Web Application Development: Building a Photo Gallery

Personal Pursuit 2 – University College Twente

Martijn Atema (s1663801)
Supervisor: Fjodor van Slooten

23 May 2024

Contents

Introduction

Web apps are a complex type of application. The browser client and server(s) need to communicate seamlessly to perform many tasks, such as storing persistent data, manipulating it, and presenting it beautifully to the user. Different parts of these tasks can be performed by different components, and many technologies exist that can be used to implement them.

In this report, I will study some of these components from the perspective of the photo gallery application, a tool that can be used for managing many users' photographs in an online environment. Specifically, I will look into frameworks, databases, and data storage, and compare competing technologies to find out which fit the product best.

Proposal

This report was created as part of a Personal Pursuit at the University College Twente (ATLAS). This chapter contains the official proposal for the project.

Basic Information

Title

Web Application Development: Building a Photo Gallery

Student

Martijn Atema

Period

PP2

Abstract

For this Personal Pursuit, I will dive into my passion of software development and improve my understanding within the field of web application development. With a focus on web frameworks, I will first analyse and compare different technologies that can be used in web applications.

This analysis will lead to a "web stack", a selection of technologies which will form the basis of the application I will develop next: a photo gallery that can manage my thousands of pictures, as well as those of anyone that wants to use it.

Topic

> What is your topic? Explain why you are passionate about or interested in this topic?

Programming software is something that I've been interested in for a long time. Already in the first years of high school, I started building small applications and websites, with Visual Basic and PHP. Through the years, I added many

more languages, technologies, and tools to that list, and figuring these out has always been a joy to me. Nevertheless, it is a field that keeps developing, and one has to keep learning to stay up-to-date.

For this Personal Pursuit, I want to further expand my understanding of web applications. Within this field, numerous meta-frameworks (libraries that combine server and client code into one framework) have emerged over the past years, and have become a popular choice for web-based software. Instead of choosing the most interesting or popular one of them, I will compare a selection to find which works best for my use case. In addition, I will compare options for other components that are essential to the application, such as databases, to come to a useful "stack" of technologies.

The application that I will apply these in shall solve an issue with another hobby of mine: photography. Over the years, I have collected tens of thousands of photographs – some with family, some with friends, and some of just nature. The photo gallery application I will make should put an end to the nightmare that is managing these and put the pictures back into the spotlight where they belong.

Learning Objectives (max. 3 objectives)

> What are the learning objectives you wish to reach in this PP? A learning objective indicates what you aim to understand, or be able to do after completion of your of your PP. One objective is given.

Objective 1

Conceive, design, and execute a goal-oriented learning experience at an academic level, inspired by a personal interest or passion.

Objective 2

Understand and explain the differences between different options of technologies, and be able to select appropriate technologies for a web application.

Objective 3

Be able to apply and integrate a web framework with other technologies to develop web applications

Explaination

> Explain how reaching these objectives goes beyond the regular ATLAS curriculum and how it contributes to you becoming a 'well-rounded' new engineer.

These objectives go beyond the regular curriculum by placing an extra focus on finding "the right tool for the job", a choice that is often made for you in regular courses (where the focus is on one specific technology), or not emphasised (like in semester projects). Especially when starting new projects, but also when

adding new components to existing projects, being able to choose the correct technologies is essential.

A well-rounded engineer should be able not only use the tools that are available to them, but also select which tools to use. Together, the learning objectives for this Personal Pursuit cover the technological process of developing an application in a structured manner: from an idea to a final product. These abilities to research, compare, select, and integrate different technologies are integral to the development process not only in this field, but anywhere.

Activities

Main activities

> What are you going to do? Specify the main activities of your PP through which you intend to reach your learning objectives.

Broadly, the activities in my Personal Pursuit can be divided into a research and development phase, which will help me reach the second and third learning objectives respectively. The first learning objective is covered by the project in its entirety.

During the research phase, the focus will be on comparing different technologies. In a web application, different components work together to perform operations. To come to a selection of technologies for the photo gallery, I will compare options in the categories of frameworks, databases, and data storage.

For each of these categories, I will establish a set of criteria and their importance to the project. These shall cover the features of the technology, but also other factors, such as the developer experience. To come to the final choices, a preselection of the most popular alternatives will be analysed and compared against these criteria. The best-fitting option in each of the categories will be taken to the next phase of the project.

The development phase will start with a brief design stage, during which I will inspect other photo galleries (both offline and online) and create a mockup of my own design. With this design, I aim to adhere to responsive design principles, to ensure the photo gallery will be usable on a broad variety of devices.

The main focus in this phase, however, lies on the development itself. I will study the chosen technologies further, and integrate them with my own code to develop a web application that is fully functional and works according to the set requirements. Afterwards, I will reflect on the chosen technologies and the development process.

Final product

> What will your final product be?

For the first phase, I will create a website that documents the different technologies that I study. This will also include my reasoning behind the technologies I choose for the second phase, as well as a reflection on these choices after completing the second phase.

The most important final product, however, will be the result of the second phase: the photo gallery application itself, which should be fully functional and satisfy the requirements set together with the supervisor.

Planning

> Add a planning with main activities and milestones.

See the appendix planning for a complete project overview, with time allocated to each of the (sub-) activites.

Supervision

> What kind of supervision do you request (for instance a process supervisor, or a supervisor with specific expertise). If you already found a (potential) supervisor, please include her/his name and contact details.

Fjodor van Slooten (https://people.utwente.nl/f.vanslooten) has agreed to be my supervisor. He is an expert in the fields of web design and application development at the UT, and also teaches courses within these fields.

Outreach

> In which ways will you share your results and learning experiences with the ATLAS community? When will you do this?

First of all, the resulting application will be available publicly, and distributed under an open-source license. This will allow everyone – both within and out of the ATLAS community – to use the application, but also to learn from the source code and make their own changes to it.

Secondly, the results of my research into the different technologies will be available as a public website. This includes the comparison of different technologies, as well as a justification and reflection of the technologies used for the photo gallery application. This will make the choice easier for others looking to make web applications, and will allow them to learn from my choices (and possibly, mistakes).

At the Expo, I could present the interactive version of the application together with an overview of the technologies and choices I made.

Design

The photo gallery will be a self-hostable application that allows its users to store photos and access them from any web browser. It is primarily targeted towards individuals, families and smaller associations that want to be able to (collaborativily) organise their photo albums centrally.

The application is mainly inspired by my own wants to organise the many thousands of photographs I have taken over the past two decades, and to share them with the people that were with me on the occasions I took them. Likewise, I would like them to be able to participate by sharing their pictures with me, to have a single place where everything is stored.

Regardless of these wishes, the photo gallery is above all a means to an end: an opportunity to dive into JavaScript frameworks and the world that surrounds them. Learning about these technologies and how to use them are the primary goals of this project.

Before coming to a final design, I will perform a brief analysis of alternative solutions in the field, to provide some perspective. Together with my own views on the application, this will lead to the final requirements ands wishes for the application, which I aim to realise during the development phase of the project.

Analysis

To come to a design and requirements for the Photo Gallery, I will take a look at a few alternatives. This includes the cloud options Google Photos (targetting consumers), Flickr (targetting amateurs) and SmugMug (targeting professionals); although these are not direct alternatives to my product, they are a good representation of what most people use. In addition, I will look at PhotoPrism, an open source solution that can be self-hosted and could be considered a direct alternative to my photo gallery. Lastly, I will take a brief look at two local options: Google Gallery for phone and Windows Photos for pc.

Interface

When opening the various applications, the differences in colour schema become immediatebly clear. Although changeable dark modes have become increasingly popular, only the local apps can change colour on demand. The other options have static colours: Google Photos and Flickr boast a light theme, while SmugMug and PhotoPrism use a darker theme (although the latter's can be changed by

the administrator). As dark mode seems to be quite a controversial topic when it comes to people's preference, I will provide a switchable option for my app.

Another thing that jumps to the eye is the fluid kind of layout most overviews use: images are not aligned into a rigid grid, but are layed out in varying sizes to show them without dcropping and yet fill the available space. Notably, PhotoPrism and Google Gallery don't use this layout, and the other apps often use grids for some more advanced features. Although I think it gives a nice look to the page, the implementation of it seems rather complicated. For my photo gallery, I will stick with the good old grid view.

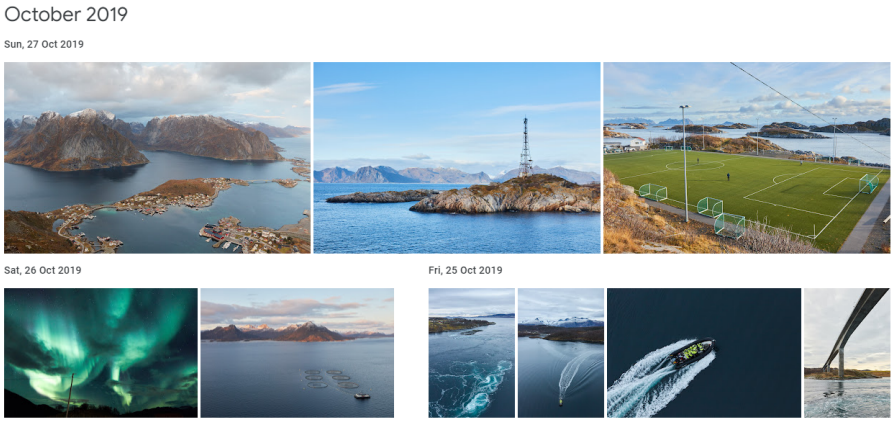

Figure: A fluid image layout in Google Photos

One aspect in which all options align, are their responsive layouts: they are made to work well on displays of different sizes, from mobile phones to desktop monitors. For modern applications, this is very much essential, and it will also be important for my photo gallery.

