to define the foreign key, e.g., `currentLocation`.

Listing 4.17. Definition of the table relationship between the users and locations tables.

```
1 location.hasMany(this.user, {foreignKey: 'currentLocation'});
2 user.belongsTo(this.location, {foreignKey: 'currentLocation'});
```

Now everything is finally defined, and the table variables can be used in the controllers to access the data. Listing 4.18 shows an example of a guest user creation. The method `registerGuest` is part of the `OdController` and in this method, the defined Sequelize user table is used to create a new user. The attributes are passed as a JSON object, and the names must exactly match the definition of the table. For further information about the usage of Sequelize, one should visit their website "https://sequelize.org/master/".

Listing 4.18. Creation of a guest user.

```
1 this._database.user.create({
2     name: identifier,
3     deviceAddress: deviceAddress,
4     deviceOS: deviceOS,
```

```
5       deviceVersion: deviceVersion,
6       deviceModel: deviceModel,
7       ipAddress: 'not set',
8       contentLanguageId: language,
9       socketId
10  });
```

4.4.7 Messages

Like mentioned before all results include a message which allows the client to quickly check if the received result was successful or if an error occurred. Listing 4.19 shows the `Message` class which is constructed very simple. The class only contains three variables, namely code, message, and date. The code and the message are passed as parameters when initializing a new object of the `Message` class. The date, on the other hand, is automatically generated in the constructor and therefore, always contains the current date and time when the message was generated.

Listing 4.19. Definition of the Message class.

```
1  export class Message
2  {
3      private code: Number;
4      private message: String;
5      private date: Date;
6
7      constructor (code, message)
8      {
9          this.code = code;
10         this.message = message;
11         this.date = new Date();
12     }
13 }
```

The message codes have been globally defined, so they only have to be changed at one point in the code. In total, there are six different types of codes. Each type has a different hundred area for its codes. For example, all success codes have a code between 200 and 299. The following table lists all types and code ranges for the messages.

Table 4.1. All message types and their corresponding code ranges.

Types	Code range
Success	200 - 299
Location	300 - 399
OD	400 - 499
Authentication	500 - 599
Code of Arms	600 - 699

4.4.8 Logging

As mentioned before GoD is using Winston to log information. Like the `Connection` class, the logger class is designed as a singleton. The logger level and the intended files are loaded from the environment variables. In the next step, a new Winston logger is created as one can see in figure 4.20:7. The desired format is JSON, and as default transports, two files are defined. File one includes all errors, and file two will include all other log events. If the server is not started in production mode, Winston also loggs to the console. This is quite useful while one debugs the server.

Listing 4.20. Definition of the Logger class.

```
1  private constructor()
2  {
3      const level = process.env.LOGGER_LEVEL;
4      const errorFile = process.env.ERROR_LOGGER_FILE;
5      const combinedFile = process.env.COMBINED_LOGGER_FILE;
6
7      this._logger = Winston.createLogger({
8          level: level,
9          format: Winston.format.json(),
10         transports:
11         [
12             new Winston.transports.File({
13                 filename: errorFile,
14                 level: 'error'
15             }),
16             new Winston.transports.File({
17                 filename: combinedFile
18             })
19         ]
20     });
21
```

```
22      if (process.env.NODE_ENV !== 'production') {
23          console.log('Winston - Enabled Console logging!');
24          this._logger.add(new Winston.transports.Console({
25              format: Winston.format.simple()
26          }));
27      }
28  }
```

4.4.9 Unit Testing

So that the functionality of the server can be guaranteed even after a change, unit tests have been defined for the server. Therefore, the Node.JS server uses the test framework Mocha.js, which makes asynchronous testing simple. "Mocha tests run serially, allowing for flexible and accurate reporting while mapping uncaught exceptions to the correct test cases" (mochajs.org 2019). Additionally, the server uses Chai as an assertion library. Chai is a behavior-driven development (BDD) / test-driven development (TTD) assertion library for node and the browser that can be delightfully paired with any JavaScript testing framework (chaijs.com 2019). God uses chai in combination with mocha for the unit tests. An assertion library is used to verify if data is correct. This simplifies the tests so that the code does not have to contain a lot of if statements.

Listing 4.21 shows the configuration for the unit tests. Before the unit tests can be started, a socket connection to the server must first be established. For this purpose, the `before` method of Mocha can be used. This method is always called before the unit tests are executed. After the tests, the connection should be terminated again. This is done in the `after` method.

Listing 4.21. Configuration for the unit tests for the server.

```
1  describe('User socket events', function()
2  {
3      var socket;
4
5      before('establish socket connection...', function(done)
6      {
7          if(process.env.https === '1')
8              socket = io.connect('https://localhost:' +
9                  process.env.SERVER_PORT);
10
11         else
12             socket = io.connect('http://localhost:' +
```

```
13                  process.env.SERVER_PORT);
14
15          socket.on('connect', function()
16          {
17              done();
18          });
19      });
20
21      after('closing␣socket␣connection...', function(done)
22      {
23          if(socket.connected)
24              socket.disconnect();
25
26          else
27              console.log('no␣connection␣to␣close...');
28
29          done();
30      });
31
32      .
33      .
34      .
35  });
```

For the actual tests, we are using the Chai assertion syntax as one can see in figure 4.22. The tests are defined with the `it()` method of Mocha. In the method, the test is first sending an event to the server as the app would do. Then an event listener is registered to receive the result of the event. The result is then tested with Chai's `expect()` method to determine if everything was successful. For example, the "registerAsGuest" event should return the status code 201. To test this, the Chai syntax looks like this: `expect(message.code).to.be.equal(201);`

Listing 4.22. Unit test for the "registerAsGuest" event.

```
1  it('registerAsGuest', function(done)
2  {
3      socket.emit('registerODGuest',
4      {
5          deviceAddress: 'deviceAddress',
6          deviceOS: 'deviceOS',
7          deviceVersion: 'deviceVersion',
```

```
8           deviceModel: 'deviceModel',
9           language: 1
10      });
11
12      socket.on('registerODGuestResult', function(result)
13      {
14          const message = result.message;
15          const data = result.data;
16
17          token = data.token;
18          user = data.user;
19          const locations = data.locations;
20
21          expect(message.code).to.be.equal(201);
22          expect(token).to.exist;
23          expect(user).to.exist;
24          expect(locations).to.exist.and.to.have.lengthOf(37);
25
26          socket.removeAllListeners('registerODGuestResult');
27          done();
28      });
29  });
```

4.4.10 Server Management

The previous sections described the programmed application for the server. This chapter now covers the used operating system (OS) on which the server is deployed. Furthermore, the used server management tools are described in more detail.

Node.js runs on multiple platforms, but after the hardware for the server are specific server components, the possible operating systems are also limited. For the exhibition of Klosterneuburg, we therefore use the server image of the Ubuntu OS with the version 16.04 LTS[13]. Ubuntu is one of the most popular operating systems across public clouds and OpenStack clouds. Ubuntu is a Linux distribution and is an easily manageable operating system.

In order to run our Node.JS application on the Ubuntu Server one has to install Node.js on the server. This can be easily done with the command `pkg install node`. Of course, the corresponding user must have root access in order to install a new package. Another

[13] http://releases.ubuntu.com/16.04/

option is the usage of the Node Version Manager (NVM). NVM is a bash script used to manage multiple versions of Node.JS. It allows you to install, uninstall, and switch different node version for different projects. This can be quite useful since sometimes specific Node.JS projects are not always using the newest version. NVM can be easily installed using the curl tool.

The next step is the installation of your desired database. By default, the Node.JS application uses an MYSQL database. The database can be easily installed with the command `sudo apt install mysql-server` and then configured with the command `sudo mysql_secure_installation`. For more details, one can visit the DigitalOcean website[14].

To manage the server one could use a process manager. For Node.js applications, a possible solution is the tool PM2. PM2 is an advanced Node.js process manager to manage your applications states, so you can start, stop, monitor, restart, and delete processes. PM2 can be installed with the Node Package Manager (NPM) which was already installed with Node.js. The command `npm install pm2@latest -g` installs PM2 globally. Now all prerequisite is fulfilled to run the Node.JS application on the Ubuntu server. Before one can start the application with pm2, the node modules have to be installed. Therefore, the command `npm install` must be executed in the project directory. After that, the Typescript files have to be compiled to Javascript files with the command `npm run tsc`. Furthermore, the ”dotenv_example” file must be renamed to ”.env” and filled out.

To start the application with pm2 run the command `pm2 start dist/index.js`. It is important to mention that the application runns as the user who entered the command. So if you start the application as root, the process is also running with root privileges. However, running an application with root privileges also poses some security risks if the application has security vulnerabilities. Therefore, it is better to start the application with a user who does not have root privileges. Note that this user can not start the application on the default reserved ports for HTTP (80) or HTTPS (433) because they require root privileges. To solve this problem you can, for example, start the server on port 8080 and then open it in the firewall. Of course, for the socket connection, the port must also be specified in the app. Another possibility is to forward the traffic from port 80/443 to port 8080. This would make port 8080 inaccessible from the outside.

