FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Modern Web Development Technologies and Approaches
Student:	Bc. Petr Pondělík
Supervisor:	Ing. Adam Vesecký
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

The goal of the thesis is to explore and analyze the development of web applications with a particular focus on REST, GraphQL, and PWA. Furthermore, another output of the thesis will be a prototype of a web application, demonstrating the use of the outlined technologies.

Requirements for the thesis:
- explore the types of web applications and their characteristics
- explore the web development approaches and technologies in the following scopes: client-side, server-side, and PWA
- design and implement a prototype of a web application with PWA elements and demonstrate the use of the analyzed technologies
- the application will consist of a client-side and a server-side application, where the latter will expose an API
- implement the same API using REST and GraphQL, and compare them based on the criteria below
- evaluate the used technologies in terms of implementation, extendability, sustainability, and testing

The design part of the thesis will follow SI methodologies.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 13 December 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Modern Web Development Technologies and Approaches

Bc. Petr Pondělík

Department of Software Engineering
Supervisor: Ing. Adam Vesecký

May 5, 2022

Acknowledgements

First of all, I would like to express gratitude and thanks to Ing. Adam Vesecký for his advices, time and guidance. My appreciation goes also towards my family and my friends for all their support during the whole studies.

Declaration

In Prague on May 5, 2022 . . . .

Citation of this thesis

Pondělík, Petr. *Modern Web Development Technologies and Approaches.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022. Also available from: ⟨`https://github.com/petr-pondelik/modern-web-development`⟩.

Abstrakt

V současné době je správná volba technologií a přístupů v oblasti vývoje webových služeb a aplikací klíčem k úspěchu. Proto je důležité mít přehled o aktuálních technologiích a přístupech, a především jim porozumět. Správně volit technologie či udržovat přehled o aktuálních trendech jsou však v současném rychle se vyvíjejícím světě webových technologií obtížné úkoly.

Cílem této práce je provést čtenáře světem současného vývoje webových aplikací. Práce cílí na to, aby čtenáři nejprve poskytla obecný úvod do této oblasti a co nejpřirozenější cestou se zaměřila na témata REST, GraphQL a PWA. Tato témata pak popíše a vysvětlí podrobně.

Praktickým výstupem této práce bude implementace prototypu webové aplikace rozděleného do komponent na straně serveru a na straně klienta. Tento prototyp bude demonstrovat využití vhodných technologií a přístupů k vývoji REST API, GraphQL API a progresivních webových aplikací.

V závěru budou jednotlivé komponenty prototypu, a tedy také technologie využité pro jejich implementaci, zhodnoceny z hlediska kvality a obtížnosti implementace, rozšiřitelnosti, udržitelnosti a testovatelnosti.

Klíčová slova vývoj webových aplikací, technologie na straně serveru, technologie na straně klienta, REST, GraphQL, API, PWA, Prisma, Nest, React, JavaScript, Node.js

Abstract

Nowadays, the right choice of technologies and approaches is crucial for success in web development. In order to make the right choice, it is important to be aware of and understand the current technologies and approaches. However, neither of the aforementioned tasks are easy in the fast-paced world of the web technology.

This thesis aims to walk the reader through the world of current web development. The aim of this thesis is first to conceptualize the topic broadly and then move on to the topics of REST, GraphQL and Progressive Web Applications in the most natural way possible. These specific topics will be thoroughly described and explained.

The practical output of the thesis will be the implementation of a prototype web application divided into client-side and server-side components. This prototype will demonstrate the use of appropriate technologies and approaches for developing REST and GraphQL APIs as well as Progressive Web Applications.

Finally, the individual components of the prototype, and thus the technology used to implement them, will be evaluated in terms of implementation quality and difficulty, extendability, sustainability and testability.

Keywords web application development, server-side technology, client-side technology, REST, GraphQL, API, PWA, Prisma, Nest, React, JavaScript, Node.js

Contents

List of Figures

List of Tables

List of Code Snippets

Introduction

The popularity of online solutions has grown significantly in the last decade. Many people and companies have changed their approaches to solving a wide range of tasks and started using online solutions. The technical requirements for these solutions are constantly growing. This leads to the emergence of new approaches, technologies, architectures and frameworks that are suitable to meet the evolving requirements.

We can notice the trend of emerging new solutions, for example, from the results of *The State of JavaScript* [1] survey. The timelines show that in 2017, there were only four front-end and a single back-end JavaScript framework of which at least 10 % of survey participants knew about. We can see that after four years, the result set has grown to 13 back-end frameworks [2] and 10 front-end frameworks [3].

It's clear that in today's world of web development, technology and approaches are evolving rapidly. Experienced developers do not have the time to follow and understand all these technologies and approaches, while novice developers find it difficult to orient themselves in these technologies and approaches. This is a problem at a time when the right choice of technologies and approaches is the key to success.

The Goal of This Thesis

This thesis aims to study and analyze web application development in client-side and server-side domains. In particular, the development of REST and GraphQL APIs along with progressive web applications. Furthermore, the thesis aims to design and implement prototype web application demonstrating the use of appropriate technologies and approaches in the aforementioned domains. Moreover, it will explore the types of web applications from different perspectives. The output of this thesis will be beneficial to beginning web developers as an entry guide to the current web development, or to web developers interested in extending or consolidating their knowledge in client-side or server-side development. It can also be beneficial for developers looking for a demonstration of API implementation using REST and GraphQL.

Chapter 1

Theoretical Analysis

This chapter provides an introduction into web application development and a broad overview of the technologies and approaches used in web application development. Our goal is to cover the topic of web development broadly and to get to the topics such as REST, GraphQL and PWA in the most natural way possible. Firstly, we provide the basic information on web applications. After that, we explain a client-server model and how it is used by web applications. Subsequently, we describe types of web applications from several different perspectives. After that, we describe JavaScript and analyze the technologies and approaches used in client-side web development. Then we describe PWAs in detail. After that, we analyze the server-side web development. Finally, we analyze REST and GraphQL in detail.

1.1 Web Applications and Websites

When it comes to web application development, it is suitable to start by explaining the similarities and differences between web applications and websites. Both web applications and websites are essentially a software that is stored on a server and is delivered via a network such as the internet. Users access them through web browsers. Moreover, developers use the same technologies to build both websites and web applications. The key difference between websites and web applications is in their purpose and the level of interactivity. Although modern websites have elements of web applications, their main purpose is to provide users with information. This implies that the provided information limit the level of website's interactivity. On the other hand, the main function of a web application is to provide users with functionality that allows them to achieve some goals. [4]

Web applications operate in a client-server model. This means that they consist of a client-side component and at least one server-side component. Server-side component typically implements application logic that uses one or more databases to retrieve and persist data. The application logic may also communicate with other server-side components. [5] [6]

Because of the client-server model, web applications require both client-side and server-side scripting. Client-side scripting deals with the information presentation and the user interface. For client-side scripting, we use browser-based [1] technologies such as JavaScript, CSS and HTML. [5]

On the other hand, the purpose of server-side scripting differs from the purpose of client-side scripting. It mostly deals with selecting the content to be returned in response to a request. It usually involves validating requests or interacting with a database, as we have already mentioned above. [6]

Web application components communicate via standardized protocols such as HTTP. [6] The following section describes a client-server model and how it is used in a basic web application configuration.

1.2 Client-Server Model

As we mentioned in the previous section, a client-server model is essential for web applications. It is a distributed model that divides tasks between nodes that provide services, called servers, and nodes that request these services, called clients. The server handles all processing related to the provided service, while the client can access this service. [7]

Clients and servers communicate using a request-response messaging pattern. In other words, the client sends a request to the server and the server responds with a response. The server perpetually listens for requests from clients. When the server receives the request, it processes that request and sends the response back to the client. [7]

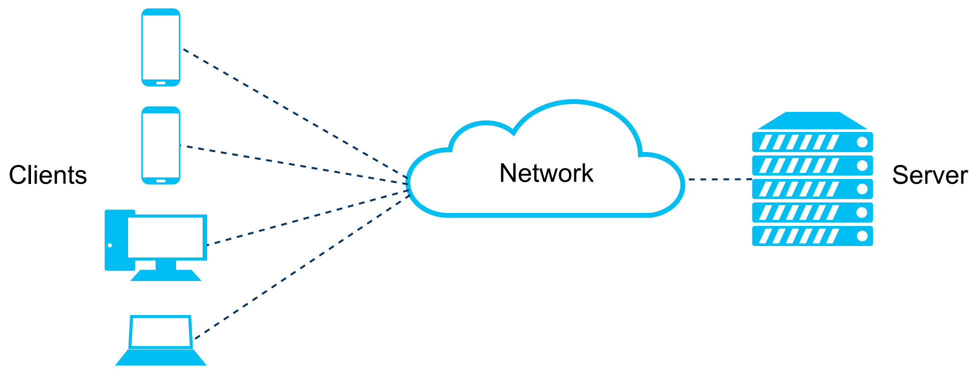


Figure 1.1: Client-Server Model

[1]The technologies that rely on a web browser to execute or interpret the code.