Organisation

Each piece of software can organise the photographs in some kind of collection. For the local options, these are equivalent to directories on the filesystem, meaning pictures can only be part of one folder at once. Online solutions are less strict, and photographs can often be part of multiple albums at the same time (PhotoPrism has both variations alongside each other, which is just confusing). Because it simplifies permissions and other logic, my photo gallery won't allow this behaviour, and photos will be part of only one album instead.

Besides albums, most tools have support for storing pictures as favourites, which is a nice addition that I wish to add. Some have a trash can where deleted items are stored for some time before being permanently deleted. Flickr, SmugMug and PhotoPrism have features that allow users to organise albums further in a structure. Google Photos has a great explore page that automatically detects shared features across all images, and allows for searching by them. However, I believe these features are too much for the initial version of the photo gallery, and they won't be added during this projet.

Photographs

All photo services allow the user to upload and store photos in their original quality, although Google also offers a feature for storing reduced-quality photos, in order to save on storage space. These full-quality pictures can be downloaded again; in addition, some services let the user download down-scaled versions. I believe uploading and downloading full-size versions are an essential part of the app, and will implement it during this project.

The software extract a varying set of metadata to show to the user. Generally, there is a set of basic metadata that is shown in a more prominent place in the interface. These include: size, date taken, location taken (if applicable), camera model, and the exposure settings of the image (aperture size, shutter speed, and ISO value). In addition to these, the services more geared towards professionals (Flickr and SmugMug) can show some more advanced information taken from the image. Although most are for viewing only, Flickr and Google Photos allow for changing the date information, and in PhotoPrism almost all metadata can be changed from the interface. During this project, I will implement basic metadata extraction to provide some information about the photos, but will keep away from any editing.

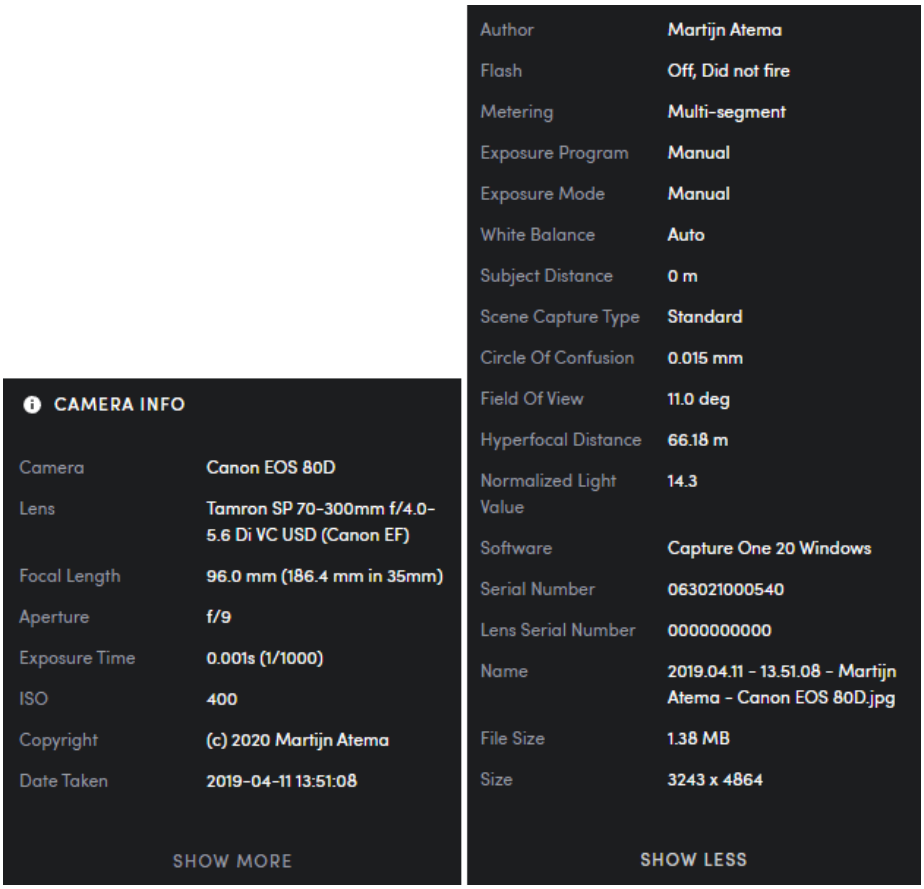

Figure: Basic and extended metadata in SmugMug

Besides image formats, the applications also offer support for varying types of

video files. In addition, PhotoPrism, Google Photos and the local applications support motion (or live) images, which are a hybrid format type that combine pictures with a short video taken at the same time. Although support for these formats is quite important for the user experience, videos are much more resource-intensive to process. Therefore, these format's won't have priority.

Users and sharing

Each of the online options has their own user system for authentication and sharing (the offline apps are single-user only) and require authentication to create albums to upload photos into. For my gallery, I will also use a built-in user database instead of relying on users of an external service.

Most solutions have some way of sharing collections with other users, by inviting them to view the album. In some cases, Google Photos being the most advanced, this can also include collaboration permissions, which means other users can add their own photos to the album. Sharing with other users is an important part of the concept of my photo gallery, and should be implemented during this project.

Besides sharing with other users, the applications also support making collections available to the public. For Flickr, this results in it showing up through the search features of the site, but generally, it gives the user a public link to share with other people. Although not a priority, I hope to also implement such an option.

Requirements

A combination of my own wishes for the product, the short analysis performed in the previous section and the learning goals of this project in its entirety lead to the following list of requirements for the product, which are divided into "hard" requirements, as well as wishes of varying priority.

Technical

- The application must be written using a JavaScript meta-framework
- The application should be locally deployable, without depending on proprietary technologies
- The application should be available in a public repository under an open source license
- The application's source should be well-documented
- The application should be written in TypeScript (*high-priority wish*)
- The application should be horizontally scalable to support very large image repositories (*low-priority wish*)
- The application should have an automated build and testing process (*low-priority wish*)

Interface

- The application interface should be responsive
- The application should display correctly in major modern browsers

- The application should have a light and dark mode
- The interface should display in the user's language (supporting English and Dutch) (*low-priority wish*)

Organisation

- Images should be organised into an album, which are viewable as one
- Images in an album should be sorted by their date
- It should be possible to move images between albums
- It should be possible to delete images from the system
- Moving and deletion should be possible for multiple images at a time (*high-priority wish*)
- Users should be able to select individual photos as their favourite and show them together (*high-priority wish*)
- It should be possible to display images on a map view (*low-priority wish*)

Photographs

- It should be possible to upload regular (JPEG) images into the application
- It should be possible to download the original images back from the system
- The application should extract basic metadata (date and time, camera settings) from the image
- The application should employ techniques to allow for quick loading (such as creating thumbnails and lazy loading)
- Besides pictures, the application should support videos and motion images (*low-priority wish*)

Users and sharing

- It should be possible for users to authenticate themselves with a username and password
- Users should only be able to create albums when given the permission
- Users should be able to give other users (precise) permissions to their albums
- It should be possible to give non-authenticated users permissions to an album using a link (*high-priority wish*)

Interface

To give the development some direction, I first created a basic mockup of the application's interface in Figma, the prototype of which is available here. The pictures below show the most important views.