With the command `pm2 list`, one can see all started applications. Figure 4.11 shows an example output of this command. As one can see, the command presents a lot of different information like the PID of the OS, the uptime of the application, or how often the application was already restarted. Furthermore, the command `pm2 logs Max-GoD -lines 1000` shows you the log file of pm2 for the corresponding application. If an error occurred and the app was restarted, one can find the error output in the PM2 log file.

[14] https://www.digitalocean.com/community/tutorials/so-installieren-sie-mysql-auf-ubuntu-18-04-de

```
prod@meeteux:~$ pm2 list
```

App name	id	mode	pid	status	restart	uptime	cpu	mem	user	watching
Max-GoD	0	fork	21492	online	0	11D	0%	92.5 MB	prod	disabled

Figure 4.11. Output of the ”pm2 list” command.

When the application is running, one can use the command `pm2 startup` to generate a startup script for all running applications. By default, the applications are not surviving a restart. Therefore, it would be necessary to manually restart the applications after the server was restarted. In order to do this automatically, the `pm2 startup` command is generating a script which is executed when the server is starting. The script restarts all applications which were running when the startup script was generated.

If an error occurs, PM2 will automatically try to restart the application. However, the administrator receives no message until now if an error has occurred. Of course, this is not very practical since the administrator currently has to check the status of the server independently regularly. To solve this problem, PM2 offers two additional tools. The first tool is a monitoring tool with the name ”pm2-server-monit”. This module automatically monitors the vital signs of the applications like CPU average usage, Network speed (input and output), Free and used memory space, and more.

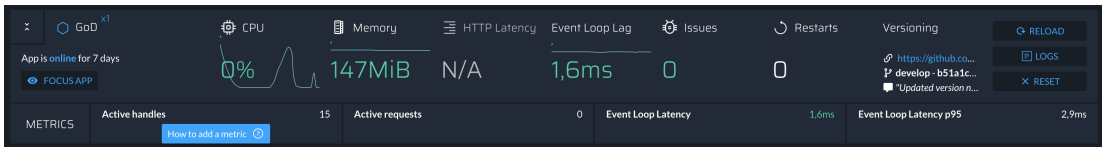


Figure 4.12. Server monitoring example of a pm2 application.

The module can be easily installed by the command `pm2 install pm2-server-monit` and does not need an additional configuration. Furthermore, PM2 offers a website to see the monitoring output. Figure 4.12 visualizes the output of the server for the Klosterneuburg exhibition. As you can see, the website displays all the parameters mentioned above in a clear visualization and in real-time.

The second module is called ”pm2-health” and it can be used to, for example, monitor if the application throws an error or if the application is restarted. After installing the module with the command `pm2 install pm2-health`, it can be configured to send emails when an error occurs, or another event such as an application reboot occurs. Therefore, the administrator does not need to monitor the server independently.

4.4.11 Continuous Deployment

To automatically test the server functionality and then deploy the application at the server, we are using the platform Codeship. Codeship uses fast, powerful Virtual Machines (VM) with preinstalled technology stacks to make starting a Continuous Integration (CI) / Continuous Deployment (CD) pipeline easy and fast. For the Klosterneuburg exhibition, Codeship provides the possibility to automatically run the defined Unit Tests whenever we are pushing changes to specific branches of the repository. If the tests were successful, Codeship is deploying the project at the designated server.

Codeship offers the user the possibility to run specific setup commands before the unit tests are executed. Setup commands can be used to e.g., load dependencies or prepare the database. For this project, we basically run the same commands which were already mentioned in section 4.4.10. Therefore, the node modules must be installed. The Typescript code must be compiled to Javascript, and the environment variables must be configured. By default, Codeship installs a MySQL database server in every VM and saves the user credentials in environment variables. If the tests were successful, Codeship provides a lot of different deployment options. For this case, the tool PM2 is used to deploy the project at a server. The deployment configuration can be stored in a ”ecosystem.config.js” file. Listing 4.23 shows the deployment code for the development server.

Listing 4.23. Deployment configuration for the PM2 tool.

```
develop: {
    user: 'node',
    host: 'god.meeteux.fhstp.ac.at',
    ref: 'origin/develop',
    repo: "https://github.com/MaximilianFHSTP/max-god.git",
    path: '/srv/develop',
    'post-deploy': 'cp ../.env ./ && npm install && npm run tsc
    && pm2 startOrRestart ecosystem.config.js --env production',
    'pre-deploy': 'git reset --hard'
}
```

In the configuration, various options can be defined. PM2 connects to the configured server using SSH and the configured user credentials. It clones the branch of the specified repository to the configured path at the server. It also offers the administrator the possibility to define pre- and post-deploy commands. Before the project is deployed, we want to reset the repository at the server in the case some files changed. After the deployment we copy the ”.env” file into the project folder, install the node modules and compile the Typescript code to JavaScript. Finally, the application is started or restarted. We can use this deployment option in our Codeship CD pipeline by installing PM2 in the VM and then

simply call the command `pm2 deploy develop`. The CD pipeline is always triggered when a change is pushed to the ”develop” or ”master” branch.

4.5 Database

This section deals with the database design of the GoD database. Not every single attribute of the database is explained, but instead, it’s more about the relationships between the individual database tables.

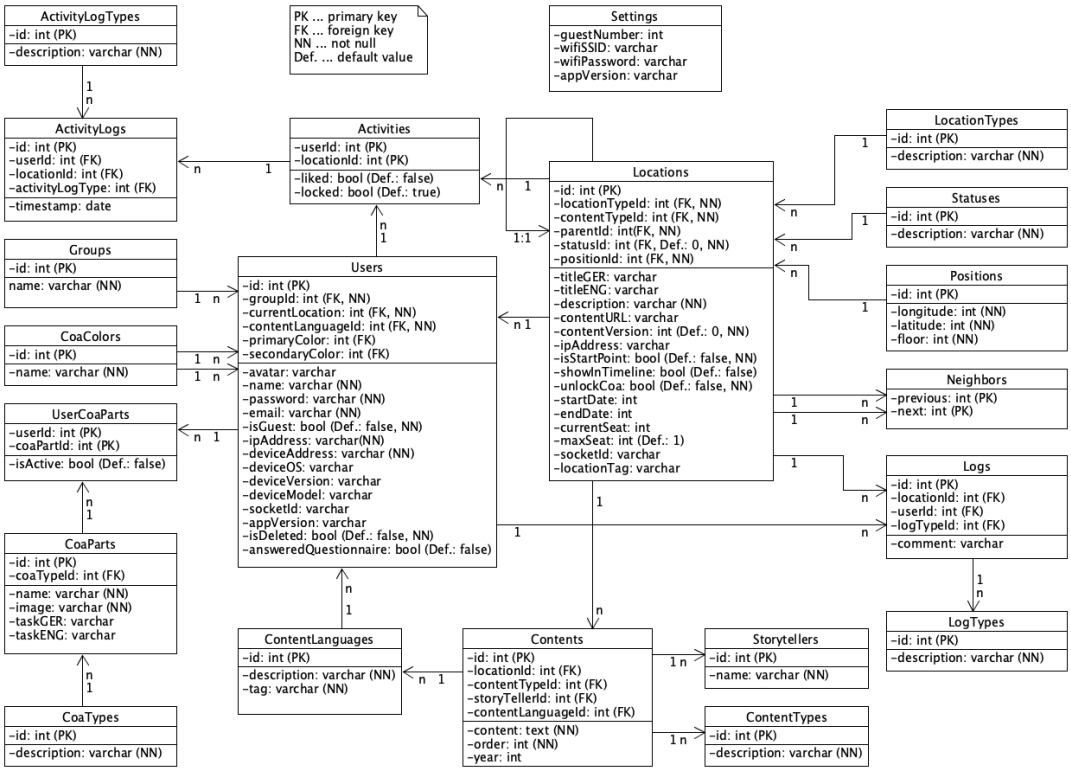


Figure 4.13. Database design of the GoD server.

The two main tables are the `Users` and the `Locations` table. Like the name suggests the `Users` table includes all relevant data of the exhibition users. The user can register with a username, email, and password and additionally, the device information of the OD of the visitor is stored in this table. Furthermore, if the user joins an active exhibit, the socket id is stored. The `Users` table has a reference to the `ContentLanguages` table to define the user’s default language for the app contents. Like mentioned before the user can unlock different parts of the Code of Arms. Therefore, there is an `n:m` connection between the `Users` table and the `CoaParts` table. Each time the visitor unlocks a COA part a new entry in the `UserCoaParts` is generated. The ”isActive” flag is used to determine which COA parts are currently used in their code of arms. Each COA part has a specific type

which is defined in the `CoaTypes` table. The COA has a primary and a secondary color which is stored in the `Users` table as the two references ”primaryColor” and ”secondaryColor”. Each user can be part of a group, and therefore the table includes a ”groupId” to identify the group which is stored in the `Groups` table.