Now that we have described the client-server model in general, we will explain how typical web application works on top of the client-server model. Considering the situation where there is no intermediary between the client and the server, we can describe the request-response process in the following steps:

1. The *Web browser* sends an HTTP request to the specified URL.
2. The *Web Server* receives the request and delegates it to the web application.
3. The *Web Application* performs the appropriate operations on the data.
4. The *Web Application* dynamically prepares the data to be sent back as part of the response.
5. The *Web Application* passes the data along with the HTTP metadata to the *Web Server* which sends an HTTP response back to the *Web Browser*.
6. The *Web Browser* processes the response and, if necessary, sends requests to fetch static resources.
7. The *Web Server* loads requested files from the file system and sends them back in response.

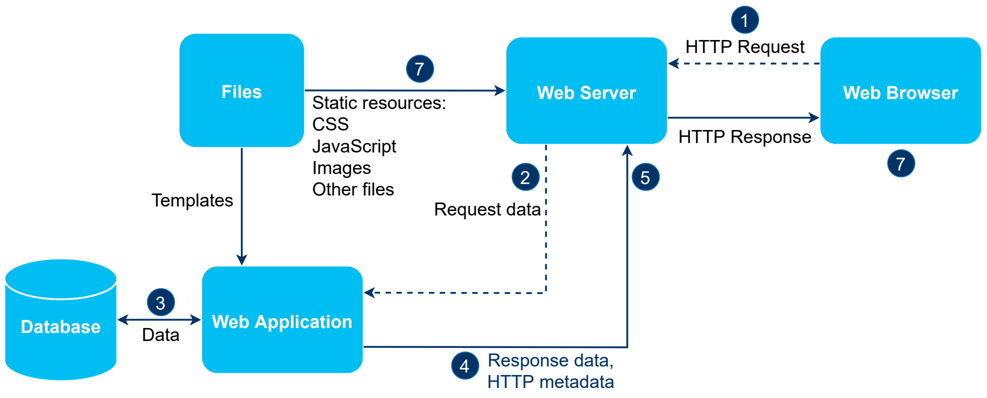


Figure 1.2: Client-Server Model for Web Application

1.3 Types of Web Applications

The preceding sections dealt with the definition of a web application and the explanation of how web applications work. This section analyzes the types of web applications from several different technical perspectives in order to provide a comprehensive overview of possible technical approaches in web application development.

Let us give a general overview before proceeding to the analysis of individual types of applications. Table 1.1 lists the perspectives we will examine and the corresponding application types. Figure 1.3 illustrates the relationships between the application types listed in this table.

Perspective	**Types**
Rendering	Server-Side Rendered (SSR) Client-Side Rendered (CSR) Hybrid Rendered (SSR & CSR)
Reloading	Multi-Page (MPA) Single-Page (SPA) Hybrids of MPA and SPA
Decoupling	Monolithic Decoupled (Headless) Microservices Oriented Micro Frontends Oriented
Adaption to Mobile Devices	Mobile-Dedicated Responsive Progressive

Table 1.1: Types of Web Applications Overview

1.3.1 From the Perspective of Rendering

Firstly we will categorize web applications according to where and how the HTML content is assembled. By assembling, we mean combining data with a template into the final content presented to the user. We recognize client-side and server-side rendered applications as well as a combination of both approaches. [8]

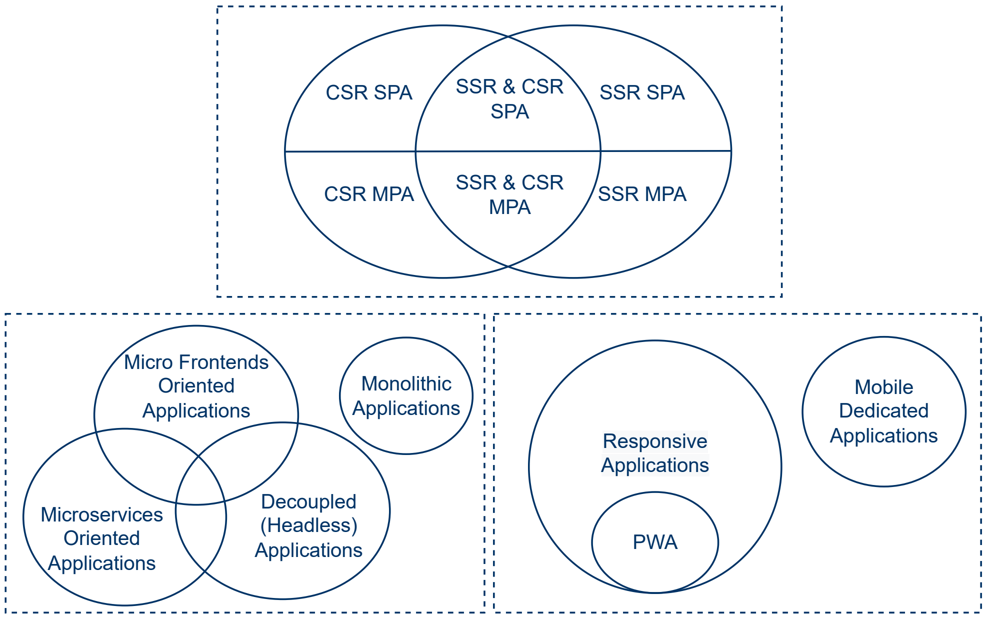

CSR SPA
SSR & CSR SPA
SSR SPA
CSR MPA
SSR & CSR MPA
SSR MPA
Micro Frontends Oriented Applications
Monolithic Applications
Microservices Oriented Applications
Decoupled (Headless) Applications
Responsive Applications
PWA
Mobile Dedicated Applications

Figure 1.3: Relations Between the Types of Web Applications

1.3.1.1 Server-Side Rendered Applications

Applications that utilize server-side rendering approach generate HTML content on each request. There are two main approaches to server-side rendering:

Template Engines Server-side technologies typically provide a way to write dynamic content into HTML. Template engines are libraries for these technologies that provide syntax for more conveniant templating. Examples of template engines are Twig, Handlebars, Pug or Jinja. [9]

Isomorphic/Universal JS This approach allows us to pre-render the default page of an application developed using a client-side JavaScript framework. This page is displayed independently of JavaScript. As soon as the JavaScript framework loads, the page becomes interactive. Examples are Next.js, Nuxt.js, Meteor.js or Angular Universal. [9]

1.3.1.2 Client-Side Rendered Applications

Client-side rendered web applications utilize asynchronous HTTP requests to retrieve data from servers and JavaScript to construct HTML in a web browser. Client-side rendering, also known as AJAX, enables us to modify the DOM without refreshing the entire page. Single-page applications, which we will describe in Section 1.3.2.2, usually take advantage of CSR. [10]

1.3.1.3 Hybrid Rendered Applications

These web applications use server-side pre-rendering. After receiving the pre-rendered content, further rendering is performed by the browser.

1.3.2 From the Perspective of Reloading

Looking at web applications from a different perspective, we can categorize them according to an architectural pattern that determines how the application handles transitions between its pages. From this point of view, we recognize multi-page and single-page web applications and web applications that combine both approaches.

