

# Implementing a Crowd-Sourced Picture Archive for Bad Harzburg

Rieke Freund, Jan Philip Rättsch, Franziska Hradilak, Benedikt Vidic, Oliver Heß, Nils Lißner, Hendrik Wölert, Jens Lincke, Tom Beckmann, Robert Hirschfeld

**Technische Berichte Nr. 149**

des Hasso-Plattner-Instituts für  
Digital Engineering an der Universität Potsdam





Technische Berichte des Hasso-Plattner-Instituts für  
Digital Engineering an der Universität Potsdam



Technische Berichte des Hasso-Plattner-Instituts für  
Digital Engineering an der Universität Potsdam | 149

Rieke Freund | Jan Philip Rättsch | Franziska Hradilak  
Benedikt Vidic | Oliver Heß | Nils Lißner | Hendrik Wölert  
Jens Lincke | Tom Beckmann | Robert Hirschfeld

## **Implementing a Crowd-Sourced Picture Archive for Bad Harzburg**

Universitätsverlag Potsdam

**Bibliografische Information der Deutschen Nationalbibliothek**

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.dnb.de/> abrufbar

**Universitätsverlag Potsdam 2023**

<http://verlag.ub.uni-potsdam.de/>

Am Neuen Palais 10, 14469 Potsdam

Tel.: +49 (0)331 977 2533 / Fax: 2292

E-Mail: [verlag@uni-potsdam.de](mailto:verlag@uni-potsdam.de)

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam.

ISSN (print) 1613-5652

ISSN (online) 2191-1665

Das Manuskript ist urheberrechtlich geschützt.

Druck: docupoint GmbH Magdeburg

**ISBN 978-3-86956-545-3**

Zugleich online veröffentlicht auf dem Publikationsserver der Universität Potsdam:

<https://doi.org/10.25932/publishup-56029>

<https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-560291>

# Abstract

Pictures are a medium that helps make the past tangible and preserve memories. Without context, they are not able to do so. Pictures are brought to life by their associated stories. However, the older pictures become, the fewer contemporary witnesses can tell these stories. Especially for large, analog picture archives, knowledge and memories are spread over many people. This creates several challenges: First, the pictures must be digitized to save them from decaying and make them available to the public. Since a simple listing of all the pictures is confusing, the pictures should be structured accessibly. Second, known information that makes the stories vivid needs to be added to the pictures. Users should get the opportunity to contribute their knowledge and memories. To make this usable for all interested parties, even for older, less technophile generations, the interface should be intuitive and error-tolerant. The resulting requirements are not covered in their entirety by any existing software solution without losing the intuitive interface or the scalability of the system.

Therefore, we have developed our digital picture archive within the scope of a bachelor project in cooperation with the Bad Harzburg-Stiftung. For the implementation of this web application, we use the UI framework React in the frontend, which communicates via a GraphQL interface with the Content Management System Strapi in the backend. The use of this system enables our project partner to create an efficient process from scanning analog pictures to presenting them to visitors in an organized and annotated way. To customize the solution for both picture delivery and information contribution for our target group, we designed prototypes and evaluated them with people from Bad Harzburg. This helped us gain valuable insights into our system's usability and future challenges as well as requirements. Our web application is already being used daily by our project partner. During the project, we still came up with numerous ideas for additional features to further support the exchange of knowledge.





# Zusammenfassung

Bilder können dabei helfen, die Vergangenheit greifbar zu machen und Erinnerungen zu bewahren, doch alleinstehende Bilder ohne Kontext erreichen das nur schwer. Der große Wert besteht in den Geschichten, die mit den Bildern verbunden sind. Je älter die Bilder jedoch werden, desto weniger Zeitzeugen können von diesen Geschichten berichten. Besonders für große analoge Bildarchive, bei denen sich das Wissen und die Erinnerungen auf viele Personen verteilen, entstehen dadurch verschiedene Herausforderungen: Zunächst müssen die Bilder digitalisiert werden, um sie vor dem Zerfall zu schützen und um sie der Öffentlichkeit zugänglich machen zu können. Da eine einfache Aufreihung aller Bilder unübersichtlich ist, sollten die Bilder in eine zugängliche Struktur gebracht werden. Des Weiteren müssen zu den Bildern bekannte Informationen, aus denen ihre Geschichten erfahrbar werden, hinzugefügt werden. Nutzende sollen die Möglichkeit haben, eigenes Wissen und Erinnerungen beizutragen. Um dies für alle Interessierten, auch für ältere, evtl. wenig technikaffine Personen, nutzbar zu machen, sollte die Oberfläche eine intuitive und fehlertolerante Nutzung ermöglichen. Die sich daraus ergebenden Anforderungen werden von keiner existierenden Softwarelösung im Gesamten abgedeckt, ohne die intuitive Oberfläche oder die Skalierbarkeit des Systems zu verlieren.

Daher haben wir im Rahmen eines Bachelorprojekts in Zusammenarbeit mit der Bad Harzburg-Stiftung ein eigenes digitales Bildarchiv entwickelt. Für die Umsetzung dieser Webapplikation nutzen wir das UI-Framework React im Frontend, welches über eine GraphQL-Schnittstelle mit dem Content Management System Strapi im Backend kommuniziert. Die Nutzung dieses Systems ermöglicht unserem Projektpartner einen effizienten Prozess vom Scannen der analogen Bilder bis zum geordneten und annotierten Darstellen für Besuchende. Um die Lösung sowohl für das Bereitstellen der Bilder als auch für das Beitragen von Informationen auf unsere Zielgruppe zuzuschneiden, haben wir Prototypen entworfen und mit Menschen aus Bad Harzburg getestet, um ihre Eindrücke auszuwerten. Mit diesen konnten wir wertvolle Erkenntnisse über die Nutzbarkeit und noch offene Herausforderungen und Anforderungen gewinnen. Unsere Webanwendung ist bei unserem Projektpartner bereits im täglichen Einsatz. Trotzdem haben wir während des Projekts noch zahlreiche Ideen für zusätzliche Funktionen erarbeitet, um den Wissensaustausch weiter zu fördern.



# Contents

<b>1</b>	<b>A Crowd-Sourced Picture Archive</b>	<b>1</b>
1.1	The Cultural Heritage Domain . . . . .	1
1.2	Crowd-Sourcing in the Cultural Heritage Domain . . . . .	2
1.3	Related Work . . . . .	4
1.4	Our Project Partner: Bad Harzburg-Stiftung . . . . .	15
1.5	Our Project Prerequisites . . . . .	16
1.6	Summary . . . . .	25
<b>2</b>	<b>Functional Constraints and Requirements</b>	<b>27</b>
2.1	Initial Project Partner Requirements . . . . .	27
2.2	Picture Management Applications . . . . .	29
2.3	Application Selection . . . . .	30
2.4	Concept for a Collaborative Picture Archive . . . . .	43
2.5	Summary . . . . .	48
<b>3</b>	<b>Evaluating Design Decisions regarding Usability</b>	<b>51</b>
3.1	Design and Implementation . . . . .	51
3.2	Empirical Evaluation . . . . .	59
3.3	Discussion . . . . .	69
3.4	Conclusion and Outlook . . . . .	71
<b>4</b>	<b>Architecture and Implementation of a Web-Based Frontend</b>	<b>73</b>
4.1	Technical Components . . . . .	73
4.2	Background . . . . .	73
4.3	Frontend Architecture . . . . .	78
4.4	Data Flow . . . . .	86
4.5	Summary . . . . .	94
<b>5</b>	<b>Strapi as a Customizable Content Management System</b>	<b>95</b>
5.1	Motivation for Using a Content Management System . . . . .	95
5.2	Introduction to Content Management Systems . . . . .	95
5.3	Introduction to Plugin Architectures . . . . .	96
5.4	Strapi as a Content Management System . . . . .	97
5.5	Customizing Strapi . . . . .	102
5.6	Discussion . . . . .	106
5.7	Summary . . . . .	115

<b>6</b>	<b>Automatic Data Migration, Testing, and Deployment</b>	<b>117</b>
6.1	Tools and Tasks in Modern Web Development . . . . .	117
6.2	Data Migration: Importing from WordPress . . . . .	118
6.3	Data Migration: Evolving the System . . . . .	124
6.4	Developer Experience . . . . .	129
6.5	Integration, Deployment, and Performance . . . . .	132
6.6	Summary . . . . .	141
<b>7</b>	<b>Low Fidelity Prototypes to Explore the Design Space</b>	<b>143</b>
7.1	Introduction . . . . .	143
7.2	Concepts . . . . .	144
7.3	Methodology . . . . .	146
7.4	User Test 1: Map vs. Timeline . . . . .	149
7.5	User Test 2: Paper Prototype for a New Feature: Stories . . . . .	156
7.6	Summary . . . . .	162
<b>8</b>	<b>Summary</b>	<b>163</b>
<b>Appendices</b>		
<b>A</b>	<b>Appendix</b>	<b>167</b>
A.1	Evaluation . . . . .	167
A.2	Queries and Filters . . . . .	170
A.3	GraphQL and Monitoring . . . . .	174
A.4	Questionnaires . . . . .	177

# 1 A Crowd-Sourced Picture Archive

In this chapter, we introduce the cultural heritage domain and the concept of “crowd-sourcing”. Following this, we examine five cultural heritage platforms with different participation levels and present our partner as well as the prerequisites for this project. Finally, we give a brief overview of our project’s implementation and the following chapters.

## 1.1 The Cultural Heritage Domain

Our project collaborated with a local community foundation to save 50,000 archived pictures from slow decay and preserve their context to make the archive useful and valuable as an accessible means to discover one’s cultural heritage.

From today’s perspective, we as living human beings can look back on a history of mankind that has lasted for several millennia. Even if we were not alive for most of that time, we can learn a lot about past events, ways of life, intellectual currents, world views, differences and similarities from today. We owe this knowledge mainly to the traces that people have left behind, as well as to the people who discover, recover, conserve, investigate, and process them. Various institutions and places retain these traces. Galleries, libraries, archives, museums, historical societies, educational institutions, monuments, memorials, squares, parks, etc. work on for the conservation and distribution of historical knowledge and cultural heritage. Just as diverse as the places where cultural heritage is preserved and knowledge is passed on are the means. For this project, we will focus on pictures.

We as a society should conserve the existing photographs as historical sources and cultural heritage. In 1826 Nicéphore Niépce took the first photograph on a pewter plate [23]. However, the concept for the camera obscura was developed hundreds of years earlier. In the following 200 years, photography has made enormous technical progress. Nowadays, a standard smartphone is enough to take a picture. In 2021, 62.61 million people in Germany were using smartphones [33]. This number has been growing steadily over the last ten years. Almost anyone can take photos without film, a darkroom, or expensive camera equipment today. This has also significantly changed people’s relationship to photos. Hardly anything happens without being photographed. In the past, when photography was more elaborate and expensive, fewer photos were taken. Digitalization can help us to conserve our history online, so our cultural heritage is protected from decay and accessible everywhere. During pandemic-related lockdowns, most of the locations mentioned above were not accessible, but thanks to the internet, they no longer have to rely solely on physical locations to distribute their content. A lot of institutions have been presenting them-

selves online for years on their websites or their social media channels. Many have been involved in digital participation before the COVID-19 pandemic. Some libraries or museums, for example, digitize their collection and make it available to everyone or certain user groups on the internet.

Oftentimes, archives of pictures come with very little context. Fortunately, for years back to the early 1900s, we have people still alive who were there. That is why we consider crowd-sourcing as an approach to save that context. As such, there is a great value in the picture archive, at least for the region, which we would like to preserve with our project. In addition, we would like to gather and share the knowledge of those who lived during the times the photos were taken.

This chapter continues with a definition of the term “crowd-sourcing” and two models classifying the participation and collaboration level for cultural heritage platforms. After that, we take a look at five existing platforms to share historical pictures and knowledge. Then we introduce our project partner, the Bad Harzburg-Stiftung, before we delineate the prerequisites of the project, such as involved parties and the legacy system. Finally, we outline some of our work from the last ten months and give an overview of the further chapters.

## 1.2 Crowd-Sourcing in the Cultural Heritage Domain

We have just presented why we should preserve our cultural heritage and history. We have also already mentioned our project including many photos that lack information and crowd-sourcing as a possible way to collect knowledge. Based on this motivation, we take a closer look at crowdsourcing in the following.

### 1.2.1 Definition of Crowd-Sourcing

The term “crowd-sourcing”<sup>1</sup> is a composition of the words “crowd” and “outsourcing”. “Crowd” describes “a large number of people gathered together” according to the Oxford Dictionary. In the case of crowd-sourcing, “gathered together” is mainly realized online but not because it is limited to online cooperation. “Outsourcing” means to “obtain (goods or a service) from an outside or foreign supplier, especially in place of an internal source” [36]. In the context of outsourcing, the first “goods or a service” that come to mind may be related to industrial production and factories, but the crowd can bring in financial resources, creative solutions for specified problems, or completion of concrete tasks.

In 2006 Jeff Howe used the term in the article “The Rise of Crowdsourcing” [26]. He characterizes it as the process of companies using the labor force of individuals online for various tasks and points out the similarity of decreasing working costs between this and outsourcing labor to third-world countries.

---

<sup>1</sup>There are variant spellings like “crowdsourcing”, but we will refer to “crowd-sourcing” in the following. In quotations, we preserve the original form.

In the article, Howe refers to the commercial crowd-sourcing platform *Amazon Mechanical Turk*<sup>2</sup>, where so-called Requesters assign Turkers to perform certain tasks and pay them in return. He strongly focuses on the aspect of outsourcing. This might be due to a lack of noncommercial, volunteer-driven projects serving the common good at this time or because of the high socio-economic significance of this process. Well-known crowd-sourcing projects are *OpenStreetMap*<sup>3</sup> and *Wikipedia*.<sup>4</sup>

### 1.2.2 Models of Crowd-Sourcing and Participation in the Cultural Heritage Domain

In their paper “Crowdsourcing in the Cultural Heritage Domain: Opportunities and Challenges”, Oomen and Aroyo distinguish the following six types of crowd-sourcing initiatives which are depicted in Table 1.1 [47]:

**Table 1.1:** Types of crowd-sourcing initiatives

Crowd-Sourcing Type	Short Definition
Correction and Transcription Tasks	Inviting users to correct and/or transcribe outputs of digitization processes
Contextualization	Adding contextual knowledge to objects, e.g. by telling stories or writing articles/wiki pages with contextual data
Completing Collection	Active pursuit of additional objects to be included in a (web)exhibit or collection
Classification	Gathering descriptive metadata related to objects in a collection (Social tagging is a well-known example.)
Co-Curation	Using inspiration/expertise of non-professional curators to create (web)exhibits
Crowdfunding	Collective cooperation of people who pool their money and other resources together to support efforts initiated by others

Before we take a look at these specific types of crowd-sourcing initiatives as well as whether and to what extent they are suitable for our project (see subsection 1.5.4), we present another possible approach to classifications in the context of participating in cultural heritage.

In “The Participatory Museum” Simon presents a different concept of public participation [54]. She builds on the existing Public Participation in Scientific Research model containing the following three kinds of projects:

<sup>2</sup><https://www.mturk.com/> (last accessed: 2022-07-13).

<sup>3</sup><https://www.openstreetmap.org/> (last accessed: 2022-07-13).

<sup>4</sup>[https://en.wikipedia.org/wiki/Main\\_Page](https://en.wikipedia.org/wiki/Main_Page) (last accessed: 2022-07-13).

1. **Contributory projects:** An institution conducting a contributory project asks its participants for a specific type of content. During the whole project, the institution takes control of the process.
2. **Collaborative projects:** A collaborative project is based on a partnership and cooperation between participants and institutional staff. The institution is still in control of the project and may design a concept or give ideas.
3. **Co-creative projects:** Co-creative projects are characterized by the fact that from the very beginning the participants cooperate with the institution as equal partners. Both parties can contribute ideas and project ambitions.

Simon extends these with a fourth project type:

4. **Hosted projects:** In a hosted project the institution provides participants with capabilities such as space, data, tools, or other means for their project. While the institution restrains itself, the participants are paramount.

### 1.3 Related Work

In general, there are endless ways to share photos in the digital world. However, we would like to share not only the pictures but also knowledge. We need a platform to share historical photos and exchange background information and views about them. Thus, we looked at various systems implementing different ways of crowd-sourcing in the cultural heritage domain and will present a selection of five in ascending order of their level of participation and user engagement. Just as we can only present a small subset of many existing systems in this limited scope, we cannot address every functionality or concept. We do not go into the search functions, because these considerations would be very extensive due to widely differing habits and practices.

#### 1.3.1 CALIsphere

*CALIsphere*<sup>5</sup> is a project of the University of California Libraries with more than 300 participating cultural heritage organizations, such as libraries, archives, museums, and historical societies. It is developed and maintained by the California Digital Library. The goal of the project is to provide a common platform for California's historic treasures. Visitors benefit from this because they no longer have to visit each of the institutions individually. Smaller institutions in particular can benefit from technical know-how and infrastructure. In addition to about 1.5 million pictures, numerous other items including videos, texts, and audio files are published on *CALIsphere*. To view items, *CALIsphere* offers three options. Users can browse collections or exhibitions, or search for specific pictures using keywords. Collections (see Figure 1.1) are organized by the content, format or origin of the objects. Because *CALIsphere* unites many different institutions, there are, for example, individual collections from different photographers. In *CALIsphere* there are no objects that are not assigned to a

---

<sup>5</sup><https://calisphere.org/> (last accessed: 2022-07-13).



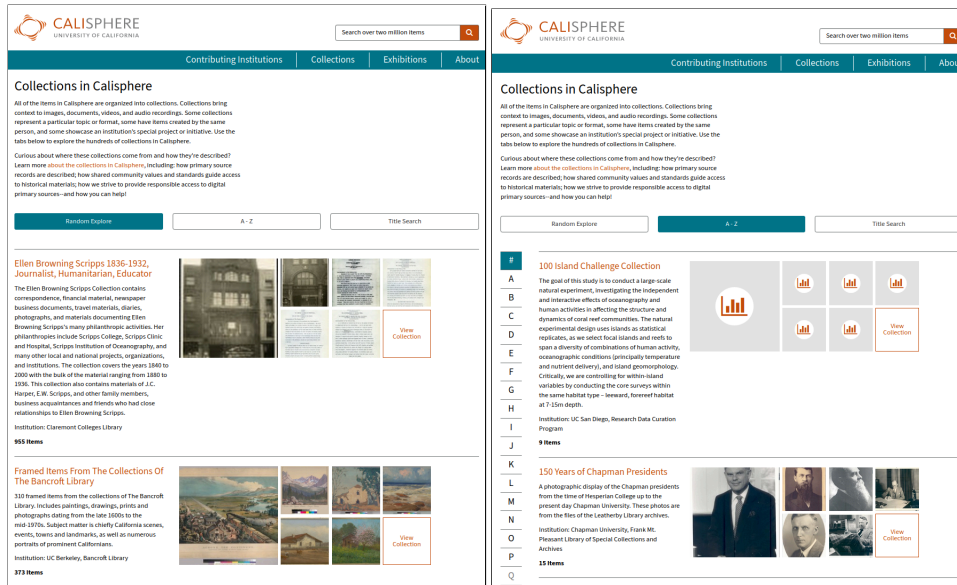


Figure 1.1: CALisphere – Collections

collection. Staff from participating institutions arranges exhibitions (see Figure 1.2) like their analog equivalents to introduce visitors to specific eras or historical topics or events.

*CALisphere* does not offer functionalities to publish comments immediately or collaborate with other users, but they provide the possibility to contact the owner of a piece of cultural heritage and send a message to share an individual story.

### 1.3.2 Project Apollo Archive on Flickr

The *Project Apollo Archive* is a *Flickr* pro-account with 62.9K followers who uploaded close to 16,000 pictures since joining the platform in September 2015.<sup>6</sup> *Flickr* calls itself “almost certainly the best online photo management and sharing application in the world”.<sup>7</sup> Visitors can take a look at their most popular photos and curated albums. It is possible to tag uploaded photos. That allows visitors to take a look at all pictures tagged with a certain keyword from this account or from all pictures on *Flickr*. Signed-up users can comment on single pictures or add them to their personal favorites visible to everyone (see Figure 1.3). We will take a more detailed look at *Flickr* and its functionalities in chapter 2.

<sup>6</sup><https://www.flickr.com/people/projectpolloarchive/> (last accessed: 2022-07-13).

<sup>7</sup><https://www.flickr.com/about> (last accessed: 2022-07-13).

# 1 A Crowd-Sourced Picture Archive

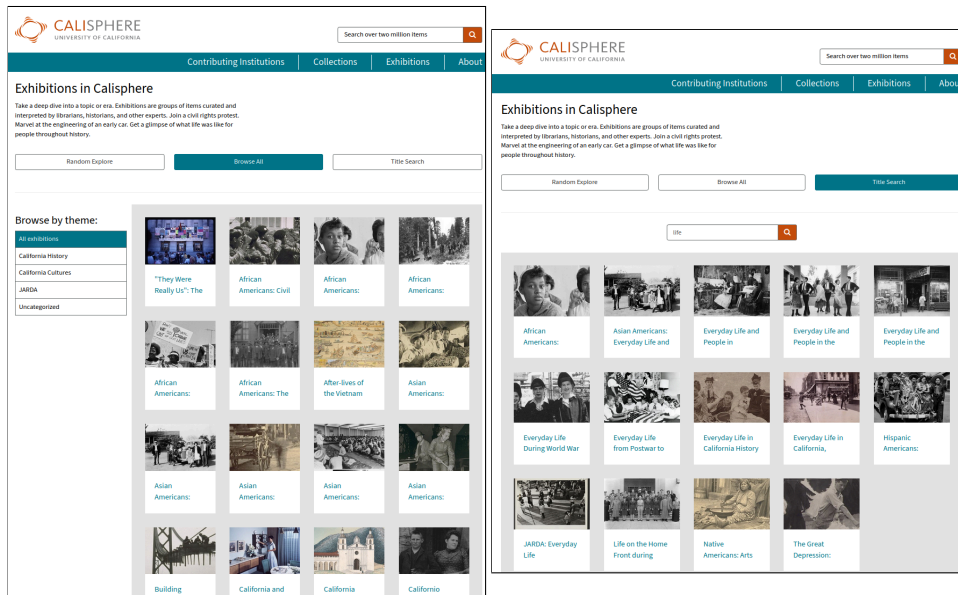


Figure 1.2: CALISphere – Exhibitions

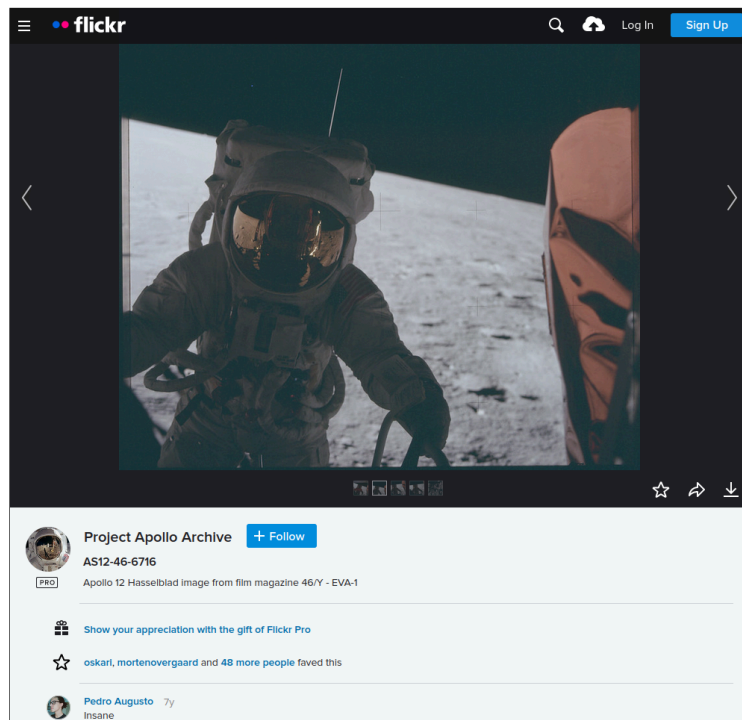


Figure 1.3: Project Apollo Archive: View of an exemplary picture with a comment

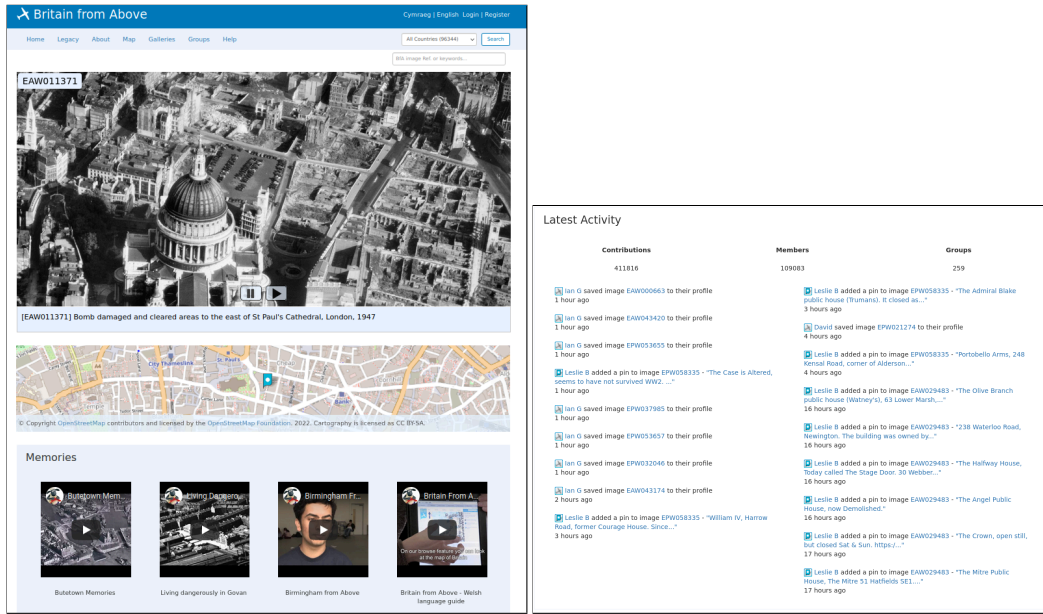


Figure 1.4: Britain from Above – Homepage and Latest activity

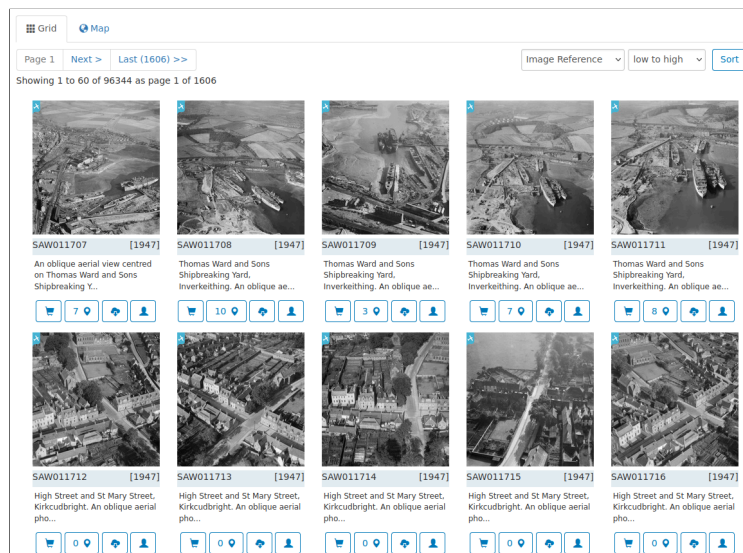


Figure 1.5: Britain from Above – Grid

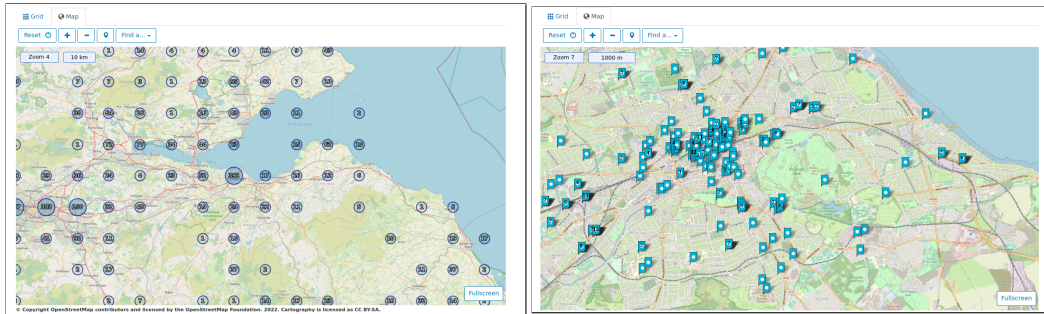


Figure 1.6: Britain from Above – Map

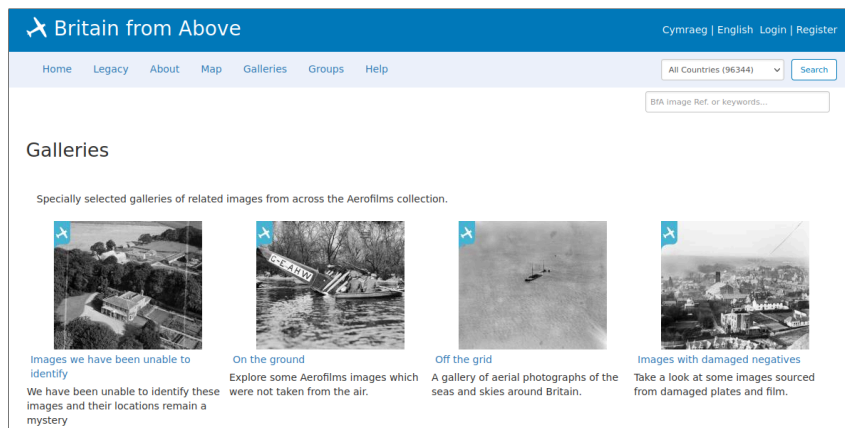


Figure 1.7: Britain from Above – Galleries

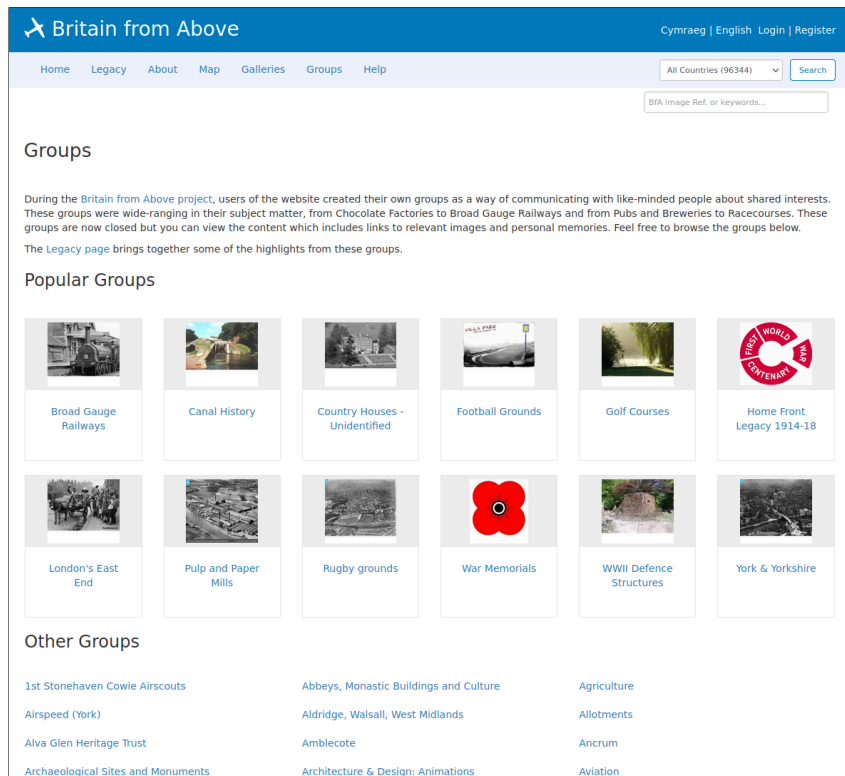


Figure 1.8: Britain from Above – Groups

### 1.3.3 Britain from Above

*Britain from Above*'s<sup>8</sup> aim was to digitalize an enormous number of aerial photographs and localize the shooting position. From 2010 to 2014, about 95,000 photos from the Aerofilms Collection were preserved within the project. *Britain from Above* is financed by the Heritage Lottery Fund, The Foyle Foundation, and other donors. Even though the project ended eight years ago, the website is still online and visitors can view the pictures. When searching or browsing, the pictures can be displayed in two ways, either in a grid or on a map (see Figure 1.5 and Figure 1.6). Because the pictures of the Aerofilms Collection are aerial photographs, they show especially landscape changes, urbanization, new construction, demolition, deindustrialization, urban development, etc. The map view is very suitable for this, as one can navigate to a region on the map and then view the developments there. Visitors can also view selected photos in separate galleries (see Figure 1.7), for instance, "Images we have been unable to identify" or "On the ground", showing no aerial photos but photos taken from the ground. Registered users can get together in groups to share common interests and background knowledge on the pictures (see Figure 1.8). Further, they can add photos to their profile. Until today, signed-up users participate actively by placing pictures with unknown shooting locations on a map by adding a pin (see Figure 1.4). *Britain from Above* links to some YouTube videos in the context of the project on its home page (see Figure 1.4).

### 1.3.4 Europeana

The primary goal behind *Europeana*<sup>9</sup> was to offer a common platform that combines the various European historical archives. *Europeana* is an initiative by the European Union that is financed by the EU's Connecting Europe Facility and the EU member states. They report 30,288,545 pictures, 22,572,573 texts, 770,665 sounds, 337,788 videos, and 8,421 3D models from more than 4,000 different institutions located all over Europe, which they provide online. They do not collaborate with every single institution but rely on aggregators. Aggregators act as interfaces between *Europeana* and the owners of the cultural heritage pieces and work either content-based or location-based. One of the content-based aggregators is the Photoconsortium<sup>10</sup> located in Italy with members and partners from across Europe. The International Consortium for Photographic Heritage accepts photographs taken by private persons or professionals. The aggregator responsible for items from Germany<sup>11</sup> is the German Digital Library.<sup>12</sup> Visitors can read blog posts giving deeper insights on the items or view public galleries, recent items, or collections ordered by themes,

<sup>8</sup><https://britainfromabove.org.uk> (last accessed: 2022-07-13).

<sup>9</sup><https://www.europeana.eu/en> (last accessed: 2022-07-13).

<sup>10</sup><https://pro.europeana.eu/organisation/photoconsortium> and <https://www.photoconsortium.net/association/en> (last accessed: 2022-07-12).

<sup>11</sup><https://pro.europeana.eu/organisation/german-digital-library> (last accessed: 2022-07-13).

<sup>12</sup><https://www.deutsche-digitale-bibliothek.de/content/ueber-uns?lang=en> (last accessed: 2022-07-13).

centuries, and organizations. On their home page, they feature the latest blog entries and galleries suitable for the season or current socio-political topics (see Figure 1.9). It is possible to create a *Europeana* account. Signed-up users can save their favorite items and create public and private galleries. A title and a description can be set for these galleries. Additionally, *Europeana* users can submit ideas for blog posts or share their individual stories accentuated by their exhibits on predefined topics.

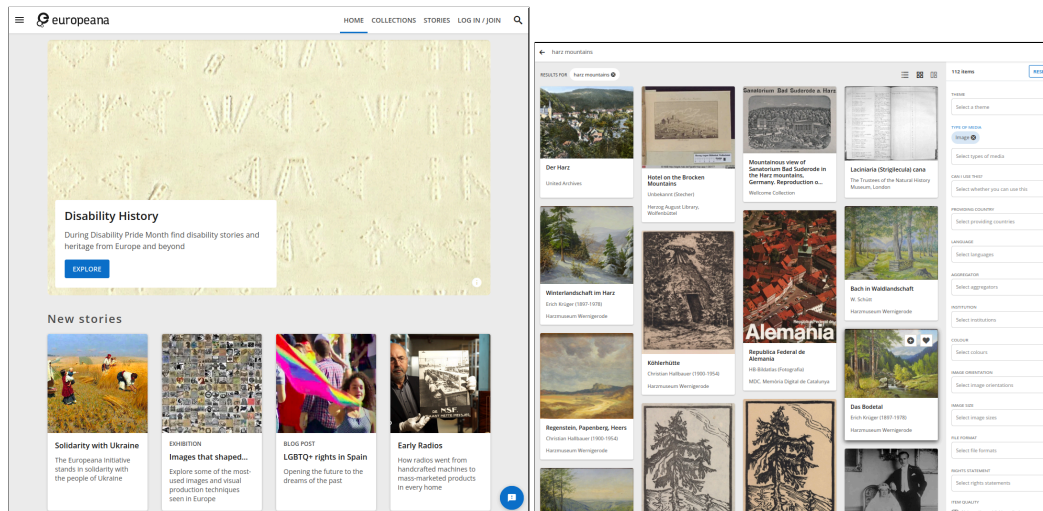


Figure 1.9: Europeana – Homepage and Searching for “harz mountains” pictures

### 1.3.5 Zooniverse

*Zooniverse*<sup>17</sup> describes itself as “the world’s largest and most popular platform for people-powered research”. The project is a collaboration between institutions from the United Kingdom and the United States, more precisely the University of Oxford, Chicago’s Adler Planetarium, and the University of Minnesota – Twin Cities (UMN) as well as volunteer participants and researchers from all over the world. It is mainly funded by “grants, as well as institutional support from Oxford, Adler, and UMN”, but they also accept donations. *Zooniverse* arose out of the astronomy project *Galaxy Zoo*, inviting the public to participate in the classification of galaxies. Nowadays *Galaxy Zoo* is one of more than 100 projects from different scholarly disciplines like

<sup>13</sup><https://www.zooniverse.org/projects/artem-dot-reshetnikov/saint-george-on-a-bike/classify> (last accessed: 2020-07-14).

<sup>14</sup><https://www.zooniverse.org/projects/effeli/node-code-breakers-looking-for-patterns-in-lymph-nodes/classify> (last accessed: 2020-07-14).

<sup>15</sup><https://www.zooniverse.org/projects/mozerm/snow-spotter/classify> (last accessed: 2020-07-14).

<sup>16</sup><https://www.zooniverse.org/projects/artem-dot-reshetnikov/saint-george-on-a-bike/classify> (last accessed: 2020-07-14).

<sup>17</sup><https://www.zooniverse.org/> (last accessed: 2022-07-13).

# 1 A Crowd-Sourced Picture Archive

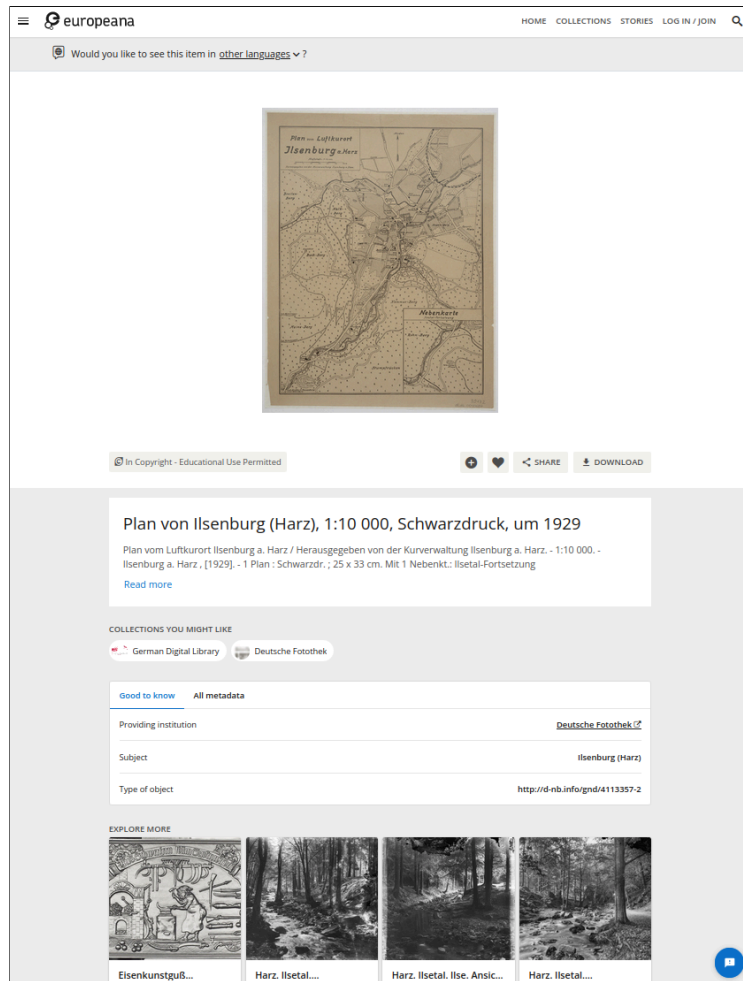


Figure 1.10: Europeana – View of a single item with associated information

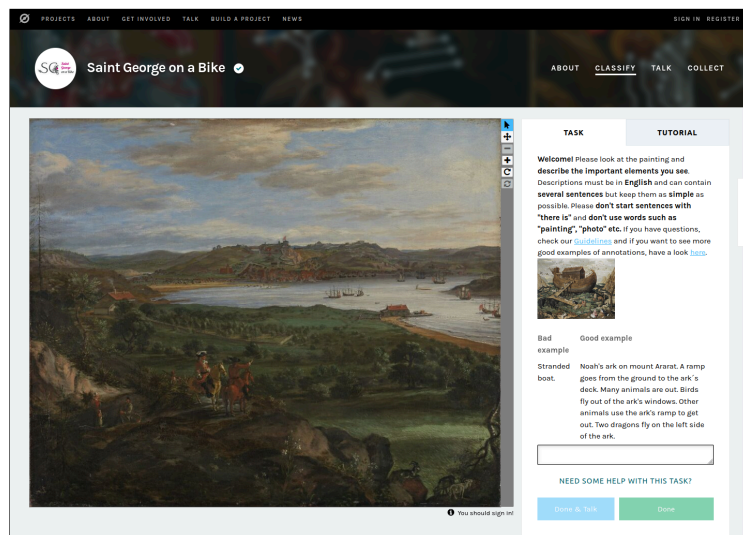


Figure 1.11: Zooniverse – Description: An exemplary description project<sup>13</sup>



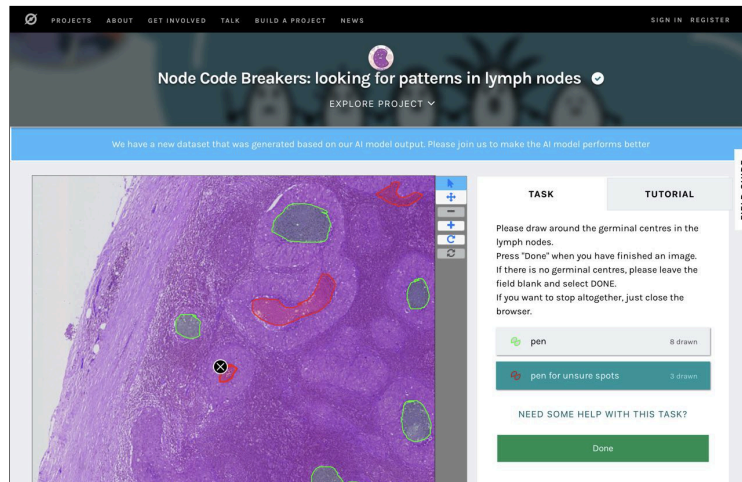


Figure 1.12: Zooniverse – Annotation: An exemplary annotation project<sup>14</sup>

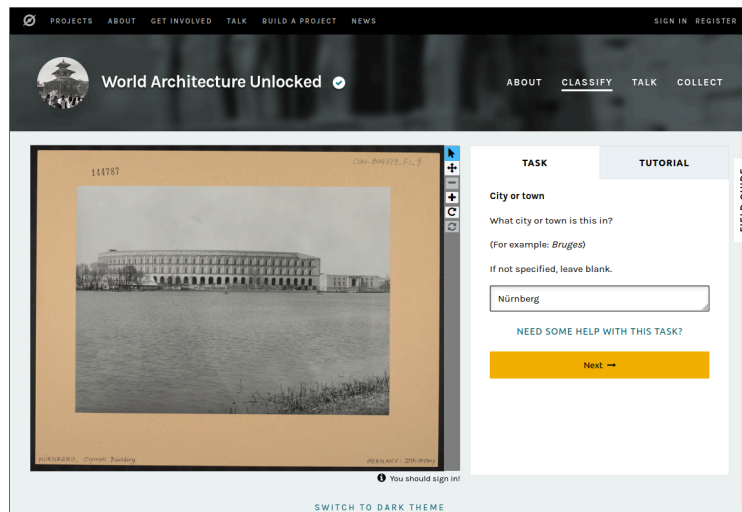


Figure 1.13: Zooniverse – Transcription: An exemplary transcription project<sup>15</sup>

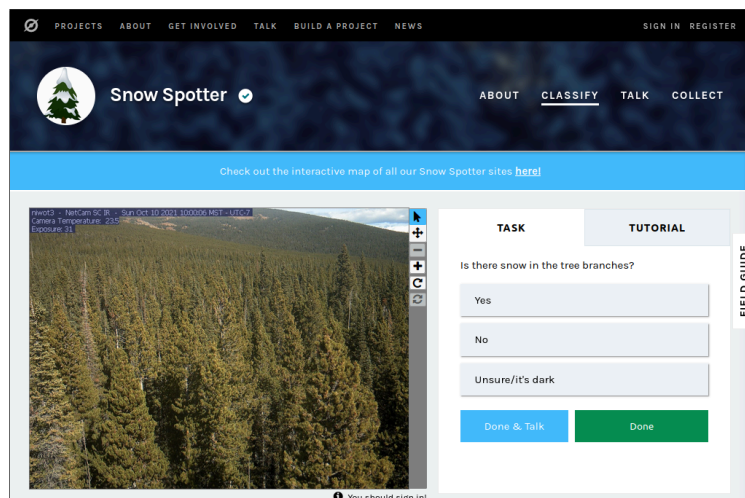


Figure 1.14: Zooniverse – Classification: An exemplary classification project<sup>16</sup>

arts, biology, history, physics, and social science. These projects can be created by everyone. They report “682,077,802 classifications so far by 2,487,137 registered volunteers”. Which tasks the users take on depends on the project in question. There are projects with transcription tasks (see Figure 1.13), contextualization (see Figure 1.11), annotation (see Figure 1.12) and classification (see Figure 1.14). What distinguishes *Zooniverse* from the previous projects is that its focus lies more on data collection, classification, and research results, rather than on individual viewing of the pictures. Still, *Zooniverse* provides the ability for visitors to access the crowd-sourcing results or browse or search processed pictures of a project themselves.

## 1.4 Our Project Partner: Bad Harzburg-Stiftung

Our project partner – the Bad Harzburg-Stiftung<sup>18</sup> – is a community foundation run by volunteers with its main focus on social engagement in Bad Harzburg. The donations are used, for example, to provide free swimming lessons for local children. The foundation received an extensive photo collection: about half a million prints that were part of photojournalist Herbert Ahrens’<sup>19</sup> estate. We distinguish prints and pictures because sometimes there are several prints of the same picture. Mainly two factors led to the intention of digitalizing the archive: firstly the decay of the physical pictures and secondly, given the age of the pictures, the dwindling number of eyewitnesses to the events captured on camera, the people knowing what and who was photographed and the stories captured in pictures. The volunteer members at the Bad Harzburg-Stiftung lack technical expertise, as nobody is a trained programmer or web developer. In 2020, one of them started the digitalization of the first pictures, sorting them into a WordPress-based gallery system (more details in subsection 1.5.2). However, as the number of scanned pictures grew, the complexity of the WordPress system and maintenance of the system moved away from his primary interest of maintaining and curating pictures and more to fixing issues that occurred on the technological side. As such, our project was formed to preserve this treasure trove of pictures and make it accessible to everyone by building a system that is intuitive to use and allows collaboration. We will present our system in the course of our report.

---

<sup>18</sup><https://bad-harzburg-stiftung.de> (last accessed: 2022-07-13).

<sup>19</sup>1910, Ahrens was born in Koblenz, Germany. During the Second World War, he worked for the Propaganda Company of the Wehrmacht. After the end of the war, Ahrens worked in Bad Harzburg and the surrounding area as a press photographer for various local and national media until he died in 1996 and gathered this accumulation of his press photographs.

## 1.5 Our Project Prerequisites

Now we focus on our project and its prerequisites. First, we introduce the parties involved and their interaction. Then we examine the legacy system to identify problems with this first system and thus potential risks or special hurdles for our new system. Eventually, we return to the types of crowdsourcing initiatives and look at potential approaches for our project.

### 1.5.1 Parties Involved

We first list the parties involved, then contextualize them and explain their connection.

- photographers
- owners of the photo collection
- people scanning the photos
- curators
- administrators
- developers
- visitors
  - contemporary witnesses
  - locals
  - historians, local history researchers

The focus of our project is on making historic photos accessible to the public. These pictures were taken by a *photographer* who might have recorded further information on the photos and maybe arranged the prints in a way suitable for their usage. The *photographers* do not necessarily interact with our system. In our case, Herbert Ahrens is not able to do so anymore.

In other cases, the *photographer* and *owner of the pictures* can be combined in one person. The *owner* is responsible for respecting existing photo rights and the rights of personality. We started our project with the Bad Harzburg-Stiftung being the only *owner*. In the meantime, this changed, as different institutions and individuals confirmed their participation by providing their photo collections. The owners of the pictures who have decided to cooperate have realized that their pictures are of interest to the public and our system seems suitable to them. Some are very proud of their collections, in which they invested a lot of time and passion. Therefore, it is important to them that pictures from their collection are recognizable as such.

The analog prints from the archive need to be digitized to make them accessible to people and to protect them from physical decay. For this, *people scanning the pictures* are needed. We will examine the scanning and uploading process in more detail in section 1.5.2.

Since a large, unordered set of pictures can easily become confusing, they need to be organized somehow. The *curators* deal with the structure and preparation of the pictures. They ensure that users are not overwhelmed by this quantity of pictures by linking the pictures with the information available to them, grouping them with other pictures, and selecting appealing suggestions for visitors.

As well as curators who take care of the platform on the content level are needed, someone has to ensure the proper functioning of the system on a technical level: the *administrator*. While the *admin* centers on the system in production, maintenance, troubleshooting for acute problems, etc., new functionalities are implemented by the *developers*. These functions can be requests of the partner, *developers'* ideas, or collaboratively designed. In the past ten months, seven bachelor students have been working part-time on the project. We started with different levels of knowledge and skills and were able to learn and try out a lot during the project that resulted in a web application, a new platform for Bad Harzburg's photos.

Now we will focus on our *visitors*: The motivation for our project was to sustain the memories and stories behind the pictures. If the memories of *contemporary witnesses* are not written down or passed on to the following generations, they are in danger of being lost. The photos in the archive stem from the second half of the 20th century. The Bad Harzburg locals live today where many of the historic photos were taken. On average, the people in Bad Harzburg are older than the German population (see Figure 1.15). Some of them may also have lived there for a long time and are also among the contemporary witnesses or at least know what the places used to look like and recognize some of the people photographed. *Historians* or *local history researchers* depend on historical sources for their work. Therefore, they often visit archives. Such a visit can be very costly: long-term registrations, travel costs, entrance fees, or tight opening hours complicate their activities. Online platforms for resources, such as some archives already deployed, facilitate their job. *Historians* from anywhere can now access the pictures digitally and all they need is an appropriate device and an internet connection. For their research, they rely on powerful and efficient search functions.