The second most important table is the Locations Table. This table contains all locations of the exhibition. The following listing lists some of the most important attributes of this table:

contentURL This attribute is used to determine which UI component should be loaded in the app.

ipAddress Each exhibit needs a static IP address in order to be accessible for the app. Furthermore, the exhibit uses the address to login at GoD. After the login, GoD changes the status of the location from ”OFFLINE” to ”FREE”. It also resets the ”currentSeat” attribute to zero.

socketId After the exhibit has logged in to the server, the socket id is stored in the database so that GoD can forward messages from the users to the exhibits.

currentSeat This value is used to count the users who are currently connected with the active exhibits.

maxSeats Some exhibits have a limit on how many users can connect to it at the same time. The value of the ”currentSeat” attribute can only ever be as large as the ”maxSeat” value. When the ”currenSeat” value reaches its maximum the status of the exhibit is changed to ”OCCUPIED”.

parentId The different locations are built in a hierarchical tree structure. At the top is the museum itself, next are the different sections and finally the exhibits themselves. To represent this structure GoD is using the ”parentId” of the `Locations` table.

Each location has a specific type which was already mentioned in section 4.3. These types are stored in the table `LocationTypes`. In addition, each location has a status that gives information about the condition of the location. These statues are defined in table `Statuses`. In order to guarantee a certain sequence of locations, some exhibits have neighbors. This neighbor relationship should ensure that an exhibit in the app is only unlocked when the previous neighbor is already unlocked. Therefore, the Neighbors table is used to store this relation.

Each location has multiple contents of different types e.g., a text or an image. The contents are stored in the `Contents` table and the types are stored in the `ContentTypes` table. Each content is always associated with a language stored in the table `ContentLanguages`. Some content is presented from the perspective of a storyteller. The different storytellers are stored in the `Storytelelrs` table.

Until the app receives a Bluetooth beacon, the locations are locked in the timeline. When the beacon was received, a new entry in the `Activity` table is created. Now the location is unlocked in the timeline. Furthermore, each time a beacon of a location is received or when the user is navigating to the detail page of the location, an `ActivityLogs` entry is created. The logs help to understand how the user moves through the exhibition. Each log entry is related to an `AcitivityLogType`.

Finally, there is the `Settings` table of the database. This table stores some settings of the server. First is the attribute ”guestNumber” which is used to create a new guest user with a unique name. Then there are the fields ”wifiSSID” and ”wifiPassword” which are needed to check if the users are connected with the correct network and to show the network credentials in the app. Last but not least there is the field ”appVersion” which is used to check if the app requires an update.

4.6 Active Exhibits

The last section of the Implementation focuses on the active exhibits. The exhibition includes two different types of active exhibits. The first type is called ’activeExhibit’ and the second one is called ’notifyActiveExhibit’. Both include a separate Node.JS server which is running on the exhibit’s computer. The exhibition includes three active exhibits. Two are of type ’notifyActiveExhibit’ and the last of type ’activeExhibit’.

4.6.1 notifyActiveExhibit

For this type of exhibit, the app does not connect directly to the exhibit server. Instead, communication is via GoD. The figure 4.14 shows the behavior if a user is joining an active exhibit of this type. First, the app checks the status of the exhibit. Therefore, it requests the status at GoD. Secondly, GoD responds to the current status of the exhibit. If the exhibit has the status ”FREE” the OD can join the exhibit. Thirdly, the app notifies GoD that the OD wants to join the active exhibit. GoD updates the status of the exhibit and then fourthly, redirects the join message to the active exhibit. Additionally, GoD sends the exhibit the relevant data of the user. Why does the app not have to connect directly to this type of exhibit? The reason is that the exhibit does not need any additional data from the OD after the user has connected. The exhibit only needs to know if a user connects and furthermore the id and name of the user.

Figure 4.14 also shows the structure of the exhibit server, which is quite similar to GoD. The exhibit server is connected to GoD via a socket client. The client calls methods of the business controllers which use Sequelize to access the SQLite database. ”SQLite

is an embedded SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files"(SQLite.org 2019). Therefore, using an SQLite database is perfect for the use of the exhibit server.

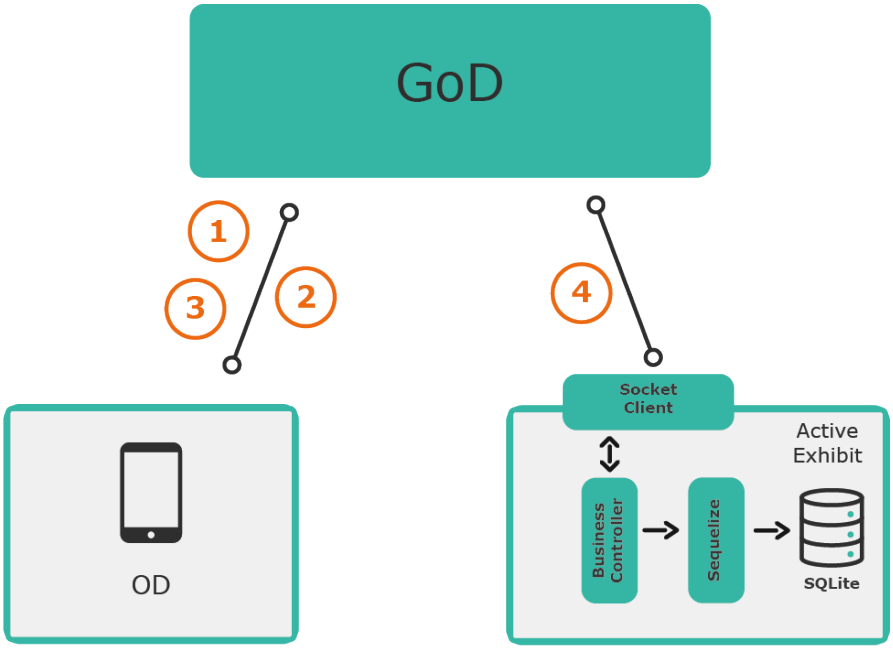


Figure 4.14. Schematic representation of the connection structure of the "notifyActiveExhibits".

The first exhibit of this type is the Legend Game of the Klosterneuburg exhibition. In this exhibit, the user has the opportunity to play the legend of Klosterneuburg. In the Legend Game, only one user can connect to the exhibit at a time. The game was developed with Unity and needs, as already mentioned, only the user information. If the user successfully recreated the legend, the exhibit server notifies GoD that the user unlocked a new part of the code of arms. GoD than tells the user that a new part was unlocked. The visitor has two possibilities two disconnect from the game. In the first approach, the user terminates the connection in the legend game. The exhibit server notifies GoD. GoD then resets the exhibit seat and reduces the current seat counter. Finally, GoD notifies the OD that it was disconnected from the exhibit.

The second exhibit is the genealogical tree visualization of the Babenberger family. This exhibit offers the user an interactive possibility to explore genealogy visualization. It provides additional information and a clearer presentation for the visitor. For this exhibit, a maximum of two users can interact at a time. This is because the exhibit has two tablets with which the users interact. The tablets update the visualization, which is projected with

a video projector. The basic behavior of the exhibit server is the same as in the Legend Game. In this exhibit, however, GoD additionally sends the position (left or right tablet) that the user has connected to.

4.6.2 activeExhibit

The exhibits of the type ”activeExhibit” need a direct connection to the exhibit server. Therefore, the connection procedure is slightly different. Figure 4.15 shows the behavior if a user is joining an active exhibit of this type. Also, with this type, the app checks first the status of the exhibit. Secondly, GoD is responding to the current status of the exhibit. If the exhibit has the status ”FREE” the OD can join the exhibit. Thirdly, the app is informing GoD that the OD wants to join the active exhibit. GoD updates the status of the exhibit. Fourthly, the app establishes a new socket connection with the exhibit server and sends it all relevant data. From that point on, all communication is directly between the OD and the exhibit. When the user wants to leave the exhibit, the user must do this in the app. The active exhibit is only terminating the connection when the app of the user is not responding anymore for 60 seconds. Therefore, the exhibit server sends a ping to the ODs at regular intervals, and if they do not respond within one minute, the server terminates the connection. Furthermore, the exhibit server informs GoD to update the status of the exhibit since the user forcefully left the exhibit.