1.3.2.1 Multi-Page Applications (MPA)

These applications generally use SSR, however they can use CSR to refresh parts of the content asynchronously. A typical MPA consists of multiple pages that the user navigates through. [11] [12]

Without utilizing a CSR approach, multi-page application sends a request to a server at every action and waits for a response. The response contains the entire new HTML page and the application reloads as soon as it receives the response. This can be inefficient. Let us imagine that we have a page with several assets [2] and a form for editing the user's profile. After submitting the form, the application requests a new HTML page and the entire page is transferred and reloaded just to display a status message. Moreover, without caching, a web browser reloads all required assets. [11] In MPA, both front-end and back-end usually depend on each other. [13]

1.3.2.2 Single-page Applications (SPA)

In contrast to MPAs, SPAs never reload or redirect. All the resources required by the application are retrieved the first time the application loads. SPAs perform each update of the page content dynamically. They communicate asynchronously with server-side applications in order to retrieve data for updates. [11] They generally feel faster than MPAs because they do not repeatedly load the entire content. They promote decoupling since front-end and back-end are strictly separated. [13] SPAs provide better user experience since they allow us to evoke a native look and feel. Because of that, this approach is suitable for Progressive Web Applications which we will analyze in Section 1.6 [12]

[2] e.g. images, CSS or JavaScript

1.3.2.3 Hybrids of MPA and SPA

We can combine both previous approaches. Such a combination can result, for example, in MPA that contains SPAs as sub-applications. [11]

1.3.3 From the Perspective of Decoupling

We can also categorize web applications according to how their logical parts are separated. This section will therefore focus on monolithic, decoupled, microservices oriented and micro frontends oriented applications. [14] [15]

1.3.3.1 Monolithic Web Applications

A typical web application consists of three layers. These are the presentation layer, the business logic layer and the data access layer. In applications with a monolithic architecture, all these layers are coupled within a single, complex codebase. Moreover, components of the logic layer which could be developed independently are tied together. [14]

Monolithic applications are not technologically flexible and make us bound to a single technological stack. They are not reliable since a failure in one functionality can cause the entire application to fail. As the application evolves, the volume of the code grows and its quality deteriorates. This reduces the speed of development. The only way to scale monolithic application is to run additional application instances and use a load balancer to distribute requests among these instances. Deployment is simple, but changing any part of the application requires to redeploy the entire application. [14]

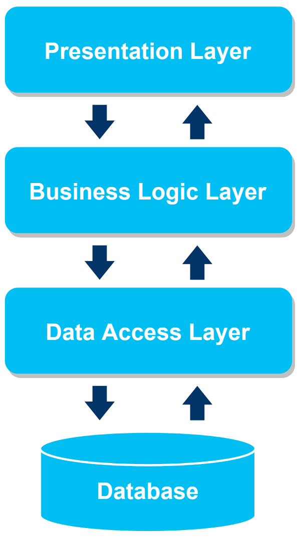


Figure 1.4: Monolithic Application

1.3.3.2 Decoupled (Headless) Web Applications

Monolithic applications tend to be difficult to maintain. Moreover, we can not clearly define the boundary between the front-end and the back-end. This means that front-end developers often need to understand how back-end solutions are implemented. [15]

Because of that, we introduce a decoupled architecture that strictly separates the presentation layer of the web application from the business logic by implementing separate applications for both layers. This approach is commonly known as the separation of front-end and back-end. Back-end and front-end applications are deployed on separate servers. The back-end application exposes an API that the front-end consumes. Such a back-end application is also called headless. This approach narrows the gap between front-end and client-side development, as well as between back-end and server-side development. [15]

It is important to note that the separation of front-end and back-end does not mean that the front-end and back-end applications are not monolithic. For decoupling within these applications, we use the architectures described in the following subsections.

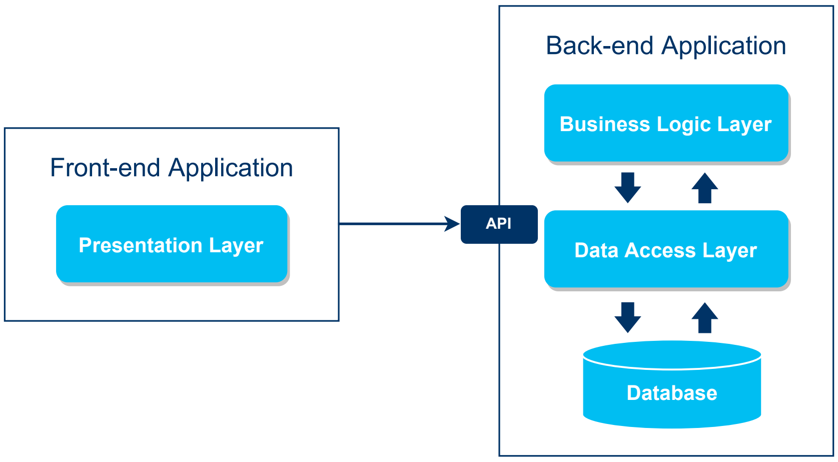


Figure 1.5: Decoupled Architecture

1.3.3.3 Microservices Oriented Web Applications

In contrast to monolithic applications, microservices oriented applications are divided into a system of small isolated services. Each of these services runs independently and exposes an API. The services then communicate with each other. We will describe microservices in more detail in Section 1.7.5.2. [14]

1.3.3.4 Micro Frontends Oriented Web Applications

Fundamentally, Micro Frontends is an architectural style that extends the idea of microservices to the front-end development. Micro Frontends use the DOM as the API that enables them to communicate. Since the Micro Frontends architecture closely relates to client-side development, we will describe it later in Section 1.5.2. [16]

1.3.4 From the Perspective of Adaption to Mobile Devices

According to the Nielsen Norman Group, we can essentially divide web applications into three categories based on how they handle adaption to mobile devices. We will expand these categories with progressive web applications, as they can be understood as an effort to behave as native applications on the target device. [17]

First of the categories is trivial. It includes web applications that fail to adapt to mobile devices. We describe other categories below. [17]

1.3.4.1 Mobile-Dedicated Web Applications

This approach provides a standalone version of the user interface that is developed specifically for mobile devices. The mobile version differs from the full one and is located under a different URL. It provides content and features suitable for mobile devices, which are usually limited compared to the full version. [17]

1.3.4.2 Responsive Web Applications

Responsive web applications represent an approach where a single implementation supports a set of distinct devices and screen sizes. In contrast with mobile-dedicated web applications, they should provide the same content and functionality for all devices.[3] They are easier to maintain because of the single codebase. However, any change requires testing on all supported devices. They may run slower due to loading content for all versions. [17]

1.3.4.3 Progressive Web Applications

Progressive web applications take web application adaptability further. They heavily use JavaScript to behave like native applications. We will describe these applications in detail in Section 1.6. [18]

[3]In practice, some responsive web applications limit content and functionality, but to a lesser extent than mobile-dedicated ones.

1.3.5 Section Summary

In this section, we explored the the types of web applications from several different perspectives. We analyzed basic characteristics of individual types of web applications and how they relate to other web application types. We learned that rendering architecture and reloading architecture are closely related and application types from these two perspectives strongly overlaps.

As we will learn in Section 1.5.2, JavaScript frameworks are very popular nowadays. One of the key benefits of JavaScript frameworks is the support for client-side rendering, which makes it easy to implement single-page applications. From the results of The State of JavaScript [19] survey, we may notice the rising usage of PWAs. In the next section, we will cover JavaScript as one of the building blocks of web development and as the key technology for the approaches mentioned above.

1.4 JavaScript

JavaScript is an essential part of web development since almost 98 % of all websites use it on client-side. [20] Moreover, according to Stack Overflow Developer Survey 2021 [21], Node.js has become the sixth most popular technology. It has even overcome web browsers as a JavaScript runtime in The State of JavaScript [22] survey.