### 1.5.2 Legacy System

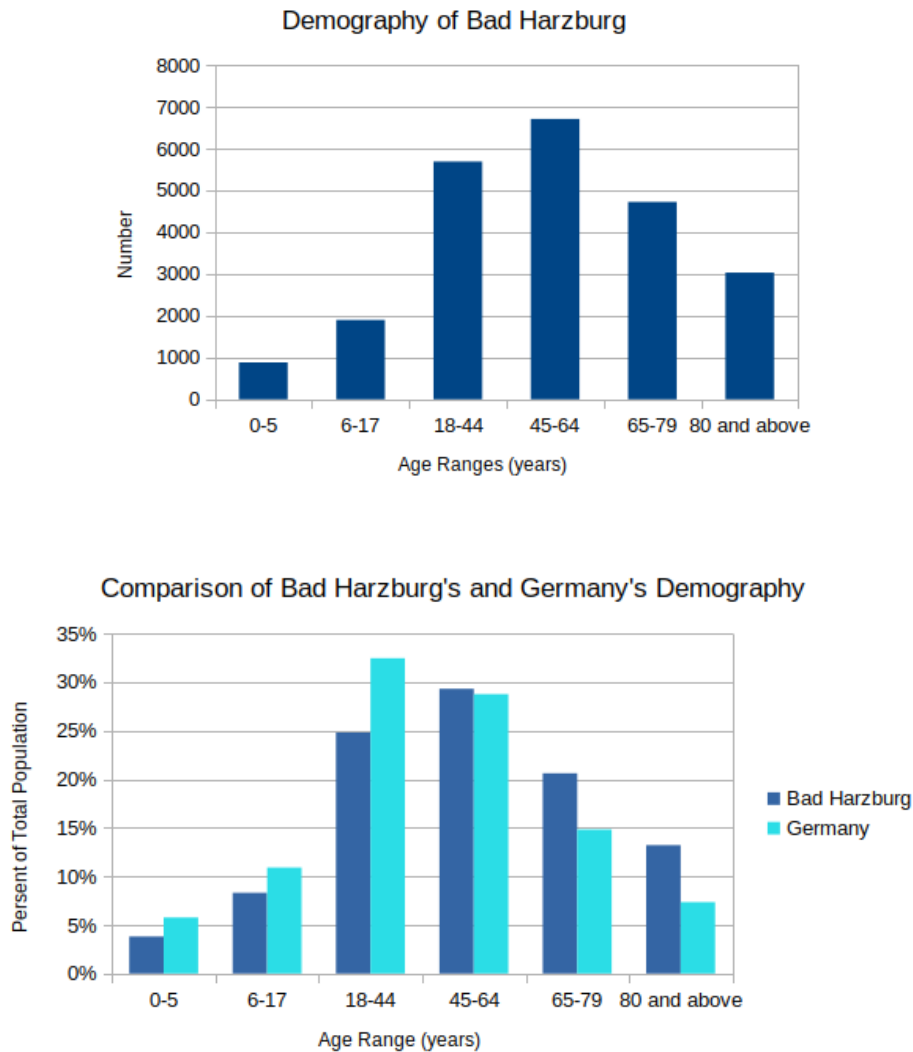
The project has a legacy platform, where around 7600 pictures have already been published. This system was based on the NextGEN gallery plugin for WordPress. In the following, we describe the legacy system and emphasize its problems.

**Upload Process** In the following, we will look at the legacy process of uploading. In the first place, the scanning person chooses a folder. There is no particular strategy for this; they simply take a folder that they consider interesting. The selected pictures are placed one by one into a special photo scanner.<sup>21</sup> With the help of the scanner software, the scanned pictures are then post-processed if necessary, for example, a specific picture section was selected. Parallel to the scanning, notes on the pictures, for instance, remarks by the photographer written on the back of the photo or further information about the picture known to the person scanning are recorded in a Microsoft Office Word document. If a curator would like to create a new gallery or album with the uploaded pictures, they have to sort them into the hierarchical

---

<sup>20</sup><https://www.stadt-bad-harzburg.de/Meine-Stadt/Zahlen-Daten-Fakten/> (last accessed: 2022-07-12).

<sup>21</sup>Plustek ePhoto Z300.



**Figure 1.15:** Demography<sup>20</sup>[55]

structure and create new preview pictures, for which the text has to be entered manually. After that, the pictures were uploaded with their associated information in the WordPress administration panel.

**UI Walk Through** Next, we consider the user interface of the website and how our visitors can interact with it. Thereby we follow the structure in Figure 1.16, where we show the navigation from the home page to a single picture.

The home page contains some general introductory information on the top, a search bar, and below all albums listed (see Figure 1.16a). First, visitors are shown a short information text about the archive. This is followed by a link for email inquiries after the archive. Underneath the keyword search function is briefly referenced.



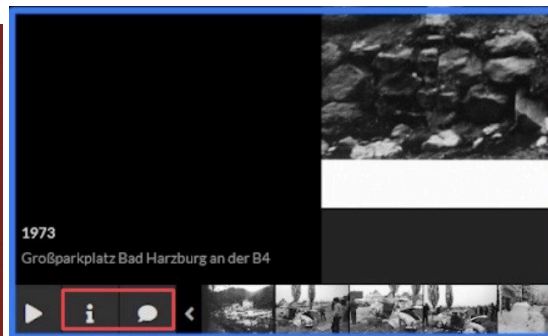
(a) Part of the home page with a search bar and the first three albums



(b) Part of an Exemplary Album Page with Subalbums and Placeholders



(c) Part of an Exemplary Gallery Page with Some Pictures



(d) Detail of an Exemplary Picture Page (see Figure 1.18) with Buttons Leading to Picture Information and Comments (marked in red)

Clicking the area shown in green takes us to the next screenshot.

**Figure 1.16:** UI walk through

Finding pictures via the search function works if users enter the exact keywords, that are linked to the picture.

Below the search bar, users see the albums created by the curator. In three columns, the gray, rectangled buttons with white font are displayed. Some albums or galleries are named after local events (e.g. “Harzburger Musiktage”, a music festival, “Salz-

& Lichterfest”, a town fair, “Galopprennwoche”, a horse racing event), places (e.g. “Rund um den Burgberg”, the Burgberg area, “Kreuz d. deutschen Ostens”, memorial cross for displaced), theme (e.g. “Sport”) etc. Subalbums often have the same title as the albums on the highest hierarchical level complemented by a year (e.g. “Salz- & Lichterfest 1968”). Visitors can navigate through hierarchically organized galleries. These galleries (see Figure 1.16c) contain 15 to 50 photos on average. When users decide to look at a picture, they can click on it. The NextGEN gallery plugin treats galleries and albums very similar, but there is one major difference. Galleries can contain photos while albums cannot. Our system is designed so that galleries are only used if needed.

While viewing a single picture, the largest part of the screen is taken up by the photo (see Figure 1.18). Visitors can navigate to the adjacent pictures with arrow buttons on the left and right side. On the bottom, there is another way of navigation, a row showing previews of the pictures in the gallery. Clicking on one of them leads there. On the left side of this row are three icons, the first starting a dia show, the second showing or hiding the information, and the third one leading to the comment section where users can read comments on a picture or leave one themselves. The information on a photo can be found in form of a title and a description above these three icons. Left-click is disabled on the pictures to prevent people from downloading the pictures in original quality. The reason for this is that press offices and media houses should contact the foundation for commercial use. Bad Harzburg residents and other private individuals can receive the pictures in their original size by mail on request. On mobile devices the same layout is used.

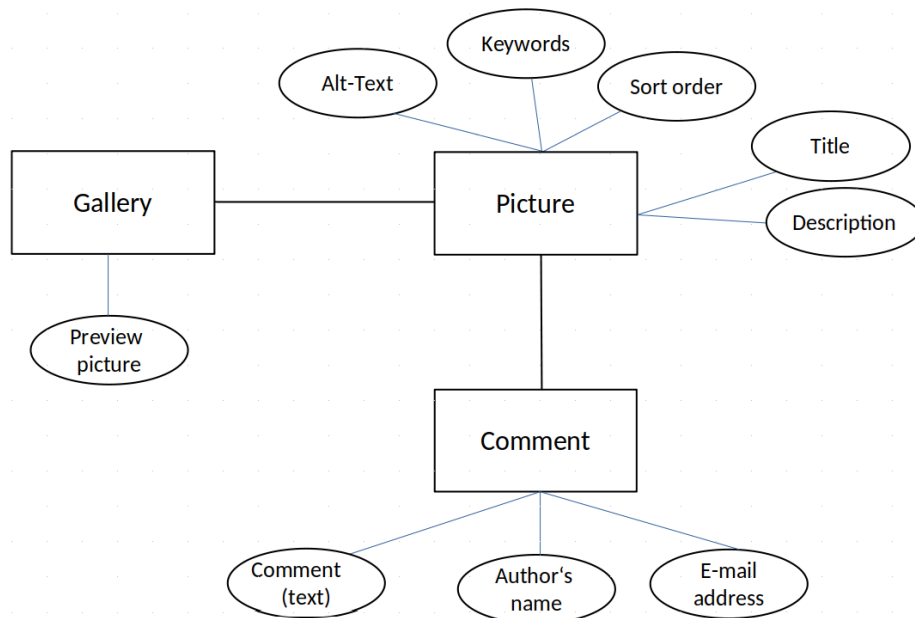


Figure 1.17: WordPress system: Data model





Figure 1.18: Exemplary picture page

**Data Model** In Figure 1.17, the attributes stored for pictures, galleries, and comments are shown respectively. For the purpose of overview and comprehensibility, some technical details are simplified. An image can only be in exactly one gallery, galleries can contain several pictures. A comment can be written only to pictures, several comments can be written to a picture.

### 1.5.3 Problems of the Legacy System

If this initial version of the system had worked without any problems, this project would probably never have been initiated. In our first meeting, our project partner already described some problems with the WordPress site. Other troubles we identified working with the old system. Most of the problems can be summarized under the following three mutually reinforcing factors:

- extensive time expenditure,
- information redundancy,
- lack of a suitable structure.

Following, we take a look at these problems. We begin with the extensive time expenditure. A single person unlocked comments, uploaded pictures, and added information. Our project partner did not succeed in attracting more volunteers to curate the site, of whom there was no shortage in principle, but who were put off by the WordPress Administration Panel, which seemed complicated, and the fear of breaking something on the site in it. As a result, the time-consuming upload process stood out a lot.

The project partner told us that he was frequently called, mailed, or asked about this in a personal conversation because users had difficulties using the website, for

example, finding further information or the comment field. This takes further time. This might not be necessary if we could make the platform intuitive for all users. Especially in the mobile version the comment field indeed was very hidden and small, so users could easily click next to it.

Despite the time involved, the upload process encouraged workarounds and mistakes and made it difficult for our project partner, as curator, to correct errors he recognized. This resulted in a not insignificant amount of photos that were linked to either incorrect, unsuitable, or irrelevant information. We summarize this kind of information as wrongly linked. Due to time constraints, several pictures were often uploaded with the same keywords, without checking whether they applied to all of them. For example, in one of the galleries, there are pictures of the two federal roads in the region. Pictures of both streets have both street names assigned to them. Thus, people who are not familiar with the streets can mistake pictures of the so-called B2 for the B4 and vice versa. As we saw in section 1.5.2, information on a picture is not only stored in the related keywords but for example in the description of the photo as well. That created information redundancy. The search only works with the exact keywords which are rare and not always correct. That makes it hard to find suitable results.

Because the project partner gradually took folders off the shelf that seemed interesting to them without a deeper strategy and digitized them, there is a lack of structure in an intermediate status of 7,600 pictures. The galleries and albums are not named uniformly, some after events, others after locations. This naming system reaches its limits when an event for which there is already an album is held in a place for which the same applies.

Two other aspects came from the lack of a suitable structure:

The data model did not include an attribute for time or date. In the case of historical photos, the date when the photo was taken is particularly important. This date is missing from many pictures, but if it is known, it is not clear where to find the date. Sometimes it is in the title, sometimes in the alt text, and sometimes in the description.

An incorrect description, inaccurate keywords, or a spelling mistake needed to be changed for every picture. Much information was saved per picture, but for instance, descriptions, keywords, and alt text often referred to more than one picture. That was not represented in the data model. The assignment of the same attributes to several pictures was not only due to the structure but also to the time-consuming upload process.

In summary, the upload process, the data structure, and the simplicity of the user interface should be improved. Better search functionality would be nice to have.

#### **1.5.4 Potential of Crowd-Sourcing Initiatives**

In subsection 1.2.2 we introduced different classifications for collaborative projects. In the following, we will return to the taxonomy of Oomen and Aroyo, the different types of crowd-sourcing, and investigate which are suitable for our project.

**Table 1.2:** Types of crowd-sourcing initiatives according to Oomen and Aroyo, How we see their potential and our ideas

Crowd-Sourcing Type	Potential	Possible Ideas
Correction and Transcription Tasks	high	transcribing handwritten annotations from the photos; commenting; editing
Contextualization	high	commenting; photo events; separate display of especially cryptic pictures; video or audio recordings
Completing Collection	medium	uploading functionality for all users
Classification	medium	allocating photos
Co-Curation	high	collection creation for all users; creation of blog posts with exhibits
Crowdfunding	low	-

**Correction and Transcription** The digitization of an extensive collection can be time-consuming and error-prone as we saw in section 1.5.2. Because of that, facilitating the upload process to reduce mistakes is essential for our project. But, since we cannot exclude all errors, allowing our users to report and correct false information related to the pictures is important as well. They could comment or directly edit the concerned piece of information.

There are also useful applications for transcription in our project: Users could help to make out the photographer's handwriting on the back of pictures or the protective envelopes. This could make the upload process faster and the curator would no longer have to figure out alone what is meant by some annotations. Another use case, where transcription could make sense, would be pictures with writing, photos of menus, street signs, banners, etc.

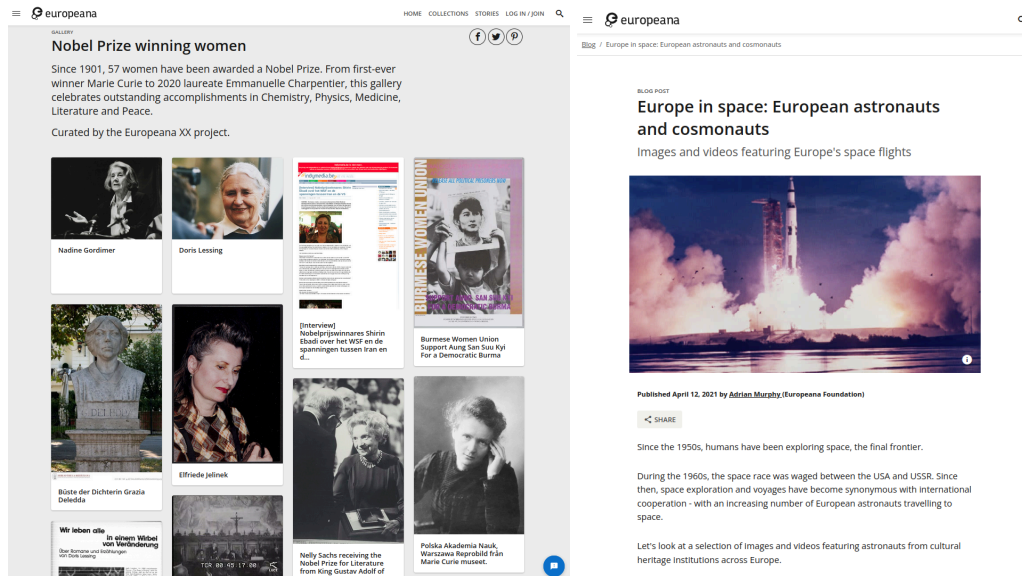
**Contextualization** As Herbert Ahrens was a press photographer, he captured local history and customs as well as regional and national events. Non-locals can hardly recognize these traditions and assess their meaning to the area. For example, pictures of the so-called finch maneuver ("Finkenmannöver" in German) can be found in the archive. These are traditional singing competitions for chaffinches, which have been held in the Harz Mountains for more than 500 years and are part of the UNESCO Intangible Cultural Heritage but are nowadays controversially discussed from the perspective of animal welfare. Background stories like this could be commented on a picture. Another interesting approach would be allowing users to record and upload video or audio, telling the stories themselves. Especially cryptic pictures could be placed very eye-catching and in connection with a call for further information on the website. Our project partner told us that they visited certain clubs and senior groups with the analog photos, who then remembered a lot while discussing together. For such events, one could build a suitable interface.

That is why contextualization may be considered as most important for our project.

**Completing Collection** Some users might have personal photos from some of the captured events in their photo albums or boxes in their attic and be interested in sharing these photos via our platform. Maybe, these photos show another detail, give further insights into a situation, or benefit other viewers differently. Another way for users to complete the collection would be with pictures from today, showing how historic places look nowadays.

**Classification** A collection of 500,000 prints is a real treasure. Nevertheless, there are so many pictures that it can quickly become confusing and chaotic to navigate through them. Both the curators benefit from an improved structure because it helps them to keep track of the entire collection, and visitors, who can take in the known information about a picture more quickly this way. Therefore, further, structured information about the pictures is beneficial.

Users could help to classify different motifs in the photos. It is also imaginable that users set pins on a map of the area allocating the pictures (comparable to Figure 1.6 and the Map prototype in chapter 7). Maybe the same approach of users classifying pictures could work for years or at least time ranges.



(a) An exemplary gallery

(b) An exemplary blog post

Figure 1.19: Europeana<sup>22</sup>

<sup>22</sup><https://www.europeana.eu/en/blog/europe-in-space-european-astronauts-and-cosmonauts> (last accessed: 2022-07-12).

**Co-Curation** Co-curation could be implemented by giving users to chance to create their personal collections and maybe provide context for them. This would allow our users to collect photos with a personal meaning. Blog posts (see Figure 1.19b and the Stories prototype in chapter 7) could also be interesting for anyone whether historian or contemporary witnesses, who would like to share a detailed and extensive story. Other users might be interested in content created by other users and discovering something new.

**Crowdfunding** As a non-profit organization supported by donations, they do not see the need for a specific crowdfunding initiative. It would be an imaginable way of financing new technical equipment such as a scanner in the future. Due to the lack of relevance for our project, we will not discuss crowdfunding further.

## 1.6 Summary

In the last months, we implemented our Crowd-Sourced Picture Archive, a crowd-sourcing platform for Bad Harzburg's historical photos. Thereby, we focused on two aspects in parallel: building prototypes and testing them and building a productive system for our project partner.

At the beginning of this chapter, we pointed out the value and importance of cultural heritage and being aware of our history. It helps us as a society to grow as we try to learn from our ancestors' faults and allows us to better understand today's circumstances. So we should protect the rare means of cultural heritage like our project's photos. Then we considered crowd-sourcing as a way of gaining knowledge about the pictures. Next, we looked at five crowd-sourcing projects in the cultural heritage domain which implement different kinds of participation. We observed that there are many projects with an active user community that succeed in presenting their large number of exhibits clearly and well-structured. After that, we explained the personas involved and our legacy system. We found different ways of crowd-sourcing which seem to be suitable features for our platform.

In chapter 2, we examine the requirements for our system. This system is a React web application. The digitalized pictures as well as the other related data are saved in a PostgreSQL database and organized by the Content Management System Strapi. Deeper insights into our tech stack, further used software, and technologies can be found in chapter 4, chapter 5, and chapter 6. There we also explain why we used React, Strapi, etc. as well as present and discuss our design decisions.

In parallel with the development, we explored the interaction of potential users with our web application and low-fidelity prototypes. Therefore we visited Bad Harzburg and had the chance to test with local and elder people. Our approaches, tests, and results are described and analyzed in chapter 3 and chapter 7.



## 2 Functional Constraints and Requirements

In chapter 1, we described the problems the Bad Harzburg-Stiftung had with digitizing the “Herbert Ahrens Archiv” and the challenges which arose when making the archive publicly available. These occurred mainly in the upload process of the images, their structuring, and the collection of the available picture information. Chapter 1 also introduced the curator as an essential role. This user group is working with the archive by digitizing the images and managing their information. In this chapter, we will describe the initial requirements for a Crowd-Sourced Picture Archive. These are meant to ensure better preservation of the images and their stories compared to the old software solution. Therefore we will take a look at existing picture management and picture sharing solutions and examine their applicability to the “Herbert Ahrens Archiv”. Based on this, we will present functional requirements for a Crowd-Sourced Picture Archive that will hopefully simplify the work of our project partner.

### 2.1 Initial Project Partner Requirements

In several meetings with our project partner, we were able to get a better insight into the domain and work out initial fundamental requirements. These requirements are divided into three classes. The first one addresses general requirements that must be met to digitize an archive of our size. The second class takes a look at usability. The third class deals with picture curation requirements and covers the upload process which was one of the main problems of the old system as well as curatorial requirements for image editing and structuring. These requirements are summarized in the following tables.

#### General Requirements

Table 2.1: General requirements

Reference	Requirement
g.1	Platform needs to be able to handle a large number of pictures
g.2	Cost for Software solution should not be high

- g.1** As already mentioned in chapter 1, the “Herbert Ahrens Archiv” itself consists of almost 500,000 images. In the long term, there is an interest in including other archives from the entire Harz area, which would further increase the number of photos.
- g.2** The archive is currently administered by the non-profit Bad Harzburg-Stiftung. Therefore, the software solution should only cost around 50€ per month or 150€ once.

### Usability Requirements

**Table 2.2:** Usability requirements

Reference	Requirement
u.1	Simple, intuitive user interface for all user groups
u.2	Mobile support
u.3	Support both general discovery and specific inquiry of pictures
u.4	Low entry barrier for contributors and archive access
u.5	Support transcribing image information at conversation pace

- u.1** As described in chapter 1, one of the main problems with the old system was the user interfaces for curators and regular users. In a new system, these interfaces should be simple, intuitive, and clear, thus making it easier to use the system for all user groups.
- u.2** Nowadays, many people use mobile devices, therefore it is important that the archive is also usable on mobile devices [56].
- u.3** With a large number of photos, it is important that the structuring of the photos remains understandable and intuitive. Furthermore, it should be possible to discover and structure photos based on different attributes.
- u.4** The archive should be freely accessible to everyone and should not require any login. Since the archive relies on outside information to preserve the stories behind the pictures, the contribution of information must be made as easy as possible.
- u.5** One use case of the system is the application at events where pictures are shown to attendees to gather additional information. For this purpose, information must be entered directly into the system at the pace of the conversation.

### Picture Curation Requirements

- p.1** The platform should not become a space for hate or misinformation. Therefore, every contribution by users should be approved by the foundation beforehand.



**Table 2.3:** Picture curation requirements

Reference	Requirement
p.1	Moderation of comments to prevent abuse
p.2	Bulk editing pictures and organizational structure
p.3	Extensible meta model
p.4	Support different curator access levels
p.5	Scanner integration and upload process has to be improved
p.6	Mark unverified information as such

- p.2** With the increasing number of pictures, it must be possible to edit multiple images at the same time to make the curation process easier and faster.
- p.3** To make the system adaptable and flexible for future requirements and functionalities, changes to the meta-model should be possible.
- p.4** To distribute the work on the archive among multiple curators, different curator roles need to be supported.
- p.5** In the past, the upload process was very time-consuming, not very efficient, and required the use of several applications see chapter 1 for a detailed analysis. However, as the photos should be saved as quickly as possible, it is necessary to simplify and automate this process where possible.
- p.6** Some elements of a picture are hard to recognize and memories of contemporary witnesses can be incorrect. Thus, some information needs to be marked as debatable to distinguish it from verified information.

## 2.2 Picture Management Applications

Based on our requirements described in section 2.1 we analyzed and looked at various picture management systems. Existing solutions in the field can be compared and distinguished on different levels, for example, platforms vs. applications, public vs. private solutions as well as for-free vs. commercial.

The first category of picture management applications considered here is personal picture archives, which focus mainly on photographers who need to sort and edit their images. Examples of these kinds of applications are Adobe Lightroom, Mylio, and digicam. Personal picture archives usually come with a variety of image editing tools. To organize pictures, they usually work with category tags, albums, people tags, and location data. These allow users to find specific images from a large number of images. Some applications come with cloud implementations to store the photos while others store the photos locally. Photo sharing is mostly supported within the respective ecosystem but not outside the system on a large scale. Applications such as Apple Photos and Google Photos also fall into the category of personal picture

archives. These applications focus on a broader audience and often come with limited storage and minimal customization options. They often have less photo editing functionality, but still allow the management and organization of photos by tags, people, and geo data.

A different type of picture management system explored here is social media platforms like Instagram, Flickr, or Facebook. These allow the upload of images in the form of posts on an existing platform. Some support organizing posts into albums, while others present posts in a single photo stream. As these are social media platforms, the focus is on interaction with other users through, for example, comment sections, direct messages, and likes. In some cases, an account must be created to access the photos, while other platforms can be used to view posts without an account. Above all these applications are characterized by the fact that no technical knowledge is necessary to present pictures and an already existing community is present.

The last category of picture management systems addressed here is website builders that feature picture management solutions. These mostly work with plugins and themes which allow users to create one's website without extensive technical knowledge in web development. WordPress is one of the most prominent examples and comes with a large community of users and plugin developers. To customize plugins or themes to meet special requirements some software development knowledge is needed.

### 2.3 Application Selection

In the following, we are going to look at concrete software solutions. During this process, we will focus on the picture upload, a picture's meta information, picture structuring, search functions, and image sharing. In the context of the requirements described in section 2.1, we will analyze the software solutions for applicability in our case. In the field of personal picture management systems, we will examine both Adobe Lightroom and Apple Photos. Among the group of social media platforms, we will look at Flickr since Flickr focuses mainly on images. With WordPress being used in 43% of all known websites it is the most popular website building tool.<sup>1</sup> Therefore we are going to review WordPress in the context of website building applications. In the last step, we will switch domains and look at the streaming service Netflix in terms of its content discoverability and content presentation characteristics.

In the analyses, we have considered the usability of the user interfaces to a limited extent. All analyzed software solutions are established products that have millions of users and were developed by experts. A detailed analysis of their user interfaces would go beyond the scope of this work. However, if there are concrete problems with the usability of any interface in our context, these were addressed.

---

<sup>1</sup><https://w3techs.com/technologies/details/cm-wordpress> (last accessed: 2022-07-13).

### 2.3.1 Personal Picture Management Systems: Adobe Lightroom CC

**General** Adobe Lightroom is a cloud-based personal picture archive that comes with a variety of photo management and photo editing features. The standard Lightroom license costs around 11.89€ a month and features 1 Terabyte of cloud photo storage with every additional Terabyte costing another 11.89€. According to Adobe, one terabyte is equivalent to about 200,000 JPEGs. Lightroom can be used across mobile, desktop, and web devices, the pictures being synced in the cloud.<sup>2</sup>

**Picture Upload** Images can be imported into Adobe Lightroom either by navigating into the corresponding folder and then selecting the photos or via drag and drop. Lightroom detects duplicates by name and issues a warning to the user before uploading them again. During the upload, a dropdown menu opens where the user can decide if they want to upload the photos to an existing, a new album, or no album at all.<sup>3</sup>

**Picture Meta-Information** A picture's metadata in Adobe Lightroom consists of a variety of attributes. These include title, description, persons, location, rating, and keyword tags. Ratings and flags are options for the curator to structure the workflow, for example, indicating that a photo is not fully curated or that it contains unverified information. Keywords in Adobe Lightroom describe the content of a picture and allow searching for pictures in the archive. Keywords can be nested and the curator can define synonyms for a keyword to make the search even more effective. Keyword sets are self-defined thematic groups of keywords that help speed up the tagging process. By selecting pictures in the grid view many pictures can be tagged simultaneously.

**Picture Structuring** In Lightroom, photos can be displayed based on different attributes. The user can display all photos, recently added photos, photos sorted by time, or sorted by people. Furthermore, Lightroom has different views for presenting photos. The view option Photo Grid shows only the thumbnail without any additional information. Square Grid shows all thumbnails in the same size, along with a status flag, a rating, and sync status. The detail shows single pictures together with the associated information. Each album in Lightroom has a title. Pictures can be added to albums by dragging them into albums. One picture can be included in more than one album. Folders can contain other folders or albums.<sup>4</sup> Lightroom has a feature that, when enabled, analyzes photos in the cloud and allocates individual

<sup>2</sup><https://www.adobe.com/de/products/photoshop-lightroom/compare-plans.html> (last accessed: 2022-07-13).

<sup>3</sup><https://helpx.adobe.com/content/help/en/lightroom-cc/using/add-photos.html> (last accessed: 2022-07-13).

<sup>4</sup><https://helpx.adobe.com/content/help/en/lightroom-cc/using/organize-photos.html> (last accessed: 2022-07-13).

faces to group photos based on detected persons. The curator can rename, merge or remove a photo from a people tag and change the tag's cover photo.<sup>5</sup>

**Search** Lightroom comes with a powerful search and filter tool. It allows the user to search for photos based on every picture attribute and interactively gives suggestions. While searching, it is possible to specify which picture attribute the user would like to search. These filter options can be set manually or set by typing "attribute:" "x" e.g "camera:" "Canon". Search results can be sorted based on a variety of attributes such as capture time, import date, or rating.

**Picture Sharing** Group albums in Lightroom allow the user to share pictures with other people and let them contribute. Albums can be shared via a link or by adding people via their email addresses. The user sharing an album can choose, whether people can only view the pictures, contribute pictures or edit and contribute. Here editing only means making changes to a photo but not changing its tags or metadata. The user that is sharing the album can also adjust the web appearance of the album thus choosing between dark and light themes and different layouts. A shared album can be viewed without an Adobe account but to comment, like, contribute, or edit pictures an account is needed. Users viewing an image in a shared album will see the title, description, format, upload date, and location. They can't see tags or persons.<sup>6</sup>

### 2.3.2 Analysis: Adobe Lightroom CC

**General** With the possibility of expanding the available storage, it would be possible in Lightroom to manage an archive of our size and beyond (Req. g.1). The cost for managing 500,000 JPEG images in Lightroom would be 35.67€ (Req. g.2).

**Usability** The general interface of Lightroom appears to be quite complex initially, which is due to a large number of features. To simplify this, a lot of icons are used and a built-in help function explains functionalities and workflows (Req. u.1). Lightroom also exists as a mobile app, and shared albums can be viewed on the phone (Req. u.2). In terms of general discovery, Lightroom supports albums as well as picture information based filtering of pictures. Folders help to organize albums but can only contain other folders or albums. Structuring with sub-albums is thereby not supported. It is not possible to view pictures on a map, based on their location data. The search and filter functions allow a specific inquiry of pictures. Since all of this doesn't apply to shared albums, these features are only available to the person creating a shared album (Req. u.3). By requiring users to have their own Adobe account to access a shared album and add comments, Lightroom is not suitable as an application for sharing an archive publicly (Req. u.4).

---

<sup>5</sup><https://helpx.adobe.com/content/help/en/lightroom-cc/using/people-view.html> (last accessed: 2022-07-13).

<sup>6</sup><https://helpx.adobe.com/content/help/en/lightroom-cc/using/save-share-photos.html> (last accessed: 2022-07-13).

**Picture Curation** Comments within a shared album are posted directly but can be deleted afterward. Drafting comments before publishing is not supported (Req. p.1). In Lightroom, multiple photos can be selected and then tagged, edited, and added to albums. This allows bulk operations to be performed rather than editing each image individually (Req. p.2). The meta-model in Lightroom cannot be extended with new attributes, thus no additional information about an image or album can be stored (Req. p.3). Shared albums allow users to edit photos or add new photos. However, image information or tags cannot be edited. This would be an important feature to split up curation tasks among multiple curators and provide different curator access levels (Req. p.4). Uploading pictures in Lightroom works manually. Direct scanner integration or auto-watched folders are not supported (Req. p.5). Ratings and flags can be used by users to mark images with unverified information as such. Concrete information cannot be marked as unverified (Req. p.6).

**Conclusion** Adobe Lightroom is primarily designed for photographers and photo editing, many of the available picture editing features are only partly relevant to our application. Direct scanning of photos into the upload interface is not supported by Lightroom. Similarly, it is not possible to auto-import pictures from a watched folder as in the desktop-only version of Lightroom Classic. Shared albums are, as the name suggests, individual albums. Making an entire archive with several folders and subfolders accessible, as necessary in our context, is therefore not possible. Moreover, the interface of shared albums cannot be changed beyond the limited options available. Thus, it would not be possible to adapt the user interface specifically to our needs.

### 2.3.3 Personal Picture Management Systems: Apple Photos

**General** Apple Photos has a lot of similarities to applications like Adobe Lightroom or Mylio. It uses a navigation bar that directs the user to “Library”, “For You”, “Albums” and “Search”. Figure 2.1 offers a view on its starting page and the navigation bar. Under “Library”, users can view photos and videos sorted by time or view all of their photos and videos. The “For You” page shows auto-curated picture collections like “Memories” and shared albums. The “Album” tab shows self-created and shared albums [2].

**Picture Upload** Pictures can be uploaded to Apple photos either by navigating into the corresponding folder and selecting them or via drag and drop.

**Picture Meta-Information** A picture in Apple Photos has a title, a description, keywords, a location, persons, information about the camera as well as a favorite tag to mark favorite pictures and add them to the corresponding album.

**Picture Structuring** Albums in Apple Photos have a name and contain photos. In addition, the application creates auto-curated albums, using image meta informa-

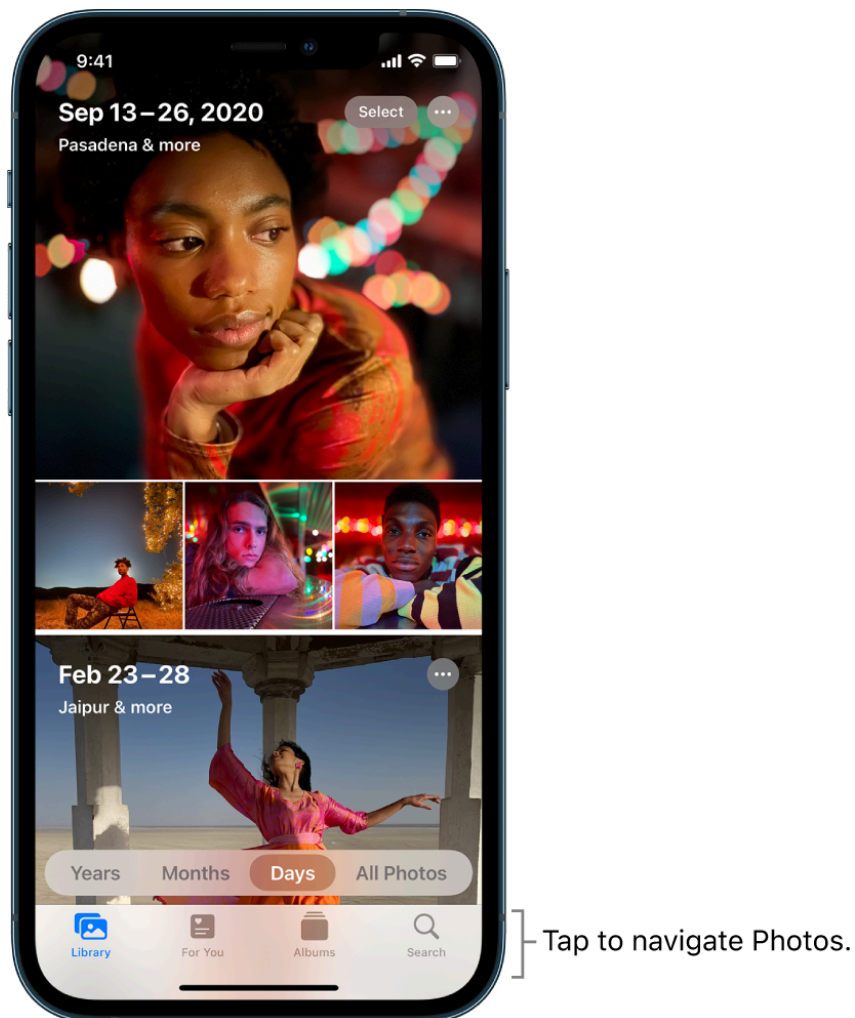


Figure 2.1: Apple Photos Starting page

tion such as location, date taken and people recognized.<sup>7</sup> In the section Memories, photos are merged into groups that are related in terms of time or location and are accompanied by music. This is how yearly reviews, memories of individual days, or compilations of photos featuring recognized individuals, are created.<sup>8</sup>

**Search** The Apple Photos search function is simple in the sense that it consists of a single search bar, but powerful in terms of its underlying functionality. The search browses all image metadata and suggests additional search terms from various categories. The results are displayed in different categories, depending on which image attribute matches the search terms. It is particularly helpful that Apple Photos rec-

<sup>7</sup><https://www.apple.com/ios/photos/> (last accessed: 2022-07-13).

<sup>8</sup><https://support.apple.com/guide/iphone/watch-memories-iphhd4f70e68f/15.0/ios/15.0> (last accessed: 2022-07-13).

ognizes objects in photos with the help of machine learning algorithms and gives results based on these.<sup>9</sup>

**Picture Sharing** Shared albums allow the user to share photos with Apple users, but also with non-Apple users, via a link. Shared albums support a variety of image formats as well as special Apple formats like “Live Photos” and can contain up to 5,000 photos.<sup>10</sup> Shared albums are not counted in the available iCloud storage of a user. When creating a shared album, the user can decide whether people who access the album via Apple Photos should be able to add their photos. Users who are logged in can comment on pictures and give them a like. People who access a shared album via a link can only view the images and their titles. They do not see comments, likes, or image meta-information and cannot add images themselves. Also, there is no possibility to search for images in a shared album. For this, the user would have to save the images in their library first.<sup>11</sup>

#### 2.3.4 Analysis: Apple Photos

**General** Apple Photos is designed for organizing private photo collections and sharing selected photos with friends and family. For this purpose, it offers a variety of helpful features and a simple user interface. Apple Photos is for free and has no general limitations on the number of pictures (Req. g.2). “Shared albums” are limited to 5,000 pictures or 1 GB in size which could lead to problems with larger albums. Additionally, the number of shared albums a user can create is limited, currently 200. This could also cause problems in a growing archive (Req. g.1).

**Usability** Mobile support is given with the iOS app and the Apple Photos web application for Android users (Req. u.2). For personal use self-created albums as well as auto-curated albums allow a good structuring of images based on different image information. However, Apple Photos does not support the hierarchical structuring of albums, which can make large albums complex and chaotic (Req. u.3). The search function of Apple Photos with its machine learning algorithms entailed makes it very easy to find photos. A similar implementation could greatly simplify finding photos in our archive and would be of great use (Req. u.3). As already mentioned, shared albums are not sufficient for our purpose to make the photos available to the public. The only collaboration features in a shared album are comments and likes, which can only be used by Apple users and are not possible via the website view of a shared album. In addition, these are posted automatically and can only be deleted afterward by the owner of the album (Req. u.4 p.1).

---

<sup>9</sup><https://support.apple.com/guide/iphone/search-in-photos-iph392d77d5f/15.0/ios/15.0> (last accessed: 2022-07-13).

<sup>10</sup><https://support.apple.com/en-us/HT202299> (last accessed: 2022-07-13).

<sup>11</sup><https://support.apple.com/guide/iphone/share-photos-with-icloud-shared-albums-iph3d2676c9/15.0/ios/15.0> (last accessed: 2022-07-13).

**Picture Curation** Apple Photos does not support bulk actions except moving multiple photos from one album to another. (Req. p.2). In Apple Photos, the image meta information is fixed and cannot be expanded (Req. p.3). The same applies to the fixed appearance of the website of shared albums, thus it is not possible to customize or extend it. Pictures and their metadata within a shared album can only be edited by the owner of the shared album. This makes it impossible to support several curators with different access levels (Req. p4). Scanning photos in Apple Photos is not directly supported by the application. Scanning and uploading photos must be done in two steps (Req. p.5)

**Conclusion** Apple Photos is not sufficient for digitizing an archive of our size. The main problems are the limited number of shared albums as the only way to share images, the missing possibility to structure albums, and the lack of bulk editing and curation tools. However, the auto-curated albums are a great feature that helps users with time-consuming curation tasks and the machine learning-assisted search can improve and provide more specific search results.

### 2.3.5 Social Media Platforms: Flickr

**General** Flickr is an online photo-sharing platform and social network. Flickr allows users to create a free account, upload pictures to their profile, and share them with the world. Flickr has similar functions to other social networks such as following people and liking a post (favorites on Flickr) to save them and show appreciation. Also, each post has a comment section where users can share their impressions, ask questions or add any information regarding a post. Flickr has a free version that allows uploading a maximum of 1000 pictures, includes ads, and allows commenting and favoriting pictures. Without an account, a user can view every public photo on the platform. The pro version costs 7,49€ a month and allows users to upload an unlimited number of pictures in full resolution, to reveal extra insights and statistics around their posts.<sup>12</sup>

**Picture Upload** Flickr has an upload interface where pictures can be added via drag and drop or by navigating to the corresponding folder.<sup>13</sup> The interface allows the user to upload and categorize many pictures at the same time. With the Flickr pro version also comes the possibility to auto upload pictures from watched folders.<sup>14</sup>

**Picture Meta-Information** During the upload process, users can add a description, tags, and people in the picture and specify privacy, licensing, and safety settings for

---

<sup>12</sup><https://www.flickrhelp.com/hc/en-us/articles/4404078815508-Reasons-to-upgrade-to-Flickr-Pro> (last accessed: 2022-07-13).

<sup>13</sup><https://www.flickrhelp.com/hc/en-us/articles/4404079632660-Upload-Photos-and-Videos-to-Flickr> (last accessed: 2022-07-13).

<sup>14</sup><https://www.flickrhelp.com/hc/en-us/articles/4404071315860-About-the-Flickr-Uploadr-for-Mac> (last accessed: 2022-07-13).



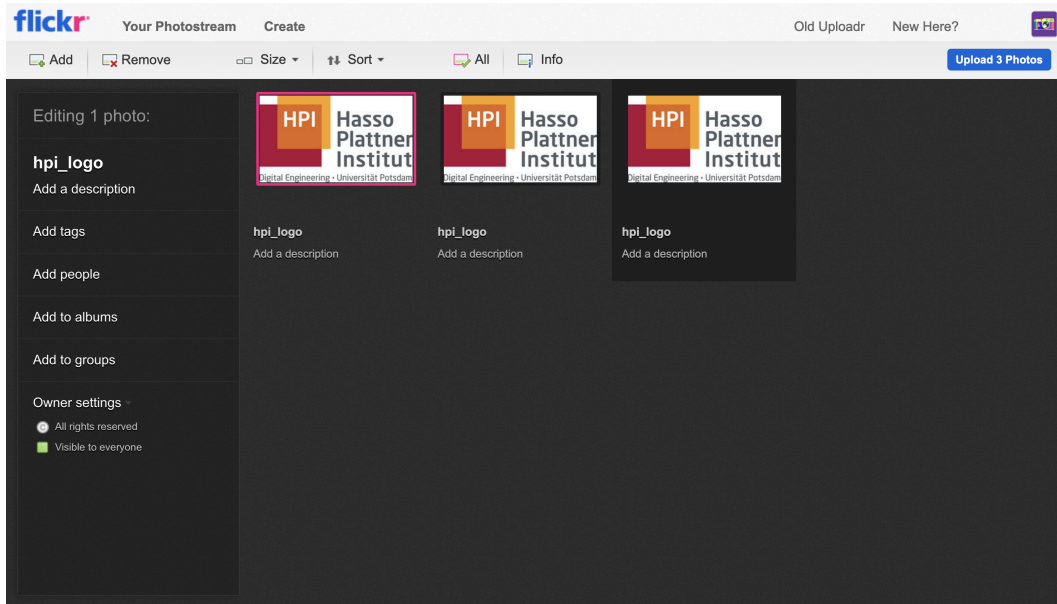


Figure 2.2: Flickr's Organizer tool

the picture. Also, pictures can be added to albums and groups. After uploading pictures, more information about location and camera settings can be added. Flickr allows the user to edit many pictures at the same time via the Organizer interface.<sup>15</sup> Figure 2.2 offers an overview of the Organizer tool and editing options. Single pictures can be edited by clicking on corresponding fields.

**Picture Structuring** Each account on Flickr has its page. Each page contains a navigation bar that links to the About page, the Photostream, Albums, Faves, and Groups. The landing page of an account is the About page. The About page of an account gives users the possibility to write something about themselves and to show 25 of their photos. Also, some statistics are shown e.g. views, most popular photos, and a testimonial section.<sup>16</sup> The photostream shows all photos of the account sorted by upload date. Albums allow to arrange images thematically into groups and have their title and description. Users also can choose the order of the images within an album. Collections can contain albums and other collections and are used to group albums.<sup>17</sup> The Faves section shows all pictures of other users that an account has liked. Galleries allow the user to share content from other users.<sup>18</sup> They are similar

<sup>15</sup><https://www.flickrhelp.com/hc/en-us/articles/4404064313620-View-and-manage-your-photos-with-the-Camera-Roll> (last accessed: 2022-07-13).

<sup>16</sup><https://www.flickrhelp.com/hc/en-us/articles/4404070311060-Get-started-with-your-About-page-in-Flickr> (last accessed: 2022-07-13).

<sup>17</sup><https://www.flickrhelp.com/hc/en-us/articles/4404064213524-Keep-your-Flickr-photos-and-videos-organized> (last accessed: 2022-07-13).

<sup>18</sup><https://www.flickrhelp.com/hc/en-us/articles/4404058722836-Galleries-Overview> (last accessed: 2022-07-13).

to albums in the sense that they have a title, and description and the owner can choose the order of the content but they only contain pictures of other users. This is certainly an interesting feature to include external and additional resources in an archive. Galleries, like single photos, have their comment section. The Group section shows all the groups that the user is part of. Groups are a way to share ideas and photos with people that are interested in the same topic and are similar to a forum.

**Search** The Flickr search bar allows the user to search for photos, people, and groups. It is possible to search for pictures on the entire platform as well as in individual profiles. When searching for pictures the user can resort to various advanced settings like searching by colors, tags, time period, licensing, safety, and image format.

**Picture Sharing** Images uploaded to Flickr are publicly available based on their privacy settings. Flickr supports four different privacy settings. Private images can only be seen by the account holder. The Friends option allows only accounts marked as friends to see the images. The same applies to the Family option. The Public option allows anyone to see the images on Flickr.

### 2.3.6 Analysis

**General** To digitize an archive of our size one would require the premium version of Flickr. With a price of 7.49€ per month and unlimited posts, Flickr meets both g.1 and g.2 of our requirements. In the past, Flickr has had repeated funding problems [37]. So there is a risk of the platform being discontinued. In this case, there is the possibility to save and download the whole archive including comments to prevent losing any information.

**Usability** Flickr can be used both on the web and as a mobile app (Req. u.1). In terms of discoverability, it is not possible in Flickr to view photos of an account on a map based on the geo data or in a time-based order. The About page of each account, featuring 25 self-selected photos and the possibility to view the most popular, most commented, or most favored pictures, offers a different entry to the archive, besides the commonly used approach of organizing albums thematically (Req. u.3). The Flickr search bar allows specific inquiries of pictures and comes with a variety of display and filter options (Req. u.3). As discussed earlier, the archive relies on users' information and stories about the pictures. Flickr main collaboration tool is the comment section which also supports responding to comments and posting pictures. In terms of layout, the comment section is rather hidden under the pictures. As we have already seen in chapter 1, a similar design decision in the past caused users not to find the comment section. A different approach to collaboration is Flickr's group feature which allows like-minded users to discuss topics and post pictures. The Gallery feature is also a form of collaboration since it allows users to create their new albums from other people's photos. In our context, users could, for example, make a gallery out of their favorite photos or create a new group of photos to tell a

new story. The problem with these collaboration tools is that the contributor needs their Flickr account to use them (Req. u.4). It is possible to change individual picture attributes directly in the picture view of Flickr, such as album affiliation, tags, title, and description by clicking into the corresponding field. This would make it possible to present pictures and at the same time incorporate the feedback of an audience (Req. u.5).

**Picture Curation** Comments are automatically posted but can be deleted by the account holder afterward (Req. p.1). Flickr upload interface and bulk editing tools allow an intuitive and easy maintenance and upload process that fulfills our requirements (Req. p.2). Flickr's underlying data model and layout are fixed, thus it is not possible to customize the design to our user group or to provide additional information such as the photographer (Req. p.3). For example, in Flickr, it would not be possible to adapt the user interface such that the comment function is more in the foreground and viewers are more encouraged to contribute. Collaborative work by several people, who have different authorization levels and thus perform different tasks, is not supported. An archive on Flickr is associated with one account that has all access rights thereby division of labor in the curation process is difficult to implement (Req. p.4). Currently, one can not directly scan and upload pictures to Flickr but similar behavior can be achieved with watched folders (Req p.5).

### 2.3.7 Conclusion

Flickr supports a variety of useful features that enable knowledge sharing. Groups allow users to create small forums on specific topics. Galleries allow users to create new albums with photos from other users. However, comments are posted automatically and can only be written by registered users. Albums and collections do not allow hierarchical structuring with sub-albums and Flickr is hardly customizable. Therefore Flickr is not an applicable solution in our context.

### 2.3.8 Website Builder: WordPress

**General** WordPress is an open-source content management system written in PHP and originally designed for blog publishing. Today, the area of application also includes online stores, forums, and content galleries. WordPress has a large variety of built-in features and external plugins which are created by external developers and companies. Some of them are free, others are fee-based. These plugins implement different functionalities and make WordPress very variable. WordPress has a variety of themes that allow the user to define the look of the site without any programming knowledge. There are themes that come from WordPress directly and others that come from external developers. Also, themes can be free or fee-based.

WordPress has a built-in picture and gallery feature. Each gallery has a title and description. The gallery feature allows the curator to upload pictures via drag and drop or by navigating to the corresponding folder. The number of pictures in a row and the spacing between each picture within a gallery can be chosen. Galleries have

a “link to” option that controls what happens after clicking on a picture. “Attachment Page” opens a large version of the picture and shows the picture name as well as a comment section. “Media File” is similar but allows clicking through all pictures in the gallery in the form of a slide show. Each picture in a gallery has a description and an alt-text. The alt-text is supposed to be an explanation of the image content, help visually impaired users, and is displayed if an image doesn’t load. Pictures can be edited and arranged and an HTML/CSS class can be defined. The gallery works well for a few photos that are supposed to visually support a blog post or online store, but the gallery reaches its limits with many photos. The feature does not support albums or tags. In addition, the built-in search block of WordPress does not search within galleries but only in individual photos.<sup>19</sup> However, to digitize a large archive like ours and make it accessible, it is important to structure, tag, and search for photos. The design of the gallery feature is quite simple but the picture information and comment section are not very visible. For a better implementation of the archive in WordPress one would either have to write a special plugin or use one of the existing picture and gallery plugins. In the following, we will take a closer look at the plugin NextGEN Gallery.

**NextGEN Gallery** With more than 600,000 active installations NextGEN developed by Imagely is Wordpress’s most popular gallery plugin.[weblinkhttps://wordpress.org/plugins/nextgen-gallery/](https://wordpress.org/plugins/nextgen-gallery/) NextGEN Pro costs \$139 per year and comes with different styles and different ways of organizing pictures.<sup>20</sup> WordPress comes with six built-in user roles.<sup>21</sup> These allow users to access various functionalities such as creating posts, moderating comments, or creating pages. NextGEN is building on these roles to restrict access to custom functionalities such as creating albums and galleries as well as uploading images.<sup>22</sup> NextGEN also allows commenting on pictures and lets the admin decide whether comments should be drafted or directly posted [42].

**Picture Upload** Pictures can be uploaded to NextGEN via drag and drop or by selecting a picture or entire folder from the user’s computer.

**Picture Meta-Information** Pictures have an ID, thumbnail, filename, title, description, tags, and the possibility to set a price for potential e-commerce.

**Picture Structuring** Albums in NextGEN can contain other albums or galleries.<sup>23</sup> An Album has a title, description, preview picture, and the option to link to a page when clicking on the album. Galleries store pictures similar to albums, they have their title, description, a thumbnail, and with the pro version the option to add a

---

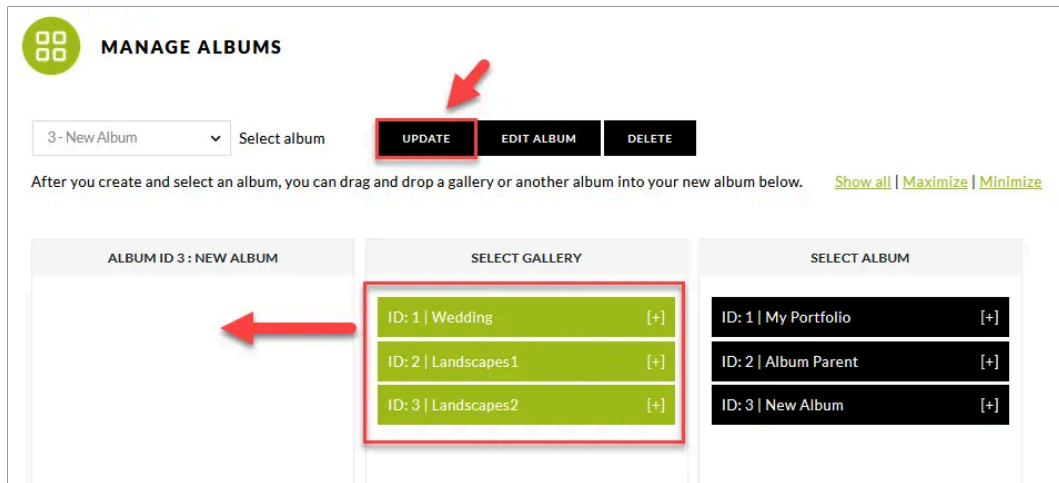
<sup>19</sup><https://wordpress.org/support/article/the-wordpress-gallery> (last accessed: 2022-07-13).

<sup>20</sup><https://www.imagely.com/pricing/> (last accessed: 2022-07-13).