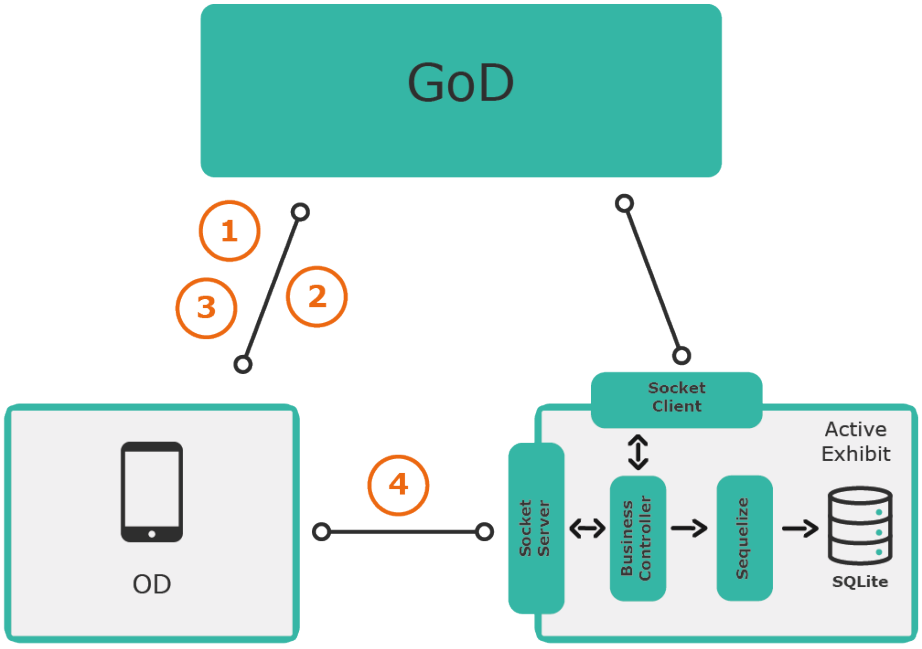


Figure 4.15. Schematic representation of the connection structure of the ’activeExhibits”.

The structure of the server is similar to the notify type with the only exception that the server also includes a Socket.io server like GoD. The notify server does not need a Socket.io server since the ODs do not need a direct connection with the exhibit. But this type needs a direct connection, and therefore it is necessary that the ”activeExhibit” types include a Socket.io server.

For the exhibition of Klosterneuburg, one active exhibition was implemented. The exhibit is a quiz with several questions about the facts of the exhibition. The exhibit permanently displays a question and its answer options on a screen. Each question has a timer, and when it has expired, the correct answer will be displayed on the screen. In addition, a visualization shows how often the users have selected which answer option. The question and answers are also displayed on the OD of the visitor. The user uses the app to answer the questions and then sees in the app if the answer was correct. On a second screen, there is also a high score list of all previously attended visitors.

5 Evaluation

So far this work presented, in addition to the similar work in the field of MDE and BYOD, several museums which offer interactive exhibitions. Furthermore, various technical possibilities and a concept for a MDE were presented. The last chapter presented a proof of concept for implementing an MDE with a focus on BYOD. The various areas such as server, app and exhibits were described in detail. This chapter covers the evaluation of the Klosterneuburg exhibition. The exhibition is running in the Klosterneuburg Abbey since March, and the evaluation phase is lasted from March to the end of July 2019. The evaluation focuses only on the technical functioning of the MDE and not on the user experience. This chapter includes the hardware specification of the computers used for the server and the exhibits. Furthermore, the chapter includes the bugs found in the evaluation period and their implemented solutions. Finally, the chapter presents some statistics about the exhibition, like the number of app downloads, the total number of registered users, and more.

5.1 Hardware

Of course, the different parts of the exhibition need computer components on which the applications can run. For the server, the exhibition uses a Hewlett Packard Enterprise (HPE) server. To be more specific an HPE ProLiant ML110 Gen10 tower server. The performance of this server is more than sufficient since it does not have to process thousands of requests at a time. The following list shows the most important hardware components of the server.

CPU: Intel(R) Xeon(R) Bronze 3106 CPU @ 1.70GHz

RAM: 32GiB DDR4 RAM

Storage: 2x 240GB SATA konfiguriert in RAID1

OS: Ubuntu 16.04

For the exhibit computers, we used different types of INTEL® NUC computers. For the exhibition parts which do not require great performance NUC7I5DNKE computers where

used. For example, we used these computers for the tablets of the genealogical tree. The hardware specifications are as follows:

CPU: Intel Core i5 3.5 GHz

RAM: 16GiB DDR4 RAM

Storage: 250GB M.2 SSD

OS: Windows 10

For the parts which need some more performance, we used the NUC7I7DNKE computers. These were used for the quiz or the genealogical tree visualization. The hardware specifications are mostly the same apart from a more powerful CPU.

CPU: Intel Core i7 4.2 GHz

RAM: 16GiB DDR4 RAM

Storage: 250GB M.2 SSD

OS: Windows 10

Finally, we needed a more powerful computer for the legend game. Since the legend game is the only exhibit which needs additional graphic performance, we used a NUC8I7HVK. The NUC5 and NUC7 only have an onboard graphics card, which means that they can display the standard Windows UI. However, they cannot be used for more graphically demanding applications.

CPU: Intel Core i7 4.2 GHz

RAM: 16GiB DDR4 RAM

GPU: Radeon RX Vega M GH graphics (8M Cache, up to 4.20 GHz)

Storage: 250GB M.2 SSD

OS: Windows 10

To automatically start the applications when the computer is starting, a batch file was created. The script file checks if the corresponding folder exists and if the node modules are installed. If that is the case, the script starts the exhibit servers and then the corresponding application e.g., the legend game.

5.2 Server Utilization

In section 4.4.10 it was described that the tool ”pm2-server-monit” is used to monitor the server load. Figure 4.12 showed an example output of this tool. The most important metrics for the application are the used CPU capacity, the needed memory, the Event Loop Latency, and the number of restarts. These metrics were monitored during the evaluation period. The server's event loop cannot handle multiple tasks at the same time. Instead, tasks are queued and then processed one at a time. This means that a single task can be stopped because the execution of the previous task takes a long time. The metric Event Loop Latency indicates the average waiting time.

However, the tool only allows the monitoring of the data in real-time. If an administrator wants to view a timeline of the collected data, he/she has to switch from the free version to a paid one. However, since a subscription would cost a lot, it was waived for this work. Instead, the load was observed in real-time when user tests were performed in the exhibition. Such user tests were conducted, for example, on the days 7th, 25th and 26th of June. As the next section will show, the highest number of user accounts were created on these days. While observing the server, the following metrics could be detected:

CPU capacity: The server never needed more than 20% of the CPU capacity. Normally the value is between 5% and 10%.

Memory: During the user tests, the application needed a maximum of 825MiB.

Event Loop Latency: The maximum latency value was about 6ms.

Restarts: Most of the bugs which caused a server restart were found in March and April. Therefore, there were no unwanted server reboots from mid-May onwards.

Throughout the evaluation period, only one server instance ran concurrently. Usually, several instances are running on a server at the same time so that the latency time can be kept as low as possible during a period with a high number of requests. A load balancer is used so that requests can be distributed as evenly as possible to the instances. PM2 also offers the possibility to run several instances of an application simultaneously and additionally offers an integrated load balancer. However, this poses a problem because it is necessary to ensure that the requests associated with a particular session ID are related to the process from which they originate. This is due to certain transports such as XHR polling or JSONP polling, which are based on multiple requests being fired during the lifetime of the socket. If sticky balancing is not activated, an error will occur.

This problem would not occur if only the ”Websocket” transport protocol was enabled for Socket.io. That is because the underlying TCP connection is kept open until the client or server decides to terminate it. However, as it may be that a client does not support

WebSockets, the transport should not be restricted. Therefore, we have to ensure that the client is always redirected to the same node, which is called sticky-session. A tool with which this can be accomplished is called NGINX[1].

Since there are now multiple Socket.IO nodes accepting connections, we need a way to broadcast events to everyone or specific ”rooms” between these processes or computers. For this purpose, the server can use the ”socket.io-redis” adapter[2]. ”Redis is an open-source (BSD licensed), in-memory data structure store, used as a database, cache and message broker” (redis.io 2019). Thus Redis is the interface in charge of routing messages.

5.3 Bugs

As described in section 4.4.10, the server management part includes a way to send the administrator an email if an error occurs. This section covers the most important bugs which arose in the evaluation period.

One of the first problems was the missing certificate authority (CA) file. At first, the server used only the certificate and the key file to create the HTTPS server. However, a client with only these two files cannot verify the authenticity of the certificate. This meant that even the Webviews of the native iOS and Android app did not want to establish a connection to the server because the certificate was untrustworthy. Adding the CA files resolved this issue.

Another problem arose when a user suddenly quit the app, or the app suddenly crashed. If that happened and the user was still connected to a ” notifyActiveExhibit ”, then the user remained connected to this exhibit without a time limit. The reason was that there was no direct connection between the OD and the exhibit, and therefore, there was no ping. However, one can solve this problem because when the connection between the OD and GoD is terminated, the server can respond to this ”disconnect” event. Since we store for each user his/her current location, the server can separate the user from the exhibit. The server knows which user it is because we added the token to the socket connection. The user object is stored in the token. GoD then forwards the disconnect message to the exhibit.
It is equally essential that the user is informed when an exhibit suddenly is unavailable. However, GoD does not automatically know which socket connection belongs to an OD and which to an exhibit. However, only user connections have a token in the socket connection. If the connection, therefore, has a token, it must be a user. In addition, GoD stores the

[1] https://www.nginx.com/
[2] https://github.com/socketio/socket.io-redis

socket id of the exhibits. So if no token is verifiable, it will be checked if the id belongs to an exhibit. If so, GoD changes the status to 'OFFLINE' and informs all connected users that the exhibit is no longer available.