JavaScript is a lightweight, cross-platform scripting or programming language. Furthermore, it can be characterized as a prototype-based, multi-paradigm, single-threaded and dynamic language. JavaScript supports object-oriented, imperative and functional programming. Depending on the implementation, JavaScript can be an interpreted or just-in-time compiled language. [23] [24]

JavaScript is capable of running in different environments, such as a web browser or Node.js. It can connect to objects in the host environment and provide the developer with control over them. [25] [24]

The core language of JavaScript is defined by ECMAScript standard which ensures that the JavaScript core behaves the same in all implementations. Among other aspects, it defines a language syntax, types, standard library of objects or error handling mechanism. [24] [26] In the context of web development, we have two extensions to the core JavaScript:

Client-side JavaScript which additionally contains objects to control a web browser and Document Object Model. [24]

Server-side JavaScript that extends the core specification by objects that provide functionality specific to server-side development. [24]

1.4.1 Implementations

The first implementation of JavaScript engine was created by Brendan Eich at Netscape. The engine was implemented using `C/C++` and named SpiderMonkey. Since its release, the engine has been gradually optimized and updated to support the current ECMAScript specification. SpiderMonkey is still used today as the JavaScript engine for Mozilla Firefox. [23]

Another important JavaScript engine is Google's V8. This engine is used by Google Chrome and the current version of Opera. It is also important to mention that Node.js is based on the V8 engine. Other popular JavaScript engines include SquirellFish and Nitro for WebKit browsers, such as Apple Safari. [23]

1.4.2 ES2015

ECMAScript 2015 is the largest update to JavaScript in its history and represents a turning point in its evolution. ES2015 brought two major changes. The first is that the JavaScript specification receives regular updates. The second brings extensive changes to the language features. [27]

The breakthrough feature of ES2015 are modules that allow us to develop flexible and modular code. Another important features are classes that add syntactic sugar to the prototype-based inheritance. We should also mention better support for asynchronous programming and the possibility to define variables with a scope limited to the block in which it is defined. [27]

1.4.3 TypeScript

In recent years, the popularity of TypeScript has grown significantly. [28]. Essentialy, TypeScript is a typed superset of JavaScript. This means that it shares the syntax of JavaScript and adds typing rules on top of it. TypeScript comes with a runtime for JavaScript that provides a compile-time type checking. The runtime preserves the behavior of JavaScript runtime so if we move code from JavaScript to TypeScript, it runs the same way. Once TypeScript checks the types validity, it removes the types and compiles the code into plain JavaScript. [29]

1.4.4 Asynchronous JavaScript

Before we will continue, it is important to note that asynchronous programming is different from parallel programming. As we mentioned at the beginning of this section, JavaScript is a single-threaded language. Asynchronous programming allows us to free the thread from waiting for the result of a blocking call. JavaScript supports both synchronous and asynchronous code execution. This is made possible by the ability to dispatch an operation outside the main thread and wait for the result. The main thread is not blocked and continues to run while the asynchronous operation is beeing processed. [30]

In asynchronous programming, we must ensure that the code which depends on the results of asynchronous operations is executes only after they are completed. There are three ways to do this in JavaScript. The first uses asynchronous callbacks and gradually becomes deprecated. The second uses promises that act as a proxy for the future result of an asynchronous operation. The last uses `async/await` constructs that provide a syntax sugar for promises. [31]

1.5 Client-Side Web Development

As we mentioned in Section 1.1, web applications operate in a client-server model. That implies that web application development consists both of client-side and server-side development. In this section, we will discuss client-side web development technologies and selected approaches. Finally, we will introduce some of the popular JavaScript frameworks.

1.5.1 Key Technologies

Client-side web development is based on three essential technologies, namely HTML, CSS and client-side JavaScript.

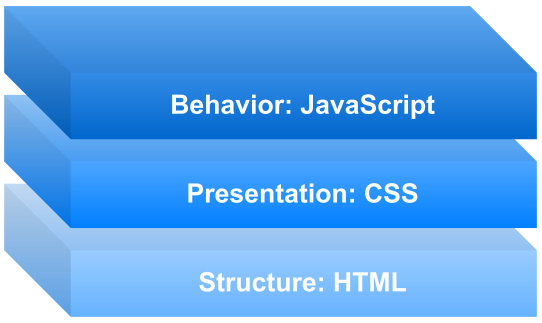


Figure 1.6: Client-Side Web Technologies Layers

1.5.1.1 HyperText Markup Language

HTML is the most basic layer of any client-side web application. It is a markup language that defines the structure of a web application using predefined elements. Most elements consist of an opening tag, a content and a closing tag. However, some elements do not require a closing tag. These elements are typically used to insert something into a document. We can categorize HTML elements as *inline* and *block*. A web browser renders the block element as a visible block of content that is separated from the preceding content by a new line. The inline element must not contain a block element. [32]

1.5.1.2 Cascading Style Sheets

Cascading Style Sheets stand for the rule-based language that specifies how HTML documents should be rendered on top of the browser's default styles. It allows us to graphically represent the document. CSS consists of a list of rules. Each rule specifies a group of styles that is applied to a set of elements. The target set of elements is specified by a selector. The group of styles is represented as a list of property-value pairs. [33]

Instead of pure CSS, we may use CSS preprocessors. CSS preprocessor is a tool that lets us to compile an extended version of CSS into a basic CSS syntax. There are several preprocessors, such as Sass, Less or Stylus. [34]

1.5.1.3 Client-Side JavaScript

JavaScript is an essential part of the client-side web development since almost 98 % of all websites use it. [20] As we mentioned in Section 1.4, it is the extension that provides APIs defined on top of the core JavaScript language. There are two different categories of these APIs, browser APIs and third-party APIs. *Browser APIs* are constructs provided directly by a web browser. Essentially, they are bridges between JavaScript and complex lower-level code. *Third-party APIs* are JavaScript codes available on the web. We must retrieve the API code in order to use it. [35]

Below we describe only the most important browser APIs. For a comprehensive overview, we can look at the reference.[4]

DOM API provides connection between JavaScript and a document that represents a website. It models the document as a logical tree. There is a node at the end of each branch that contains an object which represents an HTML element. We can access these objects and attach event handlers to them. [36]

Fetch API is a promise-based API that provides asynchronous fetching of resources. [37]

Web Storage API provides access to `sessionStorage` and `localStorage`. [38]

IndexedDB API is available within Web Workers and provides access to client-side storage for larger amounts of data. It enables efficient search through indexing. [39]

[4]https://developer.mozilla.org/en-US/docs/Web/API

Let us describe a theoretical model of the client-side JavaScript runtime, which neglects all optimizations implemented by current engines. The runtime model of the client-side JavaScript is based on stack, heap, message queue and event loop.

Heap contains allocated objects.

Stack is formed by function calls.

Message queue contains messages to be processed.

Message contains a function that is called when the message is processed.

Event loop processes the message queue, starting with the oldest message.

Message is added into the message queue anytime an event listener is triggered by an event. When the stack is empty[5], the event loop removes the oldest message from the queue and the contained function is called with the message as a parameter, causing the frame to be added on the top of the stack. The function call stack is processed until it empties. Then the event loop continues to the next message. [39]

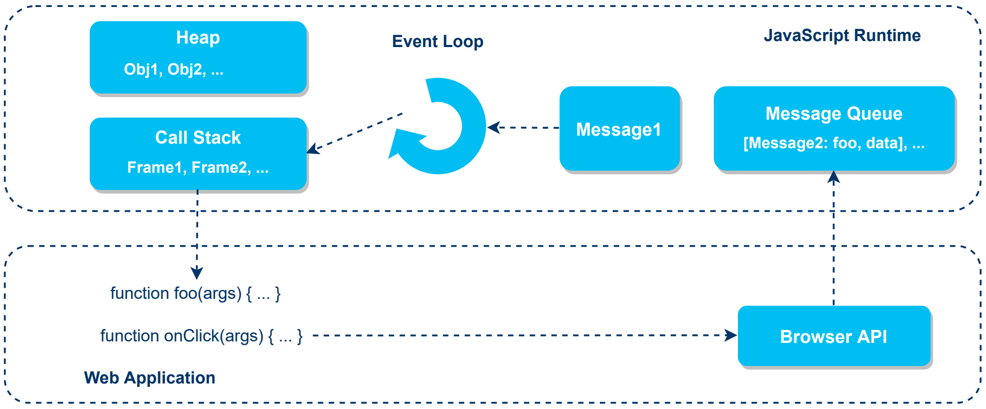


Figure 1.7: Client-Side JavaScript Runtime

1.5.2 Modern Development Approaches

There is a large number of trends in client-side web development. This thesis does not aim to analyze all existing trends. Sources [40] [41] [42] discussing web development trends include approaches such as mobile-first development, single-page applications, micro frontends or JavaScript frameworks. We will

[5]So the previous message was completely processed.

skip JavaScript frameworks and PWAs because we will cover them in detail in Section 1.5.3 and Section 1.6. Since we already discussed SPAs in section 1.3.2.2, we will omit them too.