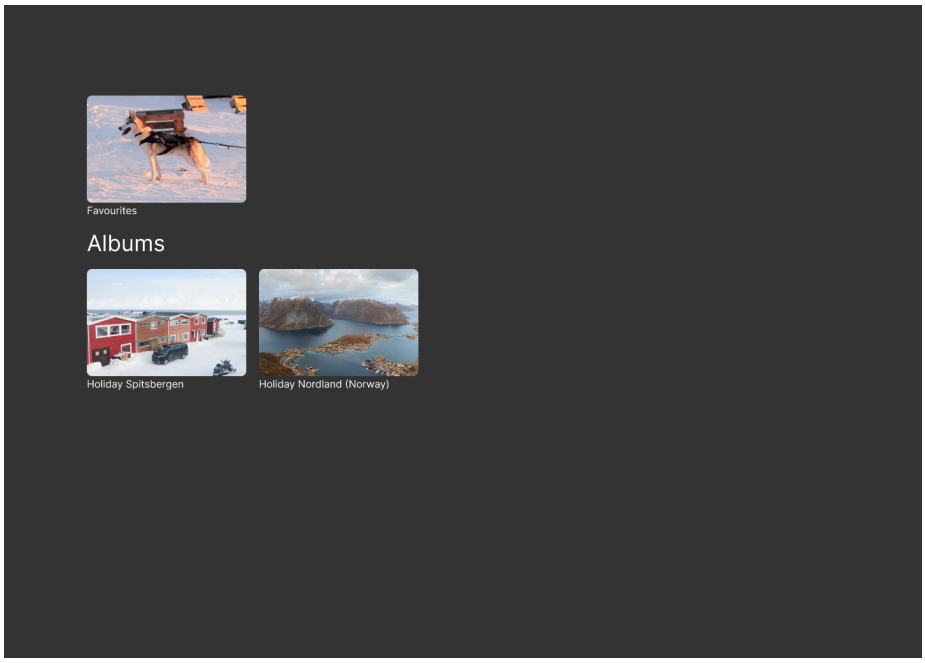

Figure: The home screen

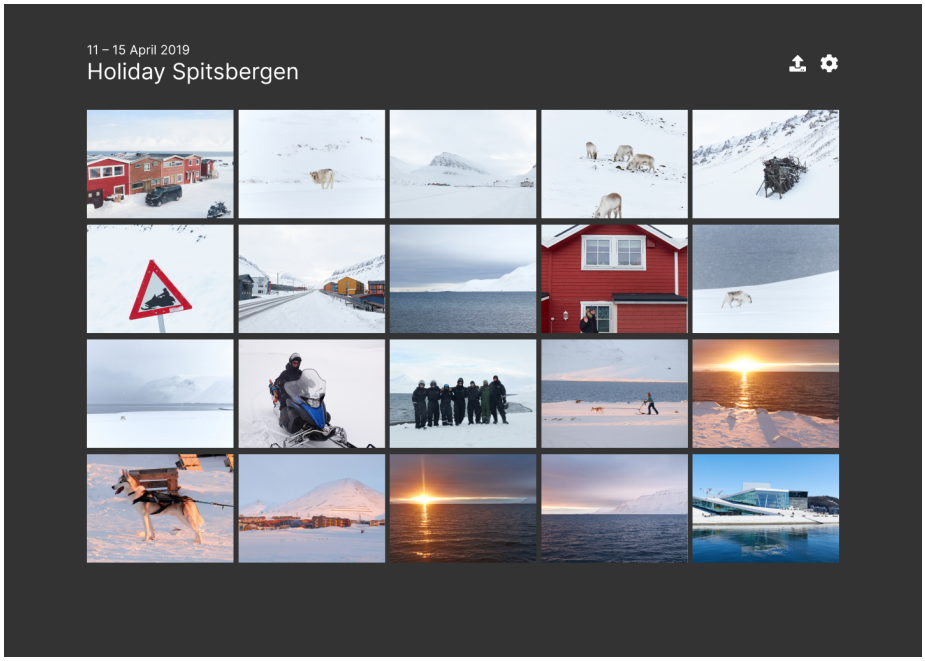

Figure: The album view

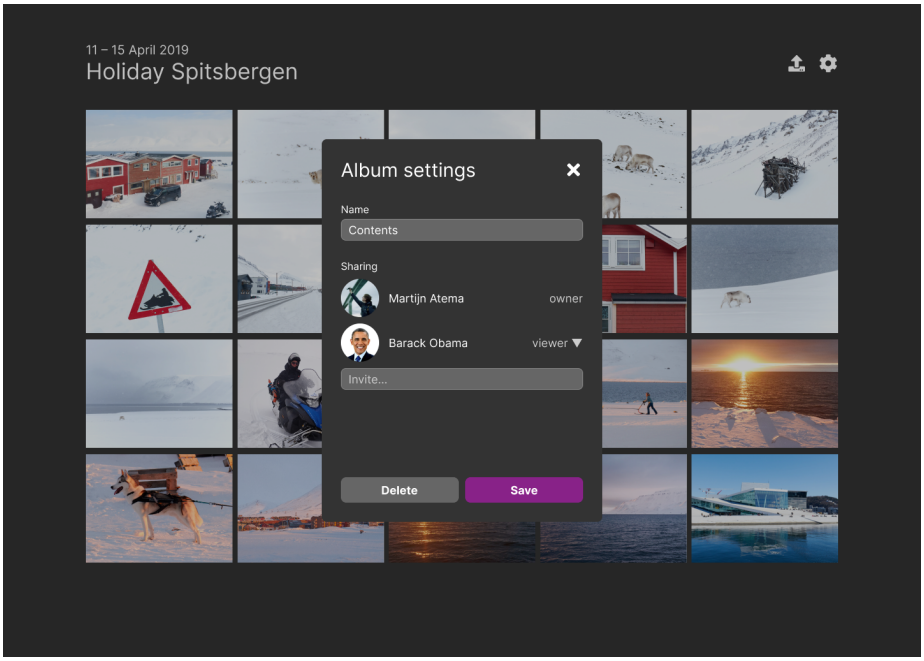

Figure: Changing an album's settings

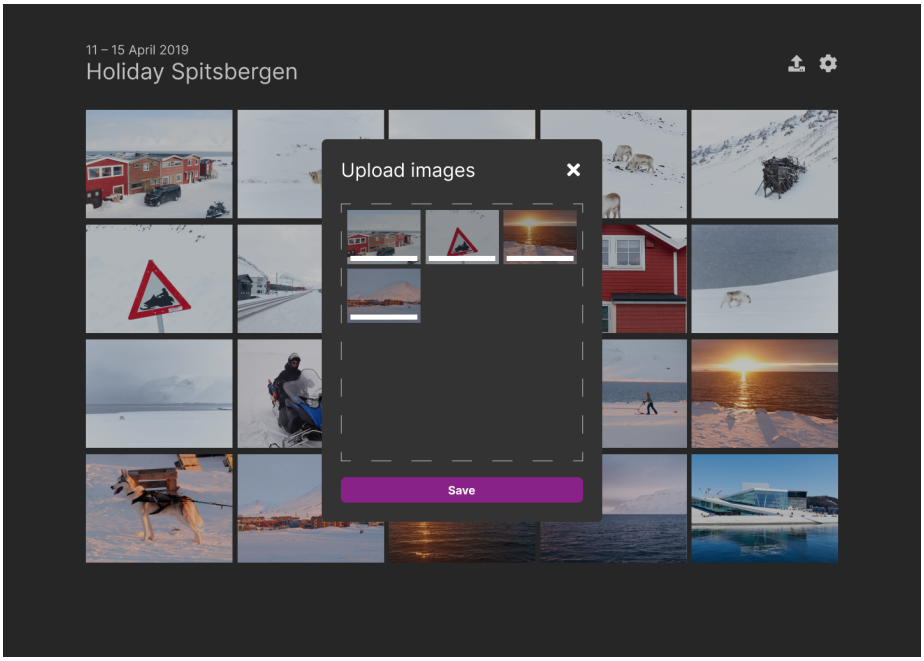

Figure: Uploading images to an album

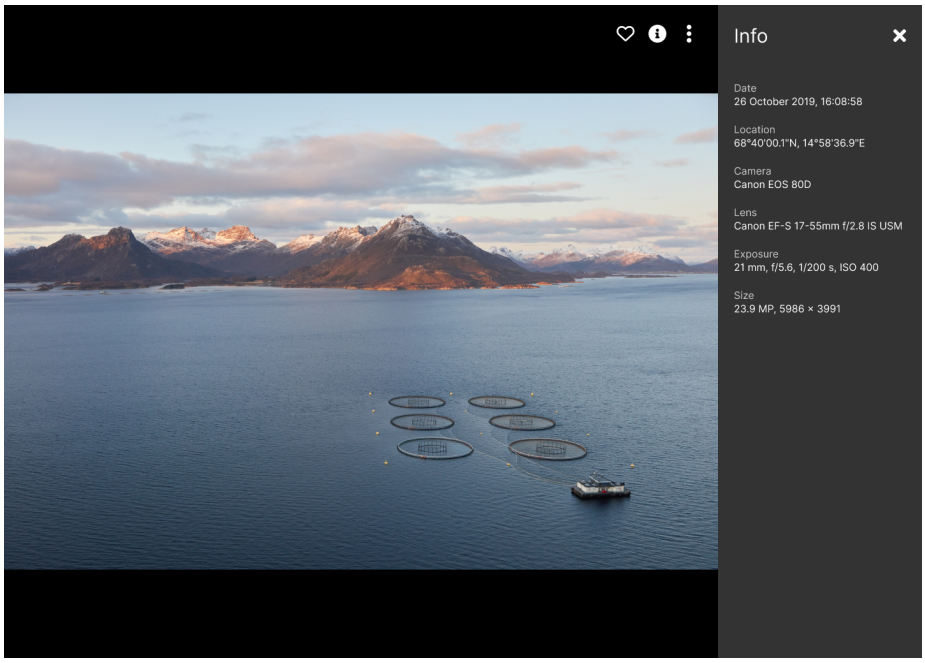

Figure: Viewing a photograph with metadata

Frameworks

At the core of a web application, there is custom-written code that provides the unique experience and binds the other components together. Some of it will run on the client (the browser) to facilitate interaction with the user. Other parts will run on the server, for example to process data and interact with other services, such as databases.

There are countless ways to shape this part of the application. For this project, I will focus on JavaScript meta-frameworks. These frameworks build on UI-frameworks (such as React or Vue) to provide additional functionality and solve some issues that come with using them on their own. Their full-stack nature means that front- and back-end code is more closely linked than in more traditional setups where the client and server are kept mostly separate.

Selection

To come to a list of frameworks for consideration, I combined selections from several recent sources[1][2][3]. From these, the most popular frameworks were chosen based on the popularity (number of stars) of the projects' repositories.

This lead to the following selection of frameworks:

- Next.js
- Gatsby
- Nuxt
- Meteor
- Astro
- Remix

Criteria

To come to an objective choice, I will compare the frameworks based on the criteria listed below. These are weighted on a scale from 1 (unimportant) to 5 (very important).

[1] D'Avanzo, Lewis [Omulabs] (23 August 2022). *What is a JavaScript Meta-framework?*
[2] Holmes, Ben [Prismic] (6 July 2022). *Unraveling the JavaScript Meta-framework Ecosystem*
[3] byby.dev (23 May 2023). *Top 11 JavaScript Fullstack Frameworks*

Target use

Weight: 5. Different frameworks target different niches of web applications and sites. The target use of the framework should be in line with my use of building a photo gallery.

Client-side interactivity

Weight: 5. As the photo gallery is an application rather than a website, smooth operation without having to reload pages is expected. The framework should be able to handle such interactivity on the client.

Data transmission

Weight: 5. The framework should assist in using data (taken from, for example, the database) across different parts of the application, both on the server and client.

Server-side rendering

Weight: 3. Frameworks can provide different types of rendering strategies (when the pages are created from the code). Server-side rendering can make sure the application is immediately available after loading the page, and allows for basic support of browsers without JavaScript.