One of the biggest problems was a bug in the socket middleware. Due to this bug, the socket events were partly received twice at the server. Among other things, this meant that the number of places for the exhibit was not increased by one; instead, it was increased by two. Therefore, users could only use one tablet at the same time for genealogy visualization. To prevent this, additional checks have been implemented for this functionality. For example, it is now additionally checked whether the current position of the user does not already correspond to the " onExhibit " position of the exhibit. If so, the second attempt to register the same position is ignored. Although discovering this problem took quite some time, it was quickly remedied. In the middleware, only a return statement had to be added.

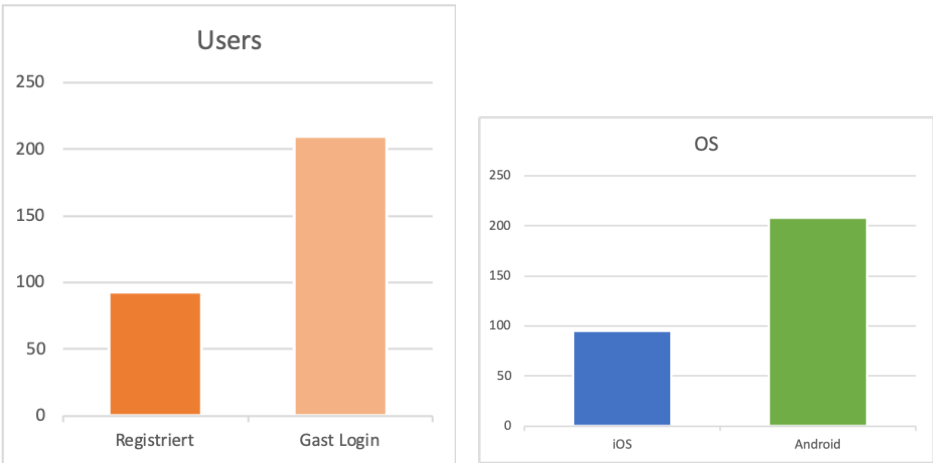
It was already mentioned that the socket id of the ODs is stored in the database. Therefore, if the exhibit disconnects the user, GoD needs to notify the OD. In the original setup, the socket id was only saved in the database at login. However, it often happened that the OD could not receive the disconnection message from GoD because the socket id had changed. This could happen if the user moved the app to the background or enabled the lock screen. To solve this problem, the socket id is saved again when the OD connects to the exhibit.

Another problem was the naming for the guest users. When a user registers as a guest, GoD automatically creates a name. The database stores a sequential number in the database that is used to create a unique username. The database requires that the username is unique. So GoD appends this sequential number to the name "Guest", e.g. "Guest123". Since a user can register with the name "Guest124", it could happen that GoD could not create a new user with the next guest number. As a result, the user could not register. To resolve this issue, a name check was implemented for the guest registry. If the name already exists in the database, then the serial number is increased by one, and the name is rechecked. That's what the server does until it can find a name that does not yet exist.

If the server had to be restarted due to an exception, then it happened from time to time that the active exhibits could no longer connect to the server. The reason for this was that although the connection could be restored the login event was not received. The solution was to have the exhibit server periodically resend the login until it finally receives an answer.

It should also be noted that in a broadcast, all clients receive the sent message. However, since clients can also connect to the server, which is not part of the app, broadcasts should be sent with caution. Although the connection itself is secure if it is an HTTPS server, this does not help if the client has also established a secure connection and then receives the

broadcast event. Therefore, no sensitive data should be sent in a broadcast event.



(a) Registered and guest users (b) Users split on iOS and Anroid

Figure 5.1. Statistic of the registered and guest users of the exhibition.

During the evaluation of user behavior in the project, it was found that it would be useful to implement additional logs. The logs should help to track how the visitors behave in the museum, how they move through the exhibition and which exhibits they look at. For this purpose, the activity logs and the regular logs were implemented. With the activity logs, one can evaluate the path of a user through the museum. The regular logs are used to record when a user logs in, automatically logs in or logs out. In addition, it is recorded whether he has answered correctly or incorrectly in the quiz and whether the visitor could find an AR target.

During the user tests that we completed during the project, we noticed several times that if Bluetooth beacons are too close to each other, the locations are triggered in the wrong order. An example of this would be the entrance of the exhibition where sections one and two partly overlap (see Figure 4.3). In order to prevent some exhibits from being triggered too early, the concept of the neighbors was introduced, which was explained in Section 4.5.

In general, during the evaluation period, it was found that the Bluetooth beacons are too inaccurate to determine the position of the visitors. In the exhibition of Klosterneuburg, we were lucky that the exhibition is like a path which the user has to follow. Therefore, we had no major problems with the beacon detection. However, at some places we had problems that beacons have partly overlaid. In addition, difficulties were discovered with the regard to the positioning of the beacons because persons or other objects can block the signals. All these factors lead to an inaccurate position detection of the system. This inaccuracy would be exacerbated if the exhibits were in a large room where the visitor can not easily follow a path.

5.4 Statistics

This section briefly shows some statistics about the exhibition for the evaluation period. For example, the app was downloaded 165 times for iOS and 276 times for Android. Figure 5.1(a) shows that in total, there were 93 registered users and 210 guest users. Figure 5.1(b) shows that 95 of these users were registered on an iOS device, and 208 were created on an Android device. Of the registered users, 32 were created on an iOS and 61 on an Android device. For the guest users, the ratio is 63 iOS users and 147 Android users. This shows very clearly that there were significantly more visitors with Android devices than iOS users.

Figure 5.2 visualizes the registered users per day. Since the exhibition is not visited very frequently, the visualization clearly shows that normally no more than five visitors are registered per day. There are only a few exceptions, but the clearest outliers are the days 7th of June with 23 users and the days 25th and 26th of June, each with 20 users. On these days e.g., school classes visited the museum to evaluate the app in the exhibition.

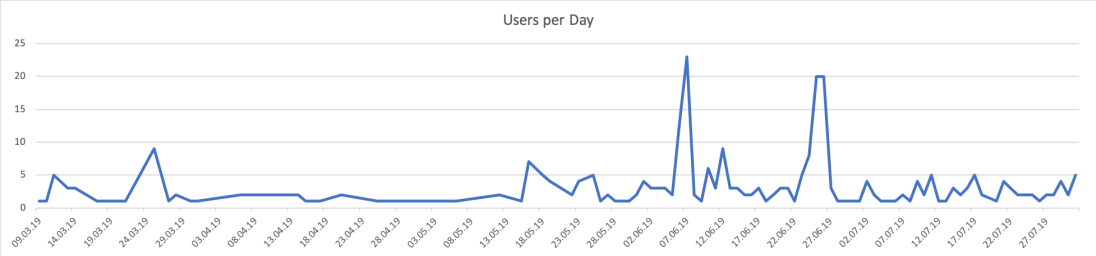


Figure 5.2. Statistic of the registered users per day.

Finally, figure 5.3 visualizes the unlocked and entered locations of the exhibition. Again, the statistics show very clearly that the users most often visited the quiz. In the app, the quiz was unlocked 306 times in the timeline and 512 times the detail view of the exhibit was opened. Of these users, 197 participated in the quiz.

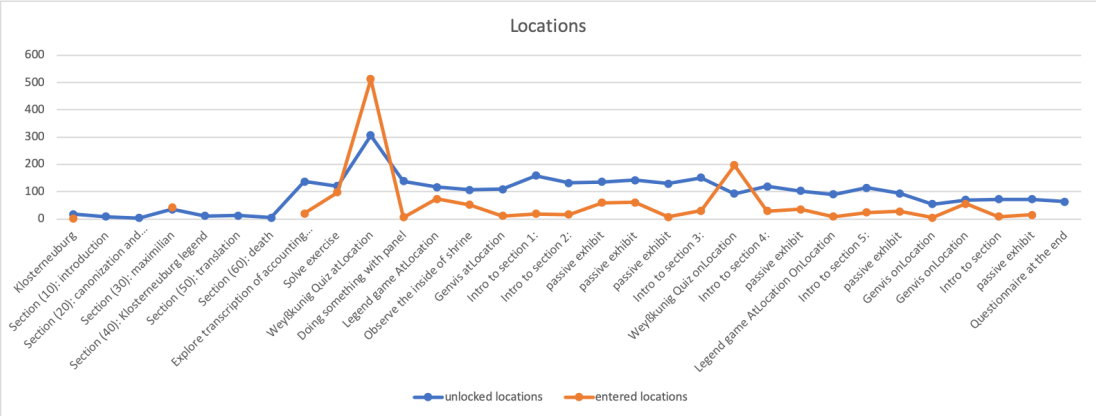


Figure 5.3. Statistic of the locations of the exhibition.

Furthermore, the statistic shows that the legend game was unlocked 116 times, and 90 users connected to the exhibit. Also, 109 users unlocked the genealogy visualization, and 56 also connected to the active exhibit. The most frequently initiated exhibit of the passive exhibits is the intro to section 1 and after that the exhibit with the title ”Miracle and Witnesses”.