1.5.2.1 Mobile-First Development

According to StatCounter Global Stats [43], more than 56 % of web traffic comes from mobile devices nowadays. This makes mobile optimization a must. Mobile-first development is based on creating a layout that is designed for mobile devices by default. This layout is gradually modified for wider viewports using layered configurations. [44]

1.5.2.2 Micro Frontends

The term Micro Frontends was first mentioned in ThoughtWorks Technology Radar in 2016. As we mentioned in Section 1.3.3.4, Micro Frontends is an architectural style that extends the idea of microservices to the front-end development. It thinks of a web application as a set of features that are developed by independent teams. Each team develops its own features end-to-end, from database to UI. [16]

There are several ideas behind Micro Frontends:

Be Technology Agnostic Each team selects and manages its technology stack independently of other teams. They may use custom elements as an interface and to encapsulate implementation details. [16]

Isolate Team Code Features developed by teams do not share runtime. They e.g. avoid global variables. [16]

Establish Team Prefixes Establish naming conventions where it is not possible to isolate. For example, namespaces in CSS or key prefixes in web storages. [16]

Favor Native Browser Features over Custom APIs Use browser events for communication instead of implementing your own publish-subscribe mechanism. [16]

Build a Resilient Site Use universal rendering and progressive enhancement to make the feature viable even if JavaScript has failed to run or has not run yet. [16]

Above we mentioned the concept of custom elements. Custom elements are a set of JavaScript APIs defined by the Web Components standard. Using these APIs, we can create custom elements that are useful for encapsulating features. Specifications of these elements serve as APIs for integration with other micro frontends. [16]

1.5.3 JavaScript Frameworks and Libraries

As we mentioned in Section 1.4.2, ES2015 brought modularity to JavaScript. This led to the emergence of many modern JavaScript frameworks which have become an essential element of modern client-side web development. They bring together tools and approaches that facilitate the development of scalable and interactive web applications. [45]

According to The 2021 State of JavaScript [3] survey, consistently the most popular JavaScript front-end framework is *React*, used by 80 % of respondents. On second place is *Angular* followed by *Vue.js*. The results show an increase in the popularity of the *Svelte* framework for which it shows very high satisfaction and interest, while usage and awareness are increasing. [3] We may see the usage ratio of the most popular frameworks in Figure 1.8.

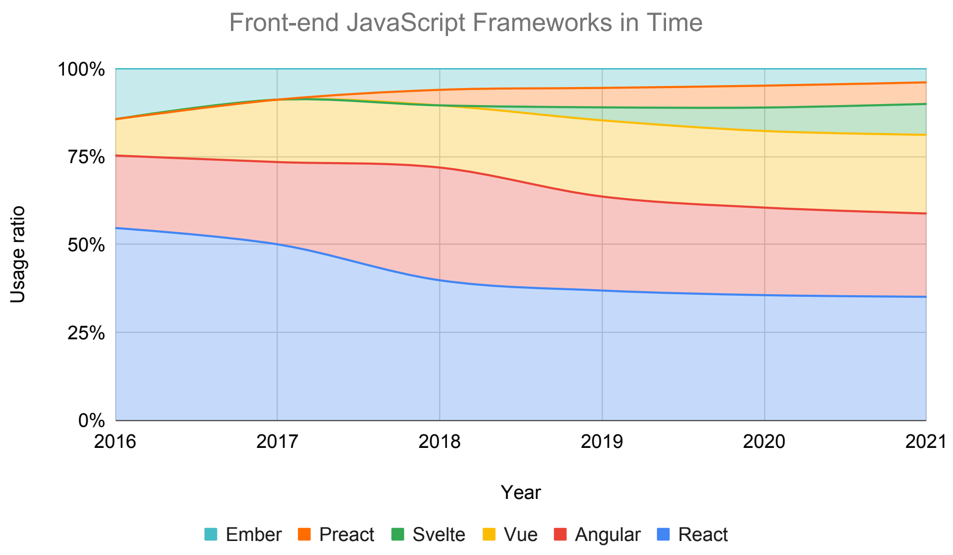


Figure 1.8: Front-end JavaScript Frameworks in Time

Below we will describe React and Vue.js based on the enormous popularity of Vue.js on GitHub [46].

1.5.3.1 React

React is a declarative, component-based JavaScript library for creating interactive UIs. The declarative nature of React is that we can declare the view of an application based on its state. React renders and updates the view of a component when its state changes. [47] It separates concerns at the level of isolated components instead of separating rendering from other UI logic, such as event handling or status updates. [48]

In React, we connect the data to the view using one-way data binding. With one-way data binding, we can use two different approaches. The first is *Component to View*, where any change to the data is reflected in the view. The second is *View to Component*, where any change in the view is reflected in the data. [49]

React interacts with HTML using a virtual DOM. [50] Moreover, it uses JSX for rendering. This technology allows us to use a declarative HTML-like code inside JavaScript to describe the rendering output. Since a web browser cannot interpret JSX, it must be compiled into calls of the specific method. This method creates objects called *React elements*. These objects describe what to render and React uses them to create and update the DOM. [51] [48]

React only provides components, DOM manipulation and component state management. Additional features such as application state management or routing are implemented externally. [50] On that basis, we should note that React itself is a library, not a framework, and is not limited to web application development[6]. Rendering for web applications is provided by *ReactDOM* library. Combination of React and ReactDOM is often considered a framework. [51]

1.5.3.2 Vue

Same as React, Vue is a declarative JavaScript framework for creating interactive UIs that implements component-based architecture. It also interacts with HTML using a virtual DOM. [52] [53]

In contrast to React, Vue supports both one-way and two-way data binding. In two-way data binding, the model automatically updates when there is a change in view, view than responds to this change in the model. [50] [54]

Essentially, we can define logic, template and styles of a component in separate files. However, the recommended way of defining a component is to use an HTML-like files called *single-file components*. [52] [55]

Vue is designed as a progressive framework. It facilitates incremental development of projects and allows us to migrate existing projects to Vue gradually. Unlike React, it allows us enhancing a static HTML without a building process. [50] [52]

[6]We can use e.g. React Native to create mobile application UI.

1.6 Progressive Web Applications

In recent years, a good-looking and easy-to-use user interface has become a must for any application to attract users and succeed. Users have become accustomed to certain standards that make them feel comfortable when using applications. As we mentioned in Section 1.5.2.1, more than 56 % of web traffic comes from mobile devices in these days. Moreover, an increasing number of mobile device users spend more than 90 % of their time using mobile applications instead of a web browser. [56] This has two implications. The first is that the web front-end must be optimized for mobile devices. Second, users expect a mobile web to behave in the same way as a native mobile application.

Some of the features of web applications differ from those of the native ones. Compared to native applications, web applications do not support any offline experience in default. PWAs attempt to narrow the gap between them by utilizing modern web capabilities to convey a native-like user experience. [18]

1.6.1 Principles of Progressive Web Applications

In general, it is not trivial to determine whether a web application is a PWA. This subsection outlines the key principles that a PWA should follow.

- be **discoverable** by search engines,
- be **installable** so that we can access a PWA from the home screen or application launcher,
- be **linkable** so that we can refer to a PWA using a URL,
- be **independent of the network** so that it can work offline,
- implement a **progressive enhancement** so that it supports older browsers and provides the best possible experience when accessed using state-of-the-art web browsers,
- support user **re-engagement** using approaches such as notifications,
- use CSS constructs to achieve responsivity,
- be delivered using HTTPS to ensure security. [18]

In order to achieve features mentioned above, we use a set of common web technologies and concepts. We will introduce these within the following section.

1.6.2 Technologies and Approaches

We develope PWAs using common web technologies. These include *service workers*, web application *manifest* and a set of Web APIs such as *Cache API*, *Push API* or *Notification API*. Furthermore, it is common to implement application shell architecture for PWA. [18] [57] We will introduce service workers, web application manifest and application shell within the following sections. Figure 1.9 illustrates the overall architecture of PWA.