TypeScript support

Weight: 3. TypeScript is a programming language that extends JavaScript with static typing. By adding type-checking, it can help prevent errors at runtime. Nowadays, it is supported by many libraries and tools in the JavaScript ecosystem, and I would very much like to use it for the photo gallery as well.

Local deployment

Weight: 2. An application written in the framework should ready to serve from one's own machine without much hastle.

Ease of use

Weight: 3. The framework should have a good documentation and not too steep learning curve, to be able to use it to build a completed application in limited time.

Community

Weight: 3. A larger community makes it easier to find answers to questions, both those previously answered and your own. In addition, large communities signify active products, which make the framework more future-proof.

Next.js

Next.js[4] is a popular React-based framework developed by Vercel, a cloud service company. After its first release in 2016, it has played a major role in shaping the category of JavaScript meta-frameworks. Seven years later, the project still gets regular major feature updates, and has now reached version 14.

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Target use *Score: 5.* Next.js has a broad target of web applications and websites.

Client-side interactivity *Score: 5.* Once the app is loaded, it behaves as a single-page application.

Data transmission *Score: 4*: Next.js has server actions which make it easy to both query and mutate server-side data from the client.

Server-side rendering *Score: 5*: Server-side rendering is a core feature of Next.js and enabled by default.

TypeScript support *Score: 5.* Next.js has built-in TypeScript support and makes it easy to create new projects with TypeScript using their own tool.

Local deployment *Score: 5.* A production build can be generated with a single command, which can then be started using the Next.js tool. With a bit of configuration, it is also possible to create a standalone build, which includes the necessary dependencies.

Ease of use *Score: 3.* The documentation clearly covers its functionality, but the many features and options that Next.js has added and changed over their 14 releases can become a bit confusing.

Community *Score: 5.* Next.js is the most popular framework of the six, and has a large community of developers that is active on GitHub, Discord, Reddit, and StackOverflow, where many questions have already been answered.

Gatsby

Gatsby[5] is a React-based framework that originated as a static site generator: a tool to create static websites based on documents and other resources. Since late 2021, with the release of version 4, the framework also started supporting rendering dynamic content. Recently, the project was acquired by Netlify, a cloud platform for web applications.

[4]Next.js. *Introduction*
[5]Gatsby. *Documentation*

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Target use *Score: 1.* The Gatsby Framework, true to its origins as a static site generator, heavly focuses on content-heavy sites, such as blogs and e-commerence where high performance is needed and interactivity is limited.

Client-side interactivity *Score: 5.* Once loaded, Gatsby provides client-side routing and rendering, providing an application-like experience.

Data transmission *Score: 2.* Gatsby has advanced GraphQL features for accessing data, but these are only available at build-time. Features for querying and modifying data at run-time are much more limited.

Server-side rendering *Score: 2.* A large part of Gatsby is based on rendering in advance. Server-side rendering was recently added to the framework, but it still very limited, and many dynamic pages still need to be rendered on the client.

TypeScript support *Score: 5.* Gatsby provides built-in TypeScript support and makes it easy to create new projects using it.

Local deployment *Score: 1.* It is easy to build the application with the Gatsby tool. However, serving pages with server code is difficult: one has to use the included tool, which is not intended for production (only testing), or rely on community plugins.

Ease of use *Score: 3.* With knowledge from React, it should be quick to get started. The documentation is generally clear, but past most attention to static site generation, while leaving much less room for interactive features.

Community *Score: 3.* Gatsby has a Discord server, as well as GitHub discussions page where questions can be asked. In addition, StackOverflow contains many questions, although it hasn't been as active recently, and many questions remain unanswered.

Nuxt

Nuxt[6] is a Vue.js-based framework inspired by Next.js (and released around the same time), and thus serving a similar purpose. It aims to provide a great developer experience in creating web applications and brands itself "The Intuitive Vue Framework".

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

[6]Nuxt. *Introduction*

Target use *Score: 5.* Nuxt has a broad focus on full-stack applications and also websites.

Client-side interactivity *Score: 5.* Once loaded, the application renders on the client and behaves as a single-page application.

Data transmission *Score: 3.* Functions are provided to aid in fetching data from the server and displaying it. It does, however, require a bit of manual work, especially when manipulating data.

Server-side rendering *Score: 5.* Server-side rendering is one of the core features of Nuxt and enabled by default.

TypeScript support *Score: 5.* Nuxt is written in TypeScript and provides rubust built-in support for types. TypeScript is the default language for new projects.

Local deployment *Score: 5.* With a single command, the Nuxt toolset can create a standalone bundle that can be used with Node.js to serve the application.

Ease of use *Score: 4.* There is a clear getting-started guide in the documentation that covers the important aspects of the framework. It builds heavily on the basics of Vue.

Community *Score: 4.* Although Nuxt is not the most popular option, it has a rather large community that is active on GitHub, Discord, and StackOverflow. These include many questions that have already been answered.

Meteor

Meteor[7] could be considered the first full-stack JavaScript meta-framework; it was released in early 2012 and has since been regularly updated with new features. Unlike most other meta-frameworks, Meteor can be used with any front-end framework, including React, Vue, Svelte and Angular, and the same code base can also be used for developing mobile and desktop applications. Meteor focuses on providing communication between the back- and front-end, including real-time features.

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Target use *Score: 5.* Meteor puts their focus on interactive web apps, specifically providing real-time features.

[7] Meteor. *Docs*

Client-side interactivity *Score: 4.* The client-side of Meteor applications behaves as a single-page application in the chosen framework. For features such as page routing, additional plugins have to be used.

Data transmission *Score: 5.* The ability to easily use (real-time) information from the (MongoDB) database in the application is one of Meteor's core features.

Server-side rendering *Score: 2.* By default, Meteor does not support server-side rendering. It can, however, be manually implemented using the chosen client-side framework.

TypeScript support *Score: 3.* Meteor provides a package for using TypeScript in your own code, and an aditional package that can provide tools for their core packages. However, a large part of the ecosystem is still JavaScript-only.

Local deployment *Score: 3.* Meteor's tooling can build an application written in the framework into standalone bundle. These bundles, however, are dependent on a specific, outdated version of Node.js, which makes it rather unflexible and difficult to use in environments that already contain newer versions of the runtime.

Ease of use *Score: 2.* Meteor has its own package ecosystem, which means things tend to work a bit different from normal. The documentation is expansive, but because many features are put into packages, things can be difficult to find.

Community *Score: 3.* Meteor has an official forum where answers to many questions can be found, in addition to those on StackOverflow. With the declining popularity of the framework, however, the community has become less active.

Astro

Astro[8] is a relatively new framework that aims to minimise the amount of JavaScript sent to the client. It introduced a frontend architecture where websites are largely rendered on the server, but contain so-called Islands of interactivity. For these Islands, Astro can work with a variety of frameworks, including React and Vue, or lightweight alternatives like Preact and Alpine.js.

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Target use *Score: 2.* Astro advertises itself as a framework for content-driven websites, such as blogs and marketing. It focuses on fast loading and search-engine optimalisation.

[8] Astro. *Why Astro?*

Client-side interactivity *Score: 3.* By default, applications work similar to classic websites, with page navigation requiring a reload. Recently, however, Astro has gotten opt-in support for client-side routing, which makes it feel more like an application. Rendering mostly remains server-side, however.

Data transmission *Score: 2.* There is support for fetching data from API routes, but integration with the interface has to be done manually.

Server-side rendering *Score: 5.* Astro focuses heavily on keeping most rendering away from the client.

TypeScript support *Score: 5.* Astro has built-in TypeScript support. Creating a new application using their wizard also makes it possible to use TypeScript.

Local deployment *Score: 4.* An official adapter plugin needs to be installed and configured to be able to build server-rendered sites, after which the application can quickly be deployed.

Ease of use *Score: 5.* Generally keeps to the basics of HTML, JavaScript and the chosen UI framework, which makes it easy to get started. Astro also has clear documentation.

Community *Score: 2.* Astro has a Discord server with quite a large active community. Finding answers on StackOverflow is difficult, as the number of questions is small and many remain unanswered.

Remix

Remix[9] is a full-stack framework built on top of a popular React routing framework. It places an emphasis on the user interface; it's router allows developers to create nested interfaces, each with their own layer of interactivity. Additionally, it leverages server-side rendering with support for traditional web standards, to allow it to even work in browsers without JavaScript.

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Target use *Score: 5.* Remix broadly targets interactive websites and applications.

Client-side interactivity *Score: 5.* Once the initial load is complete, a Remix application behaves as a single-page application and renders client-side.

Data transmission *Score: 4.* Remix makes it easy to use loaders to put information into the interface, and actions to update data from the interface.

[9]Remix. *Introduction, Technical Explanation*

Server-side rendering *Score: 5.* Remix focuses heavily on web fundamentals and is built around server-side rendering.

TypeScript support *Score: 5.* Remix is largely written in TypeScript itself and provides excellent built-in support. New projects created with their tool use TypeScript by default.

Local deployment *Score: 4.* It is easy to create a production build of the application, which can then be used with Remix's basic app server or another Node.js HTTP server using an adapter.

Ease of use *Score: 4.* Because Remix puts an emphasis on web standards, understanding of those can get one quite far. There are still concepts that you need to get used to, such as the nested routes, but the documentation is clear and helpful.

Community *Score: 3.* Remix has a Discord server with quite a large active community, as well as a GitHub discussions page where questions can be answered. Answers on StackOverflow are more difficult to find: there are not many questions, and many remain unanswered.

Conclusion

The table below shows an overview of how well the frameworks scored for each of the criteria. Totals are calculated by taking the sum of the scores multiplied by the respective weight factor.