5.5 Summary

This chapter presented the outcome of the evaluation period of the Klosterneuburg exhibition. In addition to the hardware used and the possibilities of server monitoring, the found bugs were presented. Finally, some statistics were presented which were collected during the evaluation phase. The evaluation showed that the used hardware for the exhibits and the server were sufficient. Furthermore, the used tools for the server utilization were sufficient for the general usage of the exhibition. However, with the free version of the tools no long-term statistics of the server metrics could be collected. Especially at the beginning of the exhibition, some bugs were found that hindered the operation of the exhibition. In particular, the bug with the double events was quite challenging to solve, but when the initial bugs were resolved, the exhibition ran quite stably.
As the statistics show, the majority of visitors prefer to use a guest account rather than a registered one. In the distribution of the operating system, Android has more than twice as many installations as Apple. Furthermore, for the active exhibits, the statistics have shown that especially the quiz is very popular among visitors.

6 Conclusion

This thesis presented technical concepts for multi-device ecologies in museums where the user can bring their own device (BYOD). Chapter 2 answered the questions ”Which interactive exhibits currently exist in museums?” and ”Which technical concepts are used?”. To that end, the chapter first described the meaning of the terms MDE and BYOD. Furthermore, the chapter presented similar research which deals with these topics and which provide possible technologies for usage in MDE setups. In addition, the chapter presented various different kinds of interactive exhibitions which are already implemented in museums and what technologies they use. The research showed that there are hardly any museums that offer an MDE and include the BYOD approach in their exhibitions. Furthermore, the museums mostly use active location triggering approaches like optical markers or the input of a code. Moreover, most museums use an app as a guide and do not offer an interaction with an exhibit. Hardly any museums offer any active exhibits. And if they do, then these exhibits are usually isolated and are not connected with other devices. If a museum offers an MDE, the interaction is usually only possible with preconfigured loan devices or with an ID card, which the visitor receives and with which he/she can identify himself/herself again at each station.

Chapter 3 presented the technical concept and key elements of the MEETeUX MDE. For some of the key elements the chapter showed different technical possibilities for the implementation in an MDE. The central part of the work was the implementation of an self-designed MDE with the focus on BYOD for the annual exhibition of Klosterneuburg. The exhibition consists of an app, a variety of active and passive exhibits, and a server that monitors everything. For each part of the exhibition, the concept and functionality were described in detail. In addition, it was described how to manage the server and how the CI / CD pipelines work which provide an easier and faster deployment method. Also, the concept of the database was explained and which technologies were used for the final use in the museum.
The final chapter described the evaluation phase of the exhibition. The chapter included the used hardware of the exhibits and the server. Finally, the found bugs and their solutions were described, and also some statistics of the exhibition were presented.

At the beginning of chapter 4 the key elements for the MDE of Klosterneuburg were presented. Each key element represents a research question that was answered in the Proof

of Concept chapter. The following section summarizes how each research question was answered:

Server How can one implement a server which always knows the status of the exhibition including all exhibits and the visitors ODs? More specifically, how can the server handle the communication between all devices and furthermore, decide if a user can currently interact with an exhibit or not?

The proof of concept chapter showed the implementation of the server. The MDE has been designed so that each event is transmitted to the server. Each time the OD receives a BLE beacon from an exhibit, the server is notified. Even if the user changes the language in the app or opens the detail view of an exhibit, the server will be notified. Therefore, the server stores all the data of a user that is necessary for the exhibition. The exhibits also tell the server when they are available and the server manages their status to inform the users. The use of the WebSocket technology allows the server to instantly detect when a user or exhibit loses connection to the server. Therefore, the server can update the status of the user or exhibit without the devices actively sending a message to the server. The bi-directional connection between all devices even allows the server to act as a relay and, for example, tell the OD if the user has pressed the logout button on the exhibit even if there is no direct connection between the two. In summary, the used technologies were a sufficient choice for the implementation of the server with the required specifications.

App How does an app have to be designed so that it offers the possibility to access the hardware of the device, offers a location awareness approach, provides a general user interface and includes the functionality to interact with the various exhibits?

As chapter 4 presented, the designed app consists of a special setup including a native iOS and Android app. The native part includes a webview in which an Angular project is displayed. The access to the hardware was also necessary because the location awareness approach was using the Bluetooth functionality of the smartphones. In order not to have to develop the general user interface twice for the visitors, we decided to create an Angular project for the app. Because the exhibits also used the WebSocket technology, the communication between the app and the exhibits was not a problem. In summary, the app could meet all previously defined requirements.

Guest support How can the MDE provide an option for visitors to use the app as a guest and still provide the user with the same functionality as the registered ones?

The integration of test accounts was quite simple and as the statistics showed the user were using it frequently. Since the visitors do not have to provide credentials, many users used guest accounts to explore the exhibition. Also the upgrade to a real

user account was not challenging. Furthermore, the server supports a middleware were specific WebSockets events can be only accessable for registered users.

Exhibits How can the MDE provide different types of exhibits and also include a possibility for the communication between the exhibit and the server or the app? Therefore, the communication protocol will not be limited to the TUIO protocol.

As the proof of concept showed the MDE supports three different types of exhibits, namely passive, interactive and active exhibits. However, the concept of the exhibition is designed so that the server supports a variety of exhibit types. For the communication between the exhibit and the server or the app the proof of concept is using the WebSocket technology. In the evaluation phase, it has been shown that the use of WebSockets works very well and also gives exhibits the ability to send data to the server or the ODs. Therefore, the use of TUIO was not necessary.

Device Communication Is the WebSocket technology a sufficient approach for bidirectional communication between all devices in a MDE?

The use of the WebSocket technology proved to be sufficient. As already mentioned this technology established multiple possibilities like broadcasting, disconnection alerts and a bidirectional communication between all devices. The implementation with the socket.io library was quite simple and even provides a backup mechanism for those devices which do not support the WebSocket technology.

Location Awareness Is the usage of Bluetooth and Bluetooth Low Energy beacons a sufficient approach for the location awareness?

As the evaluation phase has shown, the use of Bluetooth is not a sufficiently accurate method for the location awareness approach. At the end a lot of testing and configuration was needed to make the location awareness with Bluetooth work. Hence, it should be considered to change the location awareness approach in a future work.

Multi-Language Support How must the MDE be designed to give visitors the opportunity to experience the exhibition in different languages?

As the Proof of Concept showed the support for multiple languages must be implemented at three different parts. The text for the control elements is stored in the app itself and therefore, the switch between the languages is quite simple and fast. The contents of the exhibits are stored at the server and once the language is changed the contents of the new language will be loaded from the server. Finally, the texts for the interactive exhibits are stored in the exhibits themselves. If a user joins the exhibit with his/her OD the language will be transmitted from the OD to the exhibit and the exhibit changes the language.

6.1 Challenges

One of the biggest challenges is testing the different parts of the MDE. The difficulties start when testing the app. Unfortunately, since the app consists of a native and a web part, testing the app is not that easy. To check the complete functionality, both parts have to be tested together. In addition, the reception of the Bluetooth beacons only works if the app is installed on a real smartphone. There is no other choice than to test the app on smartphones manually.

In addition, the MDE consists of several devices that interact with each other. Therefore, it is necessary to test all parts of the MDE to ensure complete functionality. This means testing takes a long time. Even finding a bug can take much time and be tedious in some instances. If you look at the ”notifyActiveExhibit”, the app sends a message to GoD and GoD forwards it to the active exhibit. To find the bug, one has to check every part of this chain. An example of such a bug that took much time is the logout functionality which was partially not forwarded.

The other big challenge is location awareness. As already mentioned the use of BLE beacons often has an impairing inaccuracy. Therefore, a lot of testing and configuration was needed to provide the best location triggering possible. At some points of the exhibition, we needed to implement the neighbor functionality to provide a stable location triggering. Thus, changing the location awareness technology will be discussed in the next section.

Another challenge was the thick walls of the exhibition building. These led to the difficulty that the mobile phones repeatedly lost contact to the WLAN access points. For the regular use of the app, this was not a problem as GoD is also accessible through the Internet. However, the app then cannot interact with the active exhibits. If the user has just been connected to an active exhibit, the loss of connection with an access point results in a disconnection. In addition, at some points of the exhibition the visitors had no network access at all and therefore, no Internet access. As a result, visitors were sometimes unable to use the app at all.

6.2 Future Work

To sum up, the technical setup of the exhibition had some problems in the beginning, but since these problems were fixed the exhibition is running quite stably. As mentioned earlier, a big problem is the inaccuracy of beacon detection. Therefore, an important part of future work would be to revise the location awareness of the exhibition. One option would be a combination of ultrasound sounds and Bluetooth beacons, as described in section 3.4. By combining the two technologies, the accuracy could be significantly increased. An

alternative would be to use Bluetooth 5.1 as soon as it is available. The new standard includes a new feature called Direction Finding. This offers the ability to determine the direction of objects. Bluetooth thus becomes an improved navigation system. The distance to objects can be determined with the new standard down to a 1cm accuracy (Sebayang 2019).