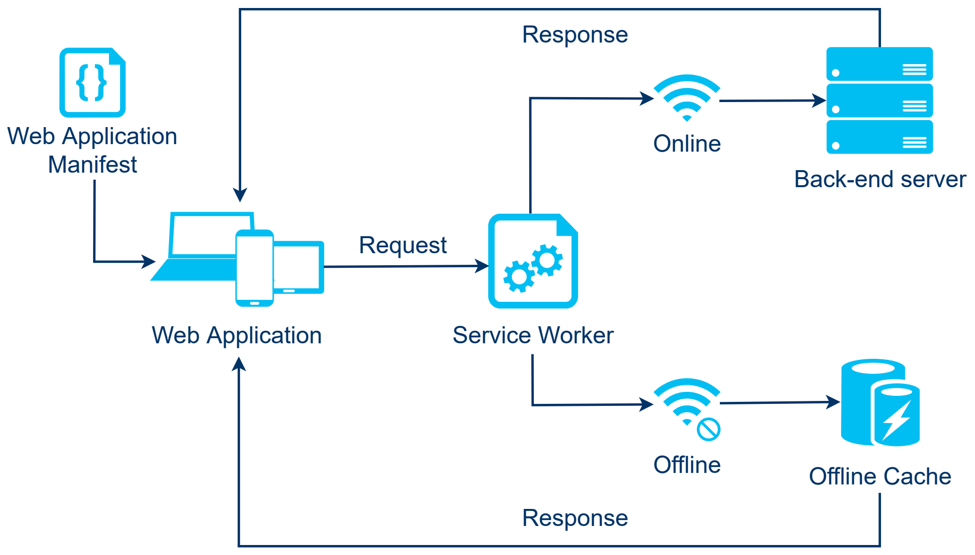


Figure 1.9: Overall PWA Architecture

1.6.2.1 Service Workers

A service worker is a script that runs in the background in a thread that is separate from the main JavaScript thread of a web application. It is able to improve the reliability and performance of the a application by providing offline access. [58] [57]

A service worker has the ability to capture and process network requests. Therefore, it can act as a network proxy between a client application and a server. On top of that, it has access to a variety of client-side APIs, such as *Cache API*, *Push API* or *Notification API*. [58] [57] As a motivation, let us describe what a service worker is able to provide:

- Because a service worker can act as a network proxy and is able to use caching and storage APIs, it allows us to control what content gets served from cache. [57]

- Using the same capabilities, the service worker is able to implement offline functionality by storing a set of requests to be handled as soon as the network is available. [57]
- Using the push and notification API, the service worker can implement system notifications. [57]

Service workers follow the concept of progressive enhancement. They enhance a website functionality through a service worker life lifecycle and do not affect the base functionality of a web application if its visited through a web browser that does not support service workers. [58] Let us describe a service worker's lifecycle:

1. A service worker's lifecycle begins with its *registration* through an application code and is based on asynchronous events. [59]
2. The first event after the registration occurs when a web browser starts *installing* a service worker which fires an `install` event. During the installation, we usually use the Cache API to precache the assets needed for offline access. [59]
3. After successful installation, the service worker becomes *activating* and the `activate` event is fired. During the activation, we typically clear the old caches if any exist. [59]
4. After the activation, the service worker changes state to *activated* and is able to handle functional events, such as fetch. [59]

1.6.2.2 Web Application Manifest

As we have already stated, PWAs should be installable. This can be achieved using the web application manifest JSON file. The manifest contains the metadata used by a web browser during the installation process, as well as by the device's interface[7] for running applications. [57] [60] The manifest may contain:

- application metadata, such as the application name,
- display configuration, such as font size, background color or splash screen,
- paths to application icons displayed within the device's interface,
- default orientation and display mode declaration. [57]

[7]e.g. home screen or application listing

1.6.2.3 Application Shell Architecture

The application shell architecture is an approach to building client-side web applications. It is based on the idea of caching a basic skeleton of the application. The basic structure of the application is stored locally and thus available offline. This structure is dynamically populated with real data using JavaScript. Application shell promotes a good performance, fast application loading and a native-like interaction regardless of the connection quality. [57]

1.7 Server-Side Web Development

Section 1.5 provided an overview of technologies and approaches for client-side web application development. This section covers server-side development in a similar way.

1.7.1 Popular Technologies

We can use a wide range of technologies for server-side development. Therefore, this section focuses only on the overview of the most popular ones. We will determine the popularity of technologies based on the popularity of its web frameworks. According to the Stack Overflow Developer Survey 2021 [21], the most popular technologies for server-side web development are *Node.js*[8] and *Python*[9]. The survey defines the popularity of technology as the number of developers who have done extensive development work with the technology in recent years or who want to work with the technology in the next year.

The following two subsections provide the information on Node.js and Python characteristics.

1.7.2 Node.js

Node.js is an event-driven, single-threaded JavaScript runtime that uses V8 JavaScript engine outside of a web browser. Node.js does not implement concurrency model by creating a new OS thread for each request. It uses unblocking asynchronous paradigm instead. This means that Node.js dispatches inherently blocking operations, such as database or filesystem access, and continues with other tasks until it receives the response. [61]

The event loop is part of the Node.js runtime and is the key to achieve non-blocking I/O operations. Whenever possible, it delegates I/O operations to the system kernel, which is usually multi-threaded and can handle multiple operations simultaneously. When the operation is complete, the kernel notifies Node.js, which adds the appropriate callback in the `poll` step of the event loop.

[8]Express
[9]Flask, Django and FastAPI

Figure 1.10 shows simplified overview of the Node.js event loop which omits the `microtask queue` that we will describe separately.

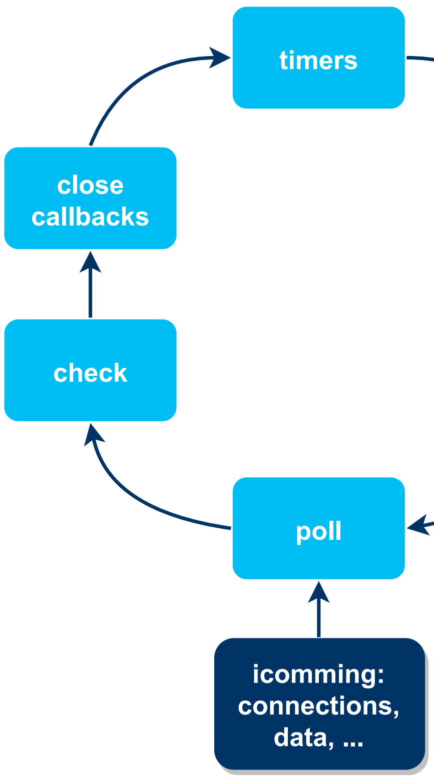


Figure 1.10: Node.js Event Loop

Each of the steps depicted in Figure 1.10 keeps a FIFO queue of callbacks to be executed. When entering a specific step, operations specific to that step are executed, followed by callbacks from the queue. [62] Let us describe the responsibilities of these steps:

timers phase executes callbacks queued by `setTimeout()` and `setInterval()`. The callbacks will be executed as soon as possible after the time *threshold* we specified has passed.

pending callbacks performs I/O callbacks postponed to the next iteration. Usually, the queue contains callbacks for system operations such as TCP errors reporting.

idle, prepare is only used internally.

poll fetches and executes I/O related events. This phase is responsible for processing the poll queue and calculating the time that blocking operations will take. Once the `poll` finishes processing the queue, it looks up the timers whose time thresholds have passed. It it find some, the event loop jumps right to `timers` phase. If there are no timers, the event loop will proceed to the `check` phase.

check performs callbacks queued by `setImmediate()`.

close callbacks executes close callbacks, such as socket closing.

Between processing queues for the steps above, the event loop checks and executes functions stored in the `microtask queue`. This queue consists of two sub-queues. The first one stores function calls delayed by `process.nextTick()` and the second one holds functions delayed by `promises`. [63]

1.7.3 Python vs Node.js

Python is an interpreted dynamic programming language. It provides high-level data types and statements that allow us to express complex operations concisely. Python provides splitting code into reusable modules.[64]

The Python interpreter is extensible with custom functions or modules implemented using C language. This feature is useful for implementing critical sections of the application where efficiency is important.[64]

Below we will compare Node.js and Python based on architecture, speed, scalability, universality and applicability.