<sup>21</sup><https://wordpress.org/support/article/roles-and-capabilities/> (last accessed: 2022-07-13).

<sup>22</sup><https://www.imagely.com/docs/roles/> (last accessed: 2022-07-13).

<sup>23</sup><https://www.imagely.com/docs/albums-vs-galleries/> (last accessed: 2022-07-13).



**Figure 2.3:** NextGENs tool for structuring albums and galleries [27]

price for an entire gallery. Also, the order of images within the gallery can be set by the user.<sup>24</sup> Via drag-and-drop, the user can add/remove galleries and albums to/from an album and sort them [27]. Figure 2.3 offers a view of NextGEN's tool for structuring albums and managing a hierarchical album structure. To manage the images within a gallery NextGEN supports a variety of functions. Bulk actions can be applied to all or individual pictures within a gallery.<sup>25</sup> Some interesting bulk actions are the possibility to watermark pictures, resize images, create new thumbnails for the pictures, import metadata from the images, or edit, delete, and overwrite tags.

**Search** NextGEN comes with its own search function that allows the user to search in tags, descriptions, and alt-text. Additionally, a boolean search is supported as well as different ways of ordering the search results. A practical search feature is that tags associated with a search term can be displayed and made clickable. This gives the user further search restrictions that lead to more precise results and thereby speeds up the search process.<sup>26</sup>

**Picture Sharing** Images uploaded and published in NextGEN are publicly accessible via the associated WordPress website.

### 2.3.9 Analysis

**General** The built-in picture and gallery features of WordPress are not sufficient for our requirements. It lacks important elements like albums, tags, and search functionality to organize the data on a thematic level as well as the presentation of image

<sup>24</sup><https://www.imagely.com/docs/basic-thumbnail/> (last accessed: 2022-07-13).

<sup>25</sup><https://www.imagely.com/docs/manage-galleries/> (last accessed: 2022-07-13).

<sup>26</sup><https://www.imagely.com/docs/search-feature/> (last accessed: 2022-07-13).

information to make an archive of our size accessible and to present information about the images properly. Photo and gallery plugins from WordPress implement many of our requirements but since they are intended to address a large group of people, they lack more specific requirements. The picture gallery plugin NextGEN costs \$139 per year and has no limitations on the number of pictures uploaded (Req. g.1, g.2).

**Usability** Problems arise when users want to customize plugins or built-in features for their specific needs. For example, Wordpress’s curator interface, where images are uploaded and edited, is also the admin interface. This interface is complex and contains many functionalities that people working with the content of the archive do not need and should not have access to (Req. u.1). To achieve such customization of a plugin the user would have to either write a custom support plugin, create or use custom hooks or override callbacks.<sup>27</sup> Changing plugin functionality can be a difficult task that is quite prone to errors. The same problem applies to the possible approach of extending the metamodel (Req. p.3). NextGEN galleries are customized to also support mobile display. However, we saw in chapter 1 that the mobile version had some issues with the visibility of the image information and the comment section (Req. u.2). Similar to the previously considered software solutions, NextGEN uses galleries to structure images. With the help of albums, hierarchical structures can be created. However, since albums themselves can only contain galleries, this structure is only conditionally usable for our use case. To achieve deep hierarchies, albums need to be nested. This can get confusing but there is no built-in functionality to better structure huge numbers of pictures (Req. u.3). The existing search function works well and is easy to add to a page. This comes with many handy settings to better customize the search to the user’s application (Req. u.3). WordPress sites are freely accessible via the internet and comments can be posted without prior login (Req. u.4). Picture information cannot be edited directly on the site. To show and edit the images during a conversation, the shown image would have to be edited parallel in the WordPress admin interface which would be a complex process (Req. u.5).

**Picture Curation** NextGEN’s commenting feature meets our requirements (Req. p.1). However, additional possibilities for contribution like self-curated galleries are missing which limits the options of implementing the concept of a collaborative archive. NextGEN’s interface for editing albums allows curators to change the organizational structure of albums and galleries quickly (Req. p.2). For photos, NextGEN supports several bulk actions that allow, among other things, to edit photo meta information and move photos from one gallery to another (Req. p.2). With the help of the user roles, it is possible to distribute the work on the archive to different persons and to determine which access they get (Req. p.4). NextGEN does not support direct

---

<sup>27</sup><https://developer.wordpress.org/> (last accessed: 2022-07-13).

scanner integration, for this, an additional support plugin would have to be written (Req. p.5).

**Conclusion** NextGEN is remarkable for its low implementation effort. It allows the users to set up a digital photo archive in the form of a website without the need for detailed technical knowledge. The plugin supports many settings and functions to adapt to different domains. The bulk actions allow the user to keep control even with an increasing number of images. For our use case, however, the lack of customizability outside of the supported settings is problematic as some of our requirements are not supported.

### **2.3.10 Different Domain: Netflix**

Netflix is an American streaming service that works via subscriptions. The Netflix Web page uses a navigation bar that links to the homepage, series, movies, the “New & Hot” category, a personal list, and a search. In the context of content discoverability, the homepage that also works as Netflix’s landing page is particularly interesting to us. The “Home” page looks different for each user and is intended to present the content of most interest to each user. Netflix has over 17,000 titles worldwide and therefore has a similar problem to ours. How do you show the user the right titles that interest them and make them stay on the page? The homepage consists of various thematically grouped films and series which are arranged in rows. This creates a two-dimensional structure. Horizontally there are titles within one category. On the vertical level, there are different categories. This makes it possible to present and navigate a large part of the content intuitively. The content of these categories can be created based on metadata such as genre or actors involved in the movies. For example, categories are latest releases, trending movies as well as continue watching. However, behavioral information is also used. For this, Netflix uses collaborative filtering algorithms that make suggestions for new titles based on the behavior of similar users. It is important that the films displayed in the different categories correspond to the user’s taste but also include diverse titles to also suggest titles to users that they might like but are different from previous titles. Netflix uses a ranking system that ranks both the categories and the titles within. To ensure diversity, the categories are not ranked in isolation, but the two rows above are also included in the ranking. To test this automated home page generation, A/B testing, as well as measured data, was used. For example, the scrolling behavior of the user can be measured and analyzed to test the utilization of the home page and the recommendations on it [1].

## **2.4 Concept for a Collaborative Picture Archive**

In section 2.3, we demonstrated that none of the applications we considered met our requirements in their totality. However, we gained valuable insights into how differ-

ent applications solve similar problems and requirements. We have also described interesting features that could be used in our application.

In the following, we will present what an application that meets those requirements could look like. The basic approach is to use a content management system and build a web application on top of it (chapter 4 and chapter 5 for more detail). This modular approach allows a flexible development of a system that fully suits the established requirements. First, we will discuss aspects concerning interface design, picture uploading, picture editing, picture structuring, and collaboration functionalities. Then, we will describe the different user roles interacting with our system and present a suggestion for an underlying data model for the system.

### **2.4.1 Interface Design Choices**

To make our application easy to navigate, a prominent navigation bar that leads to different functions like a search would be useful. This could look and work similarly to the featured navigation bar of Apple Photos, which stands out for its simplicity and comprehensibility. All of the applications considered allow viewing photos in full screen as well as browsing through photos in the form of a slide show. Both features enrich the user experience and thus are also desirable for our system. To make all available image information easily accessible, these should be displayed directly in the picture view. The same applies to the comment function. To understand the different functions more intuitively, icons and help functions that explain the use can help.

### **2.4.2 Picture Uploading**

To simplify and speed up the process of uploading new images, it must be possible for curators to scan the images directly into our application and in the next step enter the attached metadata. For this, integration into our system of the scanning software as well as picture editing software for cropping and minor changes must be supported. At the same time, we want to support the classic variant of uploading via drag and drop to upload already existing scans.

### **2.4.3 Picture Editing**

In terms of picture editing and curation, Flickr's approach works best as it supports both bulk editing of pictures and single picture editing (see subsection 2.3.5). Taking inspiration from Flickr, we aim to minimize the steps required to edit picture information by making it possible to edit the information of a single picture in the picture view itself. For this task, the curator should be able to change and add new picture information such as description, time of recording, keywords, people, and location tags directly by clicking into the corresponding fields. This will make the curation process of individual image information simple and intuitive. By implementing a similar interface for viewing and editing the images, the transition from a regular user to a curator is made easy. Thus, the concept of a collaborative picture archive



is implemented on the user interface level. With an archive of our size, it may well happen that a large number of images contain incorrect information, need to be re-sorted, or have to be re-curated. In this case, it must be possible to edit several pictures at the same time. An interface similar to Flickr's "Uploadr" interface would cover this use case. Here, multiple photos can be selected and bulk operations can be performed on them, e.g. adding new tags, changing existing information, or re-sorting the images into another collection. It is important for these operations that the curators do not have to access the database or the content management system directly to fulfill the requirement u.1 from section 2.1. To best distribute the curation process among different people, it must be possible to give multiple people access to the curator tools. To ensure aspects of data security and safety, not every person should be able to perform all of the tasks. Therefore different access levels and user roles similar to WordPress(subsection 2.3.8) should be supported. What these user roles could look like is discussed in subsection 2.4.6.

#### **2.4.4 Picture Structuring**

We have seen that the typical structuring of a large image set can be done with the help of thematic albums. All of the software solutions considered following this approach as their main method of structuring. In addition, they support the filtering of images based on other metadata such as person tags, geographic data, or temporal classification. Since the archive in its analog state is currently also organized in thematically sorted folders, a hierarchical album structure, where albums can contain both photos and sub-albums, could be a suited structure for our archive. As the pictures were mostly made in a geographically very narrow area, it could be interesting to show the pictures sorted geographically. For example, on a map or grouped by location. The same applies to a chronological classification, as the photos show the history of the Harz region throughout the 20th century. Many of the images contain people, which is why a good classification of them can be particularly interesting. Identifying people in each photo and manually tagging people is a time-consuming and error-prone process. Adobe Lightroom and Apple photos are examples of using automated people recognition to easily perform such grouping. Netflix is a great example of how a personalized homepage can help users to access titles from a large amount of content, based on their behavior. In our context, such a feature could suggest to users albums they have started browsing, new albums that have been added to the archive as well as new picture collections based on their behavior and interests. To support the specific inquiry of pictures a search functionality is going to be needed. Our metadata selection should support this use case and form the basis for search parameters. With Apple Photos we have seen a good example of how machine learning algorithms can improve the quality of search results by enriching the meta information with automatically recognized image contents.

### 2.4.5 Collaboration

As described in chapter 1, the motivation of the project is to preserve the images and their stories and make them accessible to the public. For this, our project partner depends on the knowledge of contemporary witnesses and local residents. That is why users need to be able to contribute this information. All of the applications considered in section 2.3 use commentary functions for this purpose. These allow for the simple and familiar contribution of information. Therefore, a comment function would also be useful in our application. Flickr is an example of other collaboration possibilities. Features such as likes, self-compiled albums, and groups could also encourage collaboration.

### 2.4.6 User Roles

**Photographer** The Photographer does not interact with our system directly. However, he or she took the photos and left some information that might be of interest to the archive in the pictures folder.

**Curator** Curators manage and work with the archive. They take the pictures out of their protective covers, scan them and upload them to the platform. After that, they provide all the information they received about the motive, context, location, persons, or time. When new information concerning a picture becomes apparent, they update this information. The curator also develops and maintains the organizational structure of the pictures.

**Browse User** A Browse User visits the digital archive without a specific idea or interest regarding the pictures. This type of user browses through the archive from one topic to another.

**Search User** Search Users search for certain pictures or topics they are interested in.

**Contributer** A Contributor has the right to add information to pictures but can not upload new pictures.

**Commentator** Commentators add information to a picture in the archive by writing a comment.

**Moderator** A Moderator looks over every written comment, checks the information, and prevents abuse.

**Admin** Admins maintain login credentials and access rights for curators, contributors, and moderators. Admins are also able to access the data model and the content management system.

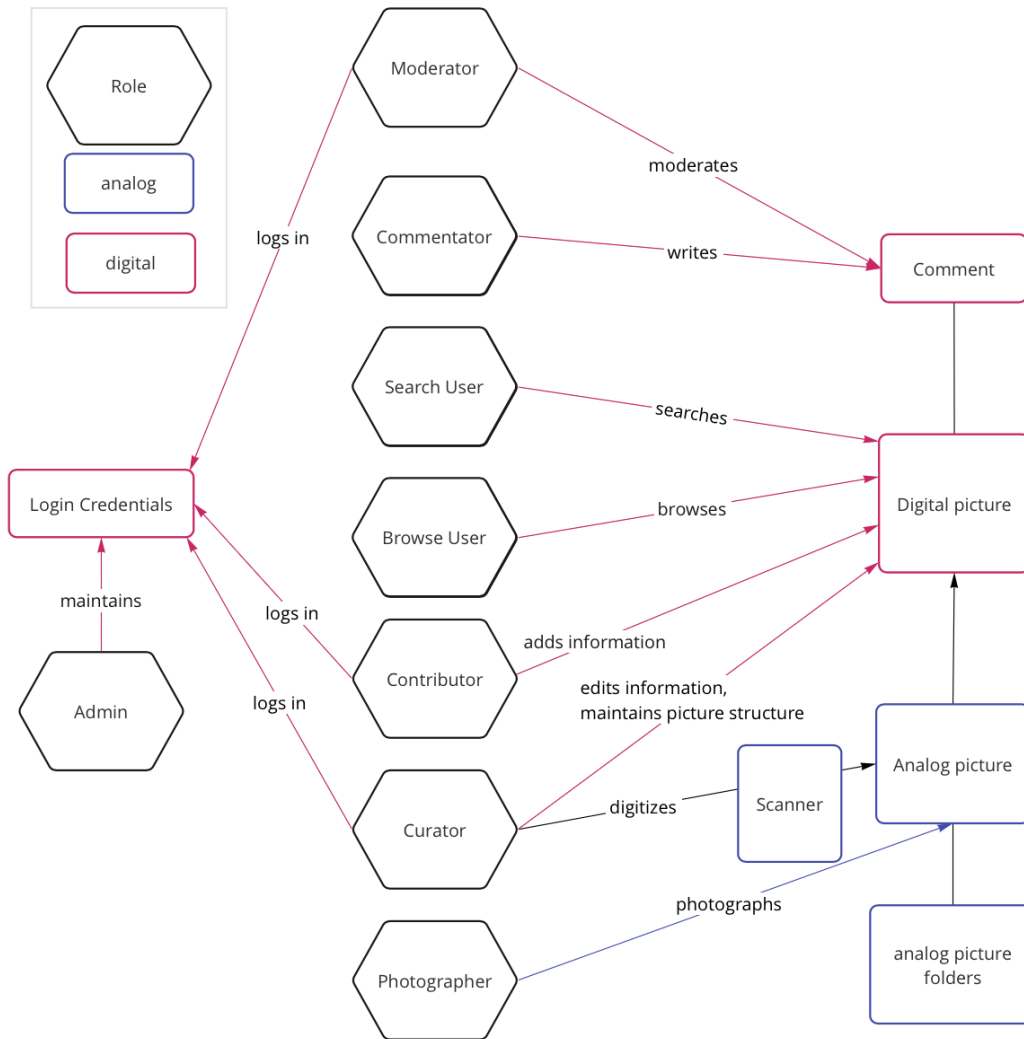
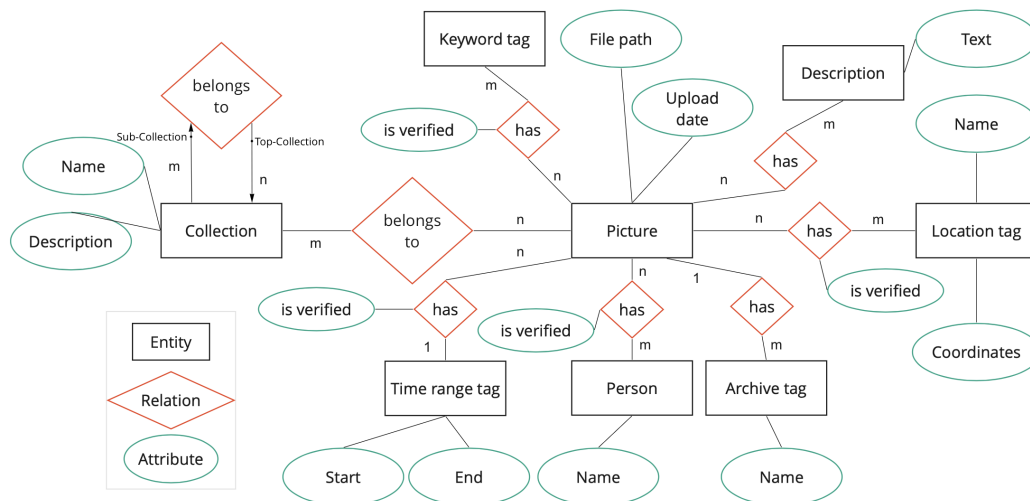


Figure 2.4: Domain model of the archive

### 2.4.7 Data Model

Figure 2.4 offers a view of the Domain Model and shows the described user groups. In section 2.3 we looked at different software solutions and their digital representation of an image. Most of them associate a picture with a title, a description, tags, persons, locations, and the camera that was used to take the photo. As we have already discussed in chapter 1, in the old system a title was stored for each image. During our analysis, we realized that a title attribute is not necessary. The titles contained mostly redundant data and the descriptions are not long enough to require a summary in form of a title. In the old system, the folder structure was expanded step by step without a concrete plan for the entire archive. This made the structure confusing and difficult to understand for new curators. Our first idea was to convert albums into category tags. These would have had a name, a description just like



**Figure 2.5:** Picture data model

albums, and an additional priority attribute. Images would have had several category tags and category tags would have been related to each other. With the priority attribute, it would have still been possible to keep the information on the hierarchical structure of albums. However, together with the project partner, we decided that we do need a clear hierarchical structuring in the form of albums since this gives especially Browse Users a good overview and gateway to our archive. That is why we came back to the initial idea of pictures belonging to collections and collections holding subcollections. Figure 2.5 offers a view of the final data model.

The data that was stored in the old system as keywords, is divided into location, keyword, person, and time range tags. This way specific search queries can be supported and the information can be presented in a better, more structured way. Location tags have coordinates in addition to the name to support the display of pictures on a map. Time range tags have a start and end date as attributes to support both accurate and inaccurate time stamps. In addition, we have added an archive tag that supports the hosting of other archives besides the Herbert Ahrens archive on our platform. Location, person, time range, and category tags have an additional “is verified” attribute on their picture to fulfill our requirement p.6 from section 2.1. This design choice will be discussed in more detail in chapter 5.

## 2.5 Summary

In this chapter, we have elaborated that none of the considered applications in the field of picture management and picture sharing fully meet our requirements. The problem lies in combining image management tools, image sharing tools, and collaborative knowledge sharing features. Based on our initial requirements and the

insights we obtained during the examination of existing applications, we presented a concept for a potential software solution. We described functionally how picture upload, picture editing, picture structuring, collaboration tools, and UI choices could be implemented in such a system. We also presented the various user roles and a data model for such a system. To improve the upload process, scanner and picture editing software need to be integrated into the application. Bulk operations for picture editing as well as an easily usable interface, for changing a single picture's details, are desirable. Contributing information should be made as easy as possible. Therefore a comment function and the possibility for users to create their albums could be implemented. To divide curation tasks among several people, different access levels should be supported, based on the roles described in subsection 2.4.6.



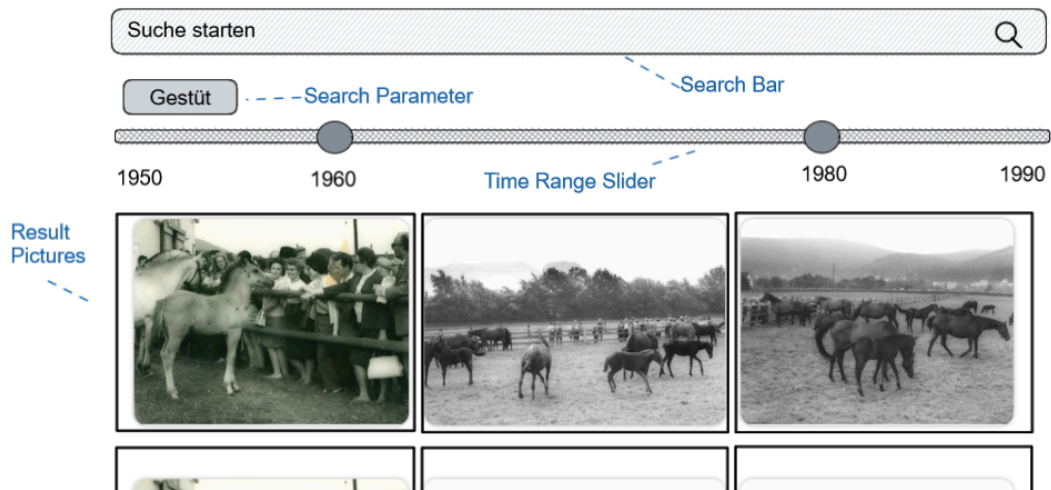
## 3 Evaluating Design Decisions regarding Usability

One important part of an application's lifecycle is usability testing. According to Levi and Conrad [34], "usability testing is the process by which human-computer interaction characteristics of a system are measured and weaknesses are identified for correction". Usability testing is therefore crucial to check whether the product or application is ready to be used and enable improvement. This is especially important for our project because of its intended public availability and usability for everyone. The goal of our application is to preserve and exchange knowledge. This includes the necessity of a visitor interface where the pictures and their information can be viewed and comments can be added and an interface for the curators to manage pictures and their information. For both, we have to fulfill certain requirements, especially regarding usability. We display the requirements of interest for this chapter in the following. To get a full list of the requirements of our application see chapter 2. The visitor interface has to be usable by everyone and be intuitive (u.1), especially the comment section. Browse and search functionalities have to be included (u.3) and mobile support has to be provided (u.2). The curator interface also has to be easy to use and both interfaces must be integrated into each other (u.1). The scanner has to be integrated into an easy workflow (p.5) and the moderation of comments has to be possible (p.1). We used two different approaches to evaluate these requirements. Qualitative evaluation was essential to verify the more specific requirement (u.3) and to discover what usability hindrances exist (u.1). Whereas quantitative evaluation was applicable for u.1 to get feedback on the general usability.

We will start with the first design prototypes followed by the description of the first implemented application design *V0* and of the curator interface design from version *V2*. Thereafter we explain our testing process with heuristics, pilot testing, user testing, and the curator interview. *V0* was adjusted to *V1* with insights from the heuristics and pilot testing. *V1* was then used in the user testing and further developed to *V2* containing the curator interface we described earlier. Lastly, we will discuss how our goals were met and draw an outlook on what still could be tested, adjusted, and implemented.

### 3.1 Design and Implementation

This section introduces important design aspects for our prototypes, client feedback, and design details of *V0*, followed by an introduction to the curator interface *V2* design.



**Figure 3.1:** Prototype 1 with time range slider. Now only pictures with time range tags between 1960 and 1980 are displayed.

### 3.1.1 Prototypes

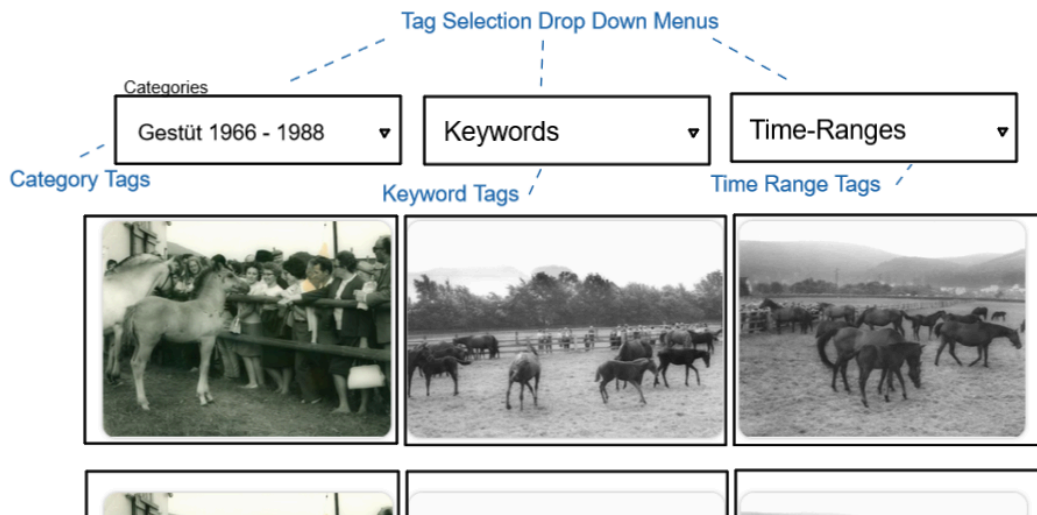
The most central features of a picture archive are displaying and accessing pictures. Therefore, we started with the picture information that we need to display together with the picture. In the legacy system, a picture can contain a description text and tags that are labels assigned to a picture by the curator and can contain information about the time (time range tags) and further keywords (keyword tags).

The pictures always had category tags that represent the picture's album. In our prototypes, we especially focused on how pictures could be accessed and structured by those tags. Prototype 1 focuses on the search design. The other two prototypes were based on the requirement to build the application for both Browse Users and Search Users (chapter 2, chapter 1). Prototype 2 forms a combined design, whereas Prototype 3 presents two separate views. Our general structuring concept was to first display all pictures and then use tag selections to narrow them down to find wanted pictures. Furthermore, the search design included a restrictive search concept where only pictures fitting all search parameters are displayed.

Prototype 1 (Figure 3.1) contains a search bar that can be used to look up pictures via the category tags. The search parameters are displayed below the search bar and are clickable. Another interesting idea was to provide a time range slider below the search bar. The idea was to support the search for pictures even when the exact date is not known. The prototype did not focus on Browse Users, yet they can scroll through all of the displayed pictures.

Prototype 2 (Figure 3.2) was exploring another idea to create a compact, reduced interface. Prototype 2's search concept consists of three drop-down menus where the user can choose from category tags, keyword tags, and time range tags. In the first two drop-downs, multi-selection is possible and the concept of a restrictive search applies. Only the pictures that have all of the selected tags are displayed. The pictures





**Figure 3.2:** Prototype 2 with tag selection drop-down menus. With “Gestüt 1966-1998” selected only pictures associated with this tag are displayed.

below can be again used to browse, but in this design, a Browse User can also look through all of the available selection categories to get inspired.

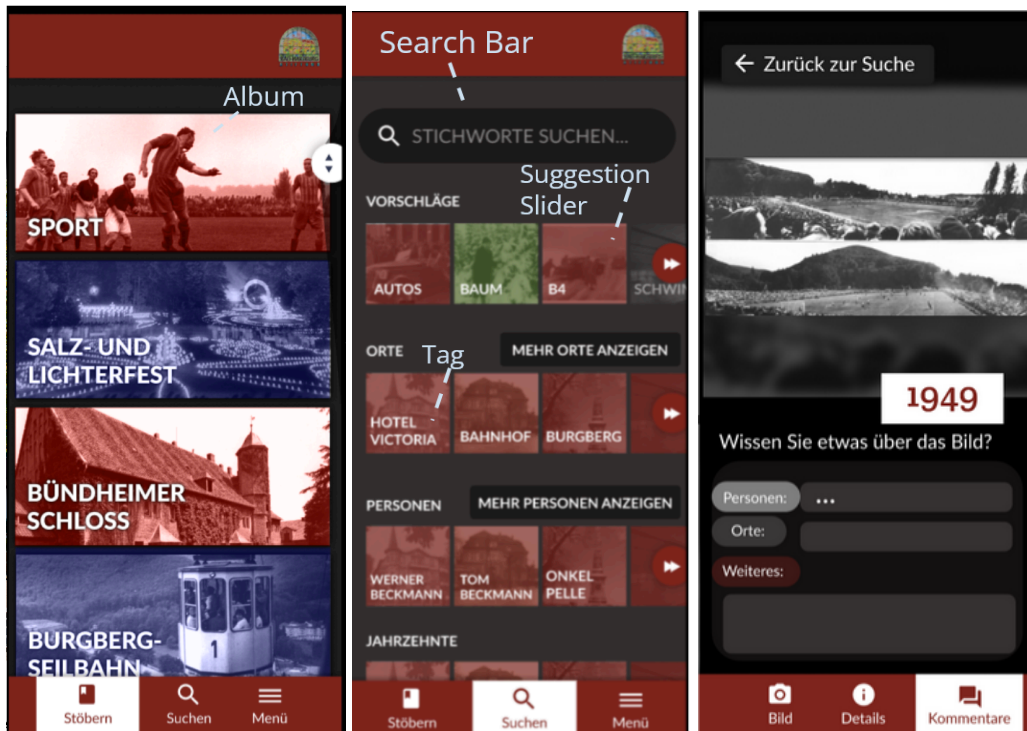
After creating the first prototypes, we wanted to explore a design that includes functionality for the Search User and the Browse User (chapter 2). To also get a first understanding of mobile design Prototype 3 was realized with a mockup for the mobile view. We focused on the two main views: the browse view and the search view. Both views are accessible via a bottom bar (Figure 3.3).

The Browse (“Stöbern”) view contains the albums seen in the old product as clickable tiles with a picture, from each respective album taken, as an illustrative background image. Thereby, we wanted to give more visual input for the browsing process and to help decide which tile to click (Figure 3.3).

The design for the search view contained a search bar on top with horizontal sliders below. The horizontal sliders contain clickable categories that are created from the tags assigned to the pictures (Figure 3.3). We conceived a general suggestion slider and three sliders for places, persons, and decades are created. The suggestion slider contains a random selection of tags when no search term is entered. When a user starts to write a search term in the search bar, the tags displayed in the suggestion slider change dynamically. Tags that fit the search term are shown.

Those horizontal sliders were discussed since various studies and research found them to be a hindrance, especially for elder people [25]. Another possibility would have been to create a hinged container per tag type. A hinged container has two modes. It can hide most of its content and only display a few tags and when clicked it opens to display more tags taking also more space. The problem is that the limited space, especially on a small mobile display, would not allow seeing all tags selection containers at once and making the navigation with opening and closing of the containers confusing. Our compromise was to add buttons on the right and left side to give users another navigation opportunity than scrolling horizontally.

When a picture is clicked, the picture view opens. Here the description and other picture information such as the time and the comment section are accessible (Figure 3.3).



**Figure 3.3:** Prototype 3 – Browse View, Search View, and Picture View with Comment section

An idea regarding the display, search and order of pictures via place tags was to use a map (Figure 3.4). Here pins are showing to which places pictures exist. This approach was not included in our first implementation before testing due to time constraints. Yet in another user test (see chapter 7) we experimented with this idea of a geographical order.

**User Feedback** We presented the prototypes described in section 2.1 to our customer to receive feedback. Here we learned that a search as in Prototype 1 or Prototype 3 is important, yet according to our customer as displayed in Prototype 1 the slider is inaccessible for older age groups (Figure 3.1). And even if the drop-down menus in Prototype 2 allow browsing through the tags, it can be cumbersome to use the drop-down menus with a big number of selection options and it is easy to forget to unselect an unwanted tag (Figure 3.2). The mobile design in Prototype 3 was received positively, especially the two views for browsing and searching.

We discussed whether the comment section in the picture view should be form-like or one input field. The Prototype 3 design held extra input fields for places,

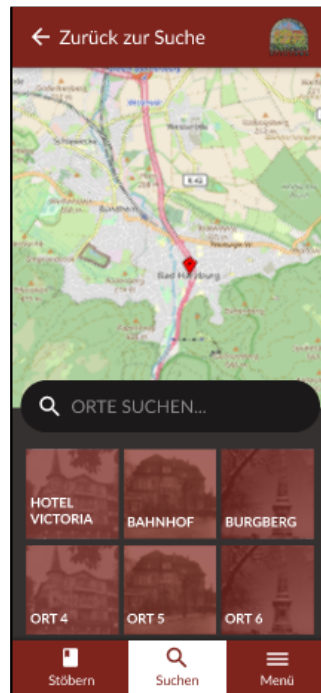


Figure 3.4: Map prototype

people, and other information. In the end, one extra input field for the name of the commentator was concluded to be very important to present ownership of the comment. The map idea was regarded as very interesting, especially for users from Bad Harzburg.

### 3.1.2 Design Details Visitor View V0

Based on the feedback on our prototypes, we started to implement V0. In this section, we display a selection of design details of V0 that we then tested for usability. Our focus in this chapter and the usability testing is set on the searching and commenting yet we also describe the browse view shortly. The search view (Figure 3.5) contains a search bar and two horizontal scrollable containers with clickable tiles. When clicking one of the latter, views with pictures connected to the selected tile keyword are displayed. Inside this view, one can search via the search bar.

**Search Bar** The search bar is presented with a filler text “search for keywords” (“Stichworte suchen”). One has to press the “Enter” key to start a search with the search term. When a search is started, a view with the result pictures or with no results is displayed. The search bar stays at its position.

**Search logic** As already stated in the description of the prototype design, we decided on a restricting search. When entering search terms, the result pictures are those that match each of the terms. When a search is started the matching pictures

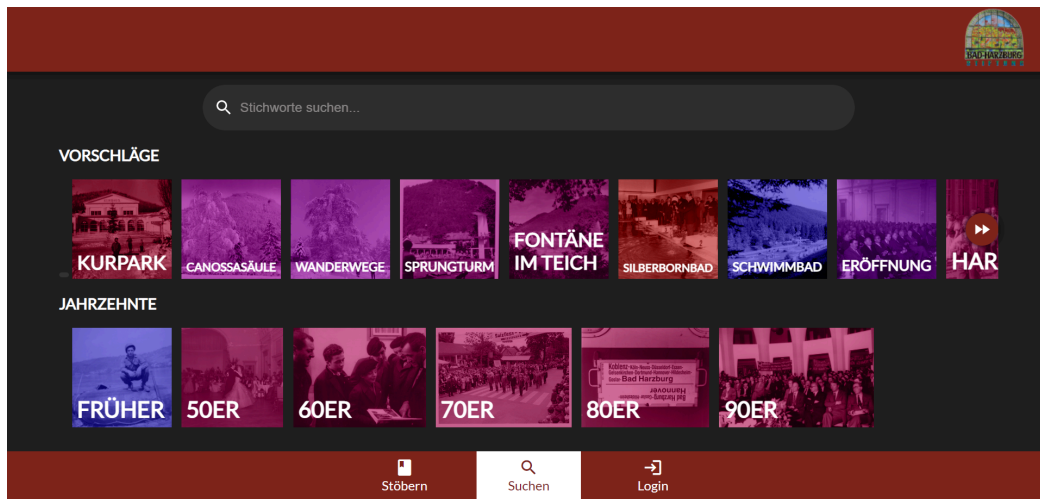


Figure 3.5: Search view in V0

or no result is displayed and one can search again inside this view adding more restricting terms. We call this a nested search. When a user wants to start a completely new search they can go backward via the “back” (“Zurück”) button until they are on the start search view again or they can click on the “search” (“Suche”) button in the Bottom Bar. The only difference to the design of the bottom bar from Prototype 3 (Figure 3.3) is that the button “Menu” is exchanged with “Login”.

**Horizontal Containers** As in our mobile design prototype, two horizontally scrolling sliders are placed below the search bar (Figure 3.5). The user can navigate through the keywords using buttons on the right-hand and left-hand sides. One slider displays suggestions that change dynamically with the search term the user writes into the search bar. Each time the search term is altered, the suggestions fitting to the search terms are reloaded and only tags that contain the search term or “No suggestions found for this search query...” are displayed. The other slider contains decades from the 50s to the 90s and an earlier decade tile is presented allowing users to use the chronological order. The original design also included person and place sliders but was not implemented at this point due to time constraints.

**Commenting** The visibility of the comment field was a very important requirement for our customer. The archive is meant to be a way to save and create knowledge and depends on input from users via a comment section. Yet, on the old website, the comment function was rarely used due to very bad visibility and usability. Therefore, we decided to put the comment section in the same side view as the picture info and give them a more present position that is not only reached via clicking a small icon (Figure 3.6). Following our client’s statement about the accountability for comments, we put a name input field on the top of our comment input (Figure 3.6).

We decided on a single field for the input since a form would take too much, potentially unused, space. Especially with the picture information and already existing



Figure 3.6: Picture information and Comment section

comments also visible in the picture-info view. Even more so as in the mobile view space is scarce. Since the existing picture information is extremely important for every user and also for the content of potential comments, they are at the top of the view.

**Browsing** Browsing is another use case that has to be supported by our application. As already described in the mobile view mockup, we constructed one view to display albums as tiles with their text and a picture belonging to the album as a background picture. Each album can contain several sub-albums. This structure is like the structure on the old website and also shows the album description text.

### 3.1.3 Design Details Curator View V2

Our focus for the usability testing was on the regular user interface. Yet we also wanted to receive feedback on our curator interface. Therefore we held an interview three weeks after introducing the curator interface to him. Before displaying our insights into this we will depict the most important design aspects.

**Curator Interface** One important design goal for the curator interface was to integrate it into the public website and to unify the views. A curator can now simply log in from all views. Every view that is available freely accessible also exists when logged in only with curating options. Furthermore, some views like the new upload view are only accessible when logged in.

In both search and browse view pictures can be deleted by the curator. In the browse view, the upload feature is now available. Figure 3.7 shows the public view of the collection “Gestüt 1966 -1988” and a view with the logged-in state with the new upload feature.

### 3 Evaluating Design Decisions regarding Usability

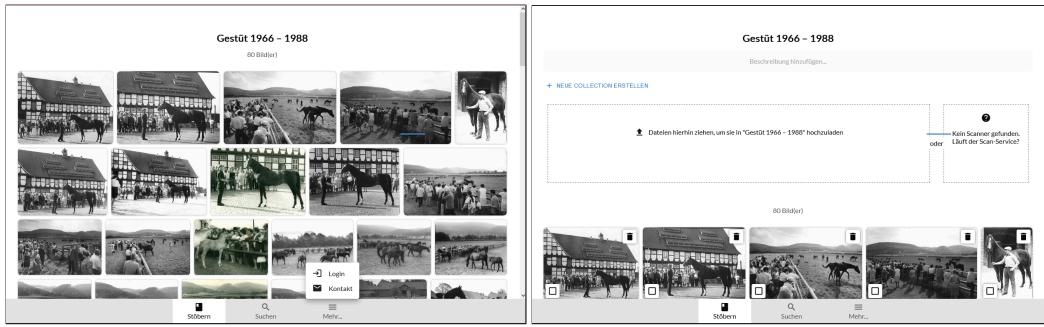


Figure 3.7: Browse album view V2, album view curator V2

#### Upload and Scan

One of the most important aspects of curating is the uploading process. It should be possible to load a set of pictures on the website or even better allow the curator to directly scan the pictures into the web archive. We created a upload interface accessible from collections and via a new upload view. The curator can now directly scan and upload pictures in one application. They can also group pictures by scanning and uploading them together. From here the picture information can be maintained. We implemented the scan functionality only for the Plustek Z3000 used by our client.

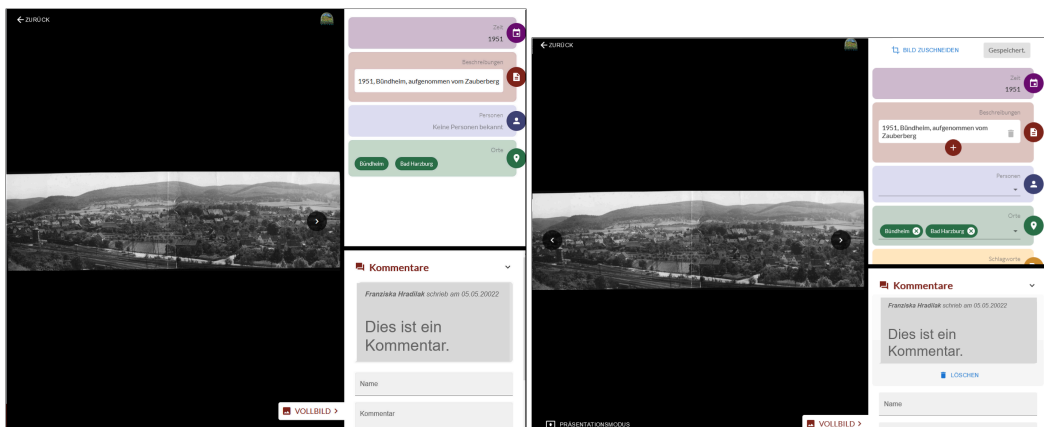
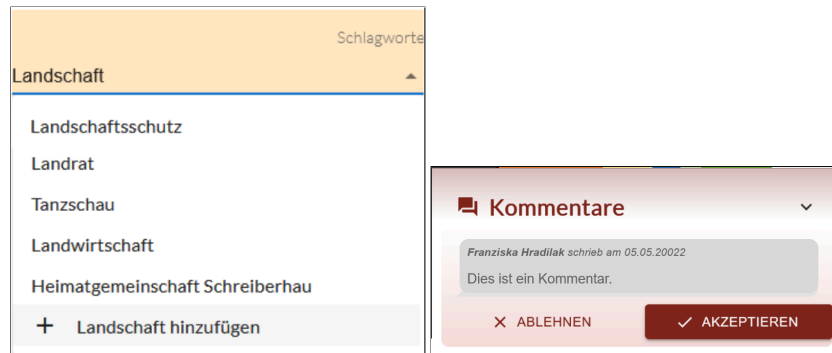


Figure 3.8: Picture view. Visitor interface and curator interface

#### Picture Information and Editing

The picture view with editing functionality opens whenever the logged-in curator clicks on a picture either found in the system or just uploaded. This allows a consistent workflow of scanning and entering the knowledge to the picture keeping both functionalities close in one application. Figure 3.8 shows the picture view in the curator version with log-in. Each tag type contains a text input field with a suggestion drop-down presenting existing tags that fit the input (Figure 3.9). Person, place, and keyword tags can be created directly inside the picture editing view, too (Figure 3.9).



**Figure 3.9:** Tag selection, comment in curator view

Below the picture details, there is the comment section that contains the unrevised comments that are yet to be published. The curator can now choose to accept or decline (Figure 3.9). They can also edit comments (e.g. misspellings), both published and not published (Figure 3.6). A comment can furthermore be deleted after being accepted. Those functionalities were designed and implemented following our client’s needs and requirements of a comment section that can be moderated.

## 3.2 Empirical Evaluation

In this section, we present the testing process, methods, and findings. We start with the testing of *V0* with Heuristics and Pilot Testing, followed by the changes we introduced in *V1* and the user testing with Bad Harzburg citizens. The evaluation of *V1* testing contains two aspects: qualitative testing and quantitative testing. In this process, we focused on evaluating the usability of commenting and searching with first-time users. Therefore we designed tasks that included searching for pictures with given information and adding comments. We take a look at the curator interface interview and finally discuss limitations.

When testing for usability, we first have to define what usability means. We use the definition from Nielsen [29]. Usability is “a quality attribute that assesses how easy user interfaces are to use. The word “usability” also refers to methods for improving ease-of-use during the design process” and is further composed of learnability, Efficiency, Memorability, Errors, and Satisfaction. We focused on learnability, Errors, and Satisfaction in our tests. They tell how intuitive and easy a first-time user can complete basic tasks, how often errors are made and how good a user can recover from them, and finally how pleased the users are [29]. Efficiency and Memorability are both not referring to the first intuitive use of the design and thus not applicable for this test that is conducted with first-time users.

### 3.2.1 Heuristics

Starting our User Testing, we wanted firstly to receive an own impression of our user interface. To get an established approach, we chose Jakob Nielsen's 10 general principles for interaction design.<sup>1</sup> It is one of the most popular and recognized methods in heuristic evaluation. The ten heuristics should be checked by experts against the system. Thereby, it is recommended to let multiple people conduct this evaluation and compare and discuss the results. Therefore, two of us checked the ten guidelines against our system separately. Then we compared and collected our findings.

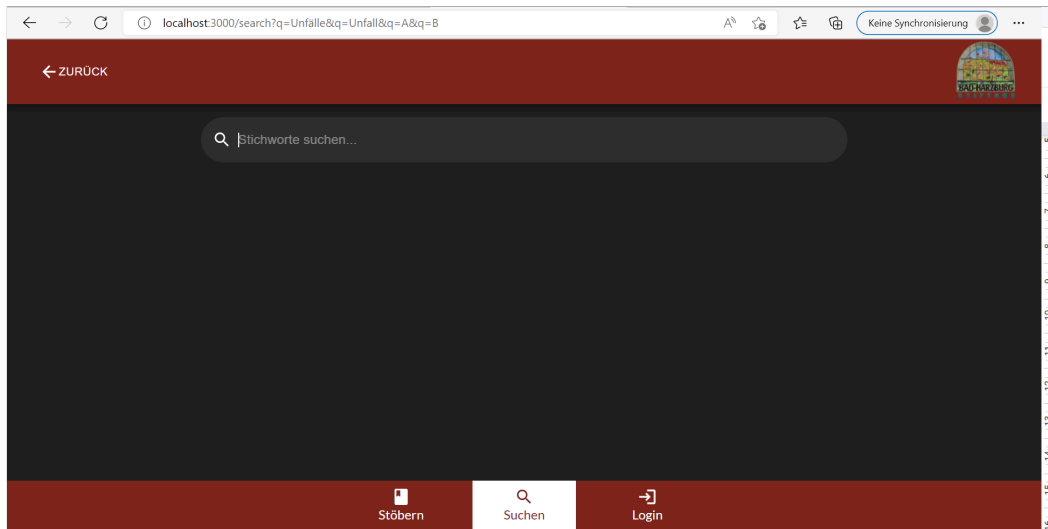


Figure 3.10: Empty search

**Insights** The first heuristic is about the “visibility of system status”. When e.g. pictures are loading a status information text (“Lädt...”) is displayed. In other cases, our application breached this heuristic. When a search is started that does not have result pictures, an empty black screen is shown (Figure 3.10). Further information about why there is nothing to be seen on the screen is missing. Also, the search bar is still displayed. And this is potentially harmful to the usability since following our restrictive search logic, from the first no-result search term onwards any other search term will result again in a no-result search. A user could enter search terms infinitely with the same black screen as an outcome. Following the 5th heuristic “Error prevention” the possibility of an infinite empty search should be prevented by e.g., making another search in an empty search impossible.

In the status quo, the search terms were only visible in the address bar, completely detached from the search bar and the focus of a user. Both problems can also be

<sup>1</sup><https://www.nngroup.com/articles/usability-testing-101/> (last accessed: 2022-07-13).



accounted for by the 9th heuristic “Help users recognize, diagnose, and recover from errors”. With no information about what was entered and why the empty screen is shown our system can leave the user confused. The not visible search terms are also problematic following the 6th heuristic “Recognition rather than recall”. Users should not have to “remember information from one part of the interface to another” so they should not have to remember what search terms they entered before.

Another hindrance was the inconsistencies caused by the design of the search view. The suggestions (“Vorschläge”) just below the search bar connote having the same meaning as search terms entered into the search bar. Sometimes when there are no fitting suggestions for the search term when starting the search with “Enter” there are result pictures displayed. The other way around it is also possible to receive no results when searching via the search bar but there are pictures displayed when you click on one suggestion. Also, the same search term in the search bar and the suggestions can yield different results.

After this first evaluation, we decided on a set of adjustments we could implement in a short time, before creating the tasks for our user testing. When no search result was produced, we removed the search bar and added the information text “There is no image matching this request” (“Es gibt kein Bild, das dieser Anfrage entspricht”). Also, an information icon with explanatory text on how the search is working was added. The wording in the search bar was changed to “search in image description” (“in Bildbeschreibung suchen”) to give a better understanding of what our search does.

### 3.2.2 User Testing

While performing the user test, one participant and two team members were seated together. One team member was operating as moderator, the other as recording clerk. The moderator was giving instructions and the recording person was noting the workflow of the participant, their verbalized thoughts, and their answers to the evaluation questions subsection A.1.4. Our participants could choose between a laptop and a tablet as test devices to make them perform the test on the one they felt more comfortable with. To receive insights into the participants’ relation to Bad Harzburg, their age group, and the use of technical devices, the participants filled out a questionnaire at the end of a testing session subsection A.1.2. To create representative test tasks we oriented ourselves on our domain and user analysis.

### 3.2.3 Pilot testing

Before testing in Bad Harzburg we conducted pilot tests, with three participants in their early 20s from our peer group, to validate our test tasks and to see if our website was ready to be tested. We learned that we had to make adjustments to our website before we could start the real test. To get a better feeling of whether our tasks would also work for participants from Bad Harzburg we conducted another pilot test with one participant in Bad Harzburg.

**Findings** Concurring with our self-evaluation the search was the biggest source of inconsistency and confusion. Our changes made before the pilot testing were not enough to fix the problem. Most of the time participants would not use the search bar as intended and only use the suggestions. Very interestingly the participants started searching multiple terms that all could fit onto the searched picture in one entry. The problem was that our search took the whole search entry as one search word and queried for it in the same sequence in the description texts. Those search entries always led to empty searches. This behavior was irritating yet they managed to change their strategy and enter only single terms. It also led to inconsistency as two terms term1 and term2 exist so that a search with “term1 term2” yields no results, yet when entering the first term1 and then after submitting the search entering “term2” there are pictures displayed.

**Adjustments (V1)** Based on Findings from our first evaluation steps we prioritized necessary changes to our website in tasks of which we present three.

We made it possible to search with multiple terms by using AND joins between each separate word from a search entry. Now one gets the same search results when entering “term1 term2” into the search bar as entering “term1” and “term2” successively. And this inconsistency is fixed.

To improve the search bar “suggestions” usability we adjusted the design combined with a change in the search logic. Therefore we changed the name of the “suggestions” into “Stichworte” (“Keywords”) and added a “Suche absenden” (“Submit search”) button into the search bar. Furthermore, we changed the search functionality so that not only description texts would be scanned but keyword tags, time range tags, place tags, and person tags too. Thus we wanted to test whether the perceived inconsistency could be solved ( Figure 3.11).

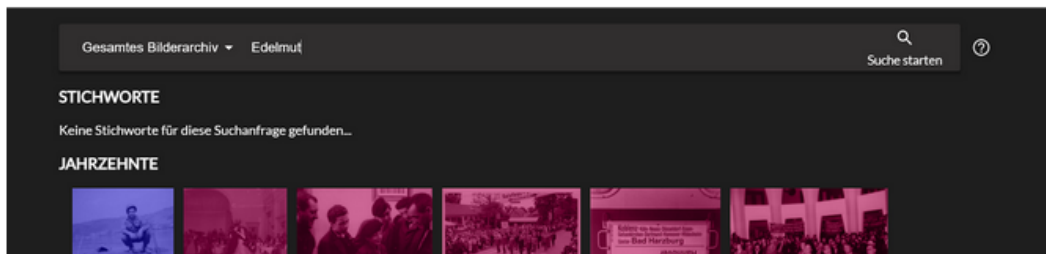


Figure 3.11: V1 search view

Addressing our findings, and also emerging from explicit participant wishes, we made the search terms visible in the form of Breadcrumbs shown below the search bar as was already included in our Prototype 1 (Figure 3.12).

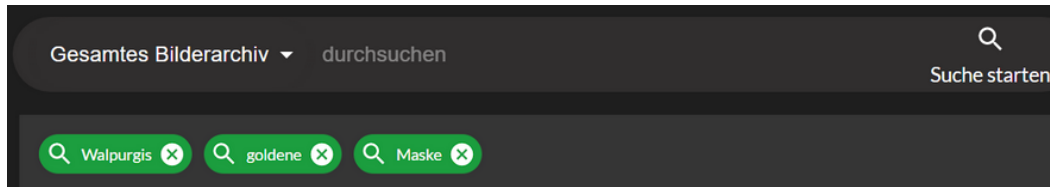


Figure 3.12: V1 search bar and breadcrumb bar

### 3.2.4 Qualitative User Testing with Think Aloud

Our first approach to usability testing was qualitative user testing. Therefore, we chose the think-aloud technique with seven tasks subsection A.1.1. In a thinking-aloud test, a participant is given a representative task to solve. While doing so the participant should “think aloud” in other words put their thoughts, ideas, and problems into words. Thinking aloud is an easy-to-use and cheap testing method. Furthermore, even with small sample sizes, we can get valuable insights and according to Nielsen, the qualitative approach is more robust against methodological errors than “statistical” studies [44].

**Walkthrough** At the beginning of each test session, the moderator explains the project’s purpose. Each task begins with a motivation (see subsection A.1.1) that the moderator reads out. The protocol clerk notes the user’s actions, and whether the user was able to finish the task within the time frame of 10 minutes. After each task, the moderator asks a set of evaluation questions inspired by “User Interview Example Questions” from the Yale University website.<sup>2</sup> In the end, the participant fills out the SUS questionnaire and then gives overall feedback to the system with a set of questions subsection A.1.2.