In addition, in a future project, a more secure method for the login of the exhibit server should be implemented. Currently, the exhibits only use their IP address to authenticate themselves as an active exhibit. The server then stores the socket id to know to which client the messages should be sent. Thus a hacker would have to find out only the IP address of an exhibit to be able to log in with it. This would lead to two problems. Firstly, the real active exhibit would then be unreachable for the user. Second, the hacker now receives the user information intended for the active exhibit. Secondly, a more secure login option should be developed. One possibility would be to implement a proper login by username and password. After that, the exhibit, as well as the OD, could get a token with which all subsequent events must be authenticated. In this setup, the username and password would have to be stored in environment variables.

Bibliography

Loke, S. W. (2003, December 15). Service-oriented device ecology workflows. In *Service-oriented computing - ICSOC 2003* (pp. 559–574). International conference on service-oriented computing. Springer, Berlin, Heidelberg. doi:10.1007/978-3-540-24593-3_38

Dini, R., Paternò, F., & Santoro, C. (2007). An environment to support multi-user interaction and cooperation for improving museum visits through games. In *Proceedings of the 9th international conference on human computer interaction with mobile devices and services - MobileHCI '07* (pp. 515–521). The 9th international conference. Singapore: ACM Press. doi:10.1145/1377999.1378062

Lim, C., Wan, Y., Ng, B., & See, C. S. (2007, May). A real-time indoor WiFi localization system utilizing smart antennas. *IEEE Transactions on Consumer Electronics*, *53*(2), 618–622. doi:10.1109/TCE.2007.381737

Tesoriero, R., Gallud, J. A., Lozano, M., & Penichet, V. M. R. (2008, March). A location-aware system using RFID and mobile devices for art museums. In *Fourth international conference on autonomic and autonomous systems (ICAS'08)* (pp. 76–81). 2008 fourth international conference on autonomic and autonomous systems (ICAS). Gosier, Guadeloupe: IEEE. doi:10.1109/ICAS.2008.38

Echtler, F., Nestler, S., Dippon, A., & Klinker, G. (2009, November). Supporting casual interactions between board games on public tabletop displays and mobile devices. *Personal and Ubiquitous Computing*, *13*(8), 609–617. doi:10.1007/s00779-009-0246-3

Ghiani, G., Paternò, F., Santoro, C., & Spano, L. D. (2009, August). UbiCicero: A location-aware, multi-device museum guide. *Interacting with Computers*, *21*(4), 288–303. doi:10.1016/j.intcom.2009.06.001

Shirazi, A. S., Döring, T., Parvahan, P., Ahrens, B., & Schmidt, A. (2009). Poker surface: Combining a multi-touch table and mobile phones in interactive card games. In *Proceedings of the 11th international conference on human-computer interaction with mobile devices and services - MobileHCI '09* (p. 1). The 11th international conference. Bonn, Germany: ACM Press. doi:10.1145/1613858.1613945

Terrenghi, L., Quigley, A., & Dix, A. (2009, November 1). A taxonomy for and analysis of multi-person-display ecosystems. *Personal and Ubiquitous Computing*, *13*(8), 583–598. doi:10.1007/s00779-009-0244-5

Hardy, R., Rukzio, E., Holleis, P., & Wagner, M. (2010). Mobile interaction with static and dynamic NFC-based displays. In *Proceedings of the 12th international conference on human computer interaction with mobile devices and services - MobileHCI '10* (p. 123). The 12th international conference. Lisbon, Portugal: ACM Press. doi:10.1145/1851600.1851623

Kray, C., Nesbitt, D., Dawson, J., & Rohs, M. (2010). User-defined gestures for connecting mobile phones, public displays, and tabletops. In *Proceedings of the 12th international conference on human computer interaction with mobile devices and services - MobileHCI '10* (p. 239). The 12th international conference. Lisbon, Portugal: ACM Press. doi:10.1145/1851600.1851640

Broll, G., Reithmeier, W., Holleis, P., & Wagner, M. (2011). Design and evaluation of techniques for mobile interaction with dynamic NFC-displays. In *Proceedings of the fifth international conference on tangible, embedded, and embodied interaction - TEI '11* (p. 205). The fifth international conference. Funchal, Portugal: ACM Press. doi:10.1145/1935701.1935743

Wang, R., Zhao, F., Luo, H., Lu, B., & Lu, T. (2011). Fusion of wi-fi and bluetooth for indoor localization. In *Proceedings of the 1st international workshop on mobile location-based service* (pp. 63–66). MLBS '11. New York, NY, USA: ACM. doi:10.1145/2025876.2025889

Seyed, T., Burns, C., Costa Sousa, M., Maurer, F., & Tang, A. (2012, November 11). Eliciting usable gestures for multi-display environments. (pp. 41–50). Proceedings of the 2012 ACM international conference on interactive tabletops and surfaces. ACM. doi:10.1145/2396636.2396643

Boring, S. & Baur, D. (2013, March). Making public displays interactive everywhere. *IEEE Computer Graphics and Applications*, *33*(2), 28–36. doi:10.1109/MCG.2012.127

Rittenbruch, M. (2013). Supporting collaboration in large-scale multi-user, 8.

Bellucci, A., Malizia, A., & Aedo, I. (2014, January 1). Light on horizontal interactive surfaces: Input space for tabletop computing. *ACM Computing Surveys*, *46*(3), 1–42. doi:10.1145/2500467

Dang, C. T. & André, E. (2014, June 17). A framework for the development of multi-display environment applications supporting interactive real-time portals. (pp. 45–54). Proceedings of the 2014 ACM SIGCHI symposium on engineering interactive computing systems. ACM. doi:10.1145/2607023.2607038

Huang, D.-Y., Chen, S.-C., Chang, L.-E., Chen, P.-S., Yeh, Y.-T., & Hung, Y.-P. (2014, July). I-m-cave: An interactive tabletop system for virtually touring mogao caves. In *2014 IEEE international conference on multimedia and expo (ICME)* (pp. 1–6). 2014 IEEE international conference on multimedia and expo (ICME). Chengdu, China: IEEE. doi:10.1109/ICME.2014.6890233

Murata, S., Yara, C., Kaneta, K., Ioroi, S., & Tanaka, H. (2014, September). Accurate indoor positioning system using near-ultrasonic sound from a smartphone. In *2014 eighth*

international conference on next generation mobile apps, services and technologies (pp. 13–18). 2014 eighth international conference on next generation mobile apps, services and technologies. doi:10.1109/NGMAST.2014.17

Winkler, C., Löchtefeld, M., Dobbelstein, D., Krüger, A., & Rukzio, E. (2014). SurfacePhone: A mobile projection device for single- and multiuser everywhere tabletop interaction. In *Proceedings of the 32nd annual ACM conference on human factors in computing systems - CHI '14* (pp. 3513–3522). The 32nd annual ACM conference. Toronto, Ontario, Canada: ACM Press. doi:10.1145/2556288.2557075

Lazik, P., Rajagopal, N., Shih, O., Sinopoli, B., & Rowe, A. (2015). ALPS: A bluetooth and ultrasound platform for mapping and localization. In *Proceedings of the 13th ACM conference on embedded networked sensor systems* (pp. 73–84). SenSys '15. event-place: Seoul, South Korea. New York, NY, USA: ACM. doi:10.1145/2809695.2809727

Seyed, T., Azazi, A., Chan, E., Wang, Y., & Maurer, F. (2015, November 15). SoD-toolkit: A toolkit for interactively prototyping and developing multi-sensor, multi-device environments. (pp. 171–180). Proceedings of the 2015 international conference on interactive tabletops & surfaces. ACM. doi:10.1145/2817721.2817750

Vepsäläinen, J., Di Rienzo, A., Nelimarkka, M., Ojala, J. A., Savolainen, P., Kuikkaniemi, K., … Jacucci, G. (2015, November 15). Personal device as a controller for interactive surfaces: Usability and utility of different connection methods. (pp. 201–204). Proceedings of the 2015 international conference on interactive tabletops & surfaces. ACM. doi:10.1145/2817721.2817745

Falk, J. H., Dierking, L. D., & Dierking, L. D. (2016). *The Museum Experience Revisited*. Routledge. doi:10.4324/9781315417851

Grubert, J., Kranz, M., & Quigley, A. (2016, December 1). Challenges in mobile multi-device ecosystems. *mUX: The Journal of Mobile User Experience*, *5*(1), 5. doi:10.1186/s13678-016-0007-y

Koukoulis, K. & Koukopoulos, D. (2016, October 31). Towards the design of a user-friendly and trustworthy mobile system for museums. In *Digital heritage. progress in cultural heritage: Documentation, preservation, and protection* (pp. 792–802). Euro-mediterranean conference. Springer, Cham. doi:10.1007/978-3-319-48496-9_63