Architecture Node.js has event-driven architecture which supports asynchronous I/O. In contrast to Node.js, Python does not implement such architecture. It can use special libraries for asynchronous I/O, but they are not part of most frameworks. [65]

Speed Both languages are interpreted, so they are generally slower than compiled languages. Node.js benefits from the performance of the V8 engine, which is optimized by Google. The fact that Node.js runs outside a web browser also has a positive effect on its performance. The non-blocking event-driven architecture allows multiple requests to be processed simultaneously. Python only supports synchronous code execution and is generally slower. [65]

Scalability Node.js supports the creation of modules and micro-services that run in separate processes and communicate using lightweight mechanisms. Python's scalability is limited. Its memory management is not thread-safe and therefore must use *Global Interpreter Lock*. This allows Python to run only one thread at a time. [65]

Applicability Node.js is widely used for web application back-end development. The advantage of Node.js is that it allows us to use JavaScript both for client-side and server-side web development. Both Python and

Node.js are cross-platform. Python may be used as a full-stack language to develop both front-end and back-end. [65]

Node.js is a popular technology used for web development that allows us to develop back-end for web applications in JavaScript which is a native technology for client-side web development. Therefore, the following section will focus on Node.js frameworks only.

1.7.4 Node.js Frameworks

From the results of The State of JavaScript [2] survey, we can notice that some of the most used Node.js frameworks are Express, Next, Nuxt, Nest, Strapi and Fastify.

Leaving aside Express, the most popular Node.js frameworks according to npm trends [66] are Next and Nest. This subsection will cover Express, Next and Nest frameworks. We will include Nest because it is a robust framework that provides a high level abstraction on the top of the HTTP server layer provided by frameworks such as Express or Fastify. [67]

1.7.4.1 Express

Express is a minimalist and flexible framework for web application development. Its simple architecture makes it easy to use for anyone who has experience with Node.js environment. It is considered a robust HTTP server framework. [68]

Features Express is an unopinionated framework. That means that the framework has no opinion about how developers use it. Because of that, developers have the freedom to experiment. This can be an advantage or a disadvantage depending on the developer's experience. It promotes rapid server-side development, especially the creation of RESTful APIs. It also natively supports NoSQL databases.[68]

Use cases It is useful to create any type of application.[68]

1.7.4.2 Next

Next was originally presented as a back-end for a React front-end, however, we can use the latest version as a full-fledged server-side framework. [68]

Features Next is built on top of React. It supports fast real-time changes propagation while development. It splits assets so that only the required JS and CSS is loaded for each page. [68]

Use cases It is especially suitable for fast and SEO-friendly applications built with React. [68]

1.7.4.3 Nest

Nest is a Node.js framework that uses a robust HTTP server framework as the underlying platform. It uses Express.js by default, but also supports Fastify. Nest.js allows us to develop scalable, maintainable and testable server-side applications. [67]

Features Nest is an opinionated framework. It effectively solves the problem of application architecture by providing higly testable, scalable, maintainable and loosely coupled architecture out of the box. Implicitly, Nest supports TypeScript, but also allows us to use pure JavaScript. It combines object-oriented, functional and functional reactive programming approaches. Although it provides abstraction over HTTP server frameworks, it still allows us to use their APIs directly. Thanks to this, we can use the libraries available for these frameworks as well. Nest is also well documented. [67]

Use cases It is useful to create any type of scalable, testable and loosely-coupled application. [68]

1.7.5 Modern Development Approaches

As server-side development is a broad topic, we will focus on a limited selection of approaches similar to Section 1.5.2. According to the same sources as in the mentioned section, we have two fundamental trends on the server side, namely API-first development and microservices architecture.

1.7.5.1 API-First Development

APIs play an important role in modern web application development that supports loosely coupled architectures. They enable mutual communication between the front-end and back-end components, as well as communication between different applications. [41]

API-first development is the approach where we prioritize APIs before other aspects of the project. This approach allows us to create reusable, modular and extensible applications. [41]

Let us briefly explain the categorization of APIs according to the architectural style.

REST architectural style is based on separating concerns of the API consumers from the API provider. Instead, REST relies on general operations over resources using underlaying network protocol. [69]

RPC is based on the idea of calling procedures on a remote system. We usually call these procedures using their identifiers. This approach is protocol-agnostic, but does not allow us to take advantage of native protocol capabilities, such as caching. [69]

Event-driven/Streaming architecture do not require direct client API calls. Instead, it provides events that clients can subscribe to. When an event occurs, the client receives new data. [69]

1.7.5.2 Microservices Architecture

As we mentioned in Section 1.3.3.3, microservices are based on dividing the application into a system of small isolated services.

Microservices are created on the basis of business responsibilities so that these responsibilities are served independently. They provide high flexibility. We can develop each microservice using a different technology. They are quite reliable because a failure of one microservice does not take down the entire application. Moreover, we can fix and re-deploy only the failing microservice. Since the codebases of the individual microservices are small, we can maintain the code well. We can scale each microservice separately. This allows us to assign additional resources only to those processes that require them. When the microservice is modified, we have to redeploy only that particular microservice. [14]

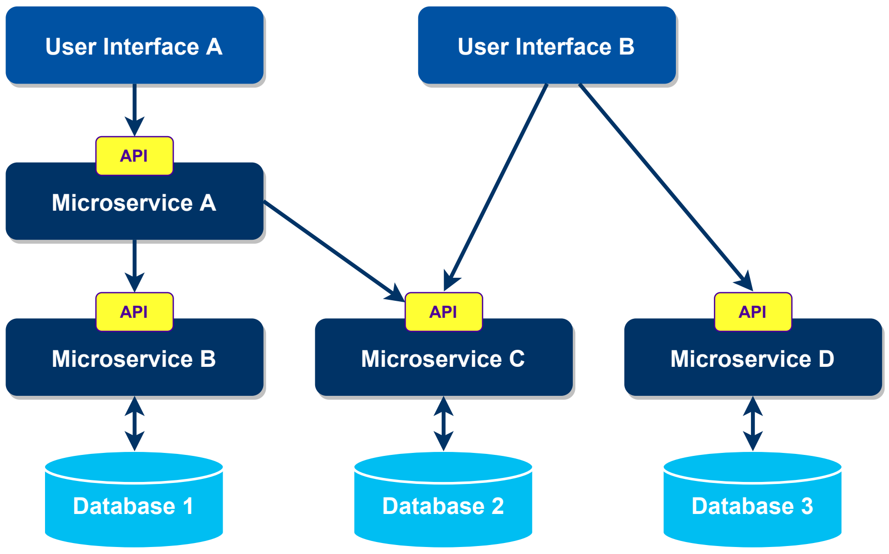


Figure 1.11: Microservices Architecture

1.8 REST

Representational State Transfer stands for an architectural style first introduced by Roy Fielding in his dissertation. In these days, REST architecture is often applied to the design of APIs for web services. A Web API complying to the REST architecture is called a REST API. A web service that provides a REST API is called RESTful. [70] The REST architecture conforms constraints outlined in the following section.

1.8.1 Constrains

In 1993, Roy Fielding began to address the problem of Web's scalability. Based on the analysis, he found that the scalability of the Web is determined by a set of constraints. Fielding grouped these constraints into following categories: [70]

- Client-server,
- Uniform interface,
- Layered system,
- Cache,
- Stateless,
- Code-on-demand. [70]

We will describe each of these categories in the subsections below.