Criterion	Weight	Next.js	Gatsby	Nuxt	Meteor	Astro	Remix
Target use	5	5	1	5	5	2	5
Client-side interactivitiy	5	5	5	5	4	3	5
Data transmission	5	4	2	3	5	2	4
Server-side rendering	3	5	2	5	2	5	5
TypeScript support	3	5	5	5	3	5	5
Local deployment	2	5	1	5	3	4	5
Ease of use	3	3	3	4	2	5	4
Community	3	5	3	4	3	2	3
Total		**134**	**81**	**129**	**106**	**94**	**131**

Based on these results, Next.js comes out on top. However, due to the small margins, Nuxt and Remix should also be considered as serious options for continuing development with.

An important difference between these three options is the client framework they are based on: Next.js and Remix use React, while Nuxt uses Vue. Of these two,

React remains the most popular[10]. In addition, I have some limited, but good experience with the framework, which makes me prefer Next.js and Remix over Nuxt.

Although Remix puts a large emphasis on developer experience and being easy to use, the popularity and larger community make me choose Next.js as the framework to develop the Photo Gallery with. Besides making it easier to find solutions to any problems I might encounter, the widespread adoption of Next.js ultimately makes it a more useful skill to have.

[10]Stack Overflow (2023). *Stack Overflow Developer Survey 2023*

Databases

Most applications need to store and access structured information, which is typically organised into a database. These databases are generally managed by a piece of software, the database management system (DBMS), through which the application can interact with the database[11].

Database systems exist in many shapes in forms. For the past decades, relational databases --- which can be queried and modified by using the structured query language (SQL) --- have been the dominant type[12]. In this type of DBMS, data is organised into tables with rows and columns[13]. Each row in the table contains a data record, which each column being a property of it. Because every data record has a unique identifier, relations can be established between records in different tables.

Recently, however, other types of databases have emerged and become more popular. These are broadly categorised as NoSQL databases, as they use other languages for accessing the data. With these database systems, data is stored differently to relational databases[14]. For example, some store data as schemaless documents with keys and values, some as graphs with nodes and edges, and some only associate keys and values.

Selection

To come to a selection of databases, I took into consideration the most popular database technologies according to the Stack Overflow Dev Survey[15] and the DB-Engines[16] ranking. These were further filtered down based on the following criteria:

- The technology must be a server that can be self-hosted.
- The technology must be broadly applicable (instead of focussing on specific types of data, such as text search, time series, or spatial data)
- The technology must be available under an open-source license.

This lead to the following final selection of databases:

[11] Oracle (27 September 2021). *What Is a Database?*
[12] Oracle (27 September 2021). *What Is a Database?*
[13] Oracle (18 June 2021). *What is a Relational Database (RDBMS)*
[14] MongoDB. *What is NoSQL?*
[15] Stack Overflow (2023). *Stack Overflow Developer Survey 2023*
[16] DB-Engines (January 2024). *DB-Engines Ranking*

- PostgreSQL
- MySQL
- MongoDB
- Redis
- MariaDB
- Cassandra

Criteria

To come to an objective choice, I will compare the database systems based on the criteria listed below.

Data integrity

Weight: 5. The database should make sure that data integrity is maintained by compling with ACID (Atomicity, Consistency, Isolation, Durability) principles. In addition, the data model should aid in keeping data items consistent.

Handling necessary data

Weight: 5. Although the data requirements of the target use of our application are not massive, the database system should still be able to comfortably handle items for hundreds of users, thousands of albums, and millions of pictures.

JavaScript support

Weight: 5. To be able to use the database from the application, it should have supporting libraries available for JavaScript. Preferrably, these should also have typing support by using TypeScript.

Performance

Weight: 3. The database system should be able to handle queries quickly, even under higher loads.

Query support

Weight: 3. Advanced query support, such as combining data from multiple sources, can avoid having to do multiple

Scalability

Weight: 2. Being able to scale the database can allow our application to grow to larger sizes by storing more information and being able to handle more requests.

PostgreSQL

PostgreSQL[17] is a popular object-relational database management system that is known for its powerful feature set and extensibility. The database makes it

[17] PostgreSQL. *About*

possible to add custom data types, procedures (even in different programming languages) and has a large ecosystem of extensions. In addition, conforms closely to the SQL standard, moreso than other relational databases.

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Data integrity *Score: 5.* Data integrity is an important feature of PostgreSQL. It is fully ACID-compliant, meaning that it has robust transactions. In additions, relations between tables can ensure that related data items remain consistent.

Handling necessary data *Score: 5.* PostgreSQL should have no trouble handling millions of rows in a table.

JavaScript support *Score: 4.* There are many libraries[18] available that support PostgreSQL, with different levels of abstraction and functionality.

Performance *Score: 4.* PostgreSQL performs very well in benchmarks[19][20], both for simple and more complex operations.

Query support *Score: 5.* There is great support for SQL, including advanced features. Joining data is a core feature of relational databases like PostgreSQL.

Scalability *Score: 2.* PostgreSQL is mainly built to be scored vertically, by increasing the machine's resources. Additionally, it supports read replicas to increase read capacity. Third-party extensions and compatible software could further expand these features.

MySQL

MySQL[21] is a popular choice of database management system that is developed by Oracle. As the name suggests, it is a relational database system, that uses SQL for managing data. Originally, MySQL was created to be fast, and can handle demanding loads. It is known for being easy to use, and a popular choice for beginning developers[22].

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

[18] npm. *Home*
[19] OnGres (July 2019). *Performance Benchmark PostgreSQL / MongoDB*
[20] Cal Mitchel [Albatross] (13 February 2023). *MariaDB vs PostgreSQL Performance Comparison*
[21] MySQL (2024). *What is MySQL?*
[22] Stack Overflow (2023). *Stack Overflow Developer Survey 2023*

Data integrity *Score: 5.* MySQL is ACID-compliant by providing robust transaction support. Relations between table columns further ensure that data items in the database remain consistent.

Handling necessary data *Score: 5.* MySQL should not have any problems in handling tables of several million rows.

JavaScript support *Score: 4.* There are many third-party libraries[23] availables for MySQL, each with their own feature set.

Performance *Score: 3.* Benchmarks[24][25][26] show that MySQL gets worse performance than its relational alternatives MariaDB and PostgreSQL.

Query support *Score: 4.* MySQL's supports SQL, but does not support some of the more advanced features that PostgreSQL supports.

Scalability *Score: 2.* To scale MySQL, the machine resources have to be increased, or read replicas can be set up to improve the request capacity of the system for read-heavy applications. Otherwise, one has to turn to other applications in the MySQL ecosystem, which can further expand these capabilities.

MongoDB

MongoDB[27] is a document-oriented database server, a type of NoSQL database that stores information as collections of documents. These documents use a format similar to JSON, and don't (necessarily) have to adhere to a schema, which makes MongoDB suitable for unstructured data, and makes it easier to make changes to the data structure. In part for these reasons, document stores (with MongoDB at the forefront) have grown massively in popularity over the last decenium.

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Data integrity *Score: 4.* Recent versions of MongoDB are ACID-compliant by allowing multi-document transactions. Although the database has all features necessary to ensure consistent data, MongoDB depends on denormalisation of data to perform optimally. As this results in duplicate data, extra care has to be taken by the application to ensure these stay in sync.

[23] npm. *Home*
[24] Jesus Castello [Stackshare] (2023). *PostgreSQL vs MySQL vs MariaDB - Help me Decide*
[25] Cal Mitchel [Albatross] (15 February 2023). *Aurora vs MariaDB vs MySQL performance and cost comparison*
[26] Jesus Castello [Stackshare] (2023). *PostgreSQL vs MySQL vs MariaDB - Help me Decide*
[27] MongoDB. *What is MongoDB?*

Handling necessary data *Score: 5.* MongoDB shouldn't have problems in managing a collection with millions of documents.

JavaScript support *Score: 5.* The JavaScript-like query language and document format of MongoDB integrate with JavaScript very well. There is an official driver available, as well as support from many third-party libraries[28] that offer additional functionality.

Performance *Score: 3.* When used properly with denormalised data, MongoDB should be very performant. However, benchmarks[29] show that it does generally lag behind PostgreSQL.

Query support *Score: 4.* MongoDB uses a powerful JavaScript-based query language that includes advanced pipelines for aggregating data. However, as MongoDB is not intended to be relational, these are not intended to be used very regularly.

Scalability *Score: 4.* Sharding is supported to distribute data between machines and increase the database's capacity. However, shards have to be defined manually by carefully picking sharding keys to ensure an even distribution.

Redis

Redis[30] is an in-memory key-value data store. It is widely applied in situations where low latency is key, such as caches, but can also be used as a document database or message broker. By just using keys for accessing data, and holding the entire store in memory, Redis can offer constant-time lookups and changes. Although memory is the primary data source, options for persistence are provided, which offer varying amounts of data protection.

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Data integrity *Score: 2.* Persistency is not neccessarily ensured due to the memory-nature of the store. Transactions in Redis offer limited guarantees for data integrity. In addition, there is no support for relations, meaning that consistency between items has to be managed by the application.

Handling necessary data *Score: 2.* As Redis is an in-memory store, the amount of data is constrained by the memory size, which tends to be much smaller than the disk size. Depending on the amount of data stored per picture, it would be able to handle the amounts, but the machine would need plenty of memory and would not be able to expand much beyond that.

[28] npm. *Home*
[29] OnGres (July 2019). *Performance Benchmark PostgreSQL / MongoDB*
[30] Redis. *Introduction to Redis*

JavaScript support *Score: 3.* There are a few third-party clients[31] available that support Redis' feature-set (including some addons). Other libraries generally have a heavy focus on caching and session storage.

Performance *Score: 5.* Since all data is stored in memory and Redis only supports basic access patterns, the store can be extremely quick.

Query support *Score: 2.* As a key-value store, Redis prioritises simplicity and speed. It has features for getting and settings values using their key, but does not support searching, relationships, or separating items into collections. There are, however, addons that do provide additional functionality close to what regular databases would offer.