Blumenstein, K., Kaltenbrunner, M., Seidl, M., Breban, L., Thür, N., & Aigner, W. (2017). Bringing Your Own Device into Multi-device Ecologies: A Technical Concept. In *Proceedings of the 2017 ACM International Conference on Interactive Surfaces and Spaces* (pp. 306–311). ISS '17. New York, NY, USA: ACM. doi:10.1145/3132272.3132279

Cecchinato, M. E., Cox, A. L., & Bird, J. (2017, May 2). Always on(line)?: User experience of smartwatches and their role within multi-device ecologies. (pp. 3557–3568). Proceedings of the 2017 CHI conference on human factors in computing systems. ACM. doi:10.1145/3025453.3025538

Nishiyama, T., Mochizuki, M., Murao, K., & Nishio, N. (2017). Hybrid approach for reliable floor recognition method. In *Proceedings of the 2017 ACM international joint conference on pervasive and ubiquitous computing and proceedings of the 2017 ACM international symposium on wearable computers* (pp. 569–576). UbiComp '17. New York, NY, USA: ACM. doi:10.1145/3123024.3124404

Petrelli, D., Marshall, M. T., O'brien, S., Mcentaggart, P., & Gwilt, I. (2017, April 1). Tangible data souvenirs as a bridge between a physical museum visit and online digital experience. *Personal and Ubiquitous Computing*, *21*(2), 281–295. doi:10.1007/s00779-016-0993-x

Urano, K., Kaji, K., Hiroi, K., & Kawaguchi, N. (2017). A location estimation method using mobile BLE tags with tandem scanners. In *Proceedings of the 2017 ACM international joint conference on pervasive and ubiquitous computing and proceedings of the 2017 ACM international symposium on wearable computers* (pp. 577–586). UbiComp '17. New York, NY, USA: ACM. doi:10.1145/3123024.3124405

Kosmopoulos, D. & Styliaras, G. (2018, July 1). A survey on developing personalized content services in museums. *Pervasive and Mobile Computing*, *47*, 54–77. doi:10.1016/j.pmcj.2018.05.002

Koukopoulos, D. & Koukoulis, K. (2018, January 1). A trustworthy system with mobile services facilitating the everyday life of a museum. *International Journal of Ambient Computing and Intelligence (IJACI)*, *9*(1), 1–18. doi:10.4018/IJACI.2018010101

Othman, M. K., Idris, K. I., Aman, S., & Talwar, P. (2018). An empirical study of visitors' experience at kuching orchid garden with mobile guide application [Advances in human-computer interaction]. doi:10.1155/2018/5740520

Park, S., Gebhardt, C., Rädle, R., Feit, A. M., Vrzakova, H., Dayama, N. R., ... Hilliges, O. (2018). AdaM: Adapting multi-user interfaces for collaborative environments in real-time. In *Proceedings of the 2018 CHI conference on human factors in computing systems* (184:1–184:14). CHI '18. event-place: Montreal QC, Canada. New York, NY, USA: ACM. doi:10.1145/3173574.3173758

Vilkomir, S. (2018, June 1). Multi-device coverage testing of mobile applications. *Software Quality Journal*, *26*(2), 197–215. doi:10.1007/s11219-017-9357-7

Wiener Gewässer. (2018). "Wiener Wasserweg"-App als virtueller Tour-Guide für die Alte Donau. Retrieved September 3, 2019, from https://www.wien.gv.at/umwelt/gewaesser/alte-donau/life/massnahmen/wasserweg.html

Android. (2019). Data and file storage overview. Retrieved August 1, 2019, from https://developer.android.com/guide/topics/data/data-storage

Angular.io. (2019). Angular - lifecycle hooks. Retrieved August 2, 2019, from https://angular.io/guide/lifecycle-hooks

Apple. (2019). Keychain services. Retrieved August 1, 2019, from https://developer.apple.com/documentation/security/keychain_services

auth0. (2019). JWT.IO. Retrieved July 25, 2019, from http://jwt.io/

Beatty, M. (2019). Dotenv [Npm]. Retrieved July 25, 2019, from https://www.npmjs.com/package/dotenv

Blumenstein, K., Breban, L., Taucher, C., Thür, N., & Seidl, M. (2019). Museums-Apps & Installationen - MEETeUX Projekt. Retrieved from http://meeteux.fhstp.ac.at/wp-content/uploads/2017/06/Recherche_MuseumsInstallationen_20170426.pdf

bmw-welt.com. (2019). BMW Museum App. Retrieved September 3, 2019, from https://www.bmw-welt.com:443/content/grpw/websites/bmw-welt_com/de/experience/popups/bmw-museum-app.html

chaijs.com. (2019). Chai. Retrieved July 25, 2019, from https://www.chaijs.com/

congstar.de. (2019). NFC - was ist das? was kann ich damit machen? | congstar. Retrieved August 11, 2019, from https://www.congstar.de/handys/technik-news-trends/nfc/

Depold, S. (2019). Sequelize. Retrieved July 23, 2019, from http://docs.sequelizejs.com/

Deutsches Museum. (2019). Deutsches Museum App. Retrieved September 4, 2019, from https://www.deutsches-museum.de/angebote/app/

Deutsches Technikmuseum. (2019). Deutsches Technikmuseum - App. Retrieved September 3, 2019, from https://sdtb.de/technikmuseum/angebote-bildung/2502/

gotlandsmuseum.se. (2019). The Gotland Museum. Retrieved September 3, 2019, from https://www.gotlandsmuseum.se/en/

Hallein, K. (2019). Der sprechende Kelte. Retrieved September 2, 2019, from https://play.google.com/store?hl=de

Hunsrück-Hochwald, N. (2019). Nationalpark-Ausstellung. Retrieved September 3, 2019, from https://www.nationalpark-hunsrueck-hochwald.de/besucher/erleben-angebote/nationalpark-ausstellung.html

Ideum. (2019). Hamline University Mississippi Multimedia Table. Retrieved September 2, 2019, from https://archive.ideum.com/creative-services/hamline-university-mississippi-multimedia-table/

Kaltenbrunner, M. (2019). TUIO. Retrieved September 2, 2019, from https://www.tuio.org/

Kennedy Space Center. (2019). Kennedy Space Center Official Guide. Retrieved September 3, 2019, from https://www.kennedyspacecenter.com/info/kennedy-space-center-official-guide

Kontakt.io. (2019). Ble beacons. Retrieved July 23, 2019, from https://kontakt.io/

Maritime Museum. (2019). Maritime Museum App. Retrieved September 3, 2019, from https://www.maritiemmuseum.nl/maritime-museum-app

MEETeUX. (2019). MEETeUX. Retrieved August 8, 2019, from https://meeteux.fhstp.ac.at/

mochajs.org. (2019). Mocha - the fun, simple, flexible JavaScript test framework. Retrieved July 25, 2019, from https://mochajs.org/

Museum Tour Guides. (2019). Natural History Museum App Guide. Retrieved September 3, 2019, from https://www.museumtourguides.com/home/product/natural-history-museum-app-guide/

Paternò, F. (2019, April 4). Concepts and design space for a better understanding of multi-device user interfaces. *Universal Access in the Information Society*, 1–24. doi:10.1007/s10209-019-00650-5

redis.io. (2019). Redis. Retrieved August 7, 2019, from https://redis.io/

Robbins, C. (2019). Winston [Npm]. Retrieved July 25, 2019, from https://www.npmjs.com/package/winston

RxJS. (2019). Retrieved August 2, 2019, from https://rxjs-dev.firebaseapp.com/

Sánchez-Adame, L. M., Mendoza, S., Viveros, A. M., & Rodríguez, J. (2019, July 26). Towards a set of design guidelines for multi-device experience. In *Human-computer interaction. perspectives on design* (pp. 210–223). International conference on human-computer interaction. Springer, Cham. doi:10.1007/978-3-030-22646-6_15

Sebayang, A. (2019). Bluetooth 5.1 Direction Finding: Bluetooth bekommt eine Richtungssuche [Golem.de]. Retrieved August 7, 2019, from https://www.golem.de/news/bluetooth-5-1-direction-finding-bluetooth-bekommt-eine-richtungssuche-1901-139030.html

Socket.IO. (2019). Socket.IO [Socket.IO]. Retrieved July 23, 2019, from https://socket.io/index.html

SQLite.org. (2019). SQLite home page. Retrieved August 6, 2019, from https://www.sqlite.org/index.html

StrongLoop, I. (2019). Express - Node.js-Framework von Webanwendungen. Retrieved July 25, 2019, from https://expressjs.com/de/

tagnology.com. (2019). WAS IST RFID? Retrieved August 11, 2019, from http://www.tagnology.com/rfid/was-ist-rfid.html

Tristan Interactive. (2019). Canadian Museum for Human Rights. Retrieved September 2, 2019, from https://apps.apple.com/us/app/canadian-museum-for-human-rights/id916923574

wels.at. (2019). Helden-der-Roemerzeit. Retrieved September 3, 2019, from https://www.wels.at/welsmarketing/tourismus/sightseeing/helden-der-roemerzeit.html

Wiggins, A. (2019). The twelve-factor app. Retrieved July 25, 2019, from https://12factor.net/config

List of Figures

List of Tables

Listings