**Study Population** The seven participants (P1–P7) from Bad Harzburg were all organized by our client from his acquaintances and friends or his acquaintance’s staff. We tested with 4 women and 3 men. As can be easily seen, our study population from Bad Harzburg is unbalanced regarding age group representation. Our main group are 60–70 years old people with 5 participants, 20–30 and 40–50 only have one representative and 50–60, younger than 20 and older than 70 representatives are missing in our study. We have thereby a mismatch between our measuring method and the Bad Harzburger population, yet Bad Harzburg has a strong tendency toward an aging society as can be seen in chapter 1. Our study focuses on participants aged over 60 and corresponds thereby with the general direction of our user population.

**Insights** Firstly, most of the participants were highly interested in the application and did express their liking of the overall look and design. Especially the combination

<sup>2</sup><https://usability.yale.edu/understanding-your-user/user-interviews/user-interview-example-questions> (last accessed: 2022-07-13).

of the search bar with the horizontal tag selection below was well received and said to be very useful since it gives flexibility and inspiration on what to look for.

#### **Search**

We could confirm the positive impact of our adjustments to the search view and the perceived inconsistencies. When formerly participants would not start a search since they didn't understand they had to use the enter button, now the participants understood that they had to use the "Suche starten" ("Submit search") button or use "Enter". This could also be impacted by the former "Vorschläge" ("suggestions") now being renamed "Stichworte" ("keywords"). Searching not only in description texts but also in keyword tags combined with the understanding of the users that the "Stichworte" is not equal to searching via the search bar was successful. Before testing in Bad Harzburg we did not know that our restrictive search logic would be different from the users' expected behavior. They did not understand that each term narrows the result down. This led to participants entering almost the right terms but one term that did not match any photo and so they received an empty search result. Only one participant got close to understanding the search concept and concluded on their own that one has to enter fewer search terms and then possibly add more (P5). Another problem was that sometimes participants did not notice misspelling words and therefore did not get any search results. The nested search was sometimes problematic. When entering a search term and then searching in those search results some users were confused why they did not get pictures for even simple and common terms like "Pferd" ("horse"). We learned thereby that the breadcrumbs were not visible enough. Yet after some time, most participants understood where this problem came from and tried their search again from the start search view. A problem our users could overcome was connected to the breadcrumb bar and search bar design. Many users clicked into the non-clickable breadcrumb bar that is below the search bar until they understood that the search bar is above. We think that this problem is caused by the similarity and closeness of the two bars (Figure 3.12).

#### **Comment Section**

Almost all of our participants were using the comment section with ease. P2 stated that even "a blind man could do that [using the comment section]" ("Das kriegt ein Blinder hin"). Some participants did not find the send button and tried to send the message via "Enter". Therefore the comment send button has to be revised.

Interestingly our participants actively and confidently used the horizontal tag bars. Contrary to our concern those horizontal bars are usable but it must be said that some participants found the horizontal design to be a bit cumbersome and slow to use. Some users wished for a vertical design or faster navigation through the tags.

#### **3.2.5 Quantitative**

Even though qualitative testing is our main focus for usability testing, we also chose to add quantitative testing methods. While performing the tasks, the recording clerk

was tracking the time and of course success or outcome of each task. And as an approach to measure usability, we used the System Usability Scale (SUS) – a standardized questionnaire that we handed out at the end of each testing session.

**System Usability Scale** The SUS is still one of the most used methods to quantify perceived usability, even though it has already been published by John Brooke in 1996 [8]. By surveying over ten papers, Sauro and Lewis found that a variety of studies have “provided evidence of the validity and sensitivity of the SUS.” Furthermore, Bangor et al. [5] and Lewis & Sauro [53] consistently found “the SUS to have reliabilities at or just over 0.90” and that, as such, this value “exceeds the typical criterion of 0.70 for measurements of sentiments” [35]. It is not specifically made for the evaluation of websites or digital systems but can be applied to technology independently. It is built of ten questions with five Likert-style chapter 7 responses, from strongly disagree (1) to strongly agree (5). The questions are formulated alternating in a positive and negative tone. From the value of the answers, the so-called SUS Score can then be calculated per participant. The SUS scale is a percentile interpretation that ranges from 0 to 100 with 100 being the best possible score.

We used the German translation of the SUS created by a crowdsourcing project initiated by Wolfgang Reinhardt [51]. Concerning the sample size, the information about what size is necessary to conduct a SUS evaluation is scarce in the literature. In general, the SUS is said to be “robust even for small sample sizes” [49]. Since our sample size was only seven, this was a beneficial factor for our use case.

We choose the SUS for being fast and easy, established, and applicable to small sample sizes, but also finally since the SUS allows us to receive a numeric value that gives a measure of the perceived usability of our application. This value allows comparison. We can compare our application to a large number of evaluation results for various applications. And we are enabled to easily compare our application to future versions and testing. We applied the SUS to support our qualitative evaluation or to reveal a mismatch since the SUS score is independent of the qualitative testing.

**SUS Evaluation Methods** Originally, a SUS score was “used primarily in one-time, isolated tests to determine a single usability [...] score for a given product or service” [5]. Here, the single SUS score was intended to be seen as a percentile value. Decades of use and research with the SUS allowed the creation of further evaluation methods based on collected data. For example, an empirical new percentile evaluation based on SUS study data was created. The average SUS value was found to be 68 [9]. To choose evaluation methods, we investigated various proposals that exist due to the long-time existence and wide use of the SUS. In the following, we describe three methods we have chosen. We focus on mappings that use grade scales or adjective scales. The adjective scale [6] was created to present an empirically created scale. While conducting SUS evaluations, an eleventh extra question was added to the SUS to evaluate the user-friendliness of the tested product since according to Bangor et al. user-friendliness “is a widely known synonym for the concept of usability” [6]. In contrast, a relative approach to grading was proposed by Sauro and Lewis [53] with data from 241 studies to evolve a curved grading scale. As summarized in

their meta paper [35] they created their scale according to common curved grading scales where a SUS score of 68 forms the center of the grade “C” range (equal to a SUS score from 62.7 to 71.0). Both the highest and lowest 15 percentile points are representing the A and F ranges, respectively. A cumulative percentile curve allows visualizing the comparison to other studies’ SUS values. Therefore, we use an open-source analysis tool provided by the Mixality Research Group that visualizes data from SUS studies.<sup>3</sup> For the cumulative percentile curve, they used data from Sauro of over 446 SUS studies [53]. An evaluation per question is not recommended by Brooke since “scores for individual items are not meaningful on their own” [8]. And this caution is confirmed by the factor analysis of Bangor et al [6]. We selected the evaluation methods to give a better understanding of the SUS results to rank and classify our application’s score.

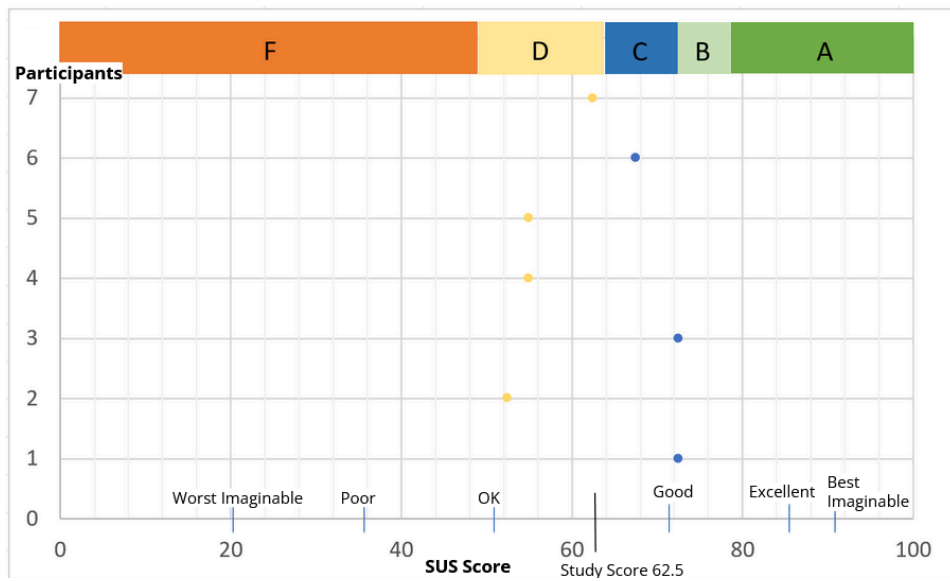
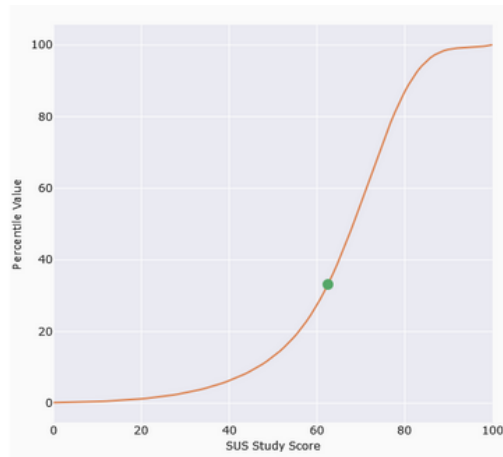


Figure 3.13: SUS results with grade scale and adjective scale mapping.

**Insights** With our study mean of 62,5 called the SUS score we get the results shown in Figure 3.13. Even though our study’s mean is below the average we receive an adjective mapping “OK” interpretable as an acceptable score. The grade mapping and percentile curve (Figure 3.14) give a more clear insight. Our study’s mean corresponds to a D on the Grade scale displaying that our SUS score is one grade step below the average C. The cumulative percentile curve visualizes that our SUS score is only at the 33,156th percentile. This hints toward significant usability problems

While the participants answered the SUS we asked them to explain their choices. In particular, the search and its confusing logic were stated as a reason to give a

<sup>3</sup><https://sus.mixality.de/> (last accessed: 2022-07-13).



**Figure 3.14:** SUS percentile curve

worse rating. One participant even stated “I would rate much higher if not for the search. That was very confusing. There should have been pictures” (“Ich würde es höher bewerten wenn das mit der Suche nicht wäre. Weil da hätte es ja dann Bilder geben müssen”). Thereby the SUS result supports the qualitative findings.

### 3.2.6 Curator V2

Our focus for the test was on the visitor interface usability. To also receive feedback on our curator interface we held an informal interview three weeks after introducing the curator interface (V2) to our client and frequent use by him. He was the only curator and therefore, his insights were especially valuable.

**Insights** The feedback was overall very positive. He stated, that the difference from the old uploading process is a “quantum leap” (“Es ist ein Quantensprung”). The former publishing process took various application switches, often led to errors and thus made him repeat work steps often. Our new interface reduces the necessary process steps drastically and the context switches between different applications to zero. This makes his work very “fluent”. Meaning that it is considerably faster, easier, and safer since he does not have to work with different information sources but directly saves the information to the pictures in the system. Now he also sees explicitly when information is missing. A further idea would be to have a bulk view where the curator could mark pictures as “with unsure information” to later have a better overview of where he still has work to do.

He is convinced that other people are also enabled to work as moderators or curators with our interface. This is a very important aspect since he wants to recruit many new people to help him with his workload. Also according to our client, our website is highly interesting for other photo archives. For further features, he wishes for a bulk curating tool, that allows him to add information, e.g. to pictures from the same album and thus with the same info text, to multiple photos at once. The

search bar is also available in the curator interface and is an important regularly used tool for his work. He has no problems using it. What he thinks to be missing is a “start a new search” button above the search bar that makes the already existing shortcut in the bottom bar more visible. He gained this insight from feedback from his acquaintances. As an outlook, we want to also add other archives to our system. Our client said, that again accountability is of utmost importance for the archive owners and that we have to find a way to make their names or archives’ names visible and attached to the pictures.

### **3.2.7 Limitations**

Our study design and also conduction held some limitations. Our study population is following a tendency in the Bad Harzburg population yet, it would have been interesting to also include participants from the age group 70 and older. Those contemporary witnesses can tell stories about times even longer ago and it is also important to make the application usable for them. Also, it would have been interesting to have more participants from younger user groups to avoid misrepresentation. Most of the participants were living in Bad Harzburg for more than 30 years, some also having been born there. Our participant group represented the main and most expected user group – retired locals from Bad Harzburg. This user group is likely to be able to contribute information to the pictures. Yet, the user group that is younger and or has no direct and long connection to Bad Harzburg and its history was not represented in this user test. Another aspect is the selection process. Most of the participants are directly acquainted with our customer and thus come from a relatively closed socio-demographic group.

For further testing with the SUS, the translation proposed by Gao et al. [21] might be more suitable since this translation was empirically tested.

In an early stage of our product, we targeted a mobile-first strategy caused by our client’s statement about mobile use in older age groups. We later focused on a unified mobile and desktop view, since our customer relativized the primary use of mobile devices by elder people, and we had the requirement to build our website not exclusively for elder people. We developed the application primarily focusing on the desktop view, adjusting the mobile view afterwards if necessary. Every participant uses mobile phones daily (subsection A.1.2) but we did not test our application in the small mobile format. When we started our testing the small mobile version was not usable. Due to time constraints, we first focused on making the version for larger screens such as PCs or tablets usable with new alterations. After testing V1 we fixed the mobile view. We also had a relatively small participant group size in contrast to at least three possible device groups (desktop, large mobile, small mobile). Allowing participants to choose between a laptop and a tablet also reduced variance between test runs.



### 3.3 Discussion

Based on our insights from our user studies and the interview with our client in the following, we discuss whether our requirements were met. Our first requirement (u.1) is good usability, so the focus of our evaluation was set on usability. We, therefore, take a look at the qualitative evaluation regarding the usability definition and the SUS evaluation.

The learnability is built on the ease of use and intuitiveness of basic tasks. Therefore, we tested with first-time users. A task almost every participant fulfilled easily was commenting. All of the participants had no problem finding and writing in the comment section. The only problem participants had was understanding how to send the comment. The comment section, therefore, presents a great improvement to the legacy application, yet further adjustments still have to improve the learnability. Our requirements of the comment section (u.1) are therefore not completely fulfilled. We assume the design with horizontal sliders to be intuitive to use since no participant had problems using them. Yet, not all participants were satisfied with them. P4 especially wished for a revised design.

The most revised designs on our application were connected to the search. We achieved an improvement when adjusting the search bar design and the suggestion slider before testing in Bad Harzburg, allowing participants to easily and intuitively understand how to start a search. Yet, the main usability problem was still unsolved. Our users are used to popular search engines like Google and Co. to enter many search terms that could fit a searched media (picture or article or other content). In these sentence-like entries not all terms have to necessarily fit the wanted search result but the best matching result, with most terms (possibly in the right order and so on) matching will be displayed. With lesser matching results displayed farther below or on other pages. Thus, they expected the search to almost always yield a result. Especially when they knew, or thought to know, that the archive must contain pictures for a topic. This happened for example with the search "Walpurgis goldene Maske". Walpurgis is a very popular and major event in the Harz. Thus a search in a press photo archive from the area containing the term "Walpurgis" has to result in pictures being displayed.

To address this problem we discussed two possible directions. One would be to try to get closer to what our users would expect and try to get closer to Google. And following Jakob's Law [14], this would be the best solution. Our search concept works contrary to the mental model of most users. According to Jakob Nielsen, the mental model represents how we expect a system to work and is founded on experience. And since the Google search logic is so omnipresent it is not recommended to introduce a new concept. Following Jon Yablonski what we found out in our user testing was to be expected to create a usability issue. Yet due to our limited time and the great effort, we would have to put into this implementation – the challenge already starts with the question of how the Google search even works – we chose the second option: The clarification of our search concept via more explicit messages. When a user enters an invalid search there should be an information text explaining that there are no pictures that match all of these terms.



**Figure 3.15:** Search with V1 with empty result

One idea is to display how many pictures are matched to a search term displayed as a breadcrumb. This way users could see which search term they have to remove to receive search results. All in all for future work on this project (with enough time on their hands) we highly recommend redesigning the search logic.

**Error** When making the search bar and empty-search design clear and searching for a no-result search impossible we reduced errors that happened this way and gave more user responses to the actual system status. When for example no result pictures are found this is communicated to the user. On the other hand, again the search functionality and the breadcrumb design are error sources. Participants did not understand the restrictive search and were thereby confused about what they had to do to receive result pictures. Also, the breadcrumb design and concept of a restrictive search were not intuitive. Therefore, some participants did not notice that they were still searching with old search terms or that the misspelling of search terms was the problem.

**Satisfaction** Even though the search problem could have left the participants unsatisfied with our website, almost all participants were pleased by the design and expressed much interest. The horizontal sliders were easy to use yet reduced satisfaction. The combination of the search and browse views and the color design was especially appreciated. On the other hand, the SUS evaluation hints toward usability problems that are also connected to the satisfaction of the participants. The participants might have wanted to be polite and not state their dissatisfaction more openly. It is also possible that we influenced the participants too much in the opposite direction and they answered the SUS more negatively than they would have normally. The SUS supports our qualitative findings otherwise. Especially the search logic poses usability problems. Therefore, the Grade D and the percentile evaluation match our general feedback. Whereas the received "OK" is closer to the general feedback from the qualitative evaluation.

Our test design did not contain small mobile devices. Only the usability for the desktop and bigger mobile versions were tested. Even though we could not perceive any difference in the usability of both versions the small mobile version still has to be tested to be able to conclude whether u.2 is met.

Our dual design with browse view and search view was created to fulfill the needs of the Browse User and the Search User u.3. Through our usability testing, we found that the search still needs improvement, but the general concept of both views was received positively. Specific testing regarding the browse usability should be performed.

### **Curator Interface**

From the interview with our client, we can overall conclude, that the curator interface satisfies his expectations and needs. We take a closer look at the usability and the other requirements. With the direct integration of the scanner into our application, the workflow and ease of use were drastically improved fulfilling p.5. We did not test the current curator interface version with a first-time user, yet our client states it to be easy to use and also believes, that new curators can understand the interface fast. Using the scanner will require an introduction for new users. Yet we can assume a general good "Learnability". By reducing context switches and removing the extra applications from the picture scanning process it did not only get faster but reduced error sources. Formerly picture information easily got lost when saving it detached from the pictures. Now our client can also specifically see which information a picture does have and does not have. Thereby faulty picture annotations can be more easily discovered and recovered from. The moderating of comments p.1 is already actively in use and approved by our customer. The curator interface usability is very good according to our client and both curator and visitor interfaces are well integrated, meeting the requirement u.1.

## **3.4 Conclusion and Outlook**

This chapter introduced design aspects and the process of validating and adjusting. We conducted usability testing with heuristics and the think-aloud method. While testing we collected qualitative feedback. A quantitative evaluation regarding the perceived usability was conducted with the SUS. We received important insights into usability problems, most importantly the search concept. We focused on the visitor interface yet we also displayed important design elements from the curator interface and the feedback. In the following, we will summarize important aspects of this chapter.

**Usability and Search** Even when our participants were overall satisfied with the visitor interface we still found major usability problems. Until testing with locals, we did not realize how substantial the search logic would be. We could see how important respecting common known concepts to connect to the users' mind map is. Due to time constraints, we then went the resource-efficient way of explaining our search concept more explicitly. Yet it would be interesting to explore ways to get closer to the familiar search concepts. Further enhancements such as the new search button would also be interesting to implement and test with participants that are acclimated to the application to get more insights about the learnability. Testing on the participant's device is also an idea to create a more natural testing environment.

**Comment section** In our user test, we wanted to know whether the participants were able to write a comment on their own. Gladly all of our participants found the comment form with ease and were able to write a comment, yet the comment send button needs revision. All in all, we created a great improvement compared to the

### *3 Evaluating Design Decisions regarding Usability*

old website. We did not test if different designs could motivate the participant to write comments, yet this could be interesting for future tests.

**Curator** From our client's feedback, we can conclude that our curator view presents a great improvement on the former functionality. Especially the direct integration of the scan process into our website creates immense ease of use. Together with the new possibility to add the picture information directly attached to it makes the whole process more efficient and also less error-prone. The most important future task from our client is bulk editing.

## 4 Architecture and Implementation of a Web-Based Frontend

In this chapter, we will examine the implementation of a crowd-sourced picture archive as a website. The main focus is on the frontend, starting with a detailed explanation of the architecture. To give a better understanding of the behavior of our system we then take a look at the data flow throughout the application. Further explanation regarding the backend follows in the subsequent chapter.

### 4.1 Technical Components

As we will discuss implementation details in this chapter, we first have to take a look at the general technical components we use. An overview of the main elements can be seen in Figure 4.1. In our backend we use a Content Management System (CMS), Strapi<sup>1</sup>, to administer our data saved in a PostgreSQL<sup>2</sup> database. To access the database Strapi uses the Knex Query Builder.<sup>3</sup> The frontend is developed using the JavaScript library React<sup>4</sup> and the client-server communication relies on GraphQL<sup>5</sup> with Apollo<sup>6</sup> as a supporting tool. As mentioned this chapter focuses on the frontend and the communication with the backend. Therefore, the server-side components will not get discussed here but in chapter 5.

### 4.2 Background

To understand the architecture and implementation decisions, we need to know about the used technical components. Therefore, we begin with an explanation of why we chose React and GraphQL and what their main concepts are.

---

<sup>1</sup><https://strapi.io/> (last accessed: 2022-07-13).

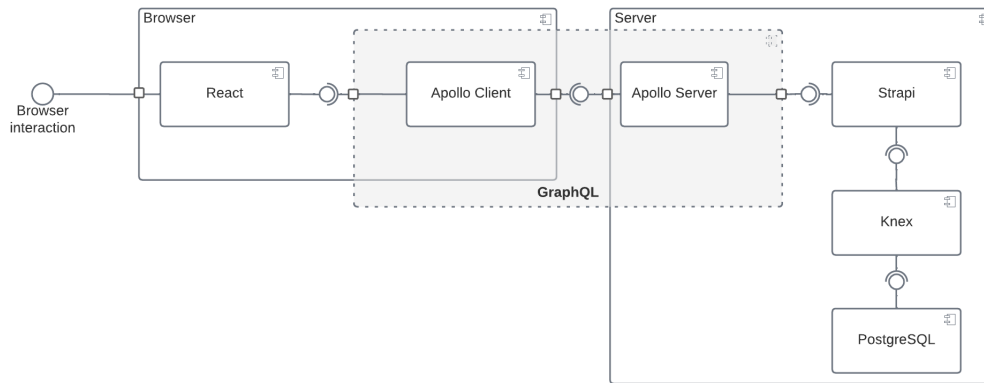
<sup>2</sup><https://www.postgresql.org/> (last accessed: 2022-07-13).

<sup>3</sup><http://knexjs.org/> (last accessed: 2022-07-13).

<sup>4</sup><https://reactjs.org/> (last accessed: 2022-07-13).

<sup>5</sup><https://graphql.org/> (last accessed: 2022-07-13).

<sup>6</sup><https://www.apollographql.com/> (last accessed: 2022-07-13).



**Figure 4.1:** The main technical components we used to implement the picture archive

### 4.2.1 React

When choosing a UI framework, in the beginning, we mainly based our decision on experience in the team. As the majority of us have never built a web application before, we decided to use React because two team members already worked with it in the past and could support the others in their learning process. Developing in React builds upon a few main characteristics that are less known from other frameworks, libraries, or programming languages.

**JSX/TSX** JSX(/TSX)<sup>7</sup> is a syntax extension to JavaScript(/TypeScript) which is recommended to use for development in React. It simplifies the combination of markup elements and logic and produces React elements that can get rendered to the DOM. JSX allows embedding any JavaScript expression by using curly braces. Moreover, JSX expressions can be treated like regular JavaScript objects and can be assigned to variables, returned from functions, and more. An example of JSX-Syntax can be seen in Listing 4.1.

**Listing 4.1:** Using JSX allows combining markup elements and JavaScript expressions

```
1 <div className='comment-container'>
2   {comments?.map(comment => (
3     <p>{comment.text}</p>
4   ))}
5 </div>
```

<sup>7</sup><https://facebook.github.io/jsx/> (last accessed: 2022-07-13).

**Composition** Composition is the core development pattern of React. The idea is to split the user interface (UI) into components that are independent and reusable elements. Contrary to the approach of separating concerns, each component combines functionality and markup. It receives an arbitrary input object called `props` and can have state. The UI element to be rendered by the component can consist of other components and plain HTML elements. The communication between different components is limited to the top-down direction because the props are read-only and the state of a component can not be accessed from other components, including the parent component.

**Function Components and Hooks** Function components are simply JavaScript functions that receive the props as arguments and return the elements that should appear in the UI. Listing 4.2 shows a basic example of a Function component. To use functionality like state or executing code on lifecycle events like initialization or unmounting, React introduced `hooks`<sup>8</sup> which provide a way to use these features in Function components. Similar results to Function components combined with hooks could have been achieved using Class components. These have slightly different syntax and already support state and lifecycle functionality. The main advantage of the hook system is that it is possible to write custom hooks that themselves can use other hooks. This allows extracting and sharing even stateful logic between different components which prevents code duplication or complicated workarounds. Furthermore, there are a few minor disadvantages of class components like the necessity to write more code to implement a class and problems with React internal optimizations<sup>9</sup>. Therefore, we decided to only use Function components in our application.

**Listing 4.2:** Simple Function component. The comments that get received as props get returned as elements to be rendered in the DOM

```

1  const CommentContainer = (comments: Comment[]) => {
2    [...]
3    return (
4      <div className='comment-container'>
5        {comments?.map(comment => (
6          <p>{comment.text}</p>
7        ))}
8      </div>
9    );
10 };

```

**Contexts** Sometimes we encountered use cases for which we wanted to share state or functionality between large component trees or maybe even the entire application. An important example is the authentication status. A lot of features involving curator

<sup>8</sup><https://reactjs.org/docs/hooks-intro.html> (last accessed: 2022-07-13).

<sup>9</sup><https://reactjs.org/docs/hooks-intro.html#motivation> (last accessed: 2022-07-13).

tasks should only be available after logging in. Various components in different parts of the application depend on their behavior on the authentication status. To avoid having to pass the same data in many branches through multiple layers of props, there are Contexts. Contexts allow sharing values for a whole tree of components. To make the data available there are Context-Providers which are components that can be wrapped around other components and get the values to be shared as props (see Listing 4.3). Every child component in this part of the component tree, no matter how deeply nested, can access this shared data by calling the useContext-hook (see Listing 4.4).

**Listing 4.3:** A Context Provider that receives the data to be shared

```
1 <AuthContext.Provider value={{ role, username, email, login, logout }}>
2   {children}
3 </AuthContext.Provider>
```

**Listing 4.4:** Every child component can access this data by calling useContext()

```
1 const { role } = useContext(AuthContext);
```

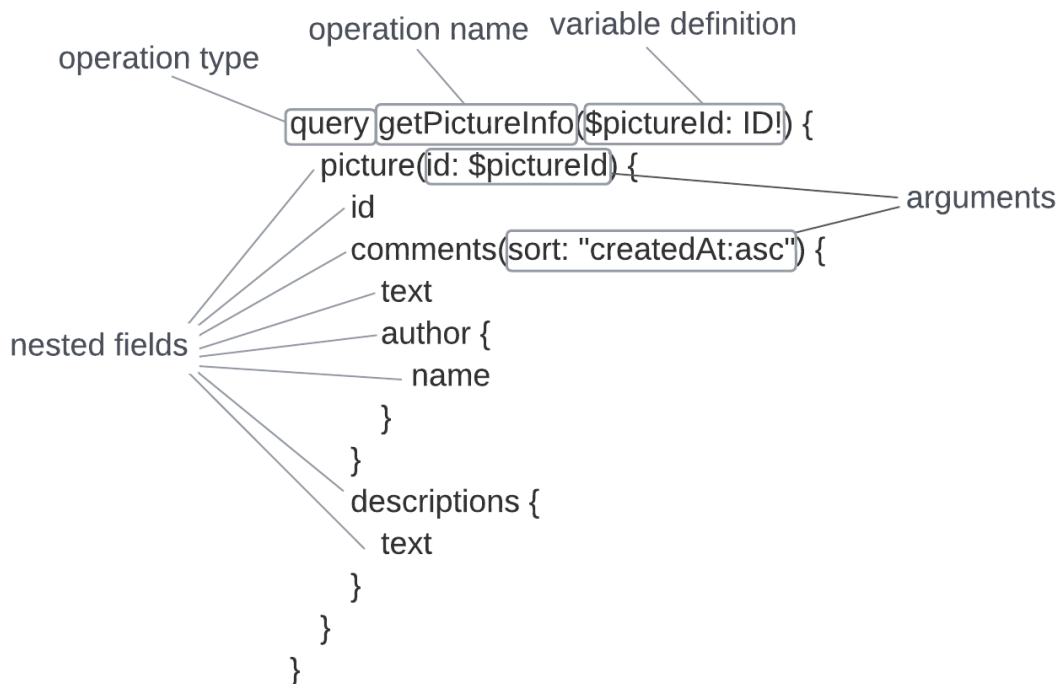
### 4.2.2 GraphQL

The React application needs to be able to fetch and edit data. As the frontend and backend are separated, the frontend can not directly access the database. Therefore, it has to call a backend API. Our CMS in the backend, Strapi, offers the two APIs REST [20] and GraphQL. In general, REST is the far more popular architectural style. Even though the team experience with REST was larger, we still chose to use the GraphQL API. This was mainly due to limits of the REST API in Strapi v3 further explained in subsection 5.6.5. While it is solvable through backend customizations, it was not possible to decide from the client side which relations to populate and which fields to select when sending a GET request. For example, we want to fetch the thumbnail URLs of a lot of pictures. Without being able to control which relations to populate and which fields to select, we would request large amounts of data per picture while the majority is unused. We have a lot of different fetches and with new features and requirements, we regularly have to add queries and adapt which fields are needed. Defining every field selection and population backend-side would mean that for each of these changes we would always have to adapt the frontend and the backend which is something we want to avoid. It is important to note that this is an issue that got resolved with the new version of Strapi. However, at the time we made the decision, it was not foreseeable when the new version would get released. Therefore, we decided to use the second API that is provided by Strapi, GraphQL.



GraphQL is unlike REST, not a collection of concepts but a query language for APIs and a server-side runtime for fulfilling those queries.

**Query Language** To interact with the API, a client has to build GraphQL operations that get sent to the server. A GraphQL operation is a string that defines which action should be performed and what data should be returned. It allows a specific and deeply nested population of relations and field selection. Each definition of a GraphQL operation consists of the following elements which can also be seen in Figure 4.2. The `operation type` defines what kind of operation is performed. There are two mainly used operation types: query and mutation. Queries are used to read data, and mutations to create, update and delete it. The `operation name` is an optional identifier that is comparable to a function name. `Fields` are used to specify which data we want to be included in the response. These fields can refer to scalar values or more complex objects. The latter requires nested fields as a sub-selection. Further, it is possible to select related objects which allow fetching data from a lot of different database tables in a single request. Each field, even deeply nested ones, can accept arguments. Which arguments can be passed is defined on the server side. Popular use cases are filtering, pagination or sorting. For these arguments, it is possible to use `variables` whose types have to get declared in the `variable definitions`.



**Figure 4.2:** An exemplary GraphQL operation definition

**Server-Side Runtime** GraphQL operations are usually sent via HTTP as POST requests to one central endpoint in the backend where they get delegated to a corresponding resolver. Resolvers are functions that back a field and are responsible for returning a result for that field. They usually call other resolvers or access a database. The backend system we use, Strapi, automatically generates resolvers for create, read, update and delete operations based on our models (see subsection 5.4.8). It is also possible to define further resolvers which then can be accessed as a field in operations. Which resolvers exist and therefore which fields can be called in operations get described in a schema which is provided by the server and can get requested by the client.

**Apollo** There are two Apollo services used in our system. The Apollo Server is used by Strapi as a runtime to provide the GraphQL API. The Apollo Client is a JavaScript library that we decided to include in our frontend as a helper for sending queries and mutations. It simplifies building requests and tracking loading, error, and network status, but most importantly we use the Apollo Client for local caching. As GraphQL operations all get sent as POST requests to the same endpoint, the URL can not be used as an identifier and therefore we can not make use of default HTTP caching. Therefore, we decided to use an external tool and chose the Apollo Client because the cache is highly customizable and there is extensive documentation.<sup>10</sup>

### 4.3 Frontend Architecture

This section gives an introduction to the frontend structure of our system. After explaining a few general guidelines we will take a look at the components of our React application, starting with a high-level overview. Further, we will examine the structure of one exemplary part of our application in more detail.

#### 4.3.1 Design Guidelines

Before talking about the specific components, we first take a look at the main ideas that we followed when implementing the picture archive as a React application. These are general rules or patterns that we used to keep consistency and simplify our structure. Some of these are common in React applications while others are specific to our architecture.

**Top-Down Data Flow** Emerging naturally from the concept of read-only props, the data flow in React applications is usually unidirectional from parent to child components. We do not use external libraries like Redux<sup>11</sup> to circumvent this. In our architecture usually, components higher up in the component hierarchy fetch

---

<sup>10</sup><https://www.apollographql.com/docs/react/> (last accessed: 2022-07-13).

<sup>11</sup><https://redux.js.org/> (last accessed: 2022-07-13).

information and pass them, sometimes partially, down to other components that further evaluate and display them.

**Views** In our architecture, Views are components that fill the entire screen except for the top and bottom bar. There is always exactly one active View that gets switched depending on the route. Each major use case is implemented as a different View. We have implemented seven Views that get explained in section 4.3.2.

**Providers** To spread data and functionality that is needed in various parts of the application we use components that we call Providers. Providers are wrapped around the entire application and utilize the `useContext`-hook introduced in section 4.2.1. Each object or function that is spread using the Context-Provider is therefore accessible in every component in the entire application. Currently, we use four different providers in our system (see section 4.3.2).

**Unified UI** To achieve a low entry barrier for contributors (see chapter 2), the interfaces for displaying and editing information should be as similar as possible. Therefore, we use for both use cases only one component that changes its behavior depending on the authentication status instead of using entirely different components or interfaces. Exceptions are curating features that are complex enough to require their own Views, like for example changing the collection hierarchy.

### 4.3.2 Component Overview

At the current state, our application contains almost 60 components. We will first take a look at the components that are at a high level in the component tree, including our Providers and Views. To give a better understanding of how Views are composed, we will then examine one View in more detail before discussing an exception from our design guidelines.



Figure 4.3: Composition of app

**Overall Structure** A high-level overview of our components can be seen in Figure 4.3. The uppermost component is the `App` which is composed of multiple `Providers`, the `TopBar` and `NavigationBar`, and a `View`. In the following, we will take a closer look at these elements. As mentioned, four important `Providers` wrap the entire application. The `AlertProvider` enables components to send temporary alerts to the user which can be success messages, warnings, errors, or just information. The `AuthProvider` grants access to login and logout functionality and the current authentication status. The authentication status is mostly retrieved to conditionally render components or alter their behavior based on the user's permissions. In the current state, our system supports two different authorization statuses, public, and curator. The `DialogProvider` allows displaying dialogs. Contrary to the alerts, dialogs require user action like a confirmation or entering input. When using a dialog a component can either define custom content or fall back to a preset. The `ApolloProvider` is a component that we did not write ourselves but follows the same concept as our `Providers`<sup>12</sup>. It uses the `useContext`-hook to provide access to the `Apollo Client` (see section 4.2.2) to send queries and mutations. These `Providers` wrap the `TopBar` and `NavigationBar` which include links to different routes. Between these two bars, there is a switch that renders a `View` depending on the route. In our application, there are currently seven different `Views`. There are three main `Views` available to an unauthorized user.

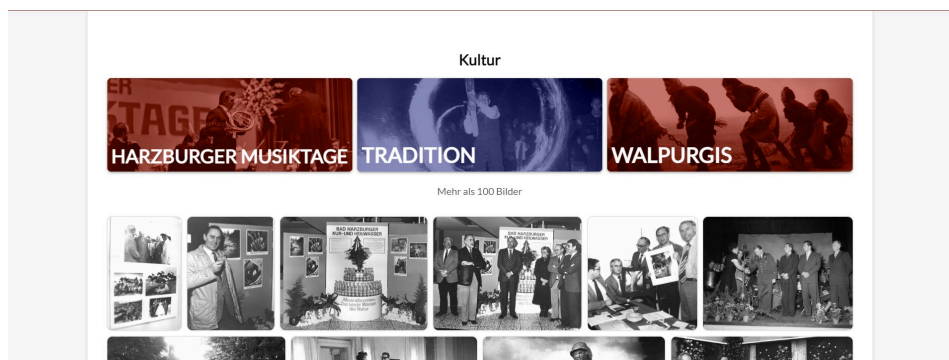


Figure 4.4: BrowseView

The `BrowseView` (see Figure 4.4) is the `View` rendered by default and displays pictures grouped in hierarchical collections. We will take a closer look at this `View` in section 4.3.2.

The `PictureView` (see Figure 4.5) displays a picture in full size and offers the possibility to write comments. Moreover, information related to the picture gets rendered in a hideable sidebar and is editable for authorized users.

<sup>12</sup><https://github.com/apollographql/apollo-client/blob/5a4b0ae202e46cf9368dd7b2fd1f0c3738f1810c/src/react/context/ApolloProvider.tsx> (last accessed: 2022-07-13).

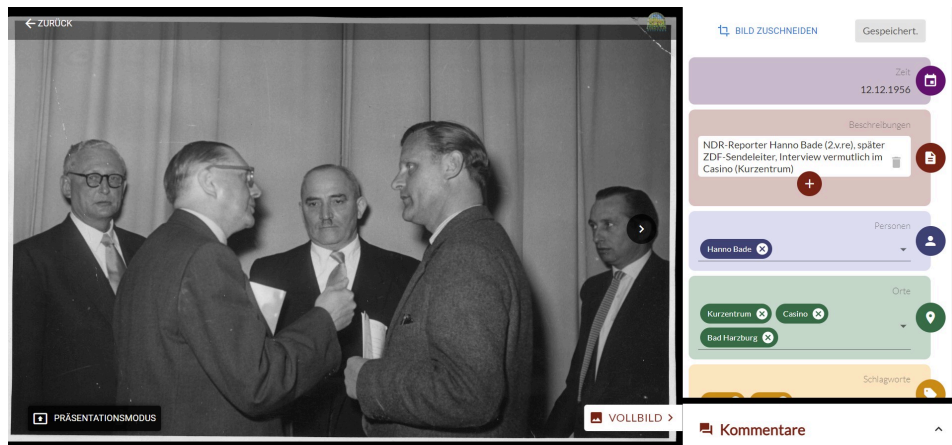


Figure 4.5: PictureView

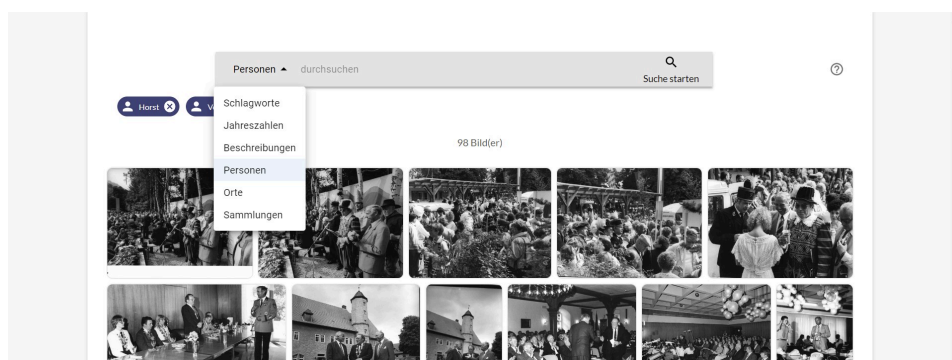


Figure 4.6: SearchView

The `SearchView` (see Figure 4.6) allows users to search for pictures that match specific information. Keywords entered in the `SearchBar` get passed on to the backend where they get converted to an actual search query. In subsection 5.6.4 we will discuss why we implemented this behavior. Moreover, four Views only provide curator functionality and are not visible to an unauthorized user.

The `UnverifiedCommentsView` (see Figure 4.7) provides an overview of which comments still have to be moderated. The component fetches all comments that have not been published yet, groups them by picture, and displays them in a table. When a user clicks on a row it opens a `PictureView` with the corresponding picture where the comments can get moderated.

The `CollectionsEditView` (see Figure 4.8) is an interface to revise the collection structure that is displayed in the `BrowseView`. The collection tree gets displayed in columns. Clicking on a collection adds a column to the right which includes its child collections. For each collection, buttons invoke mutations to rename, delete, add and merge collections. Furthermore, relations between different collections can be edited, deleted, and added.

The `TagTableView` (see Figure 4.9) is used to edit the actual tags, contrary to the picture-tag relations that get edited in the `PictureView`. The component fetches all






Noch nicht verifizierte Kommentare		Anzahl
	Bianca Goldmann: Obere Reihe: Zwischen Uwe Heinemann und Frank Thorenz; Ulrike Degener (nicht W) Mittlere Reihe: 2. v. links: Claudia Niggemann	1
	Jürgen Schneemann: Waldschwimmbad in Hohegeiss	1
	Hans Ulrich Dr.Bonewitz: Große Wurmbergschanze, Braunlage, im Hintergrund das Brockenmassiv	1
	Hans Ulrich Dr.Bonewitz: Große Wurmbergschanze, Braunlage	1
	Hans Ulrich Dr.Bonewitz: Große Wurmbergschanze, Braunlage	1

Figure 4.7: UnverifiedCommentsView

Das Herbert-Ahrens-Bilderarchiv	Personen von A-Z	B - Diverse	Hermann Baumann
Herbert Ahrens	Achterberg, Max Otto	Dieter Bandmann	+ Collection hinzufügen
Hotels & Gaststätten in Bad Harzburg	Adenauer, Dr. Konrad	Dieter Banse	+ Neue Collection erstellen
Jugend	A - Diverse	Dr. Bayer	Bestehende Collection verlinken
Kirche	Ahrens, Hermann	Günter Bängeroth	Bestehende Collection hierher verschieben
Kreuz des deutschen Ostens	Albrecht, Dr. Ernst	Hanno Bade	
Kultur	B - Diverse	Hermann Baumann	
Nationalpark	Becker, Anneliese	Jürgen Bartz	
Personen von A-Z	Benna, Lieselotte und Helmut	Mona Baptiste	
Politik	Berger, Vincenz	Rainer Barzel	
Salz- und Lichterfest	Berndt, Hermann	Rosemarie Bartels	

Figure 4.8: CollectionsEditView

tags of a certain type and displays them in a table. Each row provides functionality to rename the tag, add or remove a synonym, and delete the tag. Moreover, a user can select two tags and merge them.

The `UploadsView` (see Figure 4.10) can be used to upload pictures and edit their information before they appear in the `BrowseView`. The component fetches all pictures, that do not belong to any collection, and displays them in a grid below the upload interface.

**BrowseView** As most Views are relatively complex, they are composed of a lot of different components. In the following, we will take a look at the structure of our default View, the `BrowseView` (see Figure 4.11), to illustrate exemplarily how our Views work in detail. The `BrowseView` shows pictures grouped in collections. The component is stateless and consists of two major parts, the collection information, and a picture overview. When rendered, the `BrowseView` fetches and displays the collection attributes consisting of a title, an optional description, and the child collections. Which collection is fetched is determined by the URL path. Clicking on one of the child collections adds the name of this collection to the path and re-renders the `BrowseView` which now displays a different collection. Based on the loaded collection, the `BrowseView` builds a query filter for fetching pictures and passes it on to the `PictureScrollGrid`. The `PictureScrollGrid` is our main component to

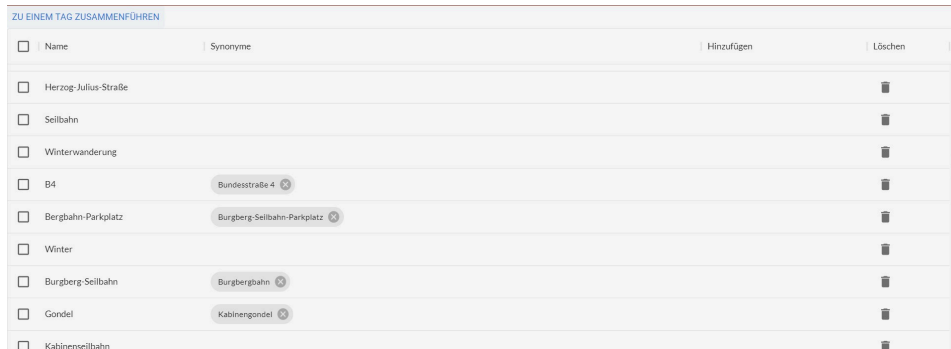


Figure 4.9: TagTableView



Figure 4.10: UploadsView

display an overview of multiple pictures and is also used in other Views like the `SearchView`. In order to make the component reusable in different contexts, the component receives a query filter as a prop and then fetches pictures from the backend using this filter. To reduce the network load this fetch is limited to 100 pictures and only if the user has scrolled to the bottom the next bulk of 100 pictures is fetched. The received pictures get passed to the `PictureGrid`, the component responsible for displaying the pictures. For authorized users, the `PictureScrollGrid` also includes a `PictureUploadArea` that allows uploading files or directly using the scanner (see subsection 6.3.2) to add pictures to the archive. Depending on where the `PictureScrollGrid` is used, the uploaded pictures immediately receive information from the context. For example, pictures uploaded in the `BrowseView` get the currently selected collection assigned which allows a curator to upload pictures directly into a certain collection. The `PictureGrid` renders each picture as a `PicturePreview` which displays a thumbnail and opens the picture in full size as a `PictureView` when clicked. The `PictureView` gets rendered as a child component of the `PictureGrid` which is an exception from our design guideline of only rendering one View at once. This is due to specific requirements which get discussed in the following section.

**PictureView as a Child Component of PictureGrid** The `PictureView` is used in two different contexts. A `PictureView` gets rendered when the `/picture` route is



Figure 4.11: Composition of BrowseView

called or when a user clicks on a picture displayed in a `PictureGrid`. These scenarios differ, as the latter involves additional requirements. After accessing a `PictureView` by clicking a picture in a `PictureGrid`, the user often wants to close the full-sized picture and return to the grid to the exact position where it was left to continue browsing. Our usual behavior of switching Views – replacing the components – would not be practical here, especially when the user already scrolled far to the bottom and possibly triggered further fetches of more pictures. There arise problems with saving and restoring the previous state of the grid which at least result in a delay for the user. Therefore, we deviate here from our principle of only rendering one View at once and allow the `PictureView` to be rendered inside of other Views. When a picture in the `PictureGrid` gets clicked, the state of the component changes and a `PictureView` gets rendered as a child component (see Listing 4.5 and Listing 4.6). This results in the `PictureView` being open and usable as usual, while in the background, there still is the `PictureGrid` rendered, but not visible. When the `PictureView` is closed the `PictureGrid` is immediately available as it does not have to be rendered again. This enables smooth transitions and even the possibility of animations.

**Listing 4.5:** Each picture in the `PictureGrid` gets rendered as a `PicturePreview`. When the `PicturePreview` gets clicked, the associated id gets selected as the “focusedPicture”.

```

1 <PicturePreview
2   picture={picture}
3   onClick={() => setFocusedPicture(picture.id)}
4   adornments={pictureAdornments}
5 />

```



**Listing 4.6:** The `PictureGrid` includes a `PictureView` that only gets rendered, when there is a `focusedPicture`

```

1 {focusedPicture &&
2   <PictureView
3     initialPictureId={focusedPicture}
4     siblingIds={pictures.map(p => p.id)}
5     onBack={() => {
6       setFocusedPicture(undefined);
7     }}
8   />
9 }
```

Another requirement emerging when using a `PictureView` in the context of multiple pictures is to allow the user to switch to the next or previous pictures without having to return to the `PictureGrid` in between. Therefore, the `PictureView` receives an optional array `siblingIds` as props which includes the ids of pictures from the context. In the `PictureView` there are navigation buttons to switch to the next or previous picture. When clicked, the `PictureView` gets rerendered with another picture id (see Listing 4.7). This is a change to our original implementation where each picture in the `PictureGrid` belonged to a different `PictureView`. That the `PictureGrid` now only includes a single `PictureView` does not represent the intuitive assumption that each picture maps to a separate element, but it has the advantage of reducing the coupling between `PictureGrid` and `PictureView`. The `PictureGrid` now does not have to manage multiple `PictureViews` that can be open, closed, or in transitioning states and the logic for switching the picture can be entirely encapsulated in the `PictureView`.

**Listing 4.7:** When a user navigates to an adjacent picture, the `PictureView` changes its selected id

```

1 const navigatePicture = (target: PictureNavigationTarget) => {
2   const targetId =
3     target === PictureNavigationTarget.NEXT
4     ? getNextPictureId(pictureId, siblingIds)
5     : getPreviousPictureId(pictureId, siblingIds);
6   if (targetId) {
7     setPictureId(targetId);
8   }
9   };
```

**Directory Structure** To give a brief introduction to our directory structure we will take a look at the most important folders in the `src` directory (see Figure 4.12).

`components` contains our React components and their related CSS files mainly grouped by Views. Each View has a separate folder including all components the

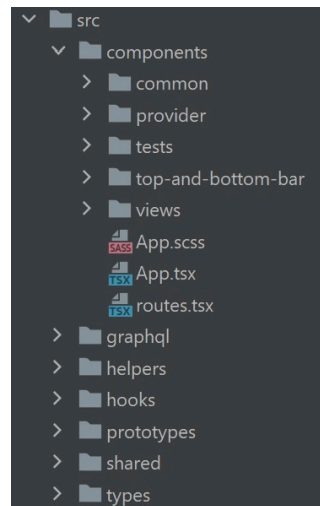


Figure 4.12: SRC directory

View is composed of. Components that are used in multiple Views like the Picture-Grid are in a common folder.

*graphql* includes the query and mutation definitions and the autogenerated query hooks.

*helpers* contains general helper functions that are used in various components. More specific helper functions that belong to certain components are placed in helper folders next to the component.

*hooks* includes our custom React hooks.

## 4.4 Data Flow

### 4.4.1 Fetching Picture Information

The previous sections introduced which technical elements exist in our application. Now we examine the interaction of these elements by looking at an exemplary data flow throughout the entire application. As it is our application's most common and basic use case, we will do that using the example of a user wanting to see a picture and associated information.

**Routing** When a user accesses our website, the `App` component gets rendered. The `App` includes a switch that renders a `View` depending on the called route and our path definitions in `routes.tsx`. For this example, we assume that the user called the `/picture/22` route. As it can be seen in Listing 4.8 this renders the `PictureView` which receives the `id` from the URL as a prop.

**Listing 4.8:** Definition of the picture route

```

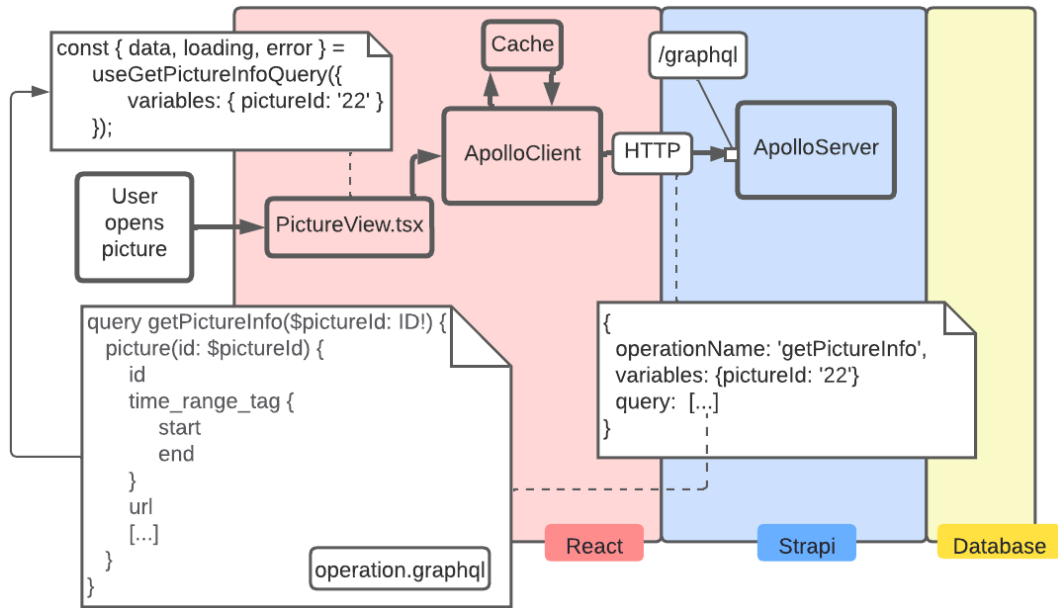
1  {
2    path: '/picture/:id',
3    render: ({ match }: RouteConfigComponentProps<{ id: '' }>) => {
4      return <PictureView initialPictureId={match.params.id} />
5    }
6  }

```

**GraphQL Query Building** The `PictureView` needs to access the data related to the picture with this `id` which we implemented by fetching the information through the GraphQL API provided by the backend. Therefore, the `PictureView` calls a hook, `useGetPictureInfoQuery`, with the picture `id`. This hook sends a GraphQL query to the API. The query that is sent (`getPictureInfo`) is defined in a separate file, `operation.graphql`, which contains all definitions of GraphQL queries and mutations. Based on this file we generated hooks (see section 6.4.1) like for example `useGetPictureInfoQuery`. This hook already includes the query definition for `getPictureInfo`, receives variables and other query options like cache behavior as props, and passes this information to the Apollo Client.