Scalability *Score: 3.* Data can be kept in a cluster, in which it is automatically split between multiple instances of Redis. However, applications do need to connect to all nodes individually, and overall capacity remains limited due to the database being stored in memory.

MariaDB

MariaDB[32] is a relational database that originated as a fork of MySQL (by its original creators) due to worries after Oracle took over the product. Originally, MariaDB aimed for drop-in compatability with MySQL. Even though that's no longer the case, it retains very good compatability --- software developed for MySQL generally works well with MariaDB and vice versa.

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Data integrity *Score: 5.* MariaDB provides strong transaction support and is ACID-compliant. Relations can be defined between tables to ensure that data in different tables is consistent.

Handling necessary data *Score: 5.* There should be no problems in handling millions of items.

JavaScript support *Score: 4.* There is an official MariaDB connector available. In addition, it is supported by many third-party libraries[33] with different features (in part due to its compatability with MySQL).

[31] npm. *Home*
[32] MariaDB. *MariaDB in brief*
[33] npm. *Home*

Performance *Score: 3.* MariaDB can handle large numbers of requests. Benchmarks[34][35][36] show that it does lag behind PostgrSQL in speed, although it performs better than its direct competitor MySQL.

Query support *Score 4.* Although MariaDB works with SQL, it misses some of the more advanced features that PostgreSQL supports.

Scalability *Score: 2.* On its own, MariaDB supports read replicas to increase the request capacity of the database. Within the MariaDB platform, there are additional products that can add further scaling support.

Cassandra

Cassandra[37] is a NoSQL database developed by Apache that focuses on being distributed. Although it is possible to run it as a single instance, one should run it as a cluster to benefit from its advantages. These clusters runs in a masterless manner, where all nodes are equal, and appear as one database system to the client. Data is automatically distributed between the nodes for optimal performance. Behind the scenes, Cassandra is a wide-column store[38], an architecture that somewhat resembles traditional (relational) databases. In a wide-column table, however, not all rows have to use the same set of columns.

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Data integrity *Score: 2.* Cassandra is not fully ACID-compliant, but does offer transactions with basic guarantees. As Cassandra is non-relational, there are no references between tables, and consistency between them has to be managed by the application itself.

Handling necessary data *Score: 5.* Cassandra is built for large datasets and should not have problems handling millions of items, even on a single node.

JavaScript support *Score: 3.* There's a complete driver library[39] available. However, support from other libraries with additional features is very limited.

Performance *Score: 3.* Benchmarks[40] show that Cassandra scores better than MongoDB on some metrics (write operations), and worse on others (read operations).

[34] Jesus Castello [Stackshare] (2023). *PostgreSQL vs MySQL vs MariaDB - Help me Decide*
[35] Cal Mitchel [Albatross] (15 February 2023). *Aurora vs MariaDB vs MySQL performance and cost comparison*
[36] Jesus Castello [Stackshare] (2023). *PostgreSQL vs MySQL vs MariaDB - Help me Decide*
[37] Apache Cassandra. *Cassandra Basics*
[38] Apache Cassandra. *Architecture Overview*
[39] npm. *Home*
[40] [Unknown bib ref: benchmark-cassandramongo]

Query support *Score 3.* The Cassandra Query Language (CQL) shares similarities with SQL. However, it is not a relational database, and does not support combining data from multiple tables (joins).

Scalability *Score: 5.* Being scalable is Cassandra's main selling point. Data distribution between nodes happens automatically and is invisible to the application. It is easy to add nodes to improve the capacity of the system.

Conclusion

The table below shows an overview of how well the database systems scored for each of the criteria. Totals are calculated by taking the sum of the scores multiplied by the respective weight factor.

Criterion	Weight	PostgreSQL	MySQL	MongoDB	Redis	MariaDB	Cassandra
Data integrity	5	5	5	4	2	5	2
Handling necessary data	5	5	5	5	2	5	5
JavaScript support	5	4	4	5	3	4	3
Performance	3	4	3	3	5	3	3
Query support	3	5	4	4	2	4	3
Scalability	2	2	2	4	3	2	5
Total		**101**	**95**	**99**	**62**	**95**	**78**

Four of the six databases end very close together at the top of these results. This includes the three relational databases (PostgreSQL, MySQL and MariaDB), which are similar in their architecture, but the NoSQL database MongoDB also scores high.

The choice between a relational database and MongoDB comes down to a trade-off between data robustness and scalability. MongoDB is generally recommended for specific purposes, such as fast prototyping, unstructured data and massive scales[41]. On the other hand, it can be challenging to design data relations and join different data types, which makes relational databases more suitable in most scenarios: "If you don't know what database to choose, then choose a relational database (MySQL or PostgreSQL)"[42]. Because the data for the Photo Gallery is naturally relational --- items like photos, albums and users are all linked --- for me, these qualities outweigh the importance of scaling in this project.

The differences between PostgreSQL, MySQL and MariaDB comes more down to the details in advanced features and implementation[43], of which the impact to the application should in most cases be limited. For the Photo Gallery, I will

[41] Charcles Wang [Fivetran] (14 June 2021). *When to use NoSQL and MongoDB*

[42] Yerem Khalatyan [InConcept Labs] (9 August 2022). *When you should NOT use MongoDB?*

[43] Hussain Nasser (6 February 2023). *Postgres vs MySQL*

continue with PostgreSQL, as it scores a bit better on performance and features, and because it fits better with my current set-up.

Even though I will be developing the application with support for just one database now, due to the similar architecture, it could be possible to add support for additional (relational) database systems in the future. In fact, this is not uncommon amongst other self-hosted applications, such as NextCloud[44].

[44]Nextcloud (2024). *System requirements*

Data Storage

Data that is stored in databases is structured and usually smaller in size. However, there may also be larger data items that need to be stored somewhere. For the photo gallery, large amounts of photographs need to be stored. With each of them being a few megabytes, having many albums worth can quickly lead to terabytes of data. When videos are added to the mix, these numbers rise even quicker.

Selection

I will focus on three ways to store such data:

- As files on a filesystem
- As objects in a repository
- In the general database

Each of these methods can be provided by several software systems, but from the perspective of the application, the actual implementation is largely abstracted away.

Criteria

To come to an objective choice, I will compare the data storage options based on the criteria listed below.

Total size

Weight: 5. The storage system should be able to handle millions of items and the resulting amounts of data, which could add up several terabytes worth, or more when videos are included.

Item size

Weight: 3. Some photographs, for example panorama's, may be rather large in size, not to mention video files. Therefore, the system should support storing files that are several gigabytes in size.

Performance

Weight: 3. The system should provide good performance in both adding files to the system and retrieving them from the system.

Accessing data

Weight: 3. The most common operation will be retrieving items from storage for the user. The service should make it easy to serve its contents.

JavaScript support

Weight: 5. The storage service should be accessible from the application and should therefore have interfaces or libraries for JavaScript (and, preferrably, TypeScript).

Scalability

Weight: 2. A data store that is able to scale can allow our application to grow to larger sizes.

Portability

Weight: 1. Being able to access the data from a different machine, or multiple machines at the same time, can be useful when the application has to transfer between servers.

File Storage

The best-known method for storing data must be files. It's the way that we use data on the devices that we use daily, such as personal computers and mobile devices. In a file-based storage system, pieces of information as stored as files, which are hierarchically arranged within (nested) directories on a filesystem[45].

File access can be implemented in different ways[46]. There are several filesystems (such as Ext4, NTFS and ZFS) that can be used to format local drives or block volumes stored on a network. In addition, there are network file protocols (such as NFS, SMB and GlusterFS) that allow for accessing files on another server (or a cluster of servers) over the network. In the end, however, most of these differences are abstracted away by the operating system, and files accross them can be accessed using the same interface from the application.

Overview

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

[45]Red Hat (1 February 2018). *File storage, block storage, or object storage?*
[46]Franc, Océane [Scaleway] (2020). *Understanding the Different Types of Storage*

Total size *Score: 4.* For most physical setups, a few terabytes of data can be easily provided by harddrives. In virtual (hosted) situations, it may be more difficult to expand to such sizes.

Item size *Score: 5.* Modern filesystems support sizes far beyond the size of images and videos.

Performance *Score: 5.* Depending on the underlying filesystem, file access can be very performant. Even in networked file situations, the data tends to be close to the server, which makes access quicker.

Accessing data *Score: 3.* Direct access to the files is not possible, but files can easily be served directly by the back-end of the application.

JavaScript support *Score: 5.* Node.js has native methods for managing files and directories.

Scalability *Score: 3.* Although it is possible to add several drives to a system to increase capacity, this cannot be done repeatedly. There exist distributed filesystems, however, which can mitigate this issue.

Portability *Score: 3.* Depending on the exact type of network storage, data could by used by several machines at the same time, or moved to another machine on the same network. Beyond that, transferring data is necessary, which should not be difficult.

Object Storage

With object storage, instead of organising data as files into folders, each piece of data is treated as an object with a unique identifier[47]. This object contains the data itself, as well as metadata that describes the data. Although a file system can be emulated with object storage, the system is in essence a flat repository, where all files are stored in one level.

These repositories are accessed over a HTTP interface. Although there are multiple providers with their own API, Amazon S3[48] has become the biggest provider, and many alternatives offer a compatible API (for example, Google[49], Wasabi[50], and Backblaze[51]). In addition, there are compatible open source self-hosted alternatives, such as MinIO[52]. For that reason, I will focus on S3-compatible services for this comparison.

[47] Red Hat (1 February 2018). *File storage, block storage, or object storage?*
[48] Amazon Web Services. *Amazon S3*
[49] Google Cloud. *Cloud Storage*
[50] Wasabi Technologies. *Wasabi Hot Cloud Storage*
[51] Backblaze. *B2 Cloud Storage*
[52] MinIO Inc. *MinIO Object Storage*

Overview

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Total size *Score: 5.* Object storage is designed to support large amounts of data.

Item size *Score: 5.* Objects up to several terabytes in size are supported, which is more than enough for our purupose.

Performance *Score: 3.* All operations with an object storage system happen over the HTTP API, which results in some overhead.

Accessing data *Score: 5.* Because object storage is HTTP-based and supports access control, users can be authorised to download files directly from the storage system.

JavaScript support *Score: 4.* There is a complete S3 SDK[53] available, which is developed by Amazon, as well as some third-party libraries.

Scalability *Score: 5.* Object stores are intended to be scaled to near-infinite sizes by adding additional nodes.

Portability *Score: 5.* Because object stores are accessed over a public API, they can easily be used by another server, even if it is on a different network (although this can result in slower performance). Migrating data between services is also generally well-supported.

Database

In our stack, there's already a component for storing data: the database. As PostgreSQL[54] also has support for storing binary data types with arbitrary data, it may seem logical to also use it for storing binary files. After all, that results in one less component to worry about.

Overview

Comparison

Each of the requirements is scored on a scale from 1 (very poor) to 5 (very good).

Total size *Score: 3.* Although the database should be able to store terabytes worth of data, disk and database limits can cause issues beyond that.

[53] npm. *Home*
[54] PostgreSQL Wiki (5 May 2021). *BinaryFilesInDB*

Item size *Score: 4.* Standard binary data types in PostgreSQL have size limits of 1GB, which may not be enough in some cases. However, PostgreSQL also has a Large Objects feature, which can automatically chunk data and support larger files by storing them in a separate system.

Performance *Score: 1.* Having to read and write larger amounts of binary data to and from the database can result in a much larger strain on its connection, which can also affect performance of normal queries.

Using the database for binary data results in a much larger strain on the database system, which can also affect performance of normal queries.

Accessing data *Score: 2.* Through PostgreSQL clients and libraries, the data can be retrieved as a stream, and then served to the clients.

JavaScript support *Score: 2.* Although there are JavaScript libraries[55] available for PostgreSQL, support for large objects is limited to a single, unmaintained library.

Scalability *Score: 1.* PostgreSQL is limitid in its capabilities for scaling. In addition, large objects are stored in a system table and therefore not replicated.

Portability *Score: 2.* The database server can be connected to by another server. Transferring the data over to another database can be a difficult and manual process.

Conclusion

The table below shows an overview of how well the storage solutions scored for each of the criteria. Totals are calculated by taking the sum of the scores multiplied by the respective weight factor.

Criterion	Weight	File	Object	Database
Total size	5	4	5	3
Item size	3	5	5	4
Performance	3	5	3	1
Accessing data	3	3	5	2
JavaScripts support	5	5	4	2
Scalability	2	4	5	1
Portability	1	3	5	2
Total		**95**	**99**	**50**

[55] npm. *Home*

File and object storage are both strong contenders. Object storage is more cloud-focussed, but at the same time allows access from across the internet. File storage is easier to start with locally, as every server has some kind of drive that files can be stored on. Although the infinite scalability of object systems is an attractive feature, local file storage should in many cases be enough for personal uses. Ultimately, this makes files storage more in line with the self-hosted nature of the Photo Gallery, which means I'll be using it to develop the application.

Development

Having chosen Next.js, PostgreSQL and file storages as the technology stack for the project, the development progress could commence. The following sections document this process with its most important choices, the results in relation to defined requirements and a reflection on the chosen technologies before taking a look at what future changes could make the product even better.

The code written as part of the development process is available publicly and openly through the GitHub repository at https://github.com/Atema/ImagiFolio.

Process

Setting up & Project Structure

Next.js makes bootstrapping a new project very easy using their `create-next-app` utility. Generally, I used the default settings that the utility suggested. Importantly, this included using TypeScript, Tailwind CSS for styling, and the app router. The latter, which replaces the original Next.js pages router, enables React's newest features, such as server components.

Although Next.js doesn't prescribe a project structure, their default examples put everything into the `app` directory that is used for defining page routes. As I prefer to keep things a bit more separated, I opted for enabling the `src` directory during bootstrapping. Within this directory, I created several folders for different elements:

- `actions`: server-side actions that are called from the client to manipulate things on the server.
- `app`: used by the app router to define pages and layouts.
- `components`: re-usable React components that can be included in pages
- `db`: methods for accessing data in the database
- `files`: methods for accessing and manipulating files on the file system
- `utils`: utility functions for use in other methods

Where useful, I created sub-directories to categorise files.

Styling and Components

As it was the default option and a framework I'd been interested in for a while, I used Tailwind CSS[56] for styling the application. This library provides utility classes that can be used to control the CSS styling of an element. An upside of this method is that the styles are directly in the code, which makes it very easy to make changes and apply new styles to an element. On the other hand, having to use many classes can also make the code feel bloated, and makes it more difficult to re-use styles.

Luckily, React is built around components, functions that contain layout, styling and code, and which can be re-used as often as one likes. To keep the code organised, I created components for often-used design elements, such as buttons and input fields, but also for larger items, such as photo lists and different types of dialogs.

Besides creating my own, as a shortcut, I used some off-the-shelf components for more complex elements, such as drop-down menus and the upload box. For these, I chose libraries (mostly Radix Primitives[57]) that did not provide any styling themselves, so that their design could be kept in line with the rest of the interface.

Database

There is a plethora of libraries available for connecting to a database, ranging from bare-bones to having a lot of features. For this project, I used Prisma[58], a library belonging to the latter category. Prisma is an ORM that allows the developer to define the data model in a schema file, and then generates types and methods for accessing the data. In addition, it can apply the schema to the database itself and handle data migrations to ensure the data in the database has the same format.

These functionalities make it very easy to interact with the data and make changes to the schema without having to worry much about updating the database. There are, however, also downsides: Prisma is heavier and has more overhead compared to leaner options, leading to a bit lower performance[59]. Originally, I considered swapping it out for a more barebones option when the database model stabilised, but due to time constraints, and as I have not had any problems with Prisma, this is off the table.

File Handling

The application needs to be able to handle lots of image files. To give the end-user the ability to decide where their images are stored, I made the image directories (per type of image) configurable. Within these image directories, images are named by their unique identifier, which ensures there are no collisions and they are easy to look up without having to find a file name. The downside

[56] Tailwind CSS. *Get started with Tailwind CSS*

[57] Radix Primitives. *Introduction*

[58] Prisma. *What is Prisma ORM?*

[59] Drizzle Team (2023). *Postgres benchmarks between Drizzle ORM and other popular ORM libraries*

is that the image files are not recognisable by their name; however, that is not an intended use-case anyway.

Next.js provides a component for optimising images on-the-go (when requesting them) or during build time. However, its options did not entirely fit the use-case of the photo gallery. Instead, I decided to generate thumbnails and previews of the images during the upload process, which ensures they are always available when the image is in the database. The conversion library, sharp[60], is the same one used internally by Next.js, and fast enough that the time added to the uploading process is generally hardly noticable. I did use the image-component for its other features, such as lazy-loading the images.

Data Fetching and Mutation

Server components are an essential part of the Next.js app router. These components are rendered on the server, and therefore have access to backend resources such as the database and filesystem. I used this method throughout the application to render dynamic data. Where interactivity is needed, the data can be passed seamlessly to client components that are hydrated on the client to allow their code to run.

To allow the user to perform actions on the backend, such as server mutations, Next.js provides server actions, which are also an integral part of the app router. These can be invoked from forms and client components, after which they run on the server and return their result. Due to their simplicity, these are generally what I used for any data mutations, although I had to rely on a more traditional POST request for uploading new files. On the server side, I used Zod[61] for validating incoming data from the client and provide meaningful error messages.

Results

Now that the development process has finished, it is time to look back at the requirements and see to what extent they have been accomplished. This section re-lists all requirements, together with a short description of how well they have been achieved.

Technical

- The application must be written using a JavaScript meta-framework

  Fulfilled. The application was written using Next.Js

- The application should be locally deployable, without depending on proprietary technologies

  Fulfilled. The application can be built and deployed using the instructions provided in the repository and is only dependent on open-source libraries.

- The application should be available in a public repository under an open source license

[60] sharp. *High performance Node.js image processing*
[61] Zod. *Documentation*

Fulfilled. The application is available on GitHub under the MIT license, a recognised open source license.

- The application's source should be well-documented

  Fulfilled. The source code is annotated with JSDoc comments to describe functionality. Documentation files in the repository describe usage of the application.

- The application should be written in TypeScript (*high-priority wish*)

  Fulfilled. The entire application is type-annotated with TypeScript, and Prisma was used to make database operations type-safe.

- The application should be horizontally scalable to support very large image repositories (*low-priority wish*)

  Unfulfilled. The app server itself itself is stateless, which means multiple instances can be spun op to deal with extra requests. However, all instances need access to the same database and iamge store, which are both difficult to scale.

- The application should have an automated build and testing process (*low-priority wish*)

  Unfulfilled. Next.js has no documentation on how to publish a pre-built version. Adding tests and automating the builds proved too much work.

Interface

- The application interface should be responsive

  Fulfilled. The interface shows data differently depending on the viewport size (whether it is viewed from a computer, phone, or something in between).

- The application should display correctly in major modern browsers

  Fulfilled. The interface works as intended on recent versions of Chromium and Firefox (on Windows and Android). More testing would be needed to verify these results across platforms, but that is beyond the scope of this project.

- The application should have a light and dark mode

  Fulfilled. The interface has both light and dark modes that switch depending on the system settings. However, there is no possibility to switch from the user-interface via a setting.