**Client-Server Communication** At first, the Apollo Client checks whether the requested information can already be found in the local cache. If that is the case, there is no request sent to the server and it continues with section 4.4.1. When the information is not in the cache, the Apollo Client builds an HTTP POST-request with the `/graphql` endpoint of the backend as the target (see Figure 4.13). In the body, there is a JSON with three fields: `query`, `variables`, and `operationName`. The `query` field contains the query definition, `variables` the values of the non-constant arguments, and `operationName` the name of the operation to be executed. The latter is mainly useful for caching and could also be used when the operation definition contains multiple queries.

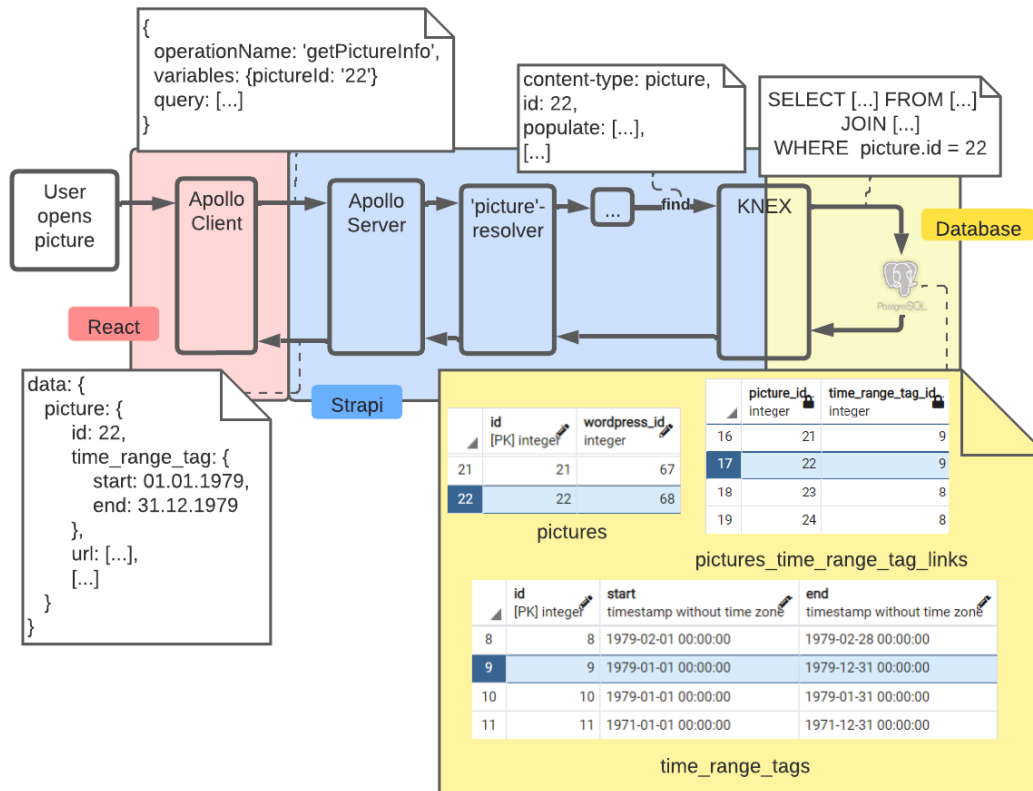
**Backend Handling** The Apollo Server receives the HTTP request and delegates the query to the corresponding resolver. The uppermost field called in `getPictureInfo` is `picture`. Therefore, the query initially gets handled by the `picture-resolver` which calls other resolvers and uses the Query Engine API (see subsection 5.4.6) to access the database using Knex as an SQL query builder. These generated SQL queries get executed on the PostgreSQL database. There are different possibilities to customize this behavior, like adapting or adding resolvers, using lifecycles, or customizing the SQL query building. We do not use these for this query but for other operations. How these customization options work and where we used them gets discussed in section 5.5.2. The results get formatted in the resolvers to fit the format defined in the GraphQL query. In the end, the `picture-resolver` returns a JSON with either an error



**Figure 4.13:** A GraphQL query gets built in the React app and sent to the backend using the Apollo Client.

or a data field, depending on whether the operation was successful. This JSON gets sent back by the Apollo Server as the body of an HTTP response (see Figure 4.14).

**Data Propagation in the React Application** Once the HTTP response arrives in the frontend the called hook, `useGetPictureInfoQuery`, updates the variables `data`, `loading`, and `error` in the `PictureView`. Updating the variables means that the `PictureView`, to stay reactive, did not `await` the result from the `fetch`. It already received values immediately after calling the hook: `data` and `error` have been undefined and `loading` has been true. When these values get updated it triggers a rerender of the UI elements that depend on the variables. Before the data can get passed down to other components it gets reformatted using the `useSimplifiedResponseData`-hook. Due to the unified-response format introduced in Strapi v4 (see subsection 5.4.7) and our workaround to implement a verified flag (see subsection 5.6.1) the data arriving in the frontend is in a format that is not practical to use. Further information regarding the different formats and the implementation of the `simplify` hook is given in section 6.4.1. The reformatted picture information gets passed top-down as props to the child components. It gets split and passed on to different components which each display some information to the user. For example, the information about the time range in which the picture was taken gets passed on through the `PictureSidebar` to the `PictureInfo` component. The `PictureInfo` contains multiple `PictureInfoFields` which each format and display a part of the information related to a picture. In the case of the time range, it is the `DateRangeSelectionField` that receives the dates, converts them to a string, and wraps them in a `div` (see Figure 4.15). React renders these

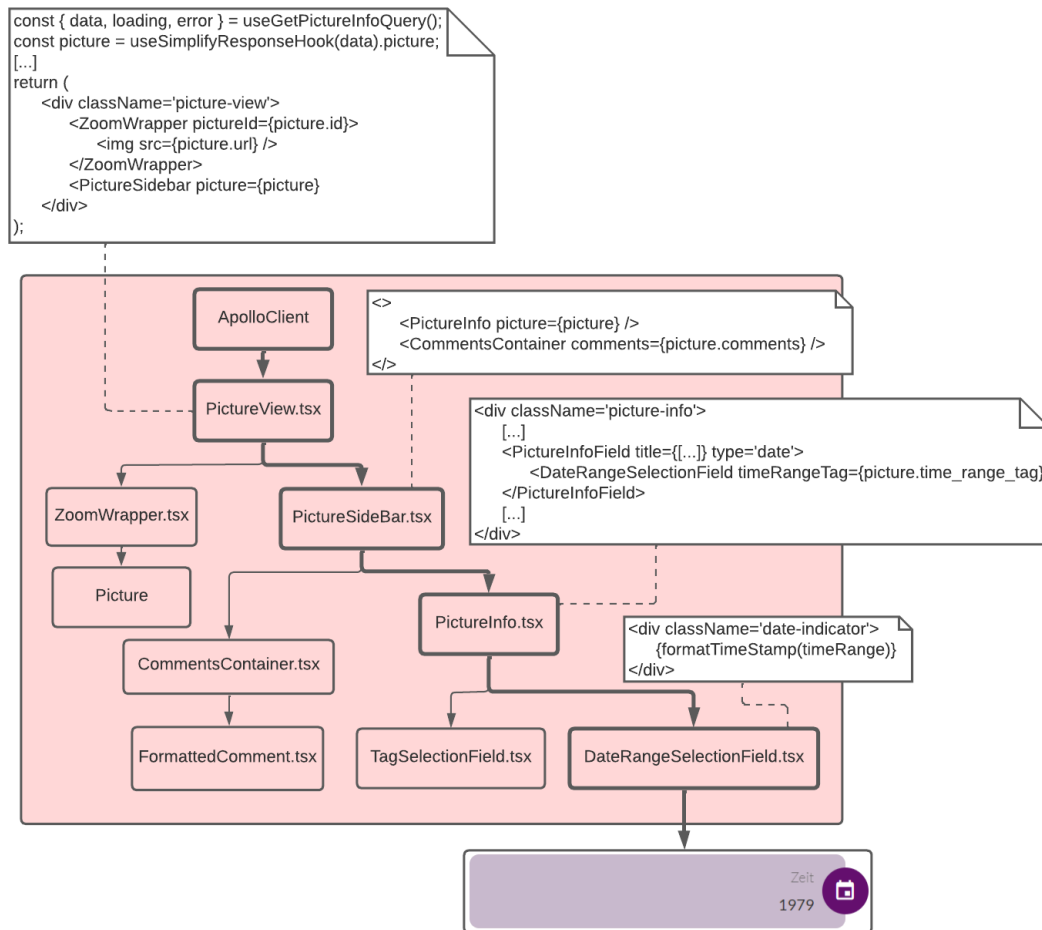


**Figure 4.14:** The backend handles the request, fetches the necessary information from the database and sends them back to the frontend.

elements combined with the associated CSS styling and the results get displayed in the browser to the user.

#### 4.4.2 Editing Picture Information

In the previous section, we have seen an example of fetching information. Many of our use cases also involve editing data which in general does not differ much from just fetching. In the following, we will take a look at the key differences. Therefore, we will examine the exemplary data flow of changing the time range associated with a picture. An authorized user visits the link to a picture. The picture information gets fetched as previously explained and the page gets rendered. The user clicks on the date which opens a time range picker. When the picker gets closed and the selected time range differs from the previously saved range, the `DateRangeSelectionField` updates its local state to immediately display the changed information to the user without having to wait for the response of the actual change happening in the backend. To forward the change to the backend, the component calls a function that it received as part of the props. This function is used to pass the information to the parent component `PictureInfo`. Here we deviate from our usual top-down data flow to manage changes to the picture information in one central component. When any

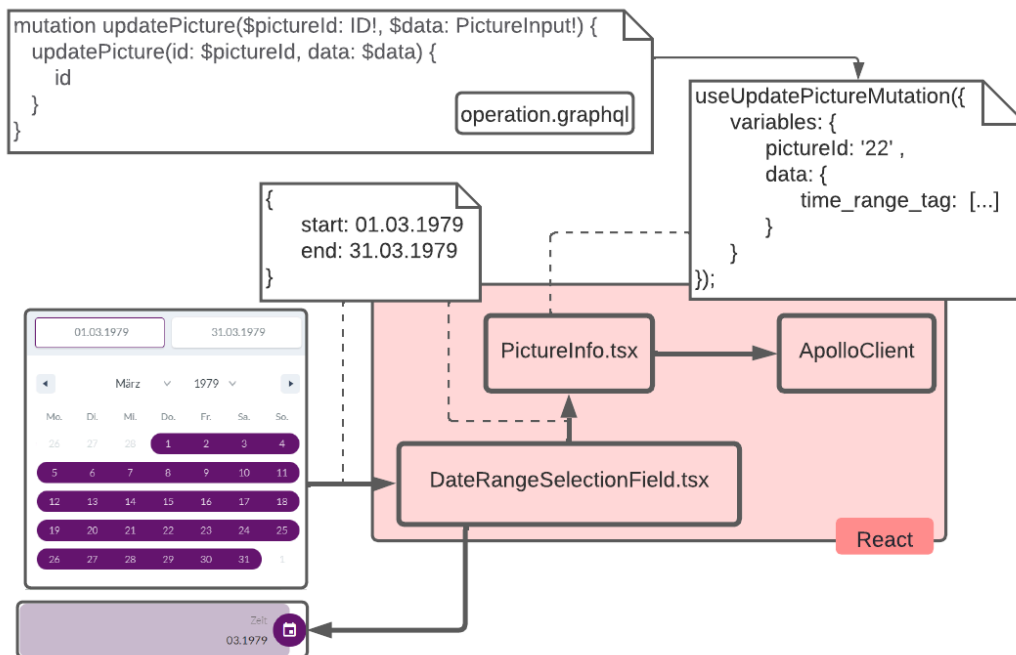


**Figure 4.15:** The received data gets simplified and propagated to the respective components. The information about the time range get passed through to the `DateRangeSelectionField` that formats and displays it.

field gets edited in a `PictureInfoField` like in this example the time range, the information gets passed up to the `PictureInfo` component which formats the changed information and calls the `useUpdatePictureMutation`-hook (see Figure 4.16). The hook is used to communicate the update with the backend and works analogously to the `getPictureInfoQuery`-hook. It is generated on the base of the mutation definition in `operation.graphql` and passes the information to the Apollo Client. The main difference is that the operation type is now a mutation and not a query and that the information with which the hook gets called not only includes the id of the picture but also the data to be changed. Moreover, we selected fewer fields that should be included in the response. The reasons for this will be explained later in this section.

The GraphQL-mutation is sent to the backend as an HTTP message in the same format as previously described for the query. The handling in the backend also works analogously. The Apollo Server delegates the mutation to the corresponding resolver of the uppermost field, `updatePicture`. This resolver calls other resolvers and uses the Query Engine API to access the database. For this specific mutation,

we did use some of the customization options. We introduced a lifecycle hook that triggers before every update of a picture entity and has the purpose of preventing the duplication of tags and removing unused ones (see section 5.5.2).

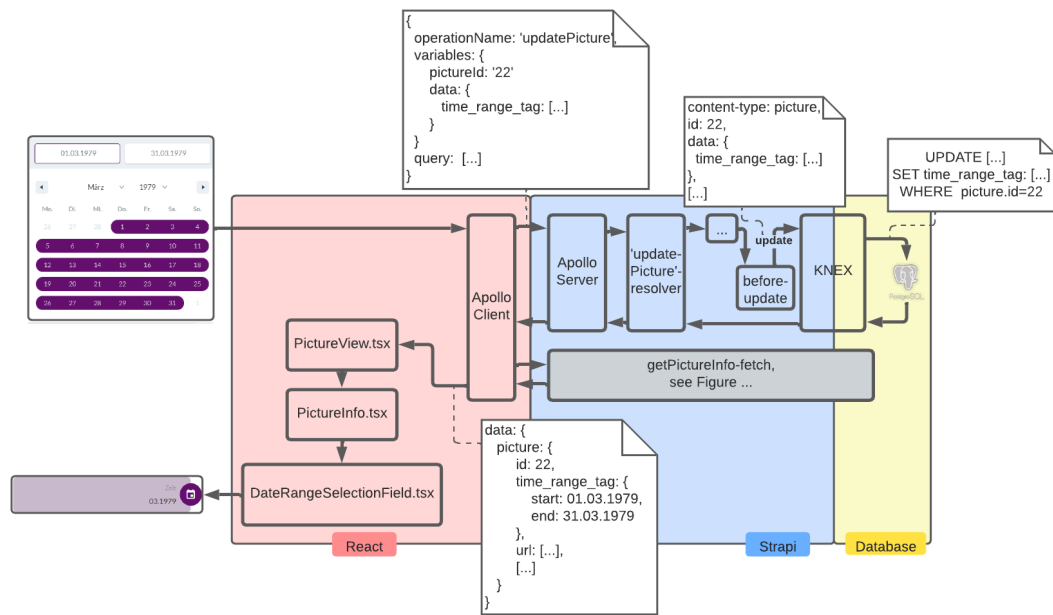


**Figure 4.16:** Changing the time range immediately updates the displayed information and then sends the changed data to the backend as a GraphQL mutation.

After executing this lifecycle function the picture update function of the Query Engine<sup>13</sup> builds two SQL queries again using Knex and executes them on the database. The first query performs the actual update. The second one fetches the updated information from the database and returns them to the resolver that called the picture update function. Which data is fetched and returned is specified with the fields in the mutation definition. Although this could be used to immediately return the updated picture information, we specified in the definition, that the mutation should not return the picture information. Instead, the response arriving in the frontend triggers a refetch of the `getPictureInfo` query (see Figure 4.17).

We implemented this behavior to maintain a single source of truth for the picture information. As explained in subsection 4.4.1, the data displayed in the `PictureInfo` component is based on the data from the `getPictureInfo` fetch in the `PictureView`. Additionally using data from a different query source, like the response of the update mutation could lead to conflicts, especially with the cache. For example, there

<sup>13</sup><https://github.com/strapi/strapi/blob/8d642957729fd35606e47a3df171852b41d437f1/packages/core/database/lib/entity-manager.js> (last accessed: 2022-07-13).



**Figure 4.17:** The sent mutation updates the information in the database and triggers a refetch of the `getPictureInfo` query.

are scenarios where closing and reopening a picture would display outdated information. Even though problems like this could be resolved by adjusting the behavior of the cache, we wanted to avoid a possible source of errors and chose the more stable option of refetching. The refetch works exactly like the usual fetch described in subsection 4.4.1. It is important to note that this was only an example of editing data. We use a large variety of updates that have different behavior, originating in differences in the expected behavior from the user perspective. While the coarse structure remains the same there are often deviations in the details. For instance, when editing a time range associated with a picture, the data sent to the API includes the actual time stamps to be saved. When editing for example the person tags associated with a picture, we first query all person tags. A curator now can select entries from this list of tags and the update mutation sent to the backend only includes the ids of the selected tags. Moreover, we use other mutations that rely more on custom backend handling. For example, we implemented merging two tags by writing a custom GraphQL resolver (see section 5.5.2) to avoid sending unnecessary many HTTP messages.

#### 4.4.3 Discussion: Layers of Indirection

As previously described our system includes several layers of indirection which make even a simple use case as displaying information to a picture a relatively complex process. In general, adding complexity is a drawback that arises the question of whether each of these layers is necessary.



**React** We chose to use the library React instead of writing plain HTML and JavaScript. As most of us had not used React before, we needed additional time to familiarize ourselves with the library. Besides learning the syntax and basic ideas, now and then we also had to refactor code parts to adapt them to React-specific development patterns which we had not been aware of as we did not know them from other languages or frameworks. Still, we evaluate using React as advantageous or even almost a necessity. The component structure introduced by React improves the structure and readability of our code and also simplifies code reusability. Components like the `PictureScrollGrid` are an often reoccurring combination of functionality and markup which would have been a lot harder to combine into a single reusable element in plain HTML and JavaScript. The reusability of components is also not limited to components we wrote ourselves. Being able to use components such as the `MUI-Autocomplete`<sup>14</sup> saved us a lot of time and unnecessary work. Moreover, we profit from the performance advantages that React provides using the virtual DOM.<sup>15</sup> Especially when editing information, often only a small part of our UI has to get rerendered instead of the entire DOM.

**Client-Server Communication** The client-server communication is the part where we have unusually many indirections. Based on a definitions file we autogenerate hooks that are used to send GraphQL queries with the Apollo Client. Incoming data gets parsed with the `useSimplifiedResponseData`-hook before it can get used in the frontend. Besides validating the GraphQL operation definitions against the current schema, the autogeneration provides types that are always consistent with the backend. As we use TypeScript this originally provided a lot of value, especially because the drawbacks in the usual development process are limited to having to execute a command after changing any operation definitions (see section 6.4.1). Our main uses of the Apollo Client are caching and unified error handling for all GraphQL-errors reaching the frontend. Using a caching tool is hardly avoidable and besides some special cases where we need a specific cache configuration, a developer does not have to be aware, that we use the Apollo Client at all. The `useSimplifiedResponseData`-hook (see section 6.4.1) is more debatable as it neither adds nor reduces complexity but moves it. The frontend can use a simpler and more intuitive format but it introduces a difference between the formats used in the React components and the operation definitions. Moreover, autogenerated types can not be used anymore, as they base on the non-simplified format from the schema. We decided to use the hook because we value the increase in the readability of large parts of the frontend code more than the drawbacks of having a less understandable `operation.graphql` file and having to manually adapt types when changing the schema.

Removing a single one of these layers of indirections would be difficult. But the overall complexity could be reduced by using REST instead of GraphQL. Initially, the understandability improves because REST is more widely known than GraphQL. Also, REST is supported by HTTP caching which removes the necessity for an exter-

---

<sup>14</sup><https://mui.com/material-ui/api/autocomplete/> (last accessed: 2022-07-13).

<sup>15</sup><https://reactjs.org/docs/faq-internals.html> (last accessed: 2022-07-13).

nal tool. The `useSimplifiedResponseData-hook` would have to remain as the cause originates in Strapi-specific behavior but it would become less confusing as there are no operation definitions and therefore fewer points of contact for the developer with the non-simplified format. In hindsight REST most likely would have been the better choice. Originally we based our decision on a problem with the REST-API that got resolved in Strapi v4 and now changing all of our API calls would require a lot of work. Whether this change would be worth the time is debatable because most of the GraphQL-specific indirections do increase the overall complexity but not the effort in the usual development workflow. Assuming a developer is familiar with both, adding a GraphQL query does not require more work than adding a request addressing the REST API.

**Strapi as a CMS** Instead of using Strapi, it would have also been possible to use another CMS or just the database with no CMS at all. The advantages and problems of using Strapi get extensively discussed in chapter 5.

### 4.5 Summary

In this chapter, we had a look at the frontend from a technical perspective. We examined the component structure of our React application which is centered around interchangeable Views that each implement a major feature. Further, we discussed the largest deviation from our usual structure and design guidelines, the Picture-View which needs special handling to enable smooth transitions between looking at a picture gallery and a full-sized picture. Moreover, we found that the data flow throughout the application includes relatively many layers of indirection, especially regarding client-server communication. While each of the different layers is necessary and can not just be removed, many of the reasons can be traced back to using GraphQL and Strapi. Future work might include changing the client-server communication from GraphQL to REST. Whether this time-intensive refactoring would be worthwhile is discussable because it would reduce the overall complexity but also would not decrease the effort in the usual development workflow.

## 5 Strapi as a Customizable Content Management System

In this chapter, we will discuss the usage of Content Management Systems for developing a crowd-sourced picture archive and the experiences we made with Strapi, an open-source representative of such systems.

### 5.1 Motivation for Using a Content Management System

To develop a crowd-sourced picture archive, we needed a system to store and manage the pictures and associated contextual information. Furthermore, we wanted to provide the end users of our application with a simple possibility for browsing through the available pictures and quickly sharing their knowledge and experiences regarding the historical context of the respective pictures.

The following sections aim to provide an overview of how Content Management Systems (CMS) can help to implement such a system, and how we used and customized Strapi, a representative of such management systems, to fulfill our project's requirements. Furthermore, the chapter includes a discussion of the most relevant technical challenges we faced with this approach throughout the project. Towards the end, the chapter then concludes on why we recommend to keep using Strapi and possible aspects that remain for future work on the implemented system.

### 5.2 Introduction to Content Management Systems

As storing and organizing different types of content, in combination with making it available on websites, is a common use case these days, there are several systems designed exactly for that purpose, called Content Management Systems. With minimal technical work needed, a CMS helps create and manage websites and website content [48]. Additionally, it offers many features that would otherwise need to be manually implemented when just setting up a database. It provides an abstraction from the data storage, which keeps users of such a CMS from the need to interact with databases directly with query languages like SQL. Along with that, other key features mentioned in an article by Oracle [48] are:

- A CMS enables the easy and collaborative management of the content of a desired website “via a single authoring and publishing environment”.
- Particularly, it offers freedom regarding the model of the content, also providing a simple way for managing digital assets like files, images, or videos.

- Additionally, Content Management Systems encapsulate the authorization layer of software applications through fine-granular control over users, roles, and permissions, providing a secure and reliable way for content management by multiple users.
- A CMS also simplifies the process of building and publishing websites with a focus on the actual content of the respective site, e.g., marketing or e-commerce websites.

Considering the requirements of our project mentioned in chapter 2 and section 5.1, especially the digital asset management and collaborative editing possibilities were key factors that influenced our decision to use a Content Management System for the project.

### 5.2.1 Monolithic vs. Headless Content Management Systems

There are two main types of Content Management Systems: traditional CMS, which are often called “monolithic”, and more modern, “headless” ones. Following considerations are based on the blog articles [57, 13, 16].

Traditional systems like WordPress<sup>1</sup> are often called “monolithic” because they are nearly full-stack environments providing not only the data storage and user interface for editing the content, but also a representational interface for the content that is accessed by the end users. However, this limits the possibilities for the display of the content on the desired website, making it on the one hand easy to quickly build and deploy a website with the help of predefined frontend templates, but on the other hand hard to only pick the features of this monolithic architecture that one wants for the application that the content is designated.

Headless Content Management Systems in comparison, are designed to be “API first”, completely separating “the backend (creation and storage) from the frontend (design and deployment)” [13]. There are well-specified application programming interfaces (APIs) for creating and managing the content, enabling the development of own, flexible user interfaces on top of these “backend-only” systems.

As already described in chapter 4, in the context of our project we wanted to develop our own frontend application, which made a headless CMS an obvious choice.

## 5.3 Introduction to Plugin Architectures

In the context of software development, the term “plugin” might not be that new. Nevertheless, we want to give a bit of background on the architectural design behind the usage of plugins. The following considerations are based on a blog article on that topic by Omar Elgabry [17].

---

<sup>1</sup><https://wordpress.org/> (last accessed: 2022-07-13).

The nature of this architecture lies in having a core system, which implements the basic business logic of the desired application, and, to extend the core application, plugins which are independent software components that come with additional features. In consequence of the fact that the plugins are separated from the core or other plugins, individual ones can be modified, added, or removed as desired. In most cases, this is possible without having to deal with resulting changes to the core or other plugins. A key point of this architecture lies in the communication between the core and the individual plugins. When implementing, developers must consider two main aspects:

1. The core needs to know which plugins are currently registered to it.
2. The interfaces between the core and the plugins must be well-defined in terms of the extension points the core provides.

Regarding the advantages of using this architectural design, Omar Elgabry also states, that “depending on how the pattern is implemented, each plugin can be deployed, tested, and scaled separately” [17], which emphasizes the independence of such plugins. However, also resulting from this is that it can be difficult to test the entire system because the behavior of multiple plugins in combination remains mostly untested. At the same time, the core itself can also be seen as a bottleneck as changes to parts of it, like the interfaces that plugins can access, might entirely break developed plugins. Further disadvantages can be that plugging in too many modules might impact the performance of the entire application and the typical consideration of which software to trust as plugins from external developers might come with security issues and additional attack vectors [18].

## 5.4 Strapi as a Content Management System

To fulfill the need for a flexible, headless Content Management System at the beginning of our project, we followed the suggestion of our project advisors to look into Strapi, short for “**Bootstrap your API**”. It is a leading representative of a headless Content Management System. It is open-source and built on top of Node.js, a browserless JavaScript application runtime. Apart from the free, open-source variant, that we used for the project, there is an enterprise edition as well.<sup>2</sup>

Strapi also comes with a command-line interface (CLI) [58, “CLI”] that can be used for various use cases regarding the following core components. For reference, all considerations below are related to the 4.2.0 version of Strapi.

### 5.4.1 The Administration Panel

Although not designed for end users, there is a React-based user interface provided by Strapi, called the Administration or Admin Panel. Figure 5.1 offers a view on its starting page.

---

<sup>2</sup><https://market.strapi.io/plugins/strapi-plugin-transformer> (last accessed: 2022-07-13).

The Admin Panel can be used for managing the content and settings of the system. And yet, using it is completely optional. It accesses the API of the independent backend just like any other external frontend application. It can also be deployed on a different server than the backend, further emphasizing the headlessness of Strapi.

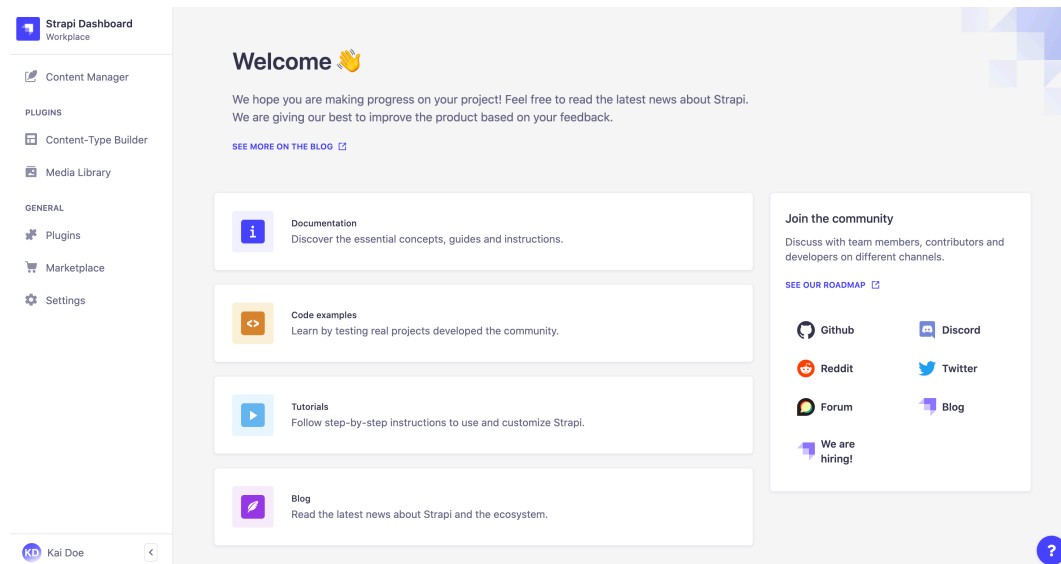


Figure 5.1: Starting page of the Administration Panel

### 5.4.2 The Media Library

As mentioned in section 5.2, a centralized hub for managing digital assets is one of the key features that come with Content Management Systems. Strapi offers such a hub with the Media Library<sup>3</sup> acting as a place to store image, video, and audio files as well as regular documents. It is also capable of creating and storing different formats of such media files. For example, for each uploaded image, several versions in different resolutions are generated.

### 5.4.3 Defining the Content Models

Just like for every CMS, the definition of the content models plays an important role in the Strapi system. It is mostly done inside a special view of the Admin Panel, the Content-type Builder. There is a detailed user guide [58, “Content-type Builder”] for this feature written by the Strapi team, which helped us on many occasions. However, we want to point out some key points about the different kinds of content types here

<sup>3</sup><https://strapi.io/features/media-library> (last accessed: 2022-07-13).

as well. For that, Table 5.1 shows a basic comparison between the three kinds of content types: Collection Types, Single Types, and Components.

**Table 5.1:** Comparison between collection types, single types, and components

	<b>Definition by Strapi</b>	<b>Example use case in our system</b>	<b>API endpoints?</b>
<b>Collection types</b>	“Collection types are content types that can manage several entries.” [58, “Content-type Builder”]	Storing tag information, e.g., the different person tags.	yes
<b>Single types</b>	“Single types are content types that can only manage one entry.” [58, “Content-type Builder”]	Managing the browse root collection, which is the default collection that gets displayed in the browse view of our frontend application described in section 4.3.2.	yes
<b>Components</b>	“Components are a data structure that can be used in multiple collection types and single types.” [58, “Content-type Builder”]	Storing the synonyms of a certain tag, e.g., aliases of a person.	no

For Single and Collection types there is also the “Draft & Publish” feature [58, “Saving & Publishing content”] for entities that should not be directly visible to non-authenticated users.

Regarding the various field types, which are used to model the attributes of a content type, we want to focus on the most important one for our project: the relational field. It enables establishing a connection to another content type. One-to-one, one-to-many, and also many-to-many relations are possible. This field type makes up the foundation of our content model, as we want to relate various types of information to the pictures type, the main content type of our system.

Based on the configurations Strapi generates a static schema file for each content type representing the model of the respective type. These JSON files can also be

adapted manually or even generated completely without the Admin Panel in the first place with the Strapi CLI.

Strapi then maps the models of these content types to relational tables on the database layer described in subsection 5.4.4. Therefore, the database schema is created automatically. Subsequent starting processes each trigger the detection of schema differences and synchronizations if needed.

But these schema migrations can not be fully automated. In most cases, changes to the schema files result in data loss. For example, that applies to the renaming or type changes of fields as in both cases it is getting translated down to a combination of removing the old column and adding a new one on the database layer. Type changes can even lead to system failures when the new type of field is not compatible with the data that is present in the associated database column. For example, this occurs if one tries to change a string field with present data into a number field. So to prevent the loss of data, custom migration code needs to be written and new fields first need to be filled with data before old fields can safely be deleted. As far as we discovered there is an interface for programmatic migration files but at the current point, it is not well documented. A possible entry point into this topic might be the Strapi migrations plugin [15]. Instead, we implemented some custom migration scripts via our custom plugin, more on that in chapter 6.

### **5.4.4 Connection to Relational Databases and the Knex Query Builder**

Regarding the data storage of one's system, Strapi is capable of connecting to the following relational databases: MySQL, MariaDB, SQLite, and PostgreSQL [58, "Installing from CLI"]. For our project, we decided to use a PostgreSQL database.

Internally, Strapi makes use of the Knex Query Builder, which provides a programmatic interface to interact with databases in the Node.js environment. It offers a simple and abstract way of building SQL statements by calling predefined methods, thereby also taking care of using prepared statements to prevent SQL injection attacks [24]. Responses of statements are automatically mapped to JavaScript objects as a part of the object-relational mapping (ORM), on which Strapi's internal interfaces are based on.

### **5.4.5 Using the Users & Permissions Plugin as an Authorization & Authentication Layer**

With the Users, & Permissions plugin (U & P) [58, "Users, Roles & Permissions"] Strapi provides an abstracted, but at the same time fine-granular control over registered users, associated roles and permissions. For example, each role can have different permissions for accessing or modifying entities of a certain content type. Configurations can easily be made in the Admin Panel or through the REST or GraphQL APIs, which get described in subsection 5.4.7 and subsection 5.4.8. Concerning the calls to these APIs, the authentication is managed by JSON Web Tokens (JWT). In the context of our project, the plugin encapsulates all our authorization



and security logic and in consequence, we could use it out of the box without having to take any further actions.

### 5.4.6 Further Internal APIs: The Entity Service and the Query Engine

Apart from the REST and the GraphQL APIs that get introduced in the next two sections, Strapi also populates internal interfaces for developers. The most important of these are the following three:

- The already introduced Knex Query Builder, usable for directly interacting with the database.
- The Query Engine API [58, “Query Engine API”] acts as an abstraction for interacting with the database and is already following the CRUD (Create, Read, Update, Delete) pattern.
- The Entity Service API [58, “Entity Service API”] makes use of the Query Engine API and further enhances it with the capability of handling aspects like the complex data structure of Components.

### 5.4.7 Accessing the Content Through the REST API

Strapi’s REST API is based on an HTTP Server that itself is based on Koa, a backend JavaScript framework for that purpose [58, “Back-end customization”]. For each content type, endpoints for default CRUD operations are generated by Strapi [58, “REST API”]. These endpoints follow the pattern of separating route configurations, controller, and service logic, whereby only the service deals with the business logic and calls the mentioned Entity Service and Query Engine APIs.

Responses follow the shape of the “Unified Response Format” [58, “REST API”], which applies to both the REST and the GraphQL API. A response is always an object consisting of a `data` and an additional `meta` or `error` key. The actual entity objects returned from the backend are put into the `data` key. Besides the `id`, the configured attributes of entities are encapsulated into the `attributes` key. The encapsulation into `data` and `attributes` keys is recursive and also applies to nested entities related to the originally requested entity. So for example, a response object for requesting a content type with a relational field looks like shown in Listing 5.1.

**Listing 5.1:** The “Unified Response Format”

```

1  {
2    "data": {
3      "id": 1,
4      "attributes": {
5        "title": "Entity A",
6        "relatedEntity": {
7          "data": {
8            "id": 2,
9            "attributes": {
10             "title": "Entity B"
11           }
12         }
13       }
14     }
15   }

```

```
14   }  
15   },  
16   "meta": {}  
17 }
```

### 5.4.8 Accessing the Content Through the GraphQL API

As already mentioned in subsection 4.2.2, Strapi also provides a GraphQL API [58, “GraphQL API”]. Internally, Strapi starts an Apollo Server instance<sup>4</sup>, which is a popular runtime capable of parsing GraphQL requests. By default, these are coming in as POST requests to the `/graphql` endpoint, which is provided by the Apollo Server instance.

With the “Shadow CRUD” feature enabled, Strapi automatically generates resolvers in analogy to the CRUD operations of a typical REST API and adds these to the GraphQL schema. When called, the registered resolvers make use of the mentioned Entity Service and Query Engine APIs directly as well. In consequence, the introduced pattern of separated route configurations, controllers, and services on, which the REST API is based, does not apply here.

### 5.4.9 Internationalization

Storing content in different languages to make the application available for international users is possible in Strapi as well. The handling for that is encapsulated in the Internationalization (i18n) plugin [58, “Internationalization (i18n)”]. For our project we decided to completely disable this plugin because our domain is a German picture archive.

## 5.5 Customizing Strapi

The Content-type Builder already offers much room for customization regarding the content of our system. Yet, there are also lots of possibilities for modifications and extensions of the behavior of the system. Note that the following considerations are not covering all of these and reflect more what we made use of in our project.

### 5.5.1 Frontend

It is possible to extend the Administration Panel introduced in subsection 5.4.1 by creating your own React components. For example, custom menu entries can be developed or new sections to existing views can be added. We did not dig deeper

---

<sup>4</sup><https://www.graphql-code-generator.com/docs/getting-started> (last accessed: 2022-07-13).

into this field, as we decided to build our frontend application and thereby hide the Admin Panel from our project partner and end users.

### 5.5.2 Backend

Even more, customizations can be made on the backend side, which again points out that Strapi is a headless CMS. In all the following cases, calls to Strapi's internal interfaces like the Query Engine as well as the Knex Query Builder are possible. These, therefore, build the foundation of creating custom code that gets integrated into the existing system.

**Modifying and Extending the REST API** As seen in subsection 5.4.7, the REST API is implemented based on the pattern of routes, controllers, and underlying services. All of these mentioned can either be extended with new custom logic or modified to configure the REST interfaces that are generated for each content type.

Just to mention a few examples, it is possible to:

- disable certain routes and their associated actions or make them public to bypass the authentication system,
- overwrite existing logic of action to add information to its results or execute side effects,
- register completely new routes and associated actions for special use cases.

In the context of our project, these extension possibilities remained mostly unused as the frontend application, introduced in chapter 4, only uses the GraphQL API.

**Modifying and Extending the GraphQL API** Strapi provides the possibility to register custom GraphQL extensions that can extend the schema partly generated by Strapi's "Shadow CRUD" feature. These extensions get defined using GraphQL Nexus<sup>5</sup>, a library that acts as a programmatic abstraction for interactions with the schema.

Regarding the actual resolving process, it is firstly possible to modify the generated resolvers for queries and mutations just like for the REST API by e.g., disabling certain actions or bypassing the authentication system. Furthermore, completely new queries and mutations can be registered. For these custom resolvers, there is the advantage of full control over the types of the arguments and the return type. Not to forget the complete control over the resolving process itself, e.g., which internal Strapi interfaces are called. Custom resolvers can also be registered to the U & P plugin to manage the authorization needed. In the context of our project, we used that possibility to register a few custom mutations for merging two tag entities to make it possible to reduce redundancy that may be present in the system.

Another customization option is using middleware. Such middleware explicitly leaves the existing resolver logic in its place and just adds custom logic around it, for

---

<sup>5</sup><https://nexusjs.org/> (last accessed: 2022-07-13).

example for logging time-related information regarding a certain resolve process or dynamically resolving information that is not available in the first place.

To point out an example, we implemented a middleware for resolving the `thumbnail` field of our collection content type. For context, collections can be in a parent-child-relation to each other. Not every collection also directly has pictures related to it. A collection can also only act as a parent to group multiple collections. So when information about a certain parent collection is queried, a representative picture needs to be chosen for it. As we do not want to modify any existing query resolving regarding collections apart from that one field, that is the perfect use case for query middleware. The implemented middleware follows a recursive approach traversing the hierarchy of child collections until at least one picture is found. At that point, one picture is chosen to be the representative of the originally requested parent collection, so in consequence, the URL of this picture is put into the `thumbnail` field of that collection.

**Extending the Model Layer: Lifecycle Hooks** On the model layer, there are opportunities for behavioral customizations as well. This can be achieved with the concept of lifecycle hooks, which the Strapi Developer Docs explain in the following way: “Lifecycle hooks are functions that get triggered when Strapi queries are called. They have triggered automatically when managing content through the Administration panel or when developing custom code using queries” [58, “Models”]. Additionally, in consequence of the fact that the model layer lies beneath the top level interfaces, both the REST and the GraphQL API trigger registered lifecycles. These hooks are each scoped for a certain content type and can be registered to run right before or after calls to the internal Query Engine regarding the respective content type are made. For example, for the `update` call, a `beforeUpdate` and a `afterUpdate` lifecycle are registrable.

During our project, we developed a custom `beforeUpdate` lifecycle for the picture type. For context, when updating a picture we also want to manage its associated tags. The main reason for that is to keep the data that is present in the system clean and manageable. New tag entities should not be created when there are already ones that reflect the new content. Furthermore, outdated tag entities should be cleaned up if they are not referenced anymore. Moreover, from the perspective of the frontend, this cleaning process should not be apparent, just like such lifecycle functions do not need to be called explicitly.

In retrospect, we realized that it could also be done as a custom mutation resolver of the GraphQL API and that this might be an even better solution. The mutation resolver marks a more separated API endpoint just for our use case, e.g., the Admin Panel would not trigger this code in contrast to the lifecycle. Additionally, we do not know which other parts of Strapi could trigger the lifecycle without us noticing, whereby the custom mutation would only be called in our frontend application. Secondly, as mentioned in subsection 5.4.8, in that way we have more control over arguments and return types. Furthermore, it could result in a better understanding of what is going on when the API call is made, as the lifecycle is still a `beforeUpdate` and the actual update of the respective picture is not even made yet at that point. On

the other hand, custom handling is closely related to the update of a picture, which the defined lifecycle is exactly scoped for. Moving that code to the point where we register our GraphQL extension could be seen as another indirection and therefore increase the complexity.

### 5.5.3 Plugins

Strapi strongly promotes the Plugin Architecture described in section 5.3. Plugins can thereby access and modify essential frontend and backend functionalities and also define their content types. From a technical perspective, Strapi plugins are independent node packages that can be installed with the help of any node package manager. Installed plugins can each be enabled and disabled, as well as provided with special configuration details inside a global config file.

**Using Plugins Developed by Strapi or the Community** Some aspects of Strapi described earlier are encapsulated into their plugins developed by the Strapi team. For example, as mentioned the U & P plugin deals with the authorization layer or the GraphQL plugin introduces the support of the GraphQL API. Partly, these come with the default installation process (e.g., U & P or i18n) [58, “Plugins”], others can be found in the Strapi Marketplace (see Figure 5.2) just like plugins developed by the community.

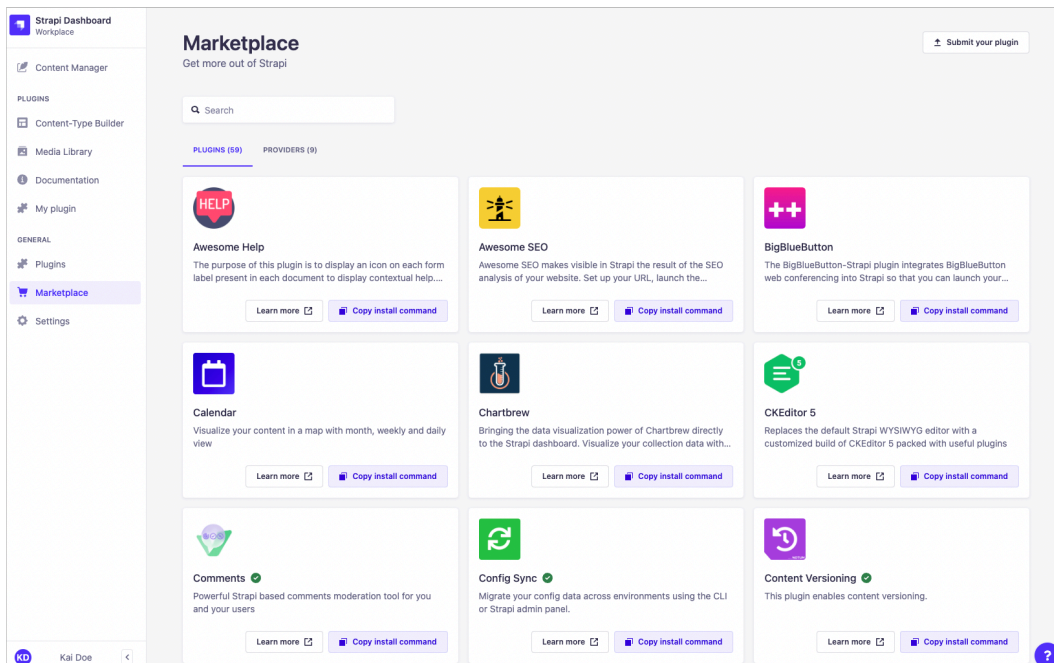


Figure 5.2: Strapi Marketplace<sup>6</sup>

**Extending Existing Plugins** There is also the possibility to modify an existing plugin, either by extending its content types or its interface e.g., by configuring new endpoints.

**Developing Own Plugins** Developing completely new plugins is possible as well. The basic directory and file structure for new plugins can be generated with the Strapi CLI. Just like external plugins, locally developed plugins can use the entry points of both frontend and backend functionalities and create their content types as well. We made use of this feature to develop a custom bulk-import functionality which gets discussed in detail later in chapter 6. Additionally, plugins are publishable to the Marketplace to support other developers who may face similar use cases.

## 5.6 Discussion

Despite the various possibilities for customizations and modifications, we faced some technical challenges while using Strapi throughout the project. The following sections discuss some of the more relevant ones of these.

### 5.6.1 Lack of Attributes on Relations

Consider the following scenario partly described in chapter 2: when tag information is added to a picture, the involved curator in some cases is uncertain whether that information is accurate in terms of the historical context of the picture. For that use case, we wanted to implement a `verified` flag for all our tag types that can be related to the picture type. In particular, that concerns the following of our content types: the keyword tags, the location tags, the person tags, and the time-range tag of a picture.

The meaning of this `verified` flag is only apparent regarding a picture-tag combination. For example, just because a person is identified in a certain picture and the person tag is then marked as `verified` in that case, there is no reason why that should also apply to a different picture that might depict the same person. Similarly, the typical example of a percentage-of-time attribute on an employee-project-relationship in a company [12] is modeled using a relationship attribute. We, therefore, derive that it makes sense to also model the `verified` flag as a relationship attribute.

Unfortunately, the Strapi Content-type Builder does not allow configuring such attributes on relationships between content types.

### 5.6.2 Possible Solutions

To mitigate this problem, we came up with various solutions. We want to focus on some of the more promising solutions and evaluate the pro and contra arguments of each one.

---

<sup>6</sup><https://docs.strapi.io/assets/img/marketplace-v4.27b6c5ad.png> (last accessed: 2022-07-13).

**1. JSON Field on the Picture Type** The main point of this approach is the management of a JSON field on the picture type which tracks the `verified` flag of all tags that are associated with a certain picture. For the convenience of the frontend, it also includes an on-demand parsing of this JSON format to booleans that are directly put inside the respective tags and vice-versa in the lifecycles of the picture type. Here are considerations of the pro and contra arguments for this approach:

#### Pro

- The information stored in the JSON can consist of much more than the `verified` boolean. Instead, any possible relationship attribute can be managed here. Additionally, these attributes can have any type that is representable in a JSON, e.g., non-boolean (Example: a very abstracted key-value store is possible).

#### Contra

- **Consistency issues:** If a tag loses its relation to certain pictures or is even deleted, the JSON fields on all prior related pictures will need to be updated to keep them synchronized with the actual state of the system. That would result in changes to the update and delete lifecycle of every tag that can have this `verified` flag.
- Strapi's JSON fields are not filterable at the moment. One could technically use string filters (like `contains`) but in a JSON there is no guarantee that the properties are always in the same order, which would be needed for the string filtering. So this approach can not support the future use case of querying e.g., all time-range tags that are somewhere in an unverified relation.
- It might be the case that collections should be taggable in the future as well. That would result in configuring the collection type with its JSON field and parsing logic in the same way which would overall result in a high amount of duplicated code.

**2. Repeatable Component as a Key-Value Store on the Picture Type** As seen in subsection 5.4.3, Strapi provides a Component data structure usable in other content types. This approach consists of managing such a Component on the picture type in a repeatable manner. Each instance of a Component on a specific picture thereby extends an existing relationship between this picture and a specific tag with the `verified` information or other potential relationship attributes. Here are pro and contra aspects for this idea:

#### Pro

- Components are (like the JSON field) flexible and suitable for any kind of key-value store with different types (e.g., non-boolean). So other relationship attributes can be managed as well.

#### Contra

- This approach results in similar consistency issues to those described in section 5.6.2.

- At the current point, there is no filtering on Components possible in the GraphQL API, which makes this approach not feasible when considering e.g., the use case of querying all time-range tags that are somewhere in an unverified relation.

**3. Additional Custom Join Types** Here, a new Collection type `join_picture_{tag type}` is created with configured many-to-many relations to the picture type and the used tag type. This join type then contains a `verified` boolean field. That is also a solution suggested by some members of the Strapi community.<sup>7</sup> Here are the pro and contra arguments:

#### Pro

- It recreates the internal representation of attributes on relationships on the database layer. Thereby, there is the freedom of which and how many relationship attributes can be described by that and which other types apart from booleans are usable.

#### Contra

- Internally, Strapi creates three different join tables per picture tag type combination with this approach (`join_pictures_join_picture_{tag type}`, `join_picture_{tag type}`, `join_join_picture_{tag type}_{tag type}s`). But there is only the need for a single join table to make attributes on relationships work.
- There is a need to manually implement a garbage collection for these join types. If an entity of these types loses one or even both of its join partners (a picture and/or tag type entity), it will become completely meaningless. However, as it is not possible to set relational fields as required in the Strapi Content-type Builder, the now unreferenced join entity will not be deleted by Strapi itself (as it is just a regular content type entity from their perspective).

#### 4. Duplicating the Prior Picture Tag Type Relation Fields With a `verified` Prefix

This solution deals with the duplication of the prior relation fields between pictures and the individual tag types with a `verified` prefix and changing the semantics of the prior relations to unverified. In consequence, the picture type then has a `{tag type}` and a `verified_{tag type}` field for each mentioned tag type. Here are the pro and contra arguments for this approach:

#### Pro

- This approach comes with no consistency issues as there is no place where outdated information is tracked when a tag gets deleted or loses a relational partner.
- From both a picture and also from the respective tag type, one can directly get the information whether their relation is verified or not. Unlike the first two

---

<sup>7</sup>Strapi GitHub Issue: Doc request: Adding properties to a relationship <https://github.com/strapi/strapi/issues/1156> (last accessed: 2022-07-09).



solutions mentioned in subsection 5.6.2, one does not need to make a lookup on a special field that just exists on one of both involved entities.

#### Contra

- It is only a solution for a single, boolean-typed relationship attribute.
- The frontend is not completely kept away from that implementation detail, when querying information, both relation fields need to be specified (GraphQL API).

**5. Manual Implementation of Attributes on Relations** Manually implementing that feature might be possible as a custom plugin, although there are complex aspects. It would require dealing with Strapi core functionalities and interfering with their internal join logic. How the core could be extended that way is not well documented. In conclusion, it is hard to estimate the research and implementation, and maintenance effort that is required for this approach.

### 5.6.3 Evaluating the Chosen Solution

Based on these considerations, we decided to implement the fourth approach, which deals with the duplication of the prior tag type relation fields to the picture type with a `verified` prefix. In comparison to the other solutions, it is the one that resulted in the fewest code changes as there is no custom garbage collection needed for any kind of outdated information. Yet, we decided to make the following two changes:

1. As mentioned, the frontend which uses the GraphQL API now needs to specify both relation fields when information about the tags of a picture is queried.
2. For convenience reasons we also thought about hiding that duplicated relation fields detail to at least the frontend parts that are associated with the view logic. More on this can be found in chapter 6.

Additionally, at this point, we decided for this approach, the `verified` flag was the only relationship attribute there was a requirement for. Hence, we evaluated the disadvantage of the chosen solution, which is only suitable for a single, boolean-typed relationship attribute, as not that significant in comparison to the number of code changes that we evaded.

In retrospect, that argument does not hold anymore. Towards the end of the project, our project partner proposed the idea of arranging e.g., all person tags that are related to a given picture in a specific order. This not only describes a non-boolean relationship attribute, as it can have any natural number of possible values but also the combination with the previous `verified` semantics is hard to reflect with this duplicated relation fields approach. We did not implement that idea during our project, so taking further actions remains for future work.

### 5.6.4 Too Generic Query Building Resulting in Rather Cost-Intensive Queries

We wanted to support a search query for the pictures content type that looks for matches of multiple search terms in all the different tag types related to the pictures type. In particular, one should be able to type in multiple search terms and shown results should have a match in at least one of the different tag types for each of these terms. Except for the time-range tags, a match thereby should be equivalent to the term contained in a case-insensitive manner. Hence, being tolerant of typos was not necessary. Additionally, the query should be part of the GraphQL API.

**First Implementation** The first implementation consisted of building a complex `filters` object for the built-in pictures GraphQL query, which is generated by Strapi like mentioned in subsection 5.4.8. For example, when “Harz” and “1954” were typed in, the resulting `filters` object looked as shown in Listing 5.2 (for the complete object see Listing A.1 in the appendix).

Now considering, as all our tag types are configured to be in relation to the picture type, there is a join-table for each picture tag type relation generated by Strapi. So filtering pictures by attributes of these tag types results in the need for multiple joins of relational tables. Strapi internally builds these necessary joins in a too generic manner. As the `filters` object shown contains entries for all individual tag types for each search term, Strapi’s current implementation instructs the Knex Query Builder to join all necessary tables (tag tables and associated join tables) together for each search term. So when searching e.g., two terms, the resulting query contains joins for all necessary tags and associated join tables twice. There is no check whether the tables need to be joined to the existing table aggregate or not. In the appendix, Listing A.2 shows a SQL query that was generated by the Knex Query Builder in that way. It reflects the scenario of the two search terms “Harz” and “1954”.

**Listing 5.2:** Basic structure of the filters object for the search scenario

```

1  {
2    "and": [
3      // Begin of search term "Harz" (a not time-related term)
4      {
5        "or": [
6          // For each tag with textual information there is a filter for the
7          // regular relation and the verified relation field. The search term
8          // should match case-insensitively in the textual attribute.
9          {
10             "TAG_WITH_TEXTUAL_INFO": {
11               "TEXTUAL_ATTRIBUTE": {"containsi": "Harz"}
12             }
13           },
14           {
15             "VERIFIED_TAG_WITH_TEXTUAL_INFO": {
16               "TEXTUAL_ATTRIBUTE": {
17                 "containsi": "Harz"
18               }
19             }
20           },

```

```

21     [...]
22   ]
23 },
24 // Begin of search term "1954" (a time-related term)
25 {
26   "or": [
27     // Time-related terms are firstly treated as regular textual terms.
28     {
29       "TAG_WITH_TEXTUAL_INFO": {
30         "TEXTUAL_ATTRIBUTE": {"containsi": "1954"}
31       }
32     },
33     {
34       "VERIFIED_TAG_WITH_TEXTUAL_INFO": {
35         "TEXTUAL_ATTRIBUTE": {"containsi": "1954"}
36       }
37     },
38     [...]
39     // For the time-related tags we also parse the time-related term into
40     // suitable start/end timestamps before.
41     {
42       "time_range_tag": {
43         "start": {"gte": "1954-01-01T00:00:00.000Z"},
44         "end": {"lte": "1954-12-31T23:59:59.000Z"}
45       }
46     },
47     {
48       "verified_time_range_tag": {
49         "start": {"gte": "1954-01-01T00:00:00.000Z"},
50         "end": {"lte": "1954-12-31T23:59:59.000Z"}
51       }
52     }
53   ]
54 }
55 ]
56 }

```

As a consequence of these unneeded joins, the table aggregate is growing with each search term and as joining is the most expensive operation for a relational database [38] that aspect slows down the response time of our system dramatically.

Table 5.2 offers a measurement of the response time of a complete GraphQL API request with this implementation for selected search scenarios. Note that apart from “1954” all search terms appear rather often in the database.

All scenarios were tested on a PostgreSQL database with the following entity counts for the different content types.

- Pictures: 9350
- Collections: 211
- Descriptions: 2916
- Keyword tags: 798
- Location tags: 186
- Person tags: 294
- Time-range tags: 1428

Already waiting ~379 seconds or ~6 minutes in a scenario with just three search terms is unacceptable for a productive version of our system.

**Table 5.2:** Measurement of the response time for selected search scenarios based on the first implementation

Number of search terms	Search terms	Response time in ms
2	"Harz", "1954"	6822
3	"Harzburg", "Harz", "1954"	379473
4	"Bad", "Harzburg", "Harz", "1954"	N/A (canceled after nearly three hours without a response)

**Developing an Optimized Solution** To solve this performance issue we decided to register a custom GraphQL Query, which acts as an endpoint for the introduced search scenario. When resolving, we manually compose a SQL statement with the Knex Query Builder, created like the model of the actual queries built by Strapi itself. At the join part, however, we make some optimizations to only join the needed tables together once. In the appendix, Listing A.3 shows a SQL query that was generated by the Knex Query Builder for that optimized implementation as well. It again reflects the scenario of the two search terms "Harz" and "1954". Now considering the same test scenarios as before, Table 5.3 shows the measured response times with this version of the implementation.

**Table 5.3:** Measurement of the response time for selected search scenarios based on the optimized implementation

Number of search terms	Search terms	Response time in ms
2	"Harz", "1954"	2195
3	"Harzburg", "Harz", "1954"	2563
4	"Bad", "Harzburg", "Harz", "1954"	2870

It is clear that by optimizing the expensive join part of the query, we improved the performance drastically. But at the same time, this solution is highly coupled to the current state of our content types and the use of a PostgreSQL database as a data store.

Nevertheless, we think this approach satisfied our needs and was therefore solving the issue we encountered with the first implementation for the described search scenario.

**Other Approaches to Searching in Combination with Strapi** Making it possible to browse and search through the content of one's system can be seen as an essential

feature modern content-based systems should provide. In this section, two external search engines get introduced that Strapi is capable of connecting to.

### Connection to an Elasticsearch Instance

Elasticsearch (ES)<sup>8</sup> is an often used, distributed search engine that supports light-weighted queries over large data sets. Along with that, it provides features like typo-tolerance and the support of synonyms.

Applied to our scenario, the approach to use it together with Strapi would look as follows.

- Set up an Elasticsearch instance to also run on our server along with the Strapi instance.
- Synchronize the Elasticsearch instance with the Strapi database state with the Strapi Elastic plugin.<sup>9</sup> With set configurations the plugin enables Strapi to connect to the Elasticsearch instance and every time changes are made to the content type entities in Strapi, the plugin keeps the ES instance synchronized with these.
- Make the search request to the ES instance over its REST API<sup>10</sup> instead of the Strapi API to quickly retrieve matching pictures.

This approach was not tested by us, but it can definitively be seen as a possibility to easily scale up the search engine for even larger data sets. The big drawback regarding our project is the payment model which currently requires a payment of \$95 a month for the standard plan.<sup>11</sup>

### Connection to a Meiliseach Instance

Meiliseach<sup>12</sup> is an open-source alternative to Elasticsearch. It runs as a separate server instance and offers features like typo-tolerance and the support of synonyms as well.

The approach to using Meiliseach in combination with Strapi is similar to the one for Elasticsearch, there is also a Strapi plugin<sup>13</sup> on the Marketplace to manage the connection and the synchronization. But besides the REST API, one can also make use of a JavaScript integration to make calls to a pre-configured client directly in the code of a frontend application.

It was not tested by us either. However, as it has that big advantage over Elasticsearch of being open-source and free to use, we would choose Meiliseach to scale up our search solution if we proceeded with the project. So that remains for future work.

<sup>8</sup><https://www.elastic.co/elasticsearch/> (last accessed: 2022-07-13).

<sup>9</sup><https://github.com/cillaeslopes/strapi-elastic> (last accessed: 2022-07-13).

<sup>10</sup><https://www.elastic.co/guide/en/elasticsearch/reference/current/search.html> (last accessed: 2022-07-13).

<sup>11</sup><https://www.elastic.co/pricing/> (last accessed: 2022-07-13).

<sup>12</sup><https://github.com/meiliseach/meiliseach-js> (last accessed: 2022-07-13).

<sup>13</sup><https://github.com/meiliseach/strapi-plugin-meiliseach> (last accessed: 2022-07-13).

### 5.6.5 Migrating Between Major Versions

When we started our project in October 2021, we began using the Strapi version 3.6.8. At that point, the Strapi developers already shifted their focus away from the 3.x.x versions, as they were working intensively on the next major version, the v4.

While developing our first prototypes (see chapter 3) we quickly found a significant issue with the API provided by the v3. We experimented with the possible filters one could set when querying entities of a content type and noticed that especially the `and` operator for combining multiple filters was not working properly. Other users also stated they had issues with that.<sup>14</sup> The Strapi developers then often mentioned they would rewrite their complete API and proposed that this should therefore be fixed with the v4.<sup>15</sup> In combination with the announcement of a limited maintenance phase of the v3 just for security reasons<sup>16</sup>, we decided to migrate as soon as possible.

Unfortunately, the official migration guides were not released until the Mid of March 2022.<sup>17</sup> As this marks a gap of three months in comparison to the first official release of the v4, which took place on November 30, 2021 [7], we started our migration approach which is described in detail later in chapter 6. Along with the new directory structure regarding plugins, which made it necessary to partly rewrite and adapt our mentioned bulk-import plugin, the main point during the migration was the introduction of the “Unified Response Format” that the REST and the GraphQL API are based on (see subsection 5.4.7). The recursive use of the `data` and `attributes` key in the response objects did not exist in v3 and the introduction sparked a lot of discussion in the community [40]. It was also a pain point, which we solved with some special handling in our frontend application. More on that can also be found in chapter 6.

### 5.6.6 Alternative Systems

There are many alternative headless Content Management Systems in the competitive market. Two popular of these are Kontent<sup>18</sup>, which even promotes itself with wordings like “Strapi Alternative - Not satisfied with Strapi?” (see Figure 5.3), and Prismic.<sup>19</sup>

Both provide a REST and a GraphQL API for accessing and modifying the stored content. However, they are not open-source projects and offer only limited, free plans<sup>20,21</sup> and seem to also not have a solution for defining custom relationship

<sup>14</sup><https://github.com/strapi/strapi/issues/10419> (last accessed: 2022-07-13).

<sup>15</sup><https://github.com/strapi/strapi/issues/9748> (last accessed: 2022-07-13).

<sup>16</sup><https://github.com/strapi/strapi/issues/11726> (last accessed: 2022-07-13).

<sup>17</sup><https://github.com/strapi/documentation/pull/790> (last accessed: 2022-07-13).

<sup>18</sup><https://kontent.ai/> (last accessed: 2022-07-13).

<sup>19</sup><https://prismic.io/> (last accessed: 2022-07-13).

<sup>20</sup><https://prismic.io/pricing> (last accessed: 2022-07-13).

<sup>21</sup><https://kontent.ai/pricing/> (last accessed: 2022-07-13).

Anzeige · kontent.ai

## Strapi Alternative - Not Satisfied with Strapi?

Companies like yours prefer using Kontent to Strapi. Find out why. Kontent by Kentico is better for your project than Strapi. Try it yourself.

#1 Headless CMS on G2 · Integrations & Extensions · Top Headless CMS Features · Build Powerful Projects

**Figure 5.3:** Kontent promoting itself as an alternative to Strapi

attributes<sup>22,23</sup> Furthermore, they lack transparency on how their connections to the underlying data stores are working, so it is questionable how the performance regarding the described search scenario would have looked like if we had chosen one of these systems.

## 5.7 Summary

Throughout this chapter, we discussed the usage of Content Management Systems and the experiences we made with Strapi, an open-source representative of such systems while developing a crowd-sourced picture archive.

In summary, one could argue that Strapi follows the “80–20 Principle” [50]. Many use cases for developing a content-based API with the help of a Content Management System are covered by the default interfaces of Strapi making it easy to quickly deploy a collaborative content system. However, there are certain advanced use cases that Strapi is not capable of by default. In cases like these discussed, custom code needs to be developed and other general customizations made. Strapi is aware of that fact and provides developers with easy entry points into their APIs and services to develop custom code and solutions efficiently. Strapi’s community is continuously growing, encouraging the culture of open-source software. In combination with our considerations regarding alternative Content Management Systems, we recommend keeping Strapi as the CMS of choice to support the picture archive.

Regarding our project, certain aspects remain for future work as well. If additional relationship attributes are to be supported, choosing an alternative solution for supporting such attributes in the first place will be necessary. Additionally, depending on how the number of digitized pictures is growing in the future, setting up an external search engine might also be one of the most important improvements to the system.

<sup>22</sup><https://kontent.ai/learn/reference/management-api-v2/#section/Linked-items-type-element> (last accessed: 2022-07-13).

<sup>23</sup><https://prismic.io/docs/core-concepts/link-content-relationship> (last accessed: 2022-07-13).





## 6 Automatic Data Migration, Testing, and Deployment

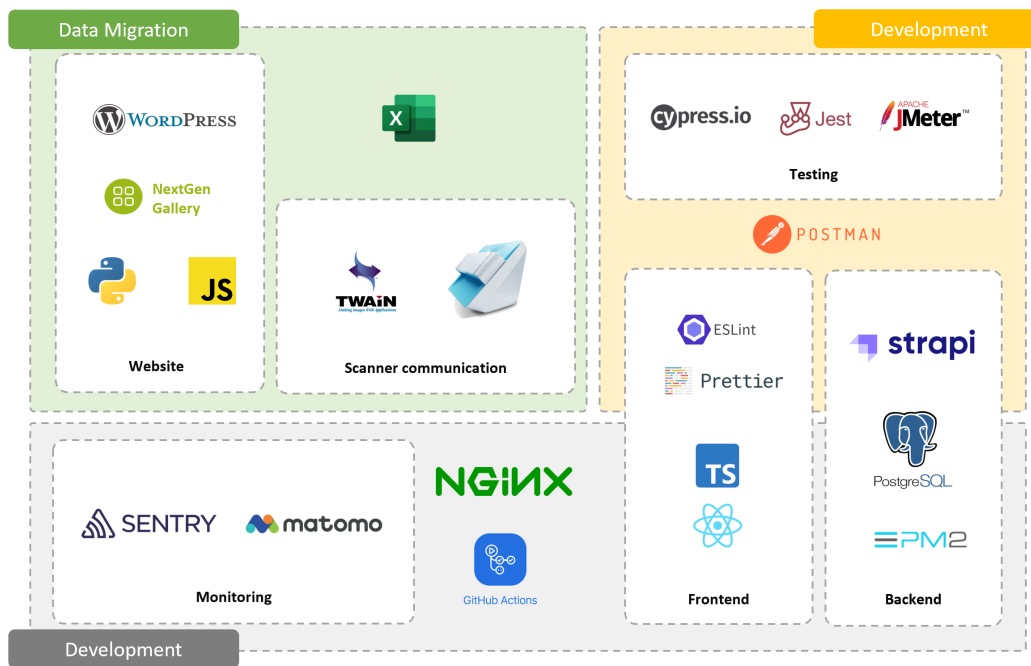
In this chapter, we will highlight how various tools and automation were used to aid the development process. We will provide an overview of three topics of importance during all stages of our project: Data migration, Developer Experience, and deployment.

### 6.1 Tools and Tasks in Modern Web Development

Developing and deploying a web application such as our crowd-sourced picture archive was a complex process for our developers. Furthermore, the growing number of components and tools available when developing the application combined with agile practices demanded a flexible approach when choosing the software components used to support said process. As seen in chapter 5, tools such as Strapi can help quickly set up a stable system and keep it up-to-date. However, there is no definite tool-based solution for developing maintainable software and providing each developer with the appropriate instruments needed to do so. Considerations had to be made during development on what to use and when. That is why it is crucial to illuminate the topics of data management, Developer Experience, and Continuous Integration and Deployment, as they have become integral parts of tackling the task of producing larger software systems [46].

The following chapter aims to provide an overview of these broader topics in the context of the development process of our software. We want to focus on highlighting the tools used and the types of considerations our team had to make. We also address a few crucial points concerning data migration since our goal was to replace a system already running and storing data. This meant thinking about how to treat the data we were working with and how to manage the multitude of forms data appeared in. An overview of all tools used, which can be separated into three distinct areas, is pictured in Figure 6.1.

The next section of this chapter will illustrate the steps needed to migrate the data from the legacy system. The third section will then discuss how we managed the data afterward and during the project. The fourth section will overview the tools which helped us during development. In the fifth section, we will illustrate the steps necessary for deploying the system and the methods we set up to monitor it. We will conclude in section six with a summary.



**Figure 6.1:** The tools used during the project. The migration of WordPress data and the role of Python and JavaScript is further explained in section 6.2

## 6.2 Data Migration: Importing from WordPress

The core focus of this work was to increase the ease of development for people participating in the project while reducing the overhead as much as possible. This can be seen in the first challenge our team faced when starting: Understanding the legacy system and migrating the data to our new data store and format. According to J. Morris [43], data migration is the “selection, preparation, extraction, transformation and permanent movement of appropriate data that are of the right quality, to the right place, at the right time and decommissioning of legacy data stores.” Data migration solves a core problem: Different systems and data stores assume varying data formats. There is a multitude of reasons for deciding to migrate data [39]:

- Combining existing data stores as a result of unifying systems,
- Upgrades to newer systems or different data stores,
- Outside regulations that lead to new requirements,
- Changes in processes that lead to new functional requirements that are no longer supported by the existing data structure.

When starting our project, we found ourselves dealing with the second of these reasons. As has already been detailed in chapter 1, while our software was developed to find a better way of solving the project partner’s requirements, it was not the first one to try. A legacy system based on the NextGEN gallery plugin for WordPress<sup>1</sup>

<sup>1</sup><https://wordpress.org/plugins/nextgen-gallery/> (last accessed: 2022-07-13).

was already in place to manage around 7.600 pictures. However, since that system did not successfully address all requirements, those pictures had to be migrated to our application to prevent reentering any already digitized information. This posed a challenge since the original system was not intended to be decommissioned until the new one was finished. Even though we agreed with the project partner that they would manage no new pictures or comments on the legacy system, we could not rule out the possibility that the initially migrated data would become stale. So we potentially had to migrate the data multiple times.

To successfully migrate data from a legacy data store to a new one, four steps are necessary [39]:

1. Exporting appropriate data from the legacy data store (selection, preparation, extraction),
2. Cleaning up the data and preparing it for import (transformation),
3. Importing the data into a new data store (permanent move to the right place),
4. Decommissioning of the legacy data store.

Following these four steps, we migrated the data from WordPress to Strapi.

### 6.2.1 Exporting from WordPress as a Legacy Data Store

The data that needed to be transferred was split across two stores forming the legacy system. While the pictures themselves, along with their metadata, were managed by the NextGen gallery plugin, the comments, as well as the folder-like structure they were placed in, were provided by the enclosing WordPress installation. This meant exporting was not as simple as gathering data from one database but rather a multi-step process.

First, the pictures themselves were stored on the server's hard drive as JPEG files and could be downloaded. The NextGen gallery organizes these files in named albums, each corresponding to a folder with the same name. These albums, the picture metadata including tags, as well as the comments were all stored in tables in an SQL database. WordPress allows the export of these tables as TSV files, which gave us three intermediate data containers (*album.tsv*, *comment.tsv*, *pictures.tsv*).

As mentioned in chapter 2, our project partner's requirements for organizing pictures were fulfilled by the options provided by NextGen gallery. They tried to solve this by embedding standalone gallery instances, each displaying one album, onto different web pages. Those pages were then connected using hyperlinks, creating the virtual experience of a hierarchical folder structure for visitors. Exporting this structure proved more complex than the previous step since the links were not created uniformly, and a central export of WordPress pages was not easily possible.

To solve this problem, we wrote a JavaScript code snippet to scrape the pages. For this script, we used the `fetch` function to get the HTML code of a target page and then utilized JavaScript's built-in `querySelectorAll` function to get description texts, `img` tags, and links leading to other pages based on their respective CSS selectors (see Listing 6.1). This function was then included in a loop iterating over a URL queue,

which eliminated cycles while traversing the page tree by checking if a URL had already been called by a previous iteration.

**Listing 6.1:** Using JavaScript’s `querySelectorAll` method to traverse the page

```

1  const allImageIds = [...doc.querySelectorAll('.ngg-gallery-thumbnail')]
2    .map(i =>
3      i.querySelector('a').getAttribute('data-image-id')
4    );
5  const allLinks = [...doc.querySelectorAll('.text a:not([data-image-id])')]
6    .map(a => a.href);

```

The export process provided us with the TSV files mentioned as well as the album structure as a JSON file. All information was present now, though it had to be reinterpreted more fittingly that more closely resembled our target data structure.

### 6.2.2 Cleaning Up and Transforming the Data

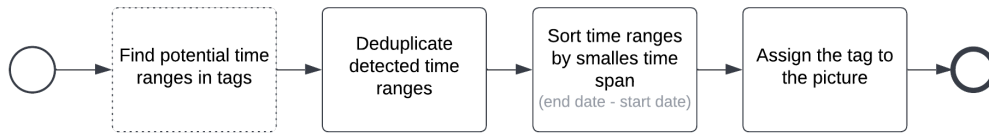
To prepare the exported data for the import, we wrote two cleanup scripts in Python. In this manner, two JSON files were produced. The first, created by `importAlbumsToJSON.py` as a result of the TSV export, mapped the albums to their respective pictures in a file called `albums.json`, a snippet of which is shown in Listing 6.2. This file stored comments, keywords, and a description for each picture alongside the filename. The second export file named `tagInfo.json` was produced by `importTagsToJSON.py` and stored the hierarchical relation between albums as recursive “children” fields, as well as additional information such as the title and description of an album. To prepare this data for import into the new system, some adjustments had to be made, such as replacing `&amp;` with `&`.

**Listing 6.2:** Extract of ‘albums.json’ file: mapping each album to its pictures and each picture to its metadata

```

1  {
2    "1": {
3      "dirname": "harzburger-musiktage",
4      "previewpic": "1",
5      "pictures": {
6        "1": {
7          "filename": "2020-06-16-11-49-0001.jpg",
8          "description": "Prof. Hermann Baumann, Naturhornwettbewerb",
9          "alttext": "Harzburger Musiktage",
10         "sortorder": "0",
11         "keywords": [],
12         "comments": []
13       },
14       [...]
15     }
16   },
17   "2": {
18     [...]

```



**Figure 6.2:** Extracting and assigning time-range tags from the other picture tags

**Handling Time-Range Tags** Since the target data model described in chapter 2 structured picture information more finely granulated than the source model, a one-to-one mapping of fields was impossible. Time-range tags are a fitting example of this because, though the time information was present on the original data (e.g., in descriptions, titles, etc.), it was not structured in a consistent, machine-readable way. Hence, we employed a heuristic to extract all possible time ranges from the source data. This heuristic was executed after the import steps mentioned in subsection 6.2.3 and its steps are pictured in Figure 6.2.

We identified potential time ranges in tags using various regular expressions (RegExp) to match common ways of expressing dates, years, and timespans found in the source data, meaning in common German date formats. This is done by checking a picture’s associated descriptions, titles, and albums against these RegExp and returning all matches. In the further steps, the potential candidates can then be reduced.

Since the heuristic always assumes the time range with the smallest time as the most fitting one, the results are not always correct. An example of this can be found in the album “Engländer in Bad Harzburg/Leave-Center 1945–1955”, whose description states “that on the 10th of April 1954, English troops came to Bad Harzburg. Many pictures in this album could only be dated as accurately as 1945–1955 or even less specific. Still, the heuristic assumed this date for every picture due to the more precise date being present in the album’s description. To deal with this, we used our tag verification method (see chapter 2) to mark every automatically generated tag as unverified so that tags can be corrected using further manual work in the future.

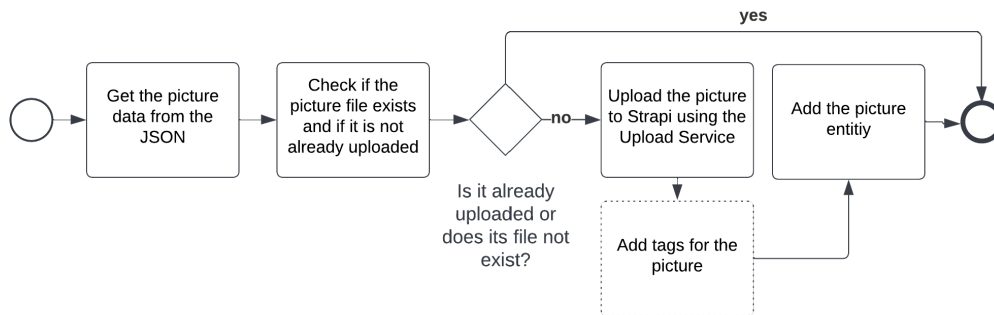
### 6.2.3 Importing Data into the New System

Using the Strapi Content-Type Builder (see subsection 5.4.3), we manually implemented content-types according to our target data model (see chapter 2), which created the necessary schema files. Having done that, we used Strapi’s built-in plugin system to create a new plugin named `bulk-import`, in which we added the endpoints needed to import the data.

First, we transferred the picture files to the server using SSH. The import process then consisted of sequentially calling three endpoints using REST requests and passing the JSON result files as POST parameters. First, the `/bulk-import/import`

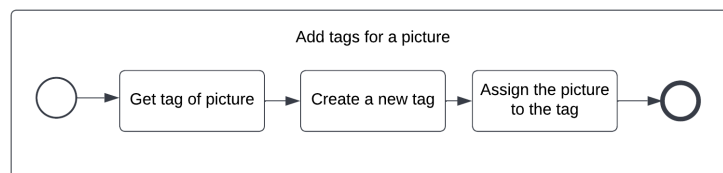
endpoint took care of creating all the necessary pictures, comments as well as tags associated with any picture: Keywords, titles, descriptions, and category tags. Please note that this reflects an older version of our data structures, as they were when we first imported the data. Further migrations must be executed as described in section 6.3 to keep this approach feasible for newer data model versions.

When called, the endpoint executes the process pictured in Figure 6.3 for each picture in each album in `albums.json`.



**Figure 6.3:** Uploading pictures using the bulk-import plugin follows this pipeline. The step of adding tags to the picture is illustrated in more detail in Figure 6.4

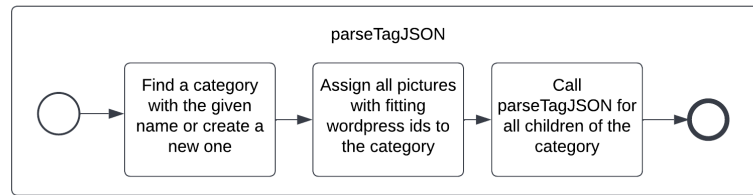
Here, we considered three potential tag types for any given picture: Title, descriptions, and keywords. For each of the tags, we first checked if a tag with the provided content already exists and, if not, created a new one (see Figure 6.4).



**Figure 6.4:** Adding tags to a picture

In that manner, we could ensure that when we created the picture, we assigned it a valid entity and also deduplicated content during the import by re-using already imported strings multiple times. We also stored the unique WordPress id for each picture so that they can be referenced in later execution steps. After completing these steps, all pictures were imported as entities. However, the album structure was not yet translated into categories for the new system. For this, we iterated over all albums once more, this time using the `tagInfo.json`, whose “children” structure allowed us to use a recursive approach for creating categories (see Figure 6.5).

Here, we matched pictures to their correct categories by identifying them via their WordPress id, which is assigned to the album in the `albums.json` file. From there on,



**Figure 6.5:** Parsing the tag json file

only the comments were missing in the new dataset. We developed another endpoint named `/bulk-import/import-comments`, which took care of this. Like the category tag endpoint, this one used the WordPress id to assign the correct comments to the right picture.

#### 6.2.4 Decommissioning the Legacy Data Store

As stated above, we wanted to keep the WordPress system running while developing the new system. That meant that the fourth step had to be deferred to a later stage, i.e., the production deployment of our software (see subsection 6.5.4). The decommissioning was completed as soon as there was no way of publicly accessing the old website.

#### 6.2.5 Sidenote: Legacy Compatibility between Strapi Versions

While our team tried to keep the possibility of migrating data from its original state completely into the latest system iteration, there was one point at which we had to take breaking changes into account: The switch from Strapi v3 to Strapi v4. Since there was no migration guide yet (see subsection 5.6.5) but we needed several features not present in the v3 release, we decided to upgrade anyway. That meant adjustments to the import script had to be made as well. So, instead of opting to migrate the already model-compliant data from our Strapi v3 instance, we dropped the data and triggered another WordPress import using the same input files as before, just using the new Strapi v4 content API.

However, during our initial testing phase, a small number of comments had already entered the system. These could not be moderated yet but also held valuable information. We added a new endpoint named `/bulk-import/import-draft-comments`, which when called migrated these comments from Strapi v3 to the v4 instance using a custom JSON format. We built this endpoint like the `import-comments` endpoint with the additional step of setting `publishedAt: null` for every new comment, thus telling Strapi the comment was a draft (see subsection 5.4.3). The imported comments were then moderated using the production system at a later stage.

## 6.3 Data Migration: Evolving the System

Having imported the original data once and prepared endpoints for later re-importing was not enough. As mentioned in section 6.2, changes to functional requirements could also lead to changes in the data structure and possibly necessary migration. A shift in domain understanding can cause such a change. During the project's runtime, such shifts took place on multiple occasions, meaning data migration became a continuous process. The main challenge was migrating from one version of a given data model to a revised version while ensuring all previous, as well as the original WordPress data, could still be used. A pipeline was created to migrate data from any state to the following one.

### 6.3.1 Migration Use Cases

Simple migrations, e.g., changing the data schema from one version to the next, were simplified using Strapi's built-in migration functionality (see subsection 5.4.3). It was only necessary to restart the Strapi application with new schema files generated by the Content-Type builder for the changes to be applied to the database automatically. However, adjusting the data itself was not as easy since, as described in subsection 5.4.3, new model fields could contain information from different prior fields. We would have lost data if we only created the fields and dropped the old ones. Currently, Strapi offers a migrations system for these kinds of challenges. However, this system was not well documented enough when we first encountered this issue. Therefore, we decided to extend our `bulk-import` script to offer additional endpoints for three specific use cases that could be called to trigger the steps of our migration pipeline. In this manner, an arbitrary state of the data model can be reached by adapting the schema files using Strapi's Content-Type builder and calling the necessary steps of the migration pipeline manually.

**Removing Titles From the Data Model** The first of those steps – accessible via `/bulk-import/migrate-titles` – was introduced once the decision was made to drop titles from our data model. Since most titles did not contain valuable information, such as the date the picture was scanned or an ascending enumeration, a large part of this data could be removed. We used the regular expression shown in Listing 6.3 to identify these titles and filter them out.

**Listing 6.3:** The regular expression used to filter out titles that are just timestamps. The file naming schema was determined by the previously used scanner software.

```
1 /(\d{4})-(\d{2}-){4}(\d{4})/gm
```

Other titles were merely duplicating information already present in a picture's description. To remove these entities, we identified the pictures associated with any given title and compared their descriptions to the title at hand. If any assigned



description's string contained the title (checking case-insensitive), the title object was removed. In a third step, we wanted to reduce the number of title entities further by consolidating titles that solely differed in a suffix number (e.g., "sommerfest 1", "sommerfest 2"). Of all remaining entities, we assumed there to be some amount of valuable information in them so that we could not just drop them. Instead, we opted to convert those titles into descriptions and assign them to their respective pictures. Since our configuration of description content types allowed us to enter formatted text, we wrapped these titles into HTML-Header-Tags (`<h1>`) so they stood out from the other descriptions. After successfully running the script, we safely removed all titles, as well as the Content-Type "title", without having to fear the loss of information.

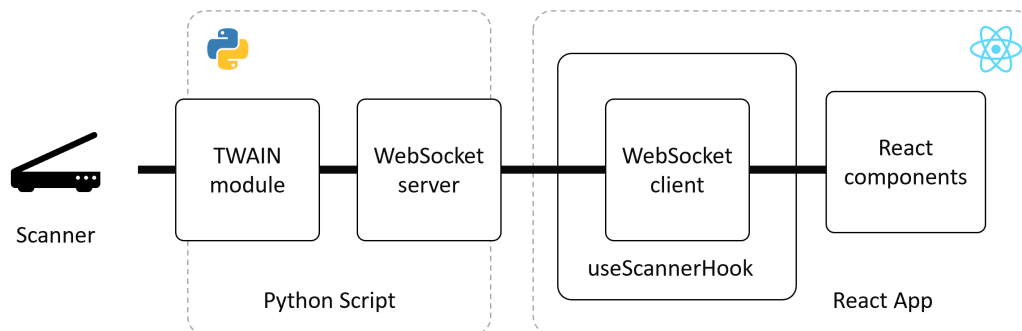
**Converting Category Tags to Collections** A similar approach was taken once we decided to switch from category tags to collections for manual picture organization. We added an endpoint at `/bulk-import/migrate-collections`, which assumes a content-type named "Collection" has already been created. It then iterated over every previous category and created a corresponding collection with the same name, description, and pictures. However, while categories worked using "related tags" (see chapter 2), collections employ a more strictly hierarchical approach. This meant that related tags of previous categories had to be mapped to child collections and parent collections of the new collection. This was only possible because our team implicitly decided earlier to use related tags hierarchically. After this script was executed, the category tags content type, as well as the data, were safely deleted. The first time we ran this script, we opted to assign each picture to the collections they were contained in as well as to the parent collections of those. However, this made querying pictures direct belonging to a collection a lot more computationally intensive at runtime, which is why we decided to reduce the relationship between pictures and their collections to just the immediate parents. For this, the `/bulk-import/reduce-picture-collection-relations` endpoint was introduced, which selected the most specific collection based on their parent-child relations and assigned it to the picture.

**Introducing Archive Tags** As a last variation point, after introducing the archive tag type, it was decided every picture should have one archive assigned, so every picture already present in the system should be assigned to a certain default archive. The new endpoint named `/bulk-import/add-default-archive-tag` took care of this by iterating over all pictures and assigning the default archive tag.

### 6.3.2 Importing Data via the Scanner

Because we did not want to halt the scanning process during project development, we introduced temporary migration measures, such as using an Excel sheet as an intermediate data store. More on this approach and its flaws can be found in subsection 6.3.3. Once the upload and curator interface was finished, data migration had taken on another meaning: Primarily migrating data from a completely analog data store, meaning physical pictures, to the digital realm without needing any

third-party data stores. The process of doing so was unoptimized before since the digitized pictures still had to be uploaded and matched to their tags manually. As explained in chapter 3, we wanted to enable direct scanning into the web application. This proved difficult since we found no easy and affordable way of interfacing with a scanner directly from a browser. A workaround employed by the solutions we found is to use an application running on the host system interfacing with the local web application via REST or Websockets.<sup>2</sup> However, these solutions were very costly and not open source, so we decided to develop a custom program for this. It is written in Python and follows the same principle as related software, shown in Figure 6.6. It uses the Python `TWAIN` module to communicate with the device, and the `websockets` module<sup>3</sup>, to connect to the browser.



**Figure 6.6:** Components necessary for scanner communication

The script must run on the host operating system to use the scanner. It then provides various operations which can be triggered by the web application sending an operation code along with the necessary parameters in the form `[operator] <...parameters>`. The `receive` function of the script then splits the received text at whitespaces (“ ”) and executes one of the operations listed in Table 6.1.

The scanner communication itself is done by addressing the `TWAIN` module functions, as shown in Listing 6.4. Here we use the `SourceManager` object provided by the library to access the selected scanner and then request the scanned image using `RequestAcquire` and `xferImageNatively`. To back up the image and prepare it for further pre-upload modifications, such as automatic cropping, we store the scanned data in a bitmap file called `tmp.bmp`. Then we execute these modification steps if needed and append the scanned image to a buffer array, which will be sent via the WebSockets after the process.

<sup>2</sup><https://asprise.com/document-scan-upload-image-browser/direct-to-server-php-asp.net-overview.html> (last accessed: 2022-07-13).

<sup>3</sup><https://github.com/aagustin/websockets> (last accessed: 2022-07-13).

**Table 6.1:** Possible operations of the scanner script

Operation	Parameters	Description	Response
list	—	List all available scanning devices	{"list": string[], "selected_scanner": number}
scan	—	Obtain one or multiple images from the currently selected scanner	<b>(Images as Blob Array)</b> see below
set_scanner	scanner_id: number	Set the currently selected scanner	{noop: true}
set_crop	auto_crop: boolean	Turn automatic cropping on/off	{noop: true}

**Listing 6.4:** The function used to interface with the scanner device. The specific implementation depends on the scanner type and only works with our test model – the Plustek ePhoto Z300.

```

1 def get_images():
2     buffer = []
3     ss = sm.OpenSource(sm.GetSourceList()[current_scanner_id])
4     scan_start_time = time.time()
5     while True:
6         try:
7             ss.RequestAcquire(0,0)
8             rv = ss.XferImageNatively()
9             if rv:
10                (handle, count) = rv
11                file_name = 'tmp.bmp'
12                twain.DIBToBMFile(handle, file_name)
13                cropFileIfNeeded()
14                with open(file_name, 'rb') as file:
15                    scan_start_time = time.time()
16                    buffer.append(file.read())
17            except (twain.excTWCC_SEQERROR, twain.excDSTransferCancelled):
18                ss.destroy()
19                ss = sm.OpenSource(sm.GetSourceList()[current_scanner_id])
20                if time.time() - scan_start_time < 5:
21                    continue
22                else:
23                    break
24            except Exception as ex:
25                [...]
26        return buffer

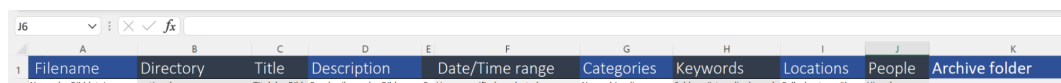
```

We wrapped the code in an endless loop to allow the scanner to repeatedly scan images without having to re-request the scan process. To be able to interrupt it, we introduced a timer. This timer is reset anytime a successful scan is made. Every time the scanner tries to obtain a scan, and no picture is available, it will throw the `twain.excTWCC_SEQERROR`. We catch this error and check if more than 5 seconds have

passed since the last successful scan. If so, we break out of the loop and send the result buffer. If not, we continue to try scanning pictures. If any other uncaught exception occurs, we display the error. If the browser requests to crop the image automatically, a call to the included cropping script named `cropImages.py` is made. It reads the `tmp.bmp` file produced by the main scanner script and crops the image according to an auto-crop Python script found on GitHub.<sup>4</sup> To make execution easier for our system’s curators, we converted the script into a Windows Executable (`.exe`), using the `nuitka` Python compiler.<sup>5</sup>

### 6.3.3 Importing Data from Excel

We wanted to make sure not to halt the scanning process of pictures while we were developing our application. However, we also wanted to make a transition to the new system and the new data structure easier than preparing another WordPress export, so we introduced an interim solution: A Microsoft Office Excel sheet prepared with columns that reflected the fields of our intended model structure (see Figure 6.7). This sheet was handed to the project partner and used to bridge the time until a sufficient upload interface could be provided. The pictures were still scanned using the previous system’s method, but instead of loose storage, the information was directly recorded in a structured way.



Filename	Directory	Title	Description	Date/Time range	Categories	Keywords	Locations	People	Archive folder
----------	-----------	-------	-------------	-----------------	------------	----------	-----------	--------	----------------

**Figure 6.7:** The sheet’s header row shows the columns that map to the tag types our new data model assumes

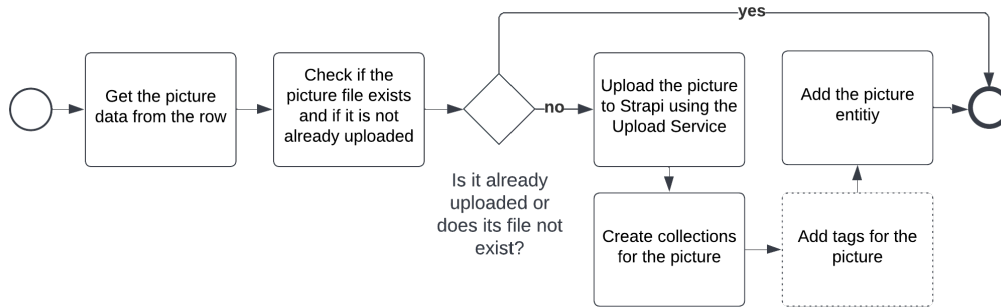
This approach was used to manage approx. 500 pictures. After the upload interface was finished, we extended the import plugin once again to add an endpoint for importing from the Excel sheet, named `/bulk-import/import-from-excel`. In this endpoint, we used the `sheetJS` node module<sup>6</sup> to convert the row data into JSON. The picture files were once again uploaded to the server via SSH, but the Excel file was passed as a POST parameter. The upload process then followed mostly the same steps as the WordPress import script, albeit with a few modifications to allow for the import of new tag types (see Figure 6.8).

Some unforeseen issues appeared during this stage. For example, we wanted to ensure the data entered was definitely in the proper format, so we added validation rules to the file name and the “Date/Time-range” column, which worked as intended. However, for the tag columns, we wanted to have the values separated by a semicolon.

<sup>4</sup><https://github.com/z80z80z80/autocrop/blob/master/autocrop.py> (last accessed: 2022-07-13).

<sup>5</sup><https://github.com/Nuitka/Nuitka> (last accessed: 2022-07-13).

<sup>6</sup><https://github.com/SheetJS/sheetjs> (last accessed: 2022-07-13).



**Figure 6.8:** Steps for importing picture metadata from the Excel sheet

But due to unfortunate communication misunderstandings and inconsistencies during the digitization process, some cells used commas instead. To mitigate this issue, we loosened our input requirements and used another RegExp to split by both commas and semicolons. This led to other problems for entries where commas were used in the intended data. Those instances had to be manually corrected after the import.

## 6.4 Developer Experience

Besides migration concerns, we also focused on providing a pleasant Developer Experience (DX), which is the perception of work a (software) developer experiences while working on a project [19]. It can be influenced by a multitude of factors, including the environment in which the development process occurs, the tools used during this process as well as the structure of the process itself. It was important to us to create a DX that enabled us to achieve high productivity. This meant that along with our agile methods, we decided on a technical development stack that should support creating maintainable software as well as possible. This included the environment in which to develop the software as well as accompanying tools for assuring compliance with self-imposed coding guidelines such as the code style.

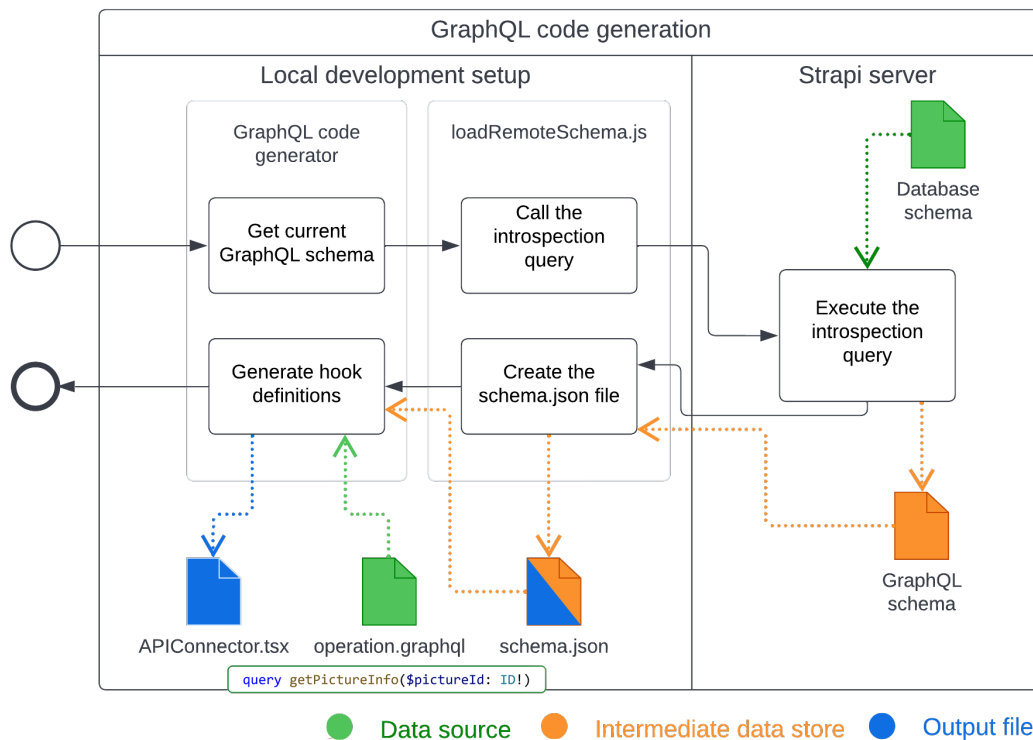
### 6.4.1 Typed Web Development with TypeScript and GraphQL

Our first decision was which programming language to use. We decided on using TypeScript, an extension of the popular JavaScript language. It was developed to make the creation of large-scale web applications more viable and improves on the default features of JavaScript by providing a module system, classes, interfaces, and a static typing system. TypeScript [41] can drastically improve the DX by providing features like on-the-fly type- and null-checking. Because of these advantages and React supporting both languages, we quickly decided on using TypeScript instead of JavaScript when setting up the project. This meant we would be sure to produce type-safe code and use the correct interfaces.

A similar thought process led us to closely connect GraphQL to our frontend since it offers type support out-of-the-box, albeit with less variation than TypeScript. We

used this to ensure every API call is valid according to our current data state and that every interaction with the API is consistent.

**Generating React Hooks from GraphQL Specifications** To automatically create the TypeScript types from the GraphQL specification, we made use of the GraphQL code generator<sup>7</sup>, which can automate the generation of typed queries, mutations, and resolvers. This module takes the current GraphQL schema as input and generates the necessary hooks for the Apollo Client. These can be imported and used in the React component, meaning API calls are always valid according to the current schema specification. A typical pipeline for generating the code follows the steps shown in Figure 6.9.



**Figure 6.9:** The pipeline used when generating the API code for the frontend. The `APIConnector.tsx` file contains Apollo hooks to be included in React code.

The Apollo Server provides an Introspection Query, which produces a schema specification when called. This specification serves as a basis for further code generation. A custom schema loader located in `load-remote-schema.js` calls the Introspection Query, writes its result in a `schema.json` file, and returns it to the code generator. The `schema.json` file only serves as a fallback in case the server is not reachable at

<sup>7</sup><https://www.graphql-code-generator.com/docs/getting-started> (last accessed: 2022-07-13).

the time of generating the API. It should not be used during deployment since it can introduce inconsistencies in the code compared to the remote schema. However, it allows the generator to function during development even when no remote schema provider is available, improving the DX. The configuration of the code generator is stored in the `codegen.yml` file, detailed in subsection A.3.1.

**Simplifying the Response Format** As explained in subsection 5.4.7, Strapi introduced a “Unified Response Format” in version 4.0.0, which enforces a recursive structure consisting of a data layer including an `id` and `attributes` field. Because the data structures used by our software heavily rely on relations between models, the layer hierarchy can become very deep, leading to long access statements like the one listed in Listing 6.5. This reduces code readability, thus impairing the DX.

**Listing 6.5:** Accessing the URL of a picture with “Unified Response Format”

```
1 const pictureUrl = data?.picture?.data?.attributes?.media.data?.attributes?.url;
```

To improve the readability of statements like this, we wanted to simplify accessing the targeted fields by simplifying the returned JSON data. For this, we introduced a `queryUtils.tsx` file containing the `flattenQueryResponseData` function. This function recursively traverses the structure and remove any data and attribute layers, reducing structures like `<object>.data.attributes.<attribute>` to `<object>.<attribute>` while maintaining attributes outside this structure such as `id`. To efficiently use this function in React components, we wrapped it inside a custom `useSimplifiedQueryResponseData` hook, returning a memoized value.

**Listing 6.6:** Using the `useSimplifiedQueryResponseData` hook inside React components makes working with the API data much simpler.

```
1 const picture: FlatPicture | undefined =  
  ↪ useSimplifiedQueryResponseData(data)?.picture;  
2 const pictureUrl = picture?.media.url;
```

Having established this functionality meant that all API calls had to follow a similar structure where the simplifying hook is called immediately after calling the GraphQL hook, as shown in Listing 6.6. This was something all developers had to take into account, but it led to increased readability across the rest of the code. It also meant that the script could perform other transformation steps on the API data, which came in handy when implementing the `verified` flag.

As discussed in subsection 5.6.1, each picture has two relations to any given tag – one for tags which can verifiably be assigned to the picture and one where that is not the case. To further increase the developer friendliness of this feature, we wanted to present this as a simple boolean flag on the relations. We used the simplifying

hook to transform the data with the `mergeVerifiedWithUnverifiedData` function. This allowed us to handle the verification state of any given relation as a boolean flag `verified`.

### 6.4.2 Development Toolchain

To set up a local development environment, the frontend, as well as the backend, have to be configured. We used the `yarn`<sup>8</sup> package manager instead of the popular `npm`. To set up the frontend, three steps are necessary: First, the new code must be cloned on the local machine. Then, the project must be installed via `yarn install`. Finally, the API connection hooks must be generated using `yarn generate-api`, which uses the GraphQL code generator as described in section 6.4.1. The latter command must run whenever the database schema, mutations, or queries change.

Despite having unanimously decided to use TypeScript as our frontend development language, our developers still used different operating systems and varying coding setups and configurations, meaning there were still many variation points. We, therefore, employed various automatic tools to maintain consistency across the produced software, e.g., in the code style. We found these tools to be essential in creating a pleasant DX.

To establish a consistent code style, we employed the code format checker `eslint`<sup>9</sup> and the code formatter `prettier`.<sup>10</sup> We also introduced a “Git-Hook” [4] that is triggered every time changes concerning the React application are committed to the repository. For this, we make use of the packages `husky`<sup>11</sup> and `pretty-quick`.<sup>12</sup> `husky` enables us to configure more flexible git hooks for different stages of the git development process, while `pretty-quick` will refer to the configured formatter but is capable of detecting which files have been changed in comparison to the last revision. Hence only changed files will be re-formatted.

While these tools improved working on the project in general, they could also be the source of inconveniences, such as problems with the IDE detecting the linting settings or strict TypeScript rules that prevented quick code fixes and checks.

## 6.5 Integration, Deployment, and Performance

While improving the development process, we also concentrated on repeatable testing and deploying the application as another major concern. This section will highlight how we set up our testing frameworks and a Continuous Integration pipeline, as well as how we built an automated deployment. Once we had a production system, we also installed tools for monitoring it. Lastly, we implemented a backup system.

---

<sup>8</sup><https://classic.yarnpkg.com/en/docs/cli/install/> (last accessed: 2022-07-13).

<sup>9</sup><https://github.com/eslint/eslint> (last accessed: 2022-07-13).

<sup>10</sup><https://github.com/prettier/prettier> (last accessed: 2022-07-13).

<sup>11</sup><https://github.com/typicode/husky> (last accessed: 2022-07-13).

<sup>12</sup><https://github.com/azz/pretty-quick> (last accessed: 2022-07-13).



### 6.5.1 Testing Frameworks and Continuous Integration

In order to ensure stability and robustness of the deployed software, we set up a Continuous Integration Pipeline using Github Actions.<sup>13</sup> For this, we decided on using two fundamental levels of abstraction for our tests: Unit testing as well as integration/e2e testing. Unit tests were written using Jest<sup>14</sup>, and the integration tests were written using Cypress.<sup>15</sup> Because the developed system consists of a frontend as well as a backend, both had to be tested. The unit tests for the frontend and backend are executed in the same actions workflow on every push to any branch. Here, linting is also done using `yarn lint`. Since the integration tests are a lot more computationally expensive, they are managed by a separate workflow that is only executed once a pull request to the `main` or `staging` branch is opened or if the request is marked as “ready for review”. However, the integration tests can also be executed manually in a Github Action anytime if necessary.

**Unit Testing** In the backend, since many parts of Strapi’s CMS functionality were used as-is and because the Strapi maintainers have implemented tests for this functionality, we focused on testing only the parts we customized, i.e., the lifecycle hooks as well as the custom GraphQL resolvers and types. We closely followed the Strapi testing guide<sup>16</sup> when creating the unit tests, only adjusting the test structure to fit the current Strapi v4 approach. In the frontend, `jest` was used as well. We decided on testing component-wise and placing the test files directly next to the component to make it easier to map tests to their respective subjects. A typical unit test then looks as shown in Listing 6.7.

**Listing 6.7:** The structure of a typical unit test. We decided on using the ‘describe-it’ pattern for testing.

```

1  it('should render the picture', async () => {
2    const { container } = renderWithAPIMocks(
3      <PictureView initialPictureId='1' />,
4      GetPictureInfoDocumentMocks
5    );
6
7    await waitFor(() => {
8      const imageTags = container.getElementsByTagName('img');
9      expect(imageTags).toHaveLength(1);
10     expect(imageTags.item(0)).toBeInTheDocument();
11     expect(imageTags.item(0)).toHaveAttribute('src', asApiPath(imageURL));
12   });
13 });

```

<sup>13</sup><https://github.com/features/actions> (last accessed: 2022-07-13).