- The interface should display in the user's language (supporting English and Dutch) (*low-priority wish*)

  Unfulfilled. Was not attempted due to time limitations.

Organisation

- Images should be organised into an album, which are viewable as one

Fulfilled. Albums are the main organisational unit, and all images are part of a single album.

- Images in an album should be sorted by their date

  Fulfilled. They are sorted by the date they are taken.

- It should be possible to move images between albums

Unfulfilled. Due to time constraints, I have unfortunately not yet been able to implement this feature.

- It should be possible to delete images from the system

  Fulfilled. It is possible to delete single images or entire albums.

- Moving and deletion should be possible for multiple images at a time (*high-priority wish*)

  Unfulfilled. This was not attempted due to time constraints.

- Users should be able to select individual photos as their favourite and show them together (*high-priority wish*)

  Unfulfilled. This was not attempted due to time constraints.

- It should be possible to display images on a map view (*low-priority wish*)

  Unfulfilled. Was not attempted due to time constraints.

Photographs

- It should be possible to upload regular (JPEG) images into the application

  Fulfilled. The application supports uploading of images in several common formats, including JPEG and HVIF.

- It should be possible to download the original images back from the system

  Fulfilled. The system stores the originally uploaded picture and allows the user to download it back with the click of a button.

- The application should extract basic metadata (date and time, camera settings) from the image

  Fulfilled. During uploading, the date, location, camera and settings are taken from the image metadata and stored. They are shown in the photo-viewing interface.

- The application should employ techniques to allow for quick loading (such as creating thumbnails and lazy loading)

  Fulfilled. As part of the upload process, the app creates thumbnails and previews to show in the interface. In addition, it employs lazy loading, such that only images on screen are loaded.

- Besides pictures, the application should support videos and motion images (*low-priority wish*)

  Unfulfilled. Was not attempted due to time constraints.

Users and sharing

- It should be possible for users to authenticate themselves with a username and password

  Fulfilled. It is possible to create accounts and login with an email address and password.

- Users should only be able to create albums when given the permission

  Fulfilled. There are separate permissions for admin, editor and viewer accounts. Only the first two can create albums.

- Users should be able to give other users (precise) permissions to their albums

  Unfulfilled. Due to time constraints, I have unfortunately not yet been able to implement this feature.

- It should be possible to give non-authenticated users permissions to an album using a link (*high-priority wish*)

  Unfulfilled. In line with the previous point, this feature has also not been implemented yet.

Conclusion

Over the span of several weeks, I have been able to fulfill many of the requirements of the product. However, it is also clear that I greatly underestimated the time necessary to implement these features. This left no time to implement most of the wishes, and even lead me to abandon two of the set requirements. Nevertheless, I believe that the application as it stands is a solid product and base for future improvements, starting with the unfulfilled requirements and wishes.

Future Changes

Although I'm very happy with the results of this project and the product I was able to produce, the photo gallery is in many ways still incomplete and open for improvements. Of course, this includes unaccomplished wishes, but also other features that were beyond the scope of this project, or came up during development. This section presents a list of items that could be used be used to further improve the software.

- **Sharing**. Being able to share albums is a core feature of the app that hasn't been realised yet, and is therefore first on the list of improvements. Besides sharing albums with other users, more fine-grained and public sharing could further improve the application.
- **Better organisation capabilities**. Being able to download, remove or move multiple images at once would be a large improvement of having to take every action one by one. In addition, this could include photos being part of multiple albums, and additional features such as user favourites.
- **Do more with metadata**. At the moment, metadata is stored and shown next to the pictures. The usefulness of these could be improved by

displaying them more playfully (for example, in map views) and by using them more broadly, for example for organising pictures.

There are also some improvements that are less visible (to most or even all users), but are still important:

- **Accessibility**. The current version was not built with accessibility in mind. Additional effort should be put in to ensure everbody is able to use the product as intended, even when using tools like screenreaders.
- **Action status and error handling**. Forms and other interactive elements do their job and show error messages when they happen. However, clearer indicators of when something is happening in the background, transitions and more granular errors (perhaps already before sending data to the server) could greatly improve the user experience.
- **Code optimisations**. Some additional optimalisations, especially in relation to caching and database queries, could improve performance and put less strain on servers, and ensure up-to-date information is shown to the user.
- **Data fetching**. Server components render the data and send it to the client all at once. This works alright, but starts gettings slower for larger albums, as a lot of data needs to be transferred. Streaming the data, or only requesting it when scrolling down, could improve the experience.

When time permits in the future, I hope to come back to ImagiFolio to implement these features and slowly evolve it into a mature piece of software.

Technology Reflection

Previously, Next.js, PostgreSQL and file storage were chosen as an "optimal" technology stack. This section looks back on the development process to see to what extent they performed as expected.

Framework

Overall, I've been quite happy with Next.js. With my basic knowledge of React and Next.js's solid documentation, it wasn't difficult to get started with building the application. Their file-based routing system makes it intuitive to create new page routes, and data flow between the server and client is mostly seamless.

Some things, however, turned out more difficult than expected. During development, I also had trouble with implementating some features, such as page transitions and keeping state after navigating pages. With Next.js's large community – my main reason for choosing the framework – I expected it to be easy to find solutions to such problems. However, due to the recent (May 2023) switch to the app router, most answers on sites like StackOverflow are outdated, and there's not always solutions available for the newer versions.

Database

During this project, I have not stumbled upon any limitations of PostgreSQL. The relational data structure makes it easy to model different data items and their connections, and provides error messages when trying to do something stupid (deleting an item that still has references elsewhere).

It must be said that with the library I used, the experience would probably have been nearly identical when using the other relational options (MySQL/MariaDB). Prisma even supports MongoDB, which could have been used in the same way, although the current data model is far from optimal for that database, requiring additional queries to achieve the same functionality.

Data Storage

As expected, native support by Node.js and all libraries make file storage a breeze to use: passing them a file path is often enough to make things work. In addition, serving files through a Next.js route handler makes it easy to perform the same permission checks as for regular pages.

The big downside of file storage is scalability. On my desktop this isn't a problem – there's enough disk space for thousands of pictures – but on my tiny webserver, additional storage comes at a high cost, and I ended up spinning up a temporary server with more space for showcasing the project. An object store that could be connected to regardless of where the app server is hosted would have been a nice alternative.

Reflection

When I first envisioned this Personal Pursuit, I put the focus entirely on learning about a single technology to develop a web application. Over time, it has grown to a much more complete and valueable project that included a structured design process, a thorough comparison of different technologies, and finally, a development stage.

Learning Objectives

At the start of the project, I set three learning goals that cover the different stages of the project:

1. *Conceive, design, and execute a goal-oriented learning experience at an academic level, inspired by a personal interest or passion.*

   With help of my supervisor, I was able to set up and follow a structured design process that allowed me to dive into the ins and outs of development using a web framework, while combining it with my passion of photography.

2. *Understand and explain the differences between different options of technologies, and be able to select appropriate technologies for a web application.*

   I have reached this goal by comparing the most popular technologies in three different categories: web frameworks, databases and data storage. Especially the comparison of frameworks – the field I was least familiar with – has been very useful in getting to understand the differences – and, not unimportantly, the similarities – between some of the most popular options.

   Although I performed a structured comparison of the different choices based on the requirements of the photo gallery, it wasn't always easy to find a single most suitable candidate, as some options were very close in their capabilities. Nevertheless, I was able to pick an appropriate set of technologies for the gallery, that turned out to work well during the development.

3. *Be able to apply and integrate a web framework with other technologies to develop web applications.*

   Using the selected technologies – Next.js as the framework, PostgreSQL

for the database, and files as a storage method – I was able to build a functional photo gallery that meets most of the stipulated requirements (and could be extended to meet the others in the future).

Starting from my basic knowledge about web development, this development process allowed me to expand my understanding of the involved technologies and how to integrate them, as well as some other tools that proved useful in the development. There is still a lot to learn when it comes to advanced features and additional libraries, but the skills I gained during this project are a great starting point for further growth.

Academic Development

Completing this Personal Pursuit and reaching these learning goals have contributed to me becoming a better engineer. Not unimportantly, by teaching me about new technologies and how to use them, as well as the field of web frameworks in general.

But above all, I learned about how to execute a design process that includes a thorough comparison of the needed technologies based on the requirements of the application. Determining what tools to use for projects is an essential skill for engineers, and this project enhanced my abilities to make such choices for all kinds of projects in the future.

Appendices

ID	Task Name	Work
0	**Personal Pursuit: Web Technologies**	**190 hrs**
1	**Proposal**	**24 hrs**
2	Initial version	20 hrs
3	Progress meeting	0 hrs
4	Final version	4 hrs
5	Deliver final proposal	0 hrs
6	**Technology research**	**72 hrs**
7	Set up information website	5 hrs
8	Define application and requirements	6 hrs
9	Select options	5 hrs
10	Analyse options	12 hrs
11	Set criteria for comparison	14 hrs
12	Progress meeting	0 hrs
13	Compare technologies	30 hrs
14	Complete technology analysis	0 hrs
15	**Design**	**14 hrs**
16	Study other photo galeries	4 hrs
17	Progress meeting	0 hrs
18	Design and wireframe interface	10 hrs
19	**Development**	**80 hrs**
20	Set up framework	8 hrs
21	Build interface	12 hrs
22	Develop database interaction	12 hrs
23	Develop data handling	12 hrs
24	Progress meeting	0 hrs
25	Combine parts	14 hrs
26	Finalise product	14 hrs
27	Complete the application	0 hrs
28	Document development	5 hrs
29	Reflect on choices	3 hrs
30	Complete documentation	0 hrs
31	Reflection meeting	0 hrs