<sup>14</sup><https://jestjs.io/> (last accessed: 2022-05-20).

<sup>15</sup><https://www.cypress.io/> (last accessed: 2022-07-13).

<sup>16</sup><https://docs.strapi.io/developer-docs/latest/guides/unit-testing.html> (last accessed: 2022-07-13).

The `renderWithAPIMocks` function used in the test is located along with some other helper functions in the `testUtils.tsx` file. It calls the default `render` method provided by the React testing library and wraps everything in an Apollo `MockedProvider`, which can load mocks from a specific definition, in our case located in files named `mocks.ts`, and intercept calls to specific GraphQL operations with pre-defined return values.<sup>17</sup> Since the `MockedProvider` depends on the mocked requests exactly matching the requests made by the components, mocks have to be kept up-to-date when changing code. This circumstance heavily impaired the Developer Experience since this had to happen quite often.

**Integration and E2E Testing** Unit tests focus on testing a very specific small code segment. To be able to test a whole user journey through our application, we employed integration/e2e tests by using Cypress. Cypress tests are stored in the top-level directory named `cypress/integration`. The advantage of using such a framework was that we were able to simulate real browser interaction and thus get as close to what a user using the application would see as possible. To achieve this goal, we needed mock data that had to reflect the real production data while also staying compact and minimal. For this, we set up a fresh system instance and created the necessary data. It was then exported to an SQL dump using `pg_dump` (see section 6.5.6). When running the integration tests, we execute a bash script that handles setup as well as teardown. As can be seen in Listing 6.8, it creates the database and loads the test data from the previous export into it.

**Listing 6.8:** Creating a database and restoring the dumped test data to it using PostgreSQL tools

```
1 createdb -h localhost -U postgres -T template0 strapi-e2e
2 pg_restore -h localhost -c -U postgres -no-owner -d strapi-e2e ./e2e/data.sql
```

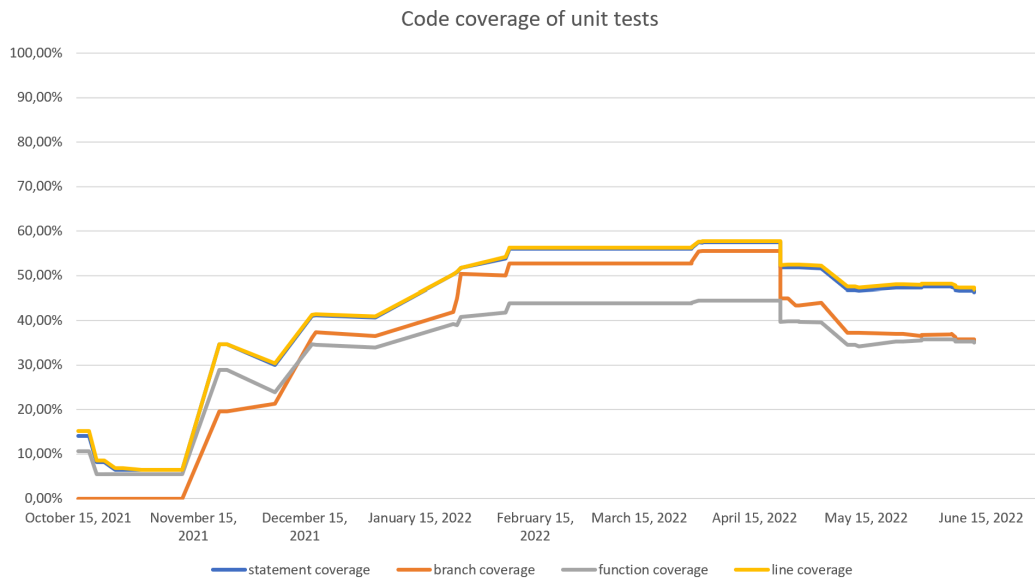
It then starts Strapi using the appropriate environment configuration (see subsection 6.5.4) and uses a busy-wait system to idle until it has started. Then, the integration tests are run using Cypress, and afterward, the Strapi process is killed.

### 6.5.2 Evaluating Test Coverage

Using the `--coverage` flag of Jest, we can plot the unit test coverage over time. This is shown in Figure 6.10. We can see that the coverage was at its highest in April 2022, at 59.5%. This rather low test coverage stems from the fact that during our initial prototype phase, we neglected to test essential code that became the basis for later developments. We never went back to fix this. However, we evaluated the added value and stability gained by the tests already written and concluded that the

<sup>17</sup><https://www.apollographql.com/docs/react/v2/development-testing/testing/> (last accessed: 2022-07-13).

most frequent errors were not being covered by the unit tests but rather problems concerning user interaction with the system. We decided to focus on integration tests to be able to test these use cases.



**Figure 6.10:** The code coverage of the unit tests. Each data point represents the test coverage at a commit on the main branch.

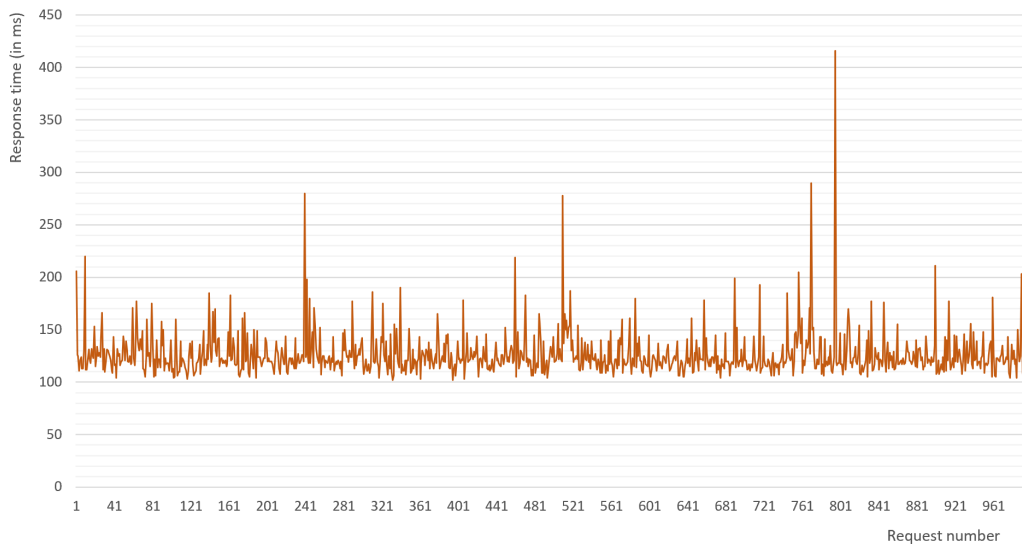
In retrospect, even though the integration tests were more difficult to set up, they allowed us to test the application in a way not possible by just employing unit tests. Therefore, use both unit and integration tests for our application.

### 6.5.3 Load Testing on the System

Not only the code's correctness but also availability was an essential factor when we evaluated our system's stability. To find the default response time a user could expect when using our application, we used Apache JMeter<sup>18</sup> to perform several load tests on the system, specifically our Strapi backend. The results of one of these tests can be seen in Figure 6.11. They show that when querying 100 pictures from the system, a response time of 127ms is expected. For this test, we simulated 100 users calling this endpoint ten times over 100 seconds.

The test query was a realistic candidate for a potentially problematic one since it is called often when visiting the website. Other queries are not executed as frequently and take similar response times. Hence, we conclude that the system can

<sup>18</sup><https://jmeter.apache.org/> (last accessed: 2022-07-13).



**Figure 6.11:** The Strapi server’s response time when querying 100 pictures via GraphQL, the mean is 127ms

withstand the expected loads. The same cannot be said for possible API calls outside this expectation.

We configured Strapi in a way that doesn’t set a maximum limit for return values so it gets easier for us to gather the responses we need from some GraphQL queries during the execution of our application, e.g., when querying subcollections for any given parent. During our load tests with approx. 15,000 pictures, we found out that such a query in large quantities can bring the server to a halt by filling the Knex transaction pool, thus effectively impairing the availability for all users. We made sure to employ Strapi’s pagination system in our application whenever the risk of querying too many entities occurs. However, since no authorization is needed for the GraphQL endpoints for querying pictures, an attacker could use them to provoke a denial of service from our server. Since a single operation for querying all pictures took more than ten seconds in our tests, and this only increased with more parallel requests, this is something that needs to be prevented in further work on the system.

### 6.5.4 Deployment

Like the Continuous Integration, our deployment was also planned to be automatic and continuous. We employ a staging strategy to deploy using two stages: “staging” and “production”. The staging instance is automatically deployed on every push to the `staging` branch, while a new production instance is deployed when pushing to `main`. We say the project is deployed once a build is available for access on our server. To make the application accessible from the web, we used the Nginx webserver.

Besides serving the built files and acting as a reverse proxy<sup>19</sup> for the backend system, it also handles secure connections using SSL by employing `certbot`<sup>20</sup> with the `--nginx` flag. The steps needed to deploy a new feature are as follows:

1. Setup a local instance of the front- and backend,
2. Check out a new feature branch,
3. Develop and test the new feature,
4. Open a pull request on the `staging` branch and request a review,
5. After merging the pull request, schedule a new production deployment using a pull request on `main`.

The deployment itself is carried out by two separate Github Actions pipelines, one for staging and one for production deployment. Each of the pipelines follows a similar structure for deploying the frontend and backend at the same time, making sure they are in sync. We outsource environment variables to `.env` files to have a single source of truth for all configurations and to avoid pushing sensitive information like the database password to the public repository. These files are already stored on the server. Furthermore, the pipelines make use of “Encrypted Action Secrets”<sup>21</sup>, small encrypted data containers that can hold information needed for the deployment, such as the SSH-Key for the server and user authorization information.

**Deploying the Frontend Application** Deploying a React application is as simple as calling `yarn build`, which triggers the `react-scripts build` command with the correct environment configuration. Here, we use the “`env-cmd`” package, which loads the correct variables into the environment. The build command uses “`Webpack`” to compile all resources into static HTML, CSS, and JavaScript files, as well as assets that are then transferred to the server using SSH and served using the Nginx webserver. The workflow pipeline, therefore, has to process these steps:

1. Check out the repository at the current state,
2. Install all project dependencies For this, we use `yarn's --frozen-lockfile` flag, which prompts it to install the modules as specified in the `yarn.lock` file without loading new versions of dependencies, thus making sure the built version behaves exactly like the development instance<sup>22</sup>,
3. Generate the API TypeScript code,
4. Compile the frontend code,
5. Transfer the files to the server using the `@appleboy/scp-action`<sup>23</sup>, overriding the old ones.

**Deploying the Backend Application** To start an instance of Strapi using the correct environment variables, we use the `cross-env` package.<sup>24</sup> Furthermore, we use

<sup>19</sup><https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/> (last accessed: 2022-07-13).

<sup>20</sup><https://certbot-prod.eff.org/> (last accessed: 2022-07-13).

<sup>21</sup><https://ghdocs-prod.azurewebsites.net/en/actions/security-guides/encrypted-secrets> (last accessed: 2022-07-13).

<sup>22</sup><https://classic.yarnpkg.com/en/docs/cli/install/> (last accessed: 2022-07-13).

<sup>23</sup><https://github.com/appleboy/scp-action> (last accessed: 2022-07-13).

<sup>24</sup><https://github.com/kentcdodds/cross-env> (last accessed: 2022-07-13).

the PM2 process manager to manage the Strapi production and staging instances. PM2 is a node-based “daemon process manager”<sup>25</sup>, which primarily solves three problems: Keeping the application running by restarting on failure, keeping track of logs, and managing the deployment. The PM2 configuration is specified in the `ecosystem.config.js` file located in the Strapi project folder. Here, the script used to start the application is defined as well as a restart policy, so the application will be restarted on failure (a maximum of 50 times). Also, in this file, the deployment stages are configured. Each stage sets SSH authentication data for logging in to the server, loading the host and user name from the environment variables, as well as the target repository to pull from and the branch to checkout. The SSH connection is secured by providing a key file named `deploy.key`, which should be located next to the configuration file. This file is produced during the Github action’s execution by reading from a secret and writing it to the correct file.

Using the command `pm2 deploy ecosystem.config.js <production|staging>`, the corresponding stage can be deployed. When this is executed, an SSH connection using the aforementioned login data is established, and the current data is fetched from the specified repository and branch. Then, `post-deploy` steps are executed. These steps include installing new dependencies if necessary (using `--frozen-lockfile` like for the frontend deployment), building the application, and then prompting the server’s PM2 instance to reload the configuration.

This makes the deployment process almost seamless and sets up the new Strapi instance automatically. The `pm2 save` command is then executed to store the currently running process list so that in case of an unexpected server restart, the processes are resumed without needing any administrative intervention.

### 6.5.5 Monitoring Deployed Applications

Despite meticulous testing and multiple deployment stages, it is not possible to assure flawless execution of the software on the end user’s hardware. More so: It should be expected that there will be errors and exceptions occurring. Not just because of the multitude of operating systems and browsers the system will run on but also because of programming errors not caught by tests or during staging.

To efficiently tackle these problems, we set up a bug tracking and monitoring solution by employing Sentry and Matomo, each offering monitoring dashboards that can be seen in subsection A.3.2. Sentry is an error and performance monitoring software that works out of the box once used as a Cloud service. Since its SDK offers support for React applications as well as Strapi in the form of a plugin, we were able to install and configure it quickly for both the front- as well as the backend of our system. Sentry works by catching error events system-wide and transmitting the event data to their servers. The source of such events is identified by a Data Source Name (DSN)<sup>26</sup>, which is automatically assigned to a Sentry project on initialization.

---

<sup>25</sup><https://github.com/Unitech/pm2> (last accessed: 2022-07-13).

<sup>26</sup><https://docs.sentry.io/product/sentry-basics/dsn-explainer/> (last accessed: 2022-07-13).

Matomo<sup>27</sup> is an open-source analytics software that markets itself as an alternative to paid options such as Google Analytics. Even though Matomo also offers a cloud-based setup, since it is a PHP-based application, we hosted it directly at the `matomo.bad-harzburg-stiftung.de` Subdomain by installing PHP on the server and serving the files using Nginx. Just like Sentry, it was sufficient to include a JavaScript snippet in our frontend, which automatically sends the tracking data to our self-hosted instance. We opted to use a minimal configuration that does not use Cookies to track our users.

For both Sentry and Matomo, we wanted to not clutter the relevant tracking data with development instance events or temporary bugs, so the tracking snippets are automatically included only when a certain environment variable is set.

### 6.5.6 Backup and Recovery

Another criterium for us when building a stable system was that we provide data stability – a part of which was the ability to recover data in case of loss. The reasons for losing data can be diverse: From human or software failures to outside attacks or hardware problems. This is why we wanted the data to be backed up regularly and automatically.

**Backup Considerations** Before creating backups, some considerations had to be made according to what, when, and how to store them. Since a working version of the codebase was available on GitHub at all times and could be restored quickly by redeploying, we deemed it sufficient to back up the current data used. This included a snapshot of the database and the current media data used by Strapi. We decided on storing data for the last 14 days, creating a new backup every night at 2:30 am since the user traffic on the application was expected to be limited at that point. Thus, the increased system load during the backup was bearable. The backups were stored in another directory on the same system. However, especially hardware failures could target the whole hard disk, including the backup files. Because of this, we later started synchronizing them to a separate server using SSH.

Furthermore, in our first approach, we created a full backup every night. This was not optimal since copying every picture took a lot of time. Moreover, the produced backups quickly grew in size, filling the available server space in less than the targeted 14 days. It is to expect that a lot of that data – especially the pictures – would be duplicated, explaining the large size of the backups. Our solution to this challenge was to use incremental backups, which only stored new data without duplicating data that was already included in a previous backup. Incremental backups make use of hard links in Unix systems, meaning that a file's contents were not duplicated with a new backup, but rather another hard link was created to point to the same location in memory [52]. This massively reduced the necessary storage space to approx. 1/10th of what was previously needed.

---

<sup>27</sup><https://matomo.org/> (last accessed: 2022-07-13).

**Creating Backups** To make creating backups more easily automatable, we created the Bash Script listed in Listing 6.9. For our approach, we used the `pg_dump` [59] tool, which is shipped with PostgreSQL. It “makes consistent backups even if the database is being used concurrently” in various file formats that can be reapplied using the `pg_restore` tool [59], which is also bundled with the PostgreSQL release. The `rsync` tool is used to back up the picture files. Using the `--link-dest` parameter enables incremental backups by comparing the files in the media directory to the files already backed up in the latest backup directory. Only files without a hard link in the last backup will be backed up again.

**Listing 6.9:** Creating a backup by dumping the current database, copying the media files, and cleaning up old backups

```

1 # Dump contents of the database
2 pg_dump -U $DB_USER -Fc $DB_NAME > "$BACKUP_TARGET/db_backup.sql"
3
4 # Sync files using incremental backup to backup directory
5 rsync -a -v -delete -link-dest=$LAST_BACKUP_DIR $MEDIA_DIR $BACKUP_TARGET
6
7 # Delete stale (> 14 days old) backups
8 find "$BACKUP_DIR"* -mtime +14 -type d -exec rm -r {} +

```

The last line of Listing 6.9 uses the `find` utility to identify subdirectories in the backup directory whose modified date is older than 14 days. The above script is executed periodically using a `cronjob`. Since our software’s production and staging instances use the same database, accessing different schemas, both are backed up simultaneously. However, only the production media directory is backed up since losing the picture files of the staging instance is not considered a risk by our team.

**Restoring Backups** A backup is only valuable if it can be restored easily. Our approach for restoring the backups was to offer another bash script, a snippet of which is shown in Listing 6.10, fulfilling this task. This script takes the path of the backup folder as a command-line argument and restores the production media files to the location specified in the `$MEDIA_DIR` variable. It also restores the whole database from the dump located at `$DUMP_PATH`, including staging data. As seen in the listing below, it creates the database if it does not exist using the `createdb` command. For our developers on Windows machines, we used a process that relied on the `pgAdmin` application, which provides a similar restoration tool.<sup>28</sup>

**Listing 6.10:** Script extract: Restoring backups by reading the backup dump and copying the files to the target media directory

<sup>28</sup><https://www.pgadmin.org/> (last accessed: 2022-07-13).



```
1 sudo createdb $DB_NAME
2 sudo pg_restore -d $DB_NAME $DUMP_PATH
3
4 # Copy files to uploads folder
5 find $MEDIA_PATH -name '*.*' -exec cp {} $MEDIA_DIR \;
```

## 6.6 Summary

In this chapter, we had a look at various decisions that had to be made for the development of the software to be as easy as possible to enable the developers to focus on creating valuable features for the project partner. We found that automation can help with this by taking care of repetitive tasks and checking and fixing code constantly. Moreover, the different areas where automation can be utilized became clear:

- Before development: data migration,
- During development: static code checkers, formatters, and Continuous Integration pipelines,
- During deployment: Continuous Deployment pipelines, process managers, and backup systems.

We found that automation can help the developers but can also be a hindrance if it does not work as intended or if too much time is invested into creating automated workflows that will not yield considerable time savings. We also saw that constant monitoring and evaluating of a running system's performance can help identify bugs and performance limitations and that different forms of tests can give mixed results.

We are providing monitoring tools and comprehensive documentation to make the life of future developers of this project easier. However, there are still areas that need to be addressed. These include the security concerns stemming from our API configuration (see subsection 6.5.3) or support for more scanner models (see subsection 6.3.2).



## 7 Low Fidelity Prototypes to Explore the Design Space

To learn more about our users and the features they would like to use, we built three prototypes. In this chapter, we introduce and discuss them. Thereby, we focus on features that can be part of future work. For all prototypes, we give an introduction as well as a classification and a walkthrough. We show our insights and discuss these results.

### 7.1 Introduction

The value of every single picture of a picture archive is increased dramatically by digitalizing it since analog pictures are victims of the process of decay. Now, to maximize the archive's intangible value, we aim to also conserve the pictures' contexts. Every single picture in the archive documents an important moment and tells a story. As these pictures are getting older, there are fewer people to remember these stories, and more pictures are endangered to lose their context, thus becoming meaningless to the viewer. In this chapter, we want to find opportunities to evolve our system in future work. Our main goal is to find and validate new feature ideas to increase our users' encouragement to consume and especially add information. We pay special attention to conserving as much knowledge and as many memories related to the pictures as possible. To achieve this goal of knowledge transfer, we need to accomplish the following subgoals:

1. The archive must be accessible to all potential users. They need a well-structured entry point to about 500,000 pictures in the archive. This is crucial to exploring the archive to learn and add knowledge and memories.
2. The system must engage its users to consume the information in the pictures.
3. It must engage its users to contribute their knowledge and memories, too.
4. Curator activities must be supported to allow checking and revisioning of added information.

We focus on the first three subgoals because they are all aimed at the system's typical users who are browsing or searching. The fourth subgoal deals with other interfaces specifically designed for curators and already discussed in chapter 3. After discussing different feature ideas that could serve our goal, we selected three of them to be consolidated. To do so, we built three prototypes. We designed two user tests to validate the prototypes and the corresponding feature ideas with participants in Bad Harzburg. Therefore, we prepared tasks and questions as well as interviews.

## 7.2 Concepts

After our system was in a state that satisfied all the basic requirements, we started to think about possible future work. Besides our project partner's ideas to improve the curator's work process, we also wanted to work on some extraordinary ideas that could push the boundaries of our system's user interaction. We took one week to collect as many ideas for potential new features as possible in a design-thinking manner. These ideas should serve as a basis for prototyping activities to find the best options for future development. In the following, we summarize our most important outcomes and discuss our choice of ideas to implement as a prototype:

- **Map:** The archive's pictures can be presented on a map of Bad Harzburg. Users see a clustering by location. This can be an interesting starting point to browse through the archive. Bad Harzburg's inhabitants can easily find places they know, and tourists can get a better overview of the city and its history. A conceivable use case is finding pictures of a user's current location.
- **Timeline:** Besides by location, we can also organize the pictures by the date they were taken. This can happen on a Timeline. Users can see the evolution of places, events, people, etc., and search for pictures related to important moments in their lives.
- **Stories:** Stories are a way of giving context to pictures. In contrast to descriptions and comments per picture, a Story creates a context with contributions to multiple pictures that build on each other. Pictures are the medium to tell the story. This way, users can share memories and/or knowledge about a topic (for example a person, an event, a local tradition, etc.) in a more extensive way. Possible stories could be about the history of the "Salz- und Lichterfest" or the life of Horst Voigt, a very dedicated inhabitant of Bad Harzburg. As users would have to contribute their own Stories, editorial work would be necessary.
- **Scavenger Hunt:** Based on the idea of a scavenger hunt, users or curators can contribute routes along important places in Bad Harzburg supported by related pictures from the archive. This feature aims at tourists, students, and other people that are eager to explore the city and learn more about its history.
- **Gamification:** Inspired by Wikipedia and StackOverflow, a scoring system can be added to our application. Users can get scores for contributing information which includes comment writing and curator activities like adding descriptions or tags. Contributing to pictures missing all information will gain the highest scores. Based on the score, a level or badge system can be implemented, and high scores can be shown. This way, users can get an additional incentive to contribute their knowledge, but editorial work becomes even more important.
- **Social Media:** Another possibility to encourage users' activity are social media aspects that create a feeling of a community among the system's users. This can involve user accounts, likes, and personal, shareable collections. Users can look at other people's profiles and their favorite pictures. Chat functionalities are possible, too.
- **Campaigns and Contests:** This feature idea is about digital events ranging from special collections that are only accessible for a limited time to calls for

action that physically involve the user. An example could be a “Recreating Pictures Contest” that invites users to send in pictures they took that show the same place, and people or pose as pictures from the archive. Other ideas are picture coloring for children and an “Easter Challenge” where eggs are hidden in some of the archive’s pictures. Such events could increase the archive’s popularity and the number and engagement of users.

The process of selecting the best ideas to be implemented as a prototype was based on two criteria:

1. Value to our main goal, which is engaging a knowledge transfer.
2. Feasibility because the prototypes needed to be developed with time constraints.

Campaigns and contests as well as Social Media functionalities are of limited value to our main goal because they aim at entertaining the user instead of promoting knowledge transfer. Gamification could increase users’ participation, but we doubted our user group’s affinity for digital games. Finding out if this assumption is correct could be interesting but we found other ideas to have a higher value to our project at this time. Scavenger Hunt is a way to promote subgoal 2, information consumption, but to a limited target group (mainly tourists), so other ideas were more important to us. Stories can provide a framework for sharing extensive knowledge. By viewing a Story, users learn more about the bigger context of pictures. Thereby, Stories might achieve a better knowledge contribution and consumption which serves subgoals 2 and 3. To prototype Stories, we have to simulate the creation as well as the viewing process. These are extensive interfaces but with paper prototyping we found a tool to simply imitate them, see section 7.5. The Map and the Timeline both propose an answer to the same question: How can we structure the huge number of about 500 000 pictures to make them as accessible as possible? The answer to this question is crucial to building a usable system and thereby significant in reinforcing users’ engagement with the archive, see subgoal 1. Sorting by time or by location could be an intuitive alternative to the known thematic structure. Creating prototypes for both views is realizable in a short time. As a result of this process, we chose the Map and the Timeline as well as Stories to be implemented as prototypes and tested with participants in Bad Harzburg.

We built our prototypes based on our software as described in chapter 4. Our Browse View allows users to view multiple thousands of already digitalized pictures structured in thematical non-disjunct collections. To each picture, curators can add descriptions and a time-range tag to give the day or known time range that the picture was taken in. Tags for people that can be seen on the picture and the place where it was taken can be included, too. The system also provides a Search View, where users can search in pictures’ descriptions or choose from a variety of decade and place filters. Users can read and write comments that need to be verified by a curator to be published.

## 7.3 Methodology

This section introduces prototyping as our main tool to work with new feature ideas. Afterward, we present the methodological approaches and tools that we used to design user tests to evaluate our prototypes.

### 7.3.1 Prototyping

We used prototyping as the main tool to work with new feature ideas. According to Budde et al., a prototype serves as “an operational model of the application system. It implements certain aspects of the future system” [11]. A prototype is built for two reasons: It serves as a “communication basis” by supporting discussions with stakeholders, and it helps to experiment with new approaches [11]. Jensen et al. state that prototyping is the process of „making and utilizing prototypes“ [31].

### 7.3.2 Qualitative User Tests

To analyze our prototypes of new feature ideas and test whether they meet our and the user group’s requirements, we designed two user tests that we performed with participants in Bad Harzburg. The Map and Timeline prototypes were tested in one user test because they are trying to solve the same problem and should be compared with each other. The Story prototype was tested individually. Every run of a user test with a participant is considered a “testing session”. Our testing methods are mainly qualitative. As discussed in chapter 3, we only have a limited number of participants. But our participants’ feedback is very valuable because they represent our user group well and they live in Bad Harzburg, most of them for their whole life. Some of them already know the old version of the archive or are eager to get to know our new system. Qualitative testing does not give statistical results but has the advantage of being robust. Jakob Nielsen states that it is not as prone to methodological errors as quantitative testing [44]. This makes it especially useful for our case. Similar to the procedure of the usability testing described in chapter 3, each of our testing sessions starts with an introduction given by one of our team members, the moderator of the testing session. A second team member is present to write a transcript for later evaluations. After the introduction, the prototypes are shown, and some tasks are posed. We use a think-aloud approach to comprehend the participants’ opinions and intents. When the participant finished working with the prototypes, an interview is conducted. In the following, these methodological approaches are discussed in more detail.

### 7.3.3 Think-Aloud

To collect as much data as possible, we chose a think-aloud approach as introduced in chapter 3. Participants are asked to share their thoughts when they see a prototype for the first time. According to Jakob Nielsen, thinking aloud “may be the single most important usability engineering method” because it helps to better understand

their conceptions and problems [44]. However, the think-aloud method has some drawbacks: It influences performance measurements, and expressing their thoughts can feel very unnatural to participants. Furthermore, participants' feedback might be misleading. Experimenters must pay attention to what users are thinking and doing while they are working with a prototype [45]. If participants stop sharing their thoughts, we intervene by asking about the reason for their actions. This helps us to better analyze their interaction with our system. As we do not want to conduct any performance measurement, thinking aloud is a very valuable method.

#### **7.3.4 Moderator and Recording Clerk**

To conduct a testing session, there are two team members with one participant in a quiet environment. One of the experimenters is the moderator, and the other one, the recording clerk, writes a transcript. The moderator takes on the communication with the participant. They introduce the testing session and the single tasks and ask all questions. The other person does not interrupt the testing session but takes notes about the actions and comments of the participant that we can evaluate in depth after the testing session. As claimed by Nielsen, it is important not to have too many people around because participants might become shy to talk [45].

#### **7.3.5 Interview**

After the use of each prototype, we interview the participant. The interview allows us to talk about special actions and statements participants do in an earlier phase when it is not possible to ask without disturbing the user. Interviews are more exploratory than questionnaires and permit us to flexibly ask follow-up questions or rephrase statements that were not understood correctly. They allow learning more about our participants' requirements than planned during preparation. As a qualitative testing method, interviews are not suitable for large numbers of participants and quantitative analysis. However, this is not necessary for our testing. To successfully conduct an insightful interview, Nielsen suggests that the moderator must pay attention not to bias the participant and to ask open-ended questions [45].

#### **7.3.6 Pilot Testing**

Similar to our procedure described in chapter 3, we had several pilot testing sessions with other students to prepare for the testing. Before conducting the testing sessions with all our participants, we had one additional pilot test with a person from our user group in Bad Harzburg. As Nielsen describes, it helped as a final rehearsal and allowed testing of our questions and tasks for understandability [45]. We did not have to implement major changes as no problems arose in the pilot tests.

### 7.3.7 Quantitative Approach for the First Experiment

In 1932, Rensis Likert introduced an ordered response scale to scientifically measure opinions. To do so, the so-called “Likert scales” provide statements to which subjects are asked to express their agreement on a metric scale from strongly agree to strongly disagree. Likert-type scales use Likert’s approach on some individual questions without the claim to synthesize all questions [32]. We use Likert-type scales as a quantitative approach for the first user test. They are utilized to check our qualitative results. We retrieved our questions from the System Usability Scale introduced in chapter 3. Initially, we planned to use this questionnaire for our prototype testing as well. But during our pilot testing sessions, we noticed that participants were overwhelmed by the number of questions and the level of detail of the System Usability Scale, especially because the prototypes are built in one simple interface and the time spent working with them was little. Our Likert scale uses the same response items as the System Usability Scale: 5 points from strongly disagree (1) to strongly agree (5).

We want the system to allow its users to get in touch with the archive’s pictures. To evaluate how well our system achieves this goal, we ask for orientation, accessibility to the archive, and clearness. In addition to these questions, we want to know how well the prototypes perform with the whole archive in mind. The scores are not only used to evaluate the two prototypes on their own but most importantly to compare them. These considerations resulted in the following four questions:

- With the help of the Timeline/Map I was able to navigate through the set of pictures,
- The Timeline/Map has brought the pictures close to me. It has given me access to the picture archive,
- The Timeline/Map is clear,
- I would like to use the Timeline/Map to look at the entire picture archive.

This questionnaire is completed twice, after dealing with each of the two prototypes. The complete, German questionnaires can be found in Figure A.4 and Figure A.5. It is important to notice the scientific discussion about the statistical evaluation of Likert-type scales. Some scientists argue that the items’ responses cannot be treated as an interval scale because they are not equidistant. Thereby, the Likert-type scale results could not be arithmetically manipulated, as Jamieson describes [30].

### 7.3.8 Study Population

At the end of each testing session, we asked the participant to fill another questionnaire, as introduced in chapter 3, about age, an affinity for technology, and relation to Bad Harzburg to get a better overview of our participant’s demography. Overall, we had 7 valid testing sessions. One additional test run was interrupted multiple times and therefore considered invalid. It may not be represented as the participant had to perform several context switches, so we will not include it in our evaluation. In one testing session, two people participated together. They often had different opinions and discussed until they found an agreement. This resulted in very inter-



esting insights that we might not have gotten if we interviewed them separately. On the other hand, their final answer to our questions and questionnaires fell on the middle ground without a strong opinion. This is also why we have 8 participants although we only conducted 7 valid testing sessions. We refer to the participants by P<sub>1</sub>–P<sub>8</sub>. Our project partner selected our participants intending to find people who are interested in the picture archive and have a relation to the city of Bad Harzburg. We also assumed that they have basic technical skills so they can operate a computer on their own. The participants are all part of our project partner's network. This causes them all to be members of educated classes and have a relation to work with print media, for example, 3 of our participants are working in a library. Six participants were born in Bad Harzburg or the surrounding area, one is living there for over 30 years and one for about 2 years. Three participants are male and five are female. Our study population has the following age distribution: One person in each age group of 21–30, 31–40, 41–50, and 51–60. Three people were between 61 and 70 years old, and none were older than 70 or younger than 21. We assume that this corresponds to our actual user group very well, although we might need to test some older users as they are the most important contemporary witnesses.

## 7.4 User Test 1: Map vs. Timeline

In this section, we deal with subgoal 1, the accessibility of our archive. This section introduces the prototypes for the Map and the Timeline and classifies them. We will first describe different prototype classifications. Afterward, we show the execution of the user testing sessions. Finally, we evaluate the qualitative interviews as well as the quantitative questionnaires and conclude.

### 7.4.1 Setting the Stage

Before we show our prototypes to the participants, we perform another user test regarding ordering and searching with analog pictures. By working with analog pictures, we want to introduce our problem domain and gather knowledge about intuitiveness in the structuring of pictures. This provides a preparation to easily get into the prototype testing. During all the phases of the test, the same random 50 pictures are used, analog as well as digital. This increases the comparability of our prototypes because there are no differences in the quality of the picture selections. To give access to the archive, a chronological and a geographical order are the only alternatives to the thematical order of the current archive that we and all our participants came up with during brainstorming sessions and the analog user test. Our prototypes are designed to explore the possibilities of a Timeline and a Map to structure the pictures of the archive. They could be potential new features to give users better access to our system and improve the quality of browsing through it.

### 7.4.2 Introducing the Prototype

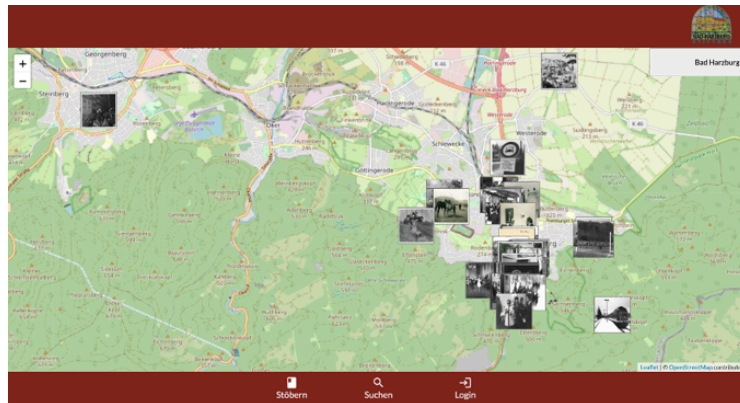


Figure 7.1: The Map prototype

The Map prototype, shown in Figure 7.1, is implemented using Leaflet, a JavaScript library for interactive maps. It is included in our existing system with a new route. We built a tool to manually place the 50 digital pictures on the Map of Bad Harzburg. The locations are not stored in the database but a JSON file. All pictures are clickable to show their descriptions. There is no more information like comments or tags given in this prototype. It is possible to interact with the Map by scrolling to zoom and dragging to move. In a pilot test, we recognized the need for a button to go back to the initial position on the Map because some users were accidentally scrolling to other places and had problems navigating back to Bad Harzburg.

Figure 7.2 shows an extract of the Timeline that is prototyped within a Miro Board, a digital whiteboard. We manually sorted the 50 given pictures by time and placed them above and below an arrow indicating the years. Each description is located next to the corresponding picture. Three pictures are missing a timestamp, so they have been positioned some distance below the Timeline and labeled with “no information available”. In both prototypes, there is one picture that is drag-and-droppable to simulate possible curator work, for example adding a new picture to the structure.

### 7.4.3 Classification of a Prototype: Background

To get a better idea of our prototypes, we classify them. There are different classifications described in the literature. We first introduce the most important classifications and then use them on the Map and Timeline prototypes as well as on the Story prototype in subsection 7.5.4.

**Fidelity** To classify prototypes, fidelity is used. According to Virzi, fidelity is “a measure of how authentic or realistic a prototype appears to the user when it is compared to the actual service” [60]. The fidelity is low if a user can easily distinguish



exemplifies one feature by implementing all layers of the application for it [10]. For example, a vertical prototype for our application could focus on comments: From typing and sending a comment from the user interface to storing it in the database many layers of the application are involved. A horizontal prototype, in contrast, could only show the user interface layer of a new feature. It does not need to interact with the database but could work with mock data.

**Classification of our Prototypes** Our Timeline is a low-fidelity prototype. As it was created with a different tool, a Miro board, it has almost no similarities to the original system. The Map, on the other hand, is a mid-fidelity prototype. We first planned to build it with a low fidelity like the Timeline, but this approach was way more elaborate for the Map. We concluded that we could get a more valuable process if we choose a higher fidelity. By integrating it into our original system we achieved a higher fidelity in comparison to building it with an external tool. Nevertheless, the Map is not high fidelity. We did not pay attention to adapting the design to our system and only the 50 preselected pictures were shown. Changes made in the prototype are not saved and some of the original system's buttons do not work, as well. For both prototypes, high fidelity is not desired. Our goal is not to find out how to build the perfect Map or Timeline, but to determine how well they perform in presenting the archive. Therefore, we want to initiate discussions about the general ideas of different structuring approaches. Our prototype development had the goal to be fast and low-cost. This was important because we knew that both the Map and the Timeline are Throwaway prototypes. We applied our prototypes in this manner: They were built to get participants' feedback and thereby refine our requirements. After the testing sessions, the prototypes will be discarded. They are not usable for deployment. If we do want to add a Timeline or a Map to our system, we must rebuild it from scratch. Both our prototypes are horizontal. They embody all necessary features but only implement one layer of the application: the user interface. They show a layout for the 50 sampled pictures and their descriptions, and provide the functionality of drag-and-dropping one picture but do not have any connection to the database layer. All data is saved in the Miro board or a JSON file.

### 7.4.4 Execution

At the beginning of each testing session, the moderator introduced our project and gave general information about the testing procedure. Before the prototypes were shown, the participant was handed 50 analog pictures and some search and order tasks. Afterward, the Map and the Timeline prototypes were presented in alternating order. The participant was asked to browse through the pictures using the respective prototype and share their thoughts. After the participant has got an overview, the moderator asked them to drag and drop one given picture to arrange it correctly. Both the time and location of the picture were given in its description. During working with a prototype, the moderator asked questions about the usability and observations of the participant's behavior. By sharing their thoughts, the participant gave a lot of input about their reception of the prototype. For example, about half of our

participants asked for picture descriptions in the Map prototype. P3 asked: “Is there a possibility to see the picture’s description?”. The moderator replied: “What do you expect: What do you have to do to see a picture’s description?”. P3 suggested clicking on a picture but does not try to do so until the moderator motivated P3 to do so. When the participant finished using one prototype, they were asked to answer the Likert scales and some more questions that would have disrupted the flow of exploring the prototypes in an earlier phase. Most of the participants also shared their thoughts while answering the questionnaires. This gave some interesting insights, but the moderator had to pay attention not to affect the participant’s answers. At the end of the testing session, the final interview was conducted. The moderator asked the participant to compare the different approaches that were discussed during the testing session and their applicability to the whole picture archive. Some follow-up questions were asked, too.

**Limitations** A possible source of errors was usability problems with Miro boards. Some subjects had issues to zoom and move the Timeline. They stated that the Timeline’s layout is small and confusing and that the navigation through the prototype was complicated. In comparison to the Miro board, the Map was easier to use. This may have influenced the perception of the Timeline prototype in a negative way. As the Map was integrated into our system’s layout, there is additional functionality that we do not want the participants to use. To one subject, we did not explain that some buttons should not be pressed. The participant clicked one of those buttons, so the moderator had to intervene to navigate back to the Map. This happened during the evaluation of the Map prototype, so we assumed it had no influence on the feedback. A few of our participants knew our system before we conducted our testing session, others did not. Thereby, they had different knowledge of the thematical order implemented in our original system. People who knew the original system had a better understanding of the picture archive’s scale. These differences may have influenced the participants’ responses. P2, P3, and P6 noticed two errors we made when placing the pictures on the Map. The casino and the sports ground moved after the pictures have been taken but we located them in their old places. They explained this circumstance to us, but we expect it not to have influenced the testing’s outcome. We furthermore think that these incidents show how fast locals notice if pictures are wrongly located on the Map and which location is shown in a picture. This could indicate that a Map also improves users’ willingness to contribute their knowledge by moving pictures to their correct location.

#### 7.4.5 Qualitative Evaluation

Older people tended to have a higher demand for chronological structures than younger users. Our three youngest participants did not like the Timeline because they had trouble getting access to it as they have no relation to the time shown in the pictures. On the other hand, two of our oldest participants liked the Timeline very much. P7 searched for pictures taken in their birth year or liked the idea of exploring the historical development of their school. Thus, we can conclude that a

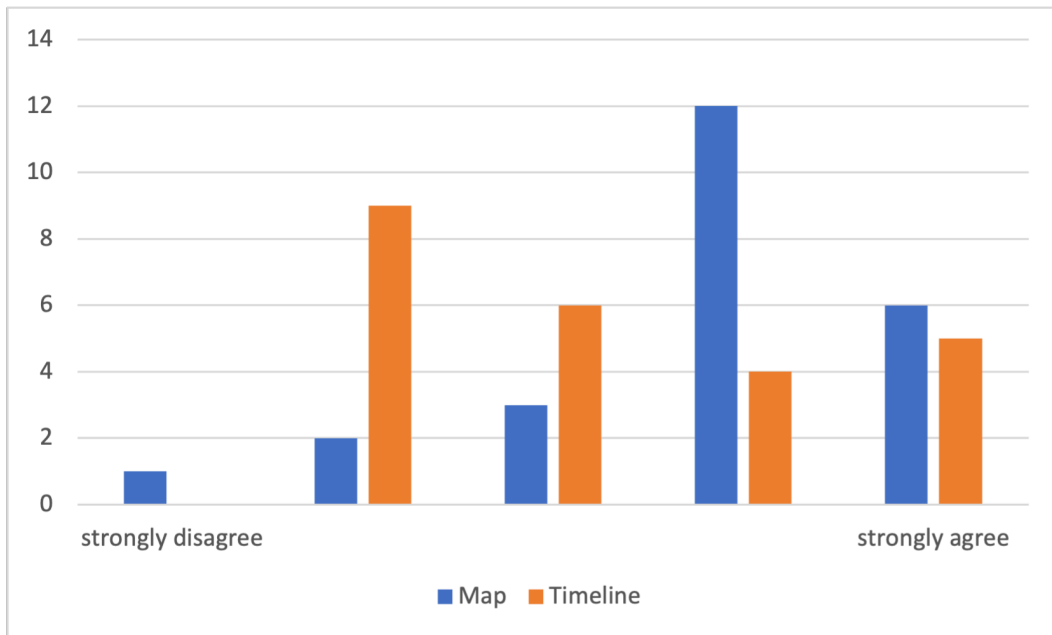
chronological structure is important to our user group which includes mainly elderly people.

Similar to the Timeline, people with a longer relation to Bad Harzburg preferred geographical structures more than people who are new to the area. P1, who moved to Bad Harzburg only two years ago, had trouble orientating on the Map. All the other participants liked using it. They enjoyed looking at places they know to see what happened there in the past. They had fun comparing the pictures to how the places look today. But all of them also stated that they do not require a picture archive to have this functionality. The Map was seen as a nice-to-have feature that is “rather entertaining than helpful”, P6 said. When comparing the Map and the Timeline all but one participant preferred the Map. Both prototypes were criticized for their lack of clarity. On the Map, the pictures are very small and overlap at first sight. Users have to zoom in to see the single pictures and click them to view them in detail. The Timeline has a similar problem: When entering the page, the pictures are very small so that a user can see the whole Timeline but must zoom in to use it properly. As the Timeline was harder to operate for most participants, this could have influenced our findings. At least for the Map, a solution to this problem could be arranging our location tags instead of the single pictures.

We also asked the participants to compare the two prototypes to a thematical structure. Participants who knew a digital version of the archive thought about its structure, and others thought about thematical structures they knew from other contexts. It could have been an improvement to our testing sessions to shortly introduce our original system to have the same preconditions for all participants. In five out of our seven testing sessions, the participant favored a thematical structure over Map and Timeline to browse the archive. All participants were familiar with thematical ordering. Thereby, it seems to be the most accessible option. The problem we noticed is that all participants would use a different thematical structure. Some preferred having many small clusters of pictures while others would like to have a limited number of big clusters. Some included geographical information like a rough subdivision by area. And some were sorted only by the pictures’ motives while others paid attention to the descriptions and the actual contents of the pictures. This makes it difficult to provide a thematical structure that fits every user’s needs. Before our user testing, we were not sure if thematical collections that are not disjunct could be confusing because users may expect them to work like known, analog photo albums. Some discussions in the testing sessions’ interviews revealed that the participants consider this idea useful instead of confusing. Some participants even suggested it themselves. This way the problem of different people having different requirements on a thematical structure can be solved.

We did not ask for a comparison of the Timeline and the decade filter our system’s Search View offers, but we expect the filter to meet the most requirements for the chronological structure. This hypothesis could be tested in future work. Another, more general but not less interesting insight that we got from observing our participants is the following: The younger and more tech-savvy participants were, the more likely they tried clicking elements if they did not know what could happen. This means, considering our elderly users, we must pay attention to clearly show

which elements are clickable and make sure that users understand what happens if they click them.



**Figure 7.3:** Cumulation of all questionnaire answers from ‘strongly disagree’ (bad) to ‘strongly agree’ (good)

#### 7.4.6 Quantitative Evaluation

To evaluate the quantitative results of our Likert-type questionnaires, we count the number of each item answered per prototype. For all four questions, ‘strongly agree’ is the answer that rates the prototype the highest while ‘strongly disagree’ indicates a dislike of some aspects of the prototype. The higher the number of ‘agree’ and ‘strongly agree’ the better the overall conception of a prototype. This way, we can draw an overall comparison of both prototypes. Figure 7.3 shows the result and illustrates that the Map has received a better rating. Even if the Map got the only ‘strongly disagree’, it received two times as many positive ratings as the Timeline. This embraces our qualitative insight that the Map is the better option to give access to the picture archive.

#### 7.4.7 Conclusion

When comparing a Map to a thematical structure, our main insight is that the Map is a nice feature that a lot of users would like to use but a thematical structure is the most valuable to give access to the archive. This means the approach we chose for our system so far is approved by our user testing results. We do not have to change

our landing page to increase the archive's accessibility and achieve our subgoal 1. A Timeline is not needed to be implemented as it seems to be of limited value to our users. The requirements arising from the wish to search chronologically can be met by our system's current functionalities. Users can filter for decades and search for exact years within our Search view. This satisfies both browse and search users. A possible extension would be the option to sort chronologically in search results or collections. In the future, a Map can be implemented to offer the user an additional functionality to explore the archive and browse through it. Therefore, a way to present the pictures concisely is needed. As suggested before, the location tags can be used to do so: On the Map, location tags would be arranged. Users would see the related pictures by clicking them. This approach could be tested in future work. No matter the exact implementation, a Map demands new functionalities for curators. Location tags or pictures must be placed precisely on the Map. This task needs very profound knowledge of Bad Harzburg and its history. The tool requires very high usability because we should not lose people who have this knowledge.

## **7.5 User Test 2: Paper Prototype for a New Feature: Stories**

In this section, we present the prototype and user test that we designed to collect feedback about Stories. Stories are introduced in section 7.2. We give a motivation for our choice of Stories to be implemented as a prototype and introduce and classify the paper prototype we built to simulate this potential new feature. Afterward, a walkthrough of the user test is described and the results are evaluated.

### **7.5.1 Setting the Stage**

As introduced in section 7.1, the main goal of our project is to put the archive's pictures into context by collecting knowledge and memories from users. Stories provide more information than comments and descriptions of single pictures. Descriptions have the following problem: They might be related to many different pictures. For example, some general information about Conrad Adenauer must be related to every picture showing the politician because a curator does not know which picture is viewed first. This makes it difficult to tell more of a context within a description. Again, the curator would not know where to write the additional information so it would end up being added to every related picture, decreasing the viewers' ease of use. Stories can solve this problem by building a context with a selection of pictures in a fixed order. We also want to allow sharing of Stories to all users. The creation of stories is elaborate because suitable pictures must be found, and a lot of information must be added. That means it is fundamental to encourage users to contribute. This needs a convenient interface to enable users to add their own Stories. The process of creating a Story includes selecting pictures, bringing them into the correct order, and adding information. Additionally, editorial work needs to be supported because all the contributions must be checked for content and spelling correctness and sources should be reviewed before publishing. To test if Stories are an improvement to our



system, we decided to build a prototype for it. We were not able to implement the whole feature. Nevertheless, we aimed at testing the whole viewing as well as the creation functionality. It is important to keep in mind that we need users to contribute Stories because otherwise there would be too little content. Therefore, we had to search for a very low-cost prototyping tool. We do not want to include general curator tasks in this prototype but focus on browsing users because we must find out if this feature is wanted at all. We want to test the option to record audio comments or contributions en passant. They can be more accessible and might improve the user experience for contributing and consuming information. Therefore, we add audio and textual content to our prototype. As discussed above, we needed to find a low-cost prototyping tool. We decide to use paper prototyping for our use case.

### 7.5.2 Background: Paper Prototypes

Paper prototypes are used to represent a product's interface and user flow. Therefore, important elements are modeled using drawings on paper. Paper prototyping focuses on supporting communication with shareholders instead of testing an implementation of a new functionality [28].

Drawings on paper cannot simulate all possible elements of a software system's user interface. For example, animations cannot be prototyped. Another drawback is the fact that people interact differently when typing on a piece of paper than they would when typing on a tablet computer. Especially typing text is cumbersome to simulate. On the other hand, sketching the user interface is faster than coding an actual application [22]. Thereby, it can be considered in an early phase of the feature development process. Especially by being very fast and with our focus on validating the idea, paper prototyping is very suitable for our use case.

### 7.5.3 Implementation

The prototype's goal is to simulate the system by showing drawings of the user interface. The moderator has to simulate the computer. The participant is asked to click on buttons on the paper as they would do with a digital application. The moderator reacts to the participant's input and exchanges the piece of paper to show the next interface. Therefore, all possible interfaces need to be prepared on paper which leads to better discussions since we can present our idea of Stories with more clearness than by just talking about them. For our case, this means we have to prepare a landing page, two Stories with some pictures each, one with textual and one with audio contributions, and some views to simulate the creation process of a Story. Additionally, we crafted a paper iPad frame to achieve a better simulation of a digital system used with touch control. A top bar with a back button reminding us of our original system is included in this frame as well.

In Figure 7.4 the landing page is shown. You can see two big buttons to view Stories, the first one with informational text, symbolized by a "T" in the upper right corner, and the second one with audio content, symbolized by a speaker symbol. The third button leads to the interface for creating new Stories. The layout is adapted

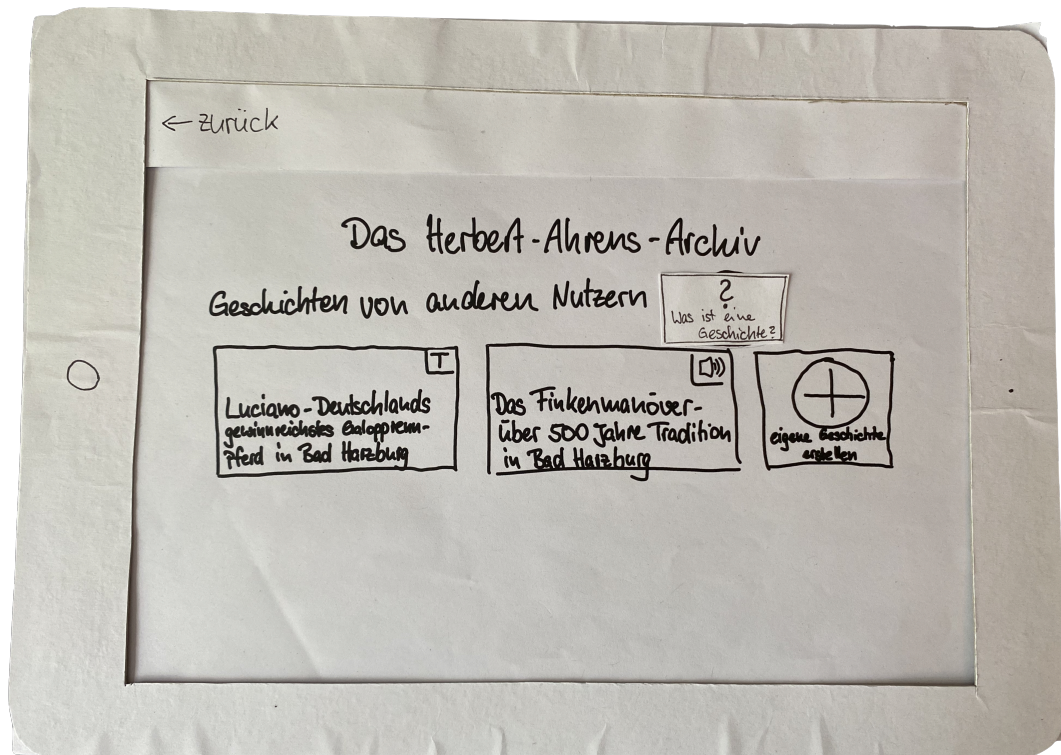


Figure 7.4: The landing page of our paper prototype with the iPad frame

to our original system’s layout with its big buttons for collections. Above those three main buttons, there is another one serving as a help button.

In Figure 7.5 we can see the interface to view a picture, show more information and comments, navigate to the last and the next picture, and hear an audio contribution that explains some details of the event shown in the picture. The audio file was recorded before and played from the moderator’s smartphone.

Figure 7.6 shows the interface that enables users to add information to a picture in a new Story. The user can choose between adding text or audio comments. There are buttons to save a contribution and to show more information and comments as well as buttons to go to the last or the next picture.

To show two example Stories we researched some facts about the “Finkenschlagen”, a very old tradition in the area around Bad Harzburg, and about the racehorse Luciano that is honored by a memorial stone in Bad Harzburg. We created an audio Story with 5 “Finkenschlagen” pictures and a textual Story with 4 pictures related to Luciano. After the participant has viewed one or both Stories, we ask them to contribute a new Story. Therefore, we prepared a text introducing an imaginary politician and summarizing four important events in his life. Corresponding to the four parts of the text, we preselected four pictures. After clicking “Create a new Story” a user sees a screen to select pictures from the archive. As it is not possible to simulate the whole archive with paper, we skip this step and directly show the four preselected pictures. Then, the user can change their order. By clicking a picture, it

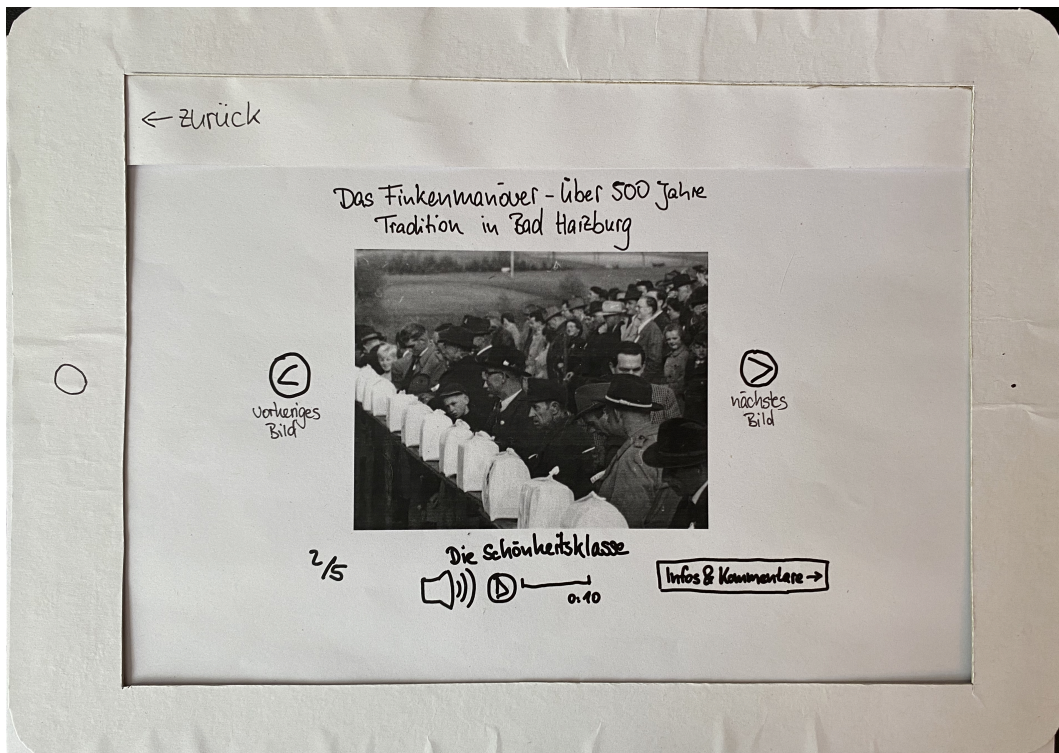


Figure 7.5: A picture in a story in our paper prototype

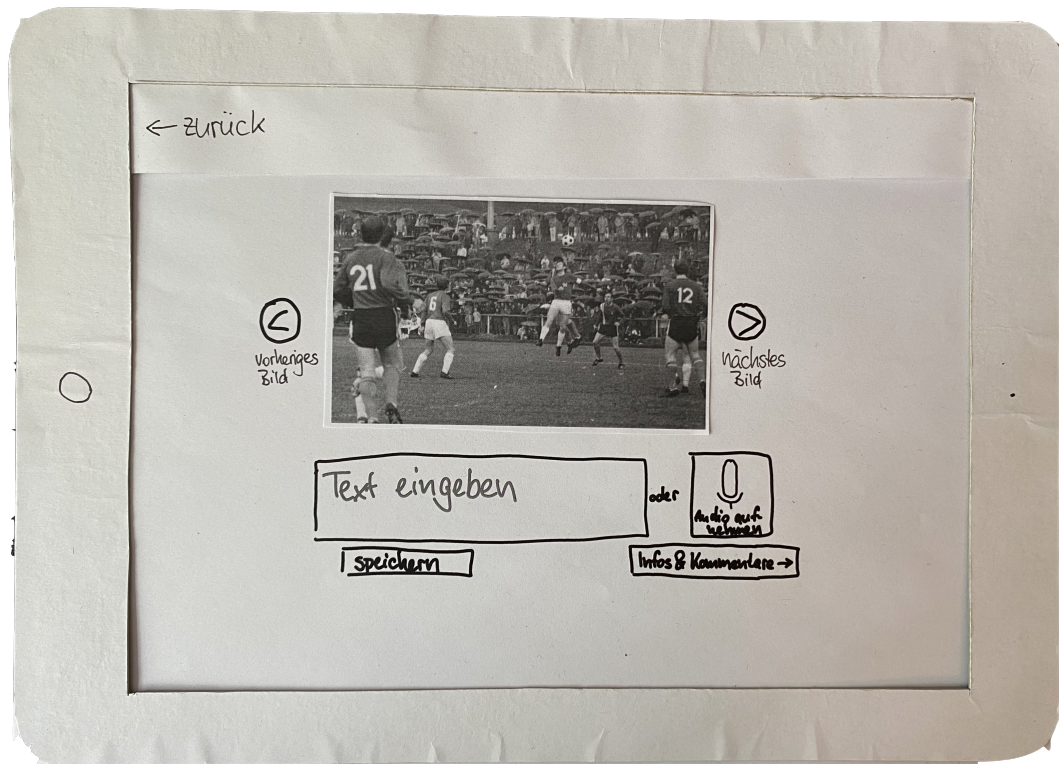
can be seen larger, and information can be added and saved. This is simulated by writing a text on a piece of paper and placing it in the correct position or pressing a start and stop button to record an audio contribution. If the Story is done the user can click “Save” and return to the landing page.

#### 7.5.4 Classification of our Prototype

As described above, this is a paper prototype. Its fidelity section 7.4.3 is low because users can easily distinguish the paper prototype from the digital system. Like the Map and Timeline prototypes, this one is a throw-away prototype (section 7.4.3) as it is not used for future development but must be discarded after testing. It is a horizontal prototype too (section 7.4.3): All needed features are simulated only in the interface layer.

#### 7.5.5 Execution

At the beginning of the testing session, the moderator presented the landing page and explained the idea of a paper prototype. The participant was asked to try to imagine working with an actual digital application while the moderator imitates a tablet computer’s response to input. Again, the participant was asked to share their thoughts, so the recording clerk was able to take precise notes. The experimenters



**Figure 7.6:** The view for adding information when creating a new story

paid special attention to the participant's reactions when being handed the prototype for the first time. In a real scenario, there would be no experimenter who can explain the new feature, so it needs to be intuitively understandable at first sight. Otherwise, too many users would leave the website without viewing a single Story because they are deterred. Some users were confused at the beginning. If a participant asked about the meaning of Stories and this landing page, the moderator hinted at the help button. After the participant read the information text, the moderator made sure that the participant does understand our idea. Most of the time, the participant clicked one of the two Stories' buttons or asked what happens if you do so. Then, the participant looked at the presented pictures and read or heard the information given. Almost all participants clicked the button to get more information and other users' comments at least once. We noticed that it is difficult to understand the origin of these descriptions and comments. It was hard to distinguish whether information belongs to the Story itself or the original picture. The same applies to comments. In our understanding, a comment on the picture belongs to the picture itself but some participants expected to write comments on the Story. When the participant finished viewing Stories and gave some feedback, the moderator explained the task to create a new Story. A piece of paper with text introducing an imaginary politician was handed to the participant.

**Limitations** When some participants asked for an explanation at the beginning of the experiment, the moderator should not have advised them to click the help button. Instead, the participants should have been asked what they expect this system to offer and what they would do if they do not understand it in the first place. Thereby, we could have learned if our users would use the help button on their own and get to know how they behave if they are overwhelmed by our system. It could have been an improvement to the testing if we prepared a second version or another prototype with the same goal. Test users then would have to compare two options similar to the discussion of Map versus Timeline in the first testing session, described in section 7.4. In our interviews, they often only told us that they like or dislike certain things. The moderator then had to ask for an explanation. Most participants were overwhelmed and confused by the task to contribute a new Story. They did not understand it correctly. They thought the written text was part of a Story like the other two they have seen. So, they tried to think of a new fact that could be the fifth part of the text. Although the moderator tried to explain the task differently after four unsuccessful attempts, only one subject completely understood it correctly.

### 7.5.6 Evaluation

Overall, the feedback was rather positive. Most participants liked the idea of Stories and the way the prototype implements them. P5 even said: "This is the best idea I can think of to tell and experience stories about the past". Others liked that this feature presents pictures livelier. Especially viewing Stories has been very enjoyable for the participants. Our participants often responded that they would contribute Stories if they knew something important. As they have a close relationship to Bad Harzburg and most of them even to the area's history, they are the perfect creators. Three participants did not come up with an idea for a new Story but assumed that they could find interesting topics if they browse through the archive. If so, they would try to make a Story. The second topic of this testing session was whether our users would like to use audio contributions. The audio could be used not only for Stories but also for comments. Surprisingly, all our participants have an aversion to voice messages and therefore use audio comments, too. But all of them admit that they know a lot of people who like to use these functionalities.

### 7.5.7 Conclusion

Our test shows that Stories could be a valuable feature to increase people's engagement and knowledge transfer on our system. The fact that all but one participants were not able to make their own Story is a vitally important aspect because the feature cannot be implemented if no one is contributing Stories. We do not know if the problem is the creation of Stories itself or if our prototype or task is misleading. Before implementing Stories as a feature in our system we should test this aspect again. Maybe higher fidelity prototypes showing different versions are the best approach. This way we could analyze if the problem is the idea of creating Stories itself or the implementation of an interface. The question of whether audio contributions are a

new requirement cannot be finally answered by our testing results. To find a solution we need to conduct another test with other participants that are not selected with the same bias subsection 7.3.8.

## 7.6 Summary

In this chapter, we examined the prototypes we built: a Map and a Timeline to structure the archive, and Stories to bring more context to the pictures. We introduced these prototypes and our methodology for testing them with potential users in Bad Harzburg. A walkthrough of both testing sessions was described before we evaluated the insights we gained from them. Now we want to draw an overall conclusion. The prototypes we chose to implement brought a lot of useful feedback to our project, especially noticing our and the participants' time constraints. We have a lot of other exciting ideas as well that would have been interesting to prototype. But we are satisfied with the outcome of our work. Other ideas can be tested in future work. In the first user test, we found that a Timeline is not required by our users even especially older participants like a chronological structure. In contrast, a Map could be an enhancement by adding a new, entertaining view to the system. In addition, we know that a thematic order is the best choice for structuring the archive. We also understood that putting a picture into multiple collections is not confusing but well understandable and even wanted by many users. In the second user test, with the help of paper prototyping, we not only learned how easily and fast we can simulate a new feature idea but also got some helpful insights. We discovered Stories to be a nice-to-have feature. Except for the aspect of our participants' missing encouragement to create their own Stories, which should be tested again, this feature seems to be a valid option to increase knowledge transfer with our system and thereby contribute to our main goal. Due to our time constraints, these insights were not put into practice. But this work explored potential future implementations. We proposed many ideas for possible new features. Three of them were prototyped and tested. This led to the outlook of the Map and Stories being features that can shape future work on this project.

## 8 Summary

Our goal was to implement a crowd-sourced picture archive in collaboration with our project partner. Our main focus was to enable the transfer of knowledge to conserve as much of the analog pictures' value as possible.

Therefore, we built a system that integrates the scanning process, a structuring and an annotation tool. This helps to engage users to consume and share their knowledge by providing a convenient user interface that allows browsing through the available pictures and quickly contribute their knowledge.

In the beginning, we defined crowd-sourcing in the cultural heritage domain and presented different crowd-sourcing platforms. Then, we introduced our project partner, the Bad Harzburg-Stiftung. We illustrated and investigated existing platforms that focus on sharing historical pictures and knowledge. Following, we examined our projects prerequisites such as the parties involved, the legacy website and potentials for crowd-sourcing.

We discussed related work and introduced the different roles interacting with our system. We categorized three kinds of photo management software: personal photo archives, photo sharing platforms and website builder. Layout, upload process and personalized structure were compared. We derived requirements for our system and in particular the data model, which the content models in the backend are based on.

We presented our qualitative and quantitative testing methods to validate the system's usability. We pointed out ways to improve the user experience by implementing the feedback we gained from usability testings and interviews. We focussed on the navigation through the user interface, the comment section and the curator tools.

Following the elaboration of system and design requirements we showed their mapping to a technical implementation. We described how we developed our frontend application with the help of the UI-framework React. We took a closer look into the architecture and the most important components. We gave an overview over the data flow within our system. Additionally, GraphQL as a query language and server runtime was introduced as the interface to our backend.

For this backend system, we discussed the usage of Content Management Systems. We introduced Strapi, an open-source representative of such systems, and described how we used and customized it. The goal was to fulfill the requirements of the data model and to provide the necessary interfaces for the frontend application. Thereby, we showed how we dealt with the most important technical challenges with the use of this kind of backend application and evaluated our solutions.

We discussed various opportunities for automations that we encountered during the development process of both the frontend and backend applications. Their goal was to simplify the developers' experience. This work introduced several tools to

improve our workflow before and during development as well as during deployment. These dealt with aspects like data migration, Continuous Integration and Continuous Deployment pipelines.

In the end, we presented three prototypes we built and tested with participants from Bad Harzburg to explore the system's design space even further. Our methodological approaches were presented and testings' outcomes were discussed. Structuring pictures by location was found to be a better approach than structuring by time, although most people like the thematical structure already implemented in the system. With the third prototype, Stories were perceived to be a promising option for future work.

This work also proposed further ideas for future work. The whole project could be shifted by implementing social media aspects found in related work like user accounts, upload of personal pictures, co-curated collections and group or forum functionalities. The initial requirements can be further refined by developing auto curated collections using machine learning approaches and new search and filter functionalities. Besides some small changes in the design, a bulk editing tool should be subject to future work to further increase the system's usability. The technical implementation can be improved by changing the client-server communication from GraphQL to REST and by using an external search engine. Regarding the deployment, security concerns and support for additional scanner models need to be addressed.



# Appendices



# A Appendix

## A.1 Evaluation

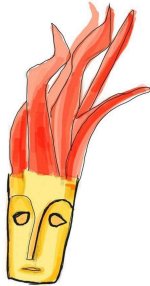
### A.1.1 Test Tasks

1. Finde dieses Bild.



2. Du warst in den 70ern bei der Einweihung eines Freibads, bei dem auch zwei Schauspieler engagiert wurden, ein Mann und eine Frau. Du erinnerst dich an eine Szene, bei der der Mann die Frau auf der einen Schulter getragen hat. Finde ein Bild davon.

3. Deine Tante hat dir immer wieder von dem Walpurgisfest erzählt, in dem dein Onkel Friedema mitgewirkt hat. Sie hat auch noch eine Maske von diesem Tag, die zu seinem Kostüm gehört hat. Die Maske ist golden und hat lange orangene Flammen, die aus dem Kopf hervor steigen. Jetzt möchtest du einmal nachschauen, ob du ein passendes Bild findest.



4. Bei der Galopprennwoche 1950 war auch Hans Günther Winkler beteiligt. Auf einem Pferd mit weißem Körper und dunklen Beinen. Dieser war ein deutscher Springreiter. Er gewann zwischen 1956 und 1976 fünf Gold- und eine Silbermedaille. Finde ein Bild von ihm und teile dein Wissen über ihn mit.
5. Dein Onkel Hans berichtet dir von einem Erlebnis aus seiner frühen Kindheit, als er beim Finkenschlagen in Hohegeiß dabei war. Finde ein passendes Bild, auf dem dein Onkel zu sehen sein könnte und merke an, dass es sich beim Finkenmanöver um eine Art Gesangswettbewerb für Buchfinken, eine über 500 Jahre alte Tradition aus dem Harz, handelt, die sogar zum immateriellen UNESCO-Kulturerbe zählt.
6. Die 1988/89 zur Miss Germany gekrönte Nicole Reinhardt stattete der Sole-Therme einen Besuch ab. Sie flüchtete aus der DDR in den Westen. Ergänze/kommentiere diese Information zu einem Bild von ihr.
7. Eine dänische Prinzessin eröffnete 1988 eine Veranstaltung in Bad Harzburg. Finde ihren Namen und die Veranstaltung.

### A.1.2 Meta Questionnaire

Fragen zu User Nr. \_\_\_\_\_ Test: \_\_\_\_\_ Datum: \_\_\_\_\_

#### Technikaffinität

Bitte **markieren** Sie:

Wie häufig benutzen Sie folgende Geräte:

- |              |     |        |                    |         |
|--------------|-----|--------|--------------------|---------|
| - Handy      | nie | selten | mehrfach die Woche | täglich |
| - Tablet     | nie | selten | mehrfach die Woche | täglich |
| - Laptop     | nie | selten | mehrfach die Woche | täglich |
| - Desktop-PC | nie | selten | mehrfach die Woche | täglich |

#### Wohnort

Bitte **markieren** Sie:

Bad- Harzburg      Harz      anderer Ort: \_\_\_\_\_

#### Bezug zu Bad Harzburg?

z.B. zur Schule in Bad Harzburg gegangen, 30 Jahre dort gelebt, erst seit kurzem in Bad Harzburg...

---

---

### A.1.3 Meta and SUS questionnaire results

Bad Harzburg = BHB

Subject	Mobile Phone	Tablet	Laptop	Desktop	Domicile	Connection to BHB	Age	SUS
0 (Pilot)	daily	never	seldom	seldom	BHB	36 years	50-60	95
1	daily	selten	seldom	weekly	Goslar	3 years	40-50	72,5
2	daily	weekly	never	daily	Domburg	2 months	20-30	52,5
3	daily	daily	seldom	never	BHB	35 years	60-70	72,5
4	daily	never	weekly	never	BHB	30 years	60-70	55
5	daily	never	weekly	weekly	Bündheim (BHB)	born in BHB	60-70	55
6	daily	daily	seldom	never	BHB	48 years	60-70	67,5
7	daily	daily	daily	never	BHB	born in BHB	60-70	62,5

### A.1.4 Evaluation Questions

1. Fragen:

1. Wie hast du die Suche empfunden?
2. Was hat dir gefallen an dem System?
3. Was hat dich eingeschränkt bzw. was war unpraktisch?
4. Was hättest du dir gewünscht?

### A.1.5 Final Evaluation Questions

1. Was fandest du gut?
2. Was hat dich gestört?
3. Was muss unbedingt geändert werden?
4. Was hat dir am besten gefallen?
5. Meta Feedback
  - Anmerkung zum Test?

## A.2 Queries and Filters

**Listing A.1:** The complete filters object for the search terms "Harz" and "1954"

```

1 {
2   "and": [{
3     "or": [{"keyword_tags": {"name": { "containsi": "Harz"}},
4           {"verified_keyword_tags": {"name": { "containsi": "Harz"}},
5           {"person_tags": {"name": { "containsi": "Harz"}},
6           {"verified_person_tags": {"name": { "containsi": "Harz"}},

```

```

7     {"collections": {"name": { "containsi": "Harz"}},
8     {"location_tags": {"name": { "containsi": "Harz"}},
9     {"verified_location_tags": {"name": { "containsi": "Harz"}},
10    {"descriptions": {"text": { "containsi": "Harz"}},
11    {"time_range_tag": {}},
12    {"verified_time_range_tag": {}}
13  ]
14 },
15 {
16   "or": [
17     {"keyword_tags": {"name": { "containsi": "1954"}},
18     {"verified_keyword_tags": {"name": { "containsi": "1954"}},
19     {"person_tags": {"name": { "containsi": "1954"}},
20     {"verified_person_tags": {"name": { "containsi": "1954"}},
21     {"collections": {"name": { "containsi": "1954"}},
22     {"location_tags": {"name": { "containsi": "1954"}},
23     {"verified_location_tags": {"name": { "containsi": "1954"}},
24     {"descriptions": {"text": { "containsi": "1954"}},
25     {"time_range_tag": {
26       "start": {"gte": "1954-01-01T00:00:00.000Z"},
27       "end": {"lte": "1954-12-31T23:59:59.000Z"}},
28     {"verified_time_range_tag": {
29       "start": {"gte": "1954-01-01T00:00:00.000Z"},
30       "end": {"lte": "1954-12-31T23:59:59.000Z"}}
31   ]
32 }
33 ]
34 }

```

**Listing A.2:** SQL Query based on the first implementation of the described search scenario

```

1  SELECT DISTINCT "t0"."published_at", "t0".* FROM "public"."pictures" AS "t0"
2  LEFT JOIN "public"."pictures_keyword_tags_links" AS "t1" ON "t0"."id" =
   ↪ "t1"."picture_id"
3  LEFT JOIN "public"."keyword_tags" AS "t2" ON "t1"."keyword_tag_id" = "t2"."id"
4  LEFT JOIN "public"."pictures_verified_keyword_tags_links" AS "t3" ON "t0"."id" =
   ↪ "t3"."picture_id"
5  LEFT JOIN "public"."keyword_tags" AS "t4" ON "t3"."keyword_tag_id" = "t4"."id"
6  LEFT JOIN "public"."pictures_time_range_tag_links" AS "t5" ON "t0"."id" =
   ↪ "t5"."picture_id"
7  LEFT JOIN "public"."time_range_tags" AS "t6" ON "t5"."time_range_tag_id" =
   ↪ "t6"."id"
8  LEFT JOIN "public"."pictures_verified_time_range_tag_links" AS "t7" ON "t0"."id" =
   ↪ "t7"."picture_id"
9  LEFT JOIN "public"."time_range_tags" AS "t8" ON "t7"."time_range_tag_id" =
   ↪ "t8"."id"
10 LEFT JOIN "public"."pictures_person_tags_links" AS "t9" ON "t0"."id" =
   ↪ "t9"."picture_id"
11 LEFT JOIN "public"."person_tags" AS "t10" ON "t9"."person_tag_id" = "t10"."id"
12 LEFT JOIN "public"."pictures_verified_person_tags_links" AS "t11" ON "t0"."id" =
   ↪ "t11"."picture_id"
13 LEFT JOIN "public"."person_tags" AS "t12" ON "t11"."person_tag_id" = "t12"."id"
14 LEFT JOIN "public"."pictures_collections_links" AS "t13" ON "t0"."id" =
   ↪ "t13"."picture_id"
15 LEFT JOIN "public"."collections" AS "t14" ON "t13"."collection_id" = "t14"."id"
16 LEFT JOIN "public"."pictures_location_tags_links" AS "t15" ON "t0"."id" =
   ↪ "t15"."picture_id"

```

```

17 LEFT JOIN "public"."location_tags" AS "t16" ON "t15"."location_tag_id" = "t16"."id"
18 LEFT JOIN "public"."pictures_verified_location_tags_links" AS "t17" ON "t0"."id" =
   ⇨ "t17"."picture_id"
19 LEFT JOIN "public"."location_tags" AS "t18" ON "t17"."location_tag_id" = "t18"."id"
20 LEFT JOIN "public"."pictures_descriptions_links" AS "t19" ON "t0"."id" =
   ⇨ "t19"."picture_id"
21 LEFT JOIN "public"."descriptions" AS "t20" ON "t19"."description_id" = "t20"."id"
22 LEFT JOIN "public"."pictures_keyword_tags_links" AS "t21" ON "t0"."id" =
   ⇨ "t21"."picture_id"
23 LEFT JOIN "public"."keyword_tags" AS "t22" ON "t21"."keyword_tag_id" = "t22"."id"
24 LEFT JOIN "public"."pictures_verified_keyword_tags_links" AS "t23" ON "t0"."id" =
   ⇨ "t23"."picture_id"
25 LEFT JOIN "public"."keyword_tags" AS "t24" ON "t23"."keyword_tag_id" = "t24"."id"
26 LEFT JOIN "public"."pictures_time_range_tag_links" AS "t25" ON "t0"."id" =
   ⇨ "t25"."picture_id"
27 LEFT JOIN "public"."time_range_tags" AS "t26" ON "t25"."time_range_tag_id" =
   ⇨ "t26"."id"
28 LEFT JOIN "public"."pictures_verified_time_range_tag_links" AS "t27" ON "t0"."id" =
   ⇨ "t27"."picture_id"
29 LEFT JOIN "public"."time_range_tags" AS "t28" ON "t27"."time_range_tag_id" =
   ⇨ "t28"."id"
30 LEFT JOIN "public"."pictures_person_tags_links" AS "t29" ON "t0"."id" =
   ⇨ "t29"."picture_id"
31 LEFT JOIN "public"."person_tags" AS "t30" ON "t29"."person_tag_id" = "t30"."id"
32 LEFT JOIN "public"."pictures_verified_person_tags_links" AS "t31" ON "t0"."id" =
   ⇨ "t31"."picture_id"
33 LEFT JOIN "public"."person_tags" AS "t32" ON "t31"."person_tag_id" = "t32"."id"
34 LEFT JOIN "public"."pictures_collections_links" AS "t33" ON "t0"."id" =
   ⇨ "t33"."picture_id"
35 LEFT JOIN "public"."collections" AS "t34" ON "t33"."collection_id" = "t34"."id"
36 LEFT JOIN "public"."pictures_location_tags_links" AS "t35" ON "t0"."id" =
   ⇨ "t35"."picture_id"
37 LEFT JOIN "public"."location_tags" AS "t36" ON "t35"."location_tag_id" = "t36"."id"
38 LEFT JOIN "public"."pictures_verified_location_tags_links" AS "t37" ON "t0"."id" =
   ⇨ "t37"."picture_id"
39 LEFT JOIN "public"."location_tags" AS "t38" ON "t37"."location_tag_id" = "t38"."id"
40 LEFT JOIN "public"."pictures_descriptions_links" AS "t39" ON "t0"."id" =
   ⇨ "t39"."picture_id"
41 LEFT JOIN "public"."descriptions" AS "t40" ON "t39"."description_id" = "t40"."id"
42 WHERE (((LOWER(CAST("t2"."name" AS VARCHAR)) LIKE LOWER(?))
43         OR (LOWER(CAST("t4"."name" AS VARCHAR)) LIKE LOWER(?))
44         OR (LOWER(CAST("t10"."name" AS VARCHAR)) LIKE LOWER(?))
45         OR (LOWER(CAST("t12"."name" AS VARCHAR)) LIKE LOWER(?))
46         OR (LOWER(CAST("t14"."name" AS VARCHAR)) LIKE LOWER(?))
47         OR (LOWER(CAST("t16"."name" AS VARCHAR)) LIKE LOWER(?))
48         OR (LOWER(CAST("t18"."name" AS VARCHAR)) LIKE LOWER(?))
49         OR (LOWER(CAST("t20"."text" AS VARCHAR)) LIKE LOWER(?)))
50 AND ((LOWER(CAST("t22"."name" AS VARCHAR)) LIKE LOWER(?))
51       OR (LOWER(CAST("t24"."name" AS VARCHAR)) LIKE LOWER(?))
52       OR ("t26"."start" >= ? AND "t26"."end" <= ?)
53       OR ("t28"."start" >= ? AND "t28"."end" <= ?)
54       OR (LOWER(CAST("t30"."name" AS VARCHAR)) LIKE LOWER(?))
55       OR (LOWER(CAST("t32"."name" AS VARCHAR)) LIKE LOWER(?))
56       OR (LOWER(CAST("t34"."name" AS VARCHAR)) LIKE LOWER(?))
57       OR (LOWER(CAST("t36"."name" AS VARCHAR)) LIKE LOWER(?))
58       OR (LOWER(CAST("t38"."name" AS VARCHAR)) LIKE LOWER(?))
59       OR (LOWER(CAST("t40"."text" AS VARCHAR)) LIKE LOWER(?))))
60 AND "t0"."published_at" IS NOT NULL)
61 ORDER BY "t0"."published_at" ASC
62 LIMIT ?

```



**Listing A.3:** SQL Query based on the optimized implementation for the described search scenario

```

1  SELECT DISTINCT "pictures".* FROM "pictures"
2  LEFT JOIN "pictures_keyword_tags_links" ON "pictures"."id" =
   ↪ "pictures_keyword_tags_links"."picture_id"
3  LEFT JOIN "pictures_verified_keyword_tags_links" ON "pictures"."id" =
   ↪ "pictures_verified_keyword_tags_links"."picture_id"
4  LEFT JOIN "keyword_tags" ON "pictures_verified_keyword_tags_links"."keyword_tag_id"
   ↪ = "keyword_tags"."id" OR "pictures_keyword_tags_links"."keyword_tag_id" =
   ↪ "keyword_tags"."id"
5  LEFT JOIN "pictures_location_tags_links" ON "pictures"."id" =
   ↪ "pictures_location_tags_links"."picture_id"
6  LEFT JOIN "pictures_verified_location_tags_links" ON "pictures"."id" =
   ↪ "pictures_verified_location_tags_links"."picture_id"
7  LEFT JOIN "location_tags" ON
   ↪ "pictures_verified_location_tags_links"."location_tag_id" = "location_tags"."id"
   ↪ OR "pictures_location_tags_links"."location_tag_id" = "location_tags"."id"
8  LEFT JOIN "pictures_person_tags_links" ON "pictures"."id" =
   ↪ "pictures_person_tags_links"."picture_id"
9  LEFT JOIN "pictures_verified_person_tags_links" ON "pictures"."id" =
   ↪ "pictures_verified_person_tags_links"."picture_id"
10 LEFT JOIN "person_tags" ON "pictures_verified_person_tags_links"."person_tag_id" =
   ↪ "person_tags"."id" OR "pictures_person_tags_links"."person_tag_id" =
   ↪ "person_tags"."id"
11 LEFT JOIN "pictures_time_range_tag_links" ON "pictures"."id" =
   ↪ "pictures_time_range_tag_links"."picture_id"
12 LEFT JOIN "pictures_verified_time_range_tag_links" ON "pictures"."id" =
   ↪ "pictures_verified_time_range_tag_links"."picture_id"
13 LEFT JOIN "time_range_tags" ON
   ↪ "pictures_verified_time_range_tag_links"."time_range_tag_id" =
   ↪ "time_range_tags"."id" OR "pictures_time_range_tag_links"."time_range_tag_id" =
   ↪ "time_range_tags"."id"
14 LEFT JOIN "pictures_descriptions_links" ON "pictures"."id" =
   ↪ "pictures_descriptions_links"."picture_id"
15 LEFT JOIN "descriptions" ON "pictures_descriptions_links"."description_id" =
   ↪ "descriptions"."id"
16 LEFT JOIN "pictures_collections_links" ON "pictures"."id" =
   ↪ "pictures_collections_links"."picture_id"
17 LEFT JOIN "collections" ON "pictures_collections_links"."collection_id" =
   ↪ "collections"."id"
18 WHERE ("keyword_tags"."name" ilike ?
19        OR "location_tags"."name" ilike ?
20        OR "person_tags"."name" ilike ?
21        OR "collections"."name" ilike ?
22        OR "descriptions"."text" ilike ?)
23 AND ("keyword_tags"."name" ilike ?
24       OR "location_tags"."name" ilike ?
25       OR "person_tags"."name" ilike ?
26       OR "collections"."name" ilike ?
27       OR "descriptions"."text" ilike ?
28       OR ("time_range_tags"."start" >= ? AND "time_range_tags"."end" <= ?))
29 AND "pictures"."published_at" IS NOT NULL
30 ORDER BY "pictures"."published_at" ASC
31 LIMIT ?

```

## A.3 GraphQL and Monitoring

### A.3.1 GraphQL Code Generator

When configuring the code generator, we use the `codegen.yml` file shown in Listing A.4 to set it up.

**Listing A.4:** The `codegen.yml` file contains the configuration of the GraphQL code generator. It sets the schema definitions source and the target of where to store the generated code (`APIConnector.tsx`).

```
1  overwrite: true
2  schema:
3    - https://bp.bad-harzburg-stiftung.de/api/graphql:
4      loader: ./src/graphql/schema/loadRemoteSchema.js
5    - "src/graphql/schema/schema.json"
6  documents: "src/graphql/*.graphql"
7  generates:
8    src/graphql/APIConnector.tsx:
9      plugins: [...]
10     config: [...]
```

We instruct the generator to use the remote schema as well as the local fallback schema. The two will be merged<sup>1</sup> and used as the generator input along with any file located in the same folder with the `.graphql` file type. In our case, the only file satisfying this condition is the `operation.graphql` file, containing the mutation and query definitions our developers wrote. According to these definitions, hooks for use in React components are generated.

---

<sup>1</sup><https://www.graphql-code-generator.com/docs/config-reference/schema-field> (last accessed: 2022-07-13).

## A.3.2 Monitoring the Application

**Figure A.1:** The sentry dashboard groups and shows all captured error messages. For each error there is a separate page detailing the stack trace as well as the clients information if it is present (see Figure A.2).

**Figure A.2:** The sentry detail page shows the stack trace, user agent information and code snippets for every error captured

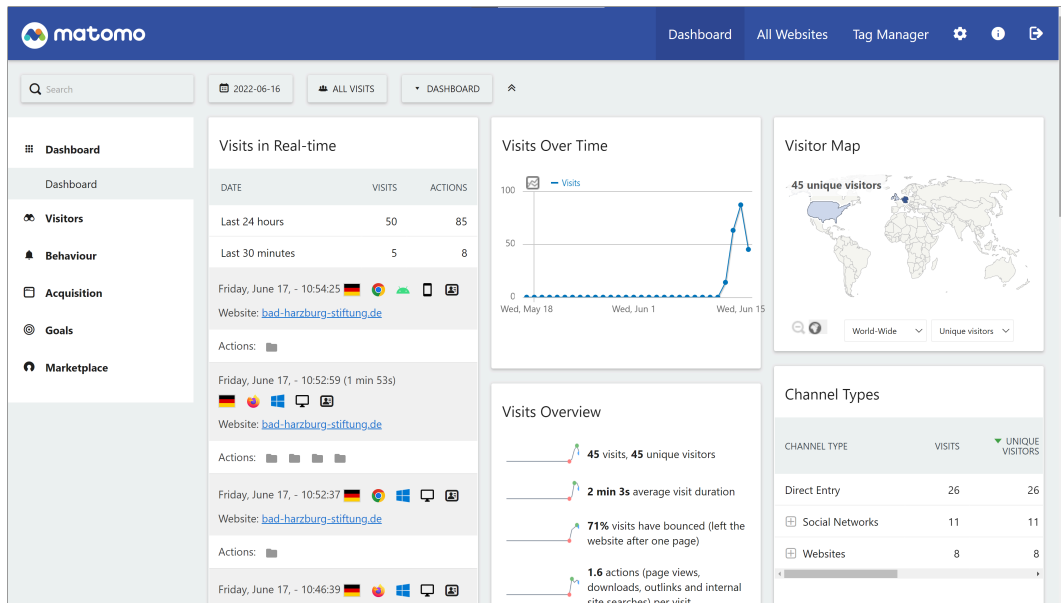


Figure A.3: The matomo dashboard shows an overview of all user interactions, the devices used and their region of origin

## A.4 Questionnaires

Durch die Karte konnte ich mich gut in der Bildermenge orientieren.

Stimme überhaupt nicht zu 1	2	3	4	Stimme voll zu 5
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Die Karte hat mir die Bilder nahe gebracht. Sie hat mir einen Zugang zum Bilderarchiv gegeben.

Stimme überhaupt nicht zu 1	2	3	4	Stimme voll zu 5
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Die Karte ist übersichtlich.

Stimme überhaupt nicht zu 1	2	3	4	Stimme voll zu 5
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Ich würde die Karte gerne nutzen, um mir das gesamte Herbert-Ahrens-Bilderarchiv anzuschauen.

Stimme überhaupt nicht zu 1	2	3	4	Stimme voll zu 5
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

**Figure A.4:** The Likert-type questionnaire we designed as a quantitative approach to rate the Map prototype

Durch den Zeitstrahl konnte ich mich gut in der Bildermenge orientieren.

Stimme überhaupt nicht zu 1	2	3	4	Stimme voll zu 5
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Der Zeitstrahl hat mir die Bilder nahe gebracht. Er hat mir einen Zugang zum Bilderarchiv gegeben.

Stimme überhaupt nicht zu 1	2	3	4	Stimme voll zu 5
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Der Zeitstrahl ist übersichtlich.

Stimme überhaupt nicht zu 1	2	3	4	Stimme voll zu 5
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Ich würde den Zeitstrahl gerne nutzen, um mir das gesamte Herbert-Ahrens-Bilderarchiv anzuschauen.

Stimme überhaupt nicht zu 1	2	3	4	Stimme voll zu 5
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

**Figure A.5:** The Likert-type questionnaire we designed as a quantitative approach to rate the Timeline prototype

## References

- [1] C. Alvino and J. Basilico. *Learning a Personalized Homepage*. Netflix Technology Blog. Apr. 2017. URL: <https://netflixtechblog.com/learning-a-personalized-homepage-aa8ec670359a> (last accessed: 2022-07-13).
- [2] Apple Support. *Anzeigen von Fotos in der App „Fotos“ auf dem iPhone*. URL: <https://support.apple.com/de-de/guide/iphone/iph3d267610/ios> (last accessed: 2022-07-13).
- [3] S. Asur and S. Huffnagel. “Taxonomy of rapid-prototyping methods and tools”. In: [1993] *Proceedings The Fourth International Workshop on Rapid System Prototyping*. IEEE, 1993. DOI: 10.1109/IWRSP.1993.263196.
- [4] Atlassian. *Git Hooks | Atlassian Git Tutorial*. URL: <https://www.atlassian.com/git/tutorials/git-hooks> (last accessed: 2022-06-28).
- [5] A. Bangor, P. T. Kortum, and J. T. Miller. “An Empirical Evaluation of the System Usability Scale”. In: *International Journal of Human-Computer Interaction* 24.6 (July 2008). DOI: 10.1080/10447310802205776.
- [6] A. Bangor, P. T. Kortum, and J. T. Miller. “Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale”. In: *Journal of Usability Studies* 4.3 (May 2009).
- [7] A. Bodin. *Announcing Strapi v4*. Nov. 30, 2021. URL: <https://strapi.io/blog/announcing-strapi-v4> (last accessed: 2022-07-11).
- [8] J. Brooke. “SUS—A Quick and Dirty Usability Scale”. In: *Usability evaluation in industry*. Edited by P. W. Jordan, B. Thomas, I. L. McClelland, and B. Weerdmeester. CRC Press, 1996.
- [9] J. Brooke. “SUS: a retrospective”. In: *Journal of usability studies* 8.2 (2013).
- [10] R. Budde and H. Zullighoven. “Prototyping revisited”. In: *COMPEURO’90: Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering – Systems Engineering Aspects of Complex Computerized Systems*. 1990. DOI: 10.1109/CMPEUR.1990.113653.
- [11] R. Budde, K. Kautz, K. Kuhlenkamp, and H. Züllighoven. “What is prototyping?” In: *Information Technology & People* 6.2/3 (Jan. 1992). DOI: 10.1108/EUM000000003546.
- [12] P. P.-S. Chen. “The entity-relationship model - toward a unified view of data”. In: *ACM Transactions on Database Systems* 1.1 (1976). DOI: 10.1145/320434.320440. URL: <https://doi.org/10.1145/320434.320440>.
- [13] Contentful. *Headless CMS explained in 1 minute*. URL: <https://www.contentful.com/r/knowledgebase/what-is-headless-cms/> (last accessed: 2022-05-18).

## References

- [14] L. Crum. “Laws of UX: Using Psychology to Design Better Products & Services”. In: *Design and Culture* 12.3 (Sept. 2020). DOI: 10.1080/17547075.2020.1822074.
- [15] T. Deplanque. *Strapi plugin migrations*. June 2022. URL: <https://github.com/TonyDeplanque/strapi-plugin-migrations> (last accessed: 2022-06-16).
- [16] Digital Curation Centre. *What is digital curation?* URL: <https://www.dcc.ac.uk/about/digital-curation> (last accessed: 2022-05-25).
- [17] O. Elgabry. *Plug-in Architecture*. May 2020. URL: <https://medium.com/omarelgabrys-blog/plugin-in-architecture-dec207291800>.
- [18] J. Esterkin. *Plug-In Architecture*. June 29, 2020. URL: <https://openclassrooms.com/en/courses/6397806-design-your-software-architecture-using-industry-standard-patterns/6896171-plugin-in-architecture>.
- [19] F. Fagerholm and J. Münch. “Developer experience: Concept and definition”. In: *2012 International Conference on Software and System Process (ICSSP)*. June 2012. DOI: 10.1109/ICSSP.2012.6225984.
- [20] R. T. Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. 2000.
- [21] M. Gao, P. Kortum, and F. L. Oswald. “Multi-Language Toolkit for the System Usability Scale”. In: *International Journal of Human–Computer Interaction* 36.20 (Dec. 2020). DOI: 10.1080/10447318.2020.1801173.
- [22] S. Gao. *Paper Prototyping — How-to, Pros & Cons, and the Struggles of Guerrilla Usability Testing*. Jan. 31, 2018. URL: <https://medium.com/@sheneral/paper-prototyping-how-to-pros-cons-and-the-struggles-of-guerrilla-usability-testing-5546dd446d5e> (last accessed: 2022-07-13).
- [23] H. Gernsheim. *A Concise History of Photography*. Courier Corporation, Jan. 1986.
- [24] HackEDU Team. *How to prevent SQL Injection vulnerabilities: How Prepared Statements Work*. Feb. 11, 2020. URL: <https://www.hackedu.com/blog/how-to-prevent-sql-injection-vulnerabilities-how-prepared-statements-work> (last accessed: 2022-07-14).
- [25] E. C. Hallett, Z. Roberts, J. Sweet, M. L. Chan, Y. Sun, W. Dick, A. Monge, and K.-P. L. Vu. “Computer Accessibility: How Individuals with Low Vision Adjust the Presentation of Electronic Text for Academic Reading”. In: *Procedia Manufacturing* 3 (2015), pages 5206–5213. DOI: 10.1016/j.promfg.2015.07.586.
- [26] J. Howe. “The Rise of Crowdsourcing”. In: *Wired* (June 2006). URL: <https://www.wired.com/2006/06/crowds/> (last accessed: 2022-07-08).
- [27] Imagely. *Working with Albums*. URL: <https://www.imagely.com/docs/albums/> (last accessed: 2022-07-13).
- [28] Institut für Konstruktionstechnik. *Papierprototyp - Methodos*. Technische Universität Braunschweig. June 30, 2017. URL: <https://methodos.ik.ing.tu-bs.de/methode/Papierprototyp.html> (last accessed: 2022-06-14).
- [29] N. Jakob. *Definition Usability*. Jan. 2012. URL: <https://www.nngroup.com/articles/usability-101-introduction-to-usability/>.



- [30] S. Jamieson. "Likert scales: how to (ab)use them". In: *Medical Education* 38.12 (Dec. 2004). DOI: 10.1111/j.1365-2929.2004.02012.x.
- [31] L. S. Jensen, A. G. Özkil, N. H. Mortensen, et al. "Prototypes in engineering design: Definitions and strategies". In: *DS 84: Proceedings of the DESIGN 2016 14th International Design Conference*. 2016.
- [32] A. Joshi, S. Kale, S. Chandel, and D. Pal. "Likert Scale: Explored and Explained". In: *British Journal of Applied Science & Technology* 7.4 (Jan. 2015). DOI: 10.9734/BJAST/2015/14975.
- [33] E. Koptyug. *Number of smartphone users in Germany 2009-2021*. URL: <https://www.statista.com/statistics/461801/number-of-smartphone-users-in-germany/> (last accessed: 2022-07-14).
- [34] M. D. Levi and F. G. Conrad. "Usability testing of world wide web sites". In: *Conference on Human Factors in Computing Systems: CHI'97 extended abstracts on Human factors in computing systems: looking to the future*. Volume 22. 27. 1997.
- [35] J. R. Lewis and J. Sauro. "Item Benchmarks for the System". In: 13.3 (2018).
- [36] Lexico. *English Dictionary*. URL: [lexico.com](https://www.lexico.com/) (last accessed: 2022-07-13).
- [37] D. MacAskill. *The world's most-beloved, money-losing business needs your help*. Dec. 2019. URL: <https://blog.flickr.net/en/2019/12/19/the-worlds-most-beloved-money-losing-business-needs-your-help/> (last accessed: 2022-07-13).
- [38] N. Mathur. *Rethink Your Master Data: The Limits of Relational Databases*. Neo4j Graph Data Platform. June 2020. URL: <https://neo4j.com/blog/rethink-your-master-data-the-limits-of-relational-databases>.
- [39] F. Matthes, C. Schulz, and K. Haller. "Testing amp; quality assurance in data migration projects". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. Sept. 2011. DOI: 10.1109/ICSM.2011.6080811.
- [40] D. Mehaffy. *Discussion regarding the complex response structure for REST & GraphQL (Developer Experience) - Discussions*. Dec. 2021. URL: <https://forum.strapi.io/t/discussion-regarding-the-complex-response-structure-for-rest-graphql-developer-experience/13400> (last accessed: 2022-07-11).
- [41] Microsoft. *The TypeScript Handbook*. URL: <https://www.typescriptlang.org/docs/handbook/intro.html> (last accessed: 2022-05-18).
- [42] W. Moris. *Why You Should Offer Image Commenting on Your Photography Website (And How to Get Started)*. Jan. 5, 2021. URL: <https://www.imagely.com/image-commenting/> (last accessed: 2022-07-13).
- [43] J. Morris. *Practical Data Migration*. BCS, The Chartered Institute, 2012.
- [44] J. Nielsen. *Thinking Aloud: The #1 Usability Tool*. URL: <https://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/> (last accessed: 2022-07-11).
- [45] J. Nielsen. *Usability engineering*. Morgan Kaufmann, 1994.
- [46] M. T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. The Pragmatic Programmers. O'Reilly Media, Feb. 2018.

## References

- [47] J. Oomen and L. Aroyo. "Crowdsourcing in the Cultural Heritage Domain: Opportunities and Challenges". In: *Proceedings of the 5th International Conference on Communities and Technologies*. C&T '11. Association for Computing Machinery, 2011. DOI: 10.1145/2103354.2103373.
- [48] Oracle. *What is a content management system (CMS)?* URL: <https://www.oracle.com/content-management/what-is-cms/> (last accessed: 2022-07-13).
- [49] K. Orfanou, N. Tselios, and C. Katsanos. "Perceived usability evaluation of learning management systems: Empirical evaluation of the System Usability Scale". In: *The International Review of Research in Open and Distributed Learning* 16.2 (Apr. 2015). DOI: 10.19173/irrodl.v16i2.1955.
- [50] V. Pareto. *Manual of Political Economy*. AM Kelley, 1971.
- [51] W. Reinhardt, E. Ruegenhagen, and B. Rummel. *System Usability Scale – jetzt auch auf Deutsch*. Feb. 16, 2016. URL: <https://blogs.sap.com/2016/02/01/system-usability-scale-jetzt-auch-auf-deutsch/>.
- [52] M. Rubel. *Easy Automated Snapshot-Style Backups with Rsync*. 2004. URL: [http://www.mikerubel.org/computers/rsync\\_snapshots/](http://www.mikerubel.org/computers/rsync_snapshots/) (last accessed: 2022-05-25).
- [53] J. Sauro and J. R. Lewis. *Quantifying the User Experience: Practical Statistics for User Research*. Morgan Kaufmann, 2016.
- [54] N. Simon. *The Participatory Museum*. Museum 2.0, 2010. URL: <https://www.participatorymuseum.org/read/>.
- [55] Statista. *Bevölkerung in Deutschland I*. URL: <https://de.statista.com/statistik/studie/id/7661/dokument/bevoelkerung-in-deutschland-i-statista-dossier/> (last accessed: 2022-07-11).
- [56] Statista. *Mobile percentage of website traffic 2021*. URL: <https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/> (last accessed: 2022-07-13).
- [57] Strapi Team. *Headless CMS explained in 5 minutes*. URL: <https://strapi.io/what-is-headless-cms> (last accessed: 2022-05-18).
- [58] Strapi Team. *Strapi User Guide*. URL: <https://docs.strapi.io/user-docs/latest/getting-started/introduction.html> (last accessed: 2022-05-25).
- [59] The PostgreSQL Global Development Group. *PostgreSQL 14.6 Documentation*. May 2022. URL: <https://www.postgresql.org/docs/14> (last accessed: 2022-07-13).
- [60] R. A. Virzi. "What can you Learn from a Low-Fidelity Prototype?" In: *Proceedings of the Human Factors Society Annual Meeting* 33.4 (Oct. 1989). DOI: 10.1177/154193128903300405.

## Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren/Redaktion
148	978-3-86956-544-6	<b>openHPI: 10 years of MOOCs at the Hasso Plattner Institute</b>	Christoph Meinel, Christian Willems, Thomas Staubitz, Dominic Sauer, Christiane Hagedorn
147	978-3-86956-533-0	<b>Modeling and formal analysis of meta-ecosystems with dynamic structure using graph transformation</b>	Boris Flotterer, Maria Maximova, Sven Schneider, Johannes Dyck, Christian Zöllner, Holger Giese, Christelle Hély, Cédric Gaucherel
146	978-3-86956-532-3	<b>Probabilistic metric temporal graph logic</b>	Sven Schneider, Maria Maximova, Holger Giese
145	978-3-86956-528-6	<b>Learning from failure: a history-based, lightweight test prioritization technique connecting software changes to test failures</b>	Falco Dürsch, Patrick Rein, Toni Mattis, Robert Hirschfeld
144	978-3-86956-526-2	<b>Die HPI Schul-Cloud – Von der Vision zur digitale Infrastruktur für deutsche Schulen</b>	Christoph Meinel, Catrina John, Tobias Wollowski
143	978-3-86956-531-6	<b>Invariant analysis for multi-agent graph transformation systems using k-Induction</b>	Sven Schneider, Maria Maximova, Holger Giese
142	978-3-86956-524-8	<b>Quantum computing from a software developers perspective</b>	Marcel Garus, Rohan Sawahn, Jonas Wanke, Clemens Tiedt, Clara Granzow, Tim Kuffner, Jannis Rosenbaum, Linus Hagemann, Tom Wollnik, Lorenz Woth, Felix Auringer, Tobias Kantusch, Felix Roth, Konrad Hanff, Niklas Schilli, Leonard Seibold, Marc Fabian Lindner, Selina Raschack
141	978-3-86956-521-7	<b>Tool support for collaborative creation of interactive storytelling media</b>	Paula Klinke, Silvan Verhoeven, Felix Roth, Linus Hagemann, Tarik Alnawa, Jens Lincke, Patrick Rein, Robert Hirschfeld
140	978-0-86956-517-0	<b>Probabilistic metric temporal graph logic</b>	Sven Schneider, Maria Maximova, Holger Giese





ISBN 978-3-86956-545-3  
ISSN 1613-5652