1.8.1.1 Client-Server

The Web is based on a client-server model that we covered in Section 1.2. Client-server constraint promotes separation of concerns. Thanks to this, we can isolate web components and implement them independently. [70]

1.8.1.2 Uniform Interface

The uniformity of the interface of web components is essential for their mutual interaction. If any of the components does not comply with the agreed standards, it is not possible to interact with it. [70] There are four constraints that ensure the interoperability of web components:

Identification of resources The interface identifies each distinct resource by a unique identifier, such as a URI. [70]

Manipulation of resources through representations Clients use the representation to manipulate the resource. The representation is not the resource itself since the resource can have multiple different representations. [70]

Self-descriptive messages A client's request message should carry enough information to describe a resource's desired state. The server's response should carry enough information to describe a resource's current state. The server may or may not accept the client's request. [70]

Hypermedia as the engine of application state A resource's representation contains hyperlinks to related resources and actions. Clients can traverse the resources and actions using these links. The hyperlink is a part of the resource's state. [70]

1.8.1.3 Layered System

These constraints take advantage of a uniform interface that allows the deployment of network components between a client and a server. These components form a layered system where no layer can see beyond the layers it interacts with. [70]

1.8.1.4 Cache

The cache constraint requires a web server to add cacheability information to each response. This helps to reduce latency and a web server's load. [70]

1.8.1.5 Stateless

Stateless constraint mandates a web server should not store the state of any client application. This means that each client has to send all required contextual information whenever it communicates with the web server. [70]

1.8.1.6 Code-On-Demand

This constraint allows a web server to send executable code to clients. The client has to be able to understand the received code in order to execute it. This means that code-on-demand establishes a coupling between the web server and its clients. Because of that, this constraint is optional. [70]

1.8.2 Resource Modeling

Any nameable Web-based concept is considered a resource. A REST API represents a set of interlinked resources which are known as the REST API's resource model. The state of the resource in a particular format and at a particular time is called resource representation. Each resource representation contains the data, the metadata and the hypermedia links which allows clients to traverse the desired states of the resource. [70] [71]

1.8.2.1 Resource Archetypes

Resource archetypes allow us to consistently communicate the structure and behavior of a resource model. [70] When designing a REST API, we encounter the following four resource archetypes:

Document A document is similar to a single database record as it represents a single concept. It represents the basis for the other archetypes. A document may have a set of sub-concepts that we model as its child resources. A document's representation commonly consists of both its attributes and links to related resources. [70]

Collection A collection is a resource directory managed by a server. Clients may suggest new resources to be added into the collection. However, the server decides whether to add the suggested resources or not. [70]

Store A store is a repository of resources managed by a specific client. The client has full control over the repository. The repository never generates a new URI. Each resource is created under the URI specified by the client. [70]

Controller We use a controller resource to model procedural concepts. The controller resource performs application-specific actions that we cannot map to CRUD operations. It accepts appropriate parameters on input. [70]

1.8.2.2 URI Path Design

Since the resource model forms a hierarchy, we can easily use URIs to describe the model. Each path segment maps to a unique resource within this hierarchy. [70] The following list provides general URI path design rules for the REST API resource model.

- use forward slash separators to indicate a resources hierarchy,
- do not use a trailing forward slash in URIs,
- use hyphens to separate resource's name segments,
- prefer lowercase letters in URI paths,
- do not include CRUD function names in URIs,
- do not include file extensions in URIs. [70]

An example of URI that identifies the resource within the resource model hierarchy may be `http://api.example.org/collection/document`. Below we list the rules for URI path segments based on the resource archetype:

- use singular noun for document names,
- use plural noun for collection names,
- use plural noun for store names,
- use verb or verb phrase for controller names. [70]

1.8.2.3 URI Query Design

The query component of a URI is assembled of a list of key-value pairs. We interpret the query component as a derivative of the resource that is specified by the path component of a URI.[70] In REST architecture, we may use the query component of a URI for the two following purposes:

- to filter collection or store,
- to paginate collection or store records. [70]

In order to accomplish filtering, we should use a query parameters to pass key-value pairs where keys indicate attributes to be filtered and values specify the desired values of the filtered attributes. For pagination, we should use query parameters to indicate page index and size. [70]

If the complexity of filters exceeds the capabilities of the query component, we should create a filtering controller related to the collection or repository. This controller may accept a filtering query through a request's body. [70]

1.8.2.4 Modelling Many-To-Many Relations

Managing relations between resources is one of the common challenges in REST. Modelling many-to-many relations is one of the more advanced, yet common tasks. Imagine a situation where we want to model relations between users and groups, where one user can belong to many groups and one group can contain many users.

Essentially, there are two major approaches to achieve this:

1. We can model our own collection for each group. This leads to relational resources on URIs `/groups/{group-id}/users/{user-id}`.
2. We can model a separate relational resource such as `/group-members`.

1.8.3 REST API Design Using HTTP

In context of web development, REST API is usually implemented on top of the HTTP, specifically using its request methods, response codes and message headers. Resource methods form the uniform interface to interact with a REST API's resource model since each HTTP method has well-defined semantics. [70] For REST API resource model it is essential if an HTTP method is safe or idempotent. We describe these two attributes below.

Safety An HTTP method is safe if it is read-only. When using a safe method, we do not request or expect any change in the resource's representation. [72]

Idempotence We call an HTTP method idempotent if an identical request leaves the resource in the same state when called repeatedly. It is important to note that idempotency does not mean that the response to the request will always be the same. For example, assume that we repeatedly call DELETE request. If the deleted resource exists at the time of the first call, the server responds with 200 OK code. On the other requests, the server responds with 404 Not Found. Important is that the other requests do not alter the server state. [72]

In Table 1.2, we categorize HTTP methods by safety and idempotence. [72]

Method	**Safe**	**Idempotent**
GET	Yes	Yes
HEAD	Yes	Yes
OPTIONS	Yes	Yes
PUT	No	Yes
DELETE	No	Yes
POST	No	No
PATCH	No	No

Table 1.2: Safe and Idempotent HTTP Methods

1.8.3.1 Methods Semantics

A REST API must not compromise the design rules outlined below and must provide a transparent interface.

GET We must use GET to retrieve a representation of a resource. The request must not contain a body. We may repeatedly send GET requests without causing any side effect. [70]

HEAD We should use HEAD to verify a resource existence or to fetch its metadata. [70]

POST We can use POST for multiple purposes. We must use it to suggest a new resource to be added into a collection. In addition, we must use POST to execute controllers. On the other hand, we must not use it to get, store or delete resources since there are dedicated methods for these purposes. [70]

PUT A client must use PUT to insert a new resource to a store and to replace or update a stored resource. The request must contain a whole resource's representation. [70]

PATCH We should use PATCH to partially update an existing resource. [70]

DELETE We must use DELETE to remove resource from its parent. [70]

OPTIONS We should use OPTIONS to retrieve resource's available interactions. The interactions are listed within the `Allow` header. [70]

From the description of the POST method it is clear that repeated calls can cause the creation of new resources. For this reason it is a non-idempotent method. Since the PATCH method performs only a partial update of a resource, it may cause side-effects if the resource contains e.g. auto-incrementing values which the request does not update.

1.8.3.2 Response Status Codes

REST APIs take advantage of standard HTTP status codes to inform clients of the results of processed requests. See the Mozilla MDN[10] for a comprehensive documentation of the HTTP status codes.

Note that if a REST API is designed on top of the HTTP, it is really a bad practice not to use HTTP status codes.

[10]https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

1.8.3.3 Metadata

In addition to the HTTP status codes, REST Architecture also uses HTTP headers to convey various metadata. Headers are available within both HTTP's request and response messages. In Section 1.8.3.1, we already mentioned that a server uses HTTP `Allow` header to inform a client about available interactions over the resource. [70]

Another use case for HTTP headers is *content negotiation*. As we mentioned in Section 1.8.1.2, a resource can have multiple representations. Content negotiation stands for the process of requesting a specific representation of a resource. In a REST API designed with HTTP, the recommended approach to content negotiation is to use of the HTTP `Content-Type` header. [73]

The last use case for HTTP headers in the REST architecture that we will mention is *caching*. HTTP provides four cache control headers. Each of them is described below.

Expires We use `Expires` header to specify an absolute expiration time of a cached resource's representation. After exceeding that time, the representation must be re-validated. [74]

Cache-Control This header specifies if a response is cacheable. It is also capable of specifying by whom and for how long by using `max-age` or `s-maxage` directives. [74]

If-None-Match This header created a conditional request. If the resource's `ETag` does not match the `ETag` sent within the request, the server will return that resource. Otherwise, HTTP 304 Not Modified is returned.

Last-Modified Response's `Last-Modified` header contains timestamp that indicates when the resource was last changes. [74]

1.8.4 HATEOAS

The REST architectural style allows us to send hypermedia links within the body of the response. Clients may use these hyperlinks to dynamically navigate through resources. [75] Essentially, this principle is the same as browsing web pages via hyperlinks. In such a scenario, we enter the website's homepage. The homepage provides us links to other sections of the website. By clicking the links, we obtain more contextual information. Analogously, a REST client sends a request to the initial API endpoint. When the client receives a response, it uses the provided links to access resources and dynamically discover available actions. [75]

1.8.4.1 Benefits

Thanks to HATEOAS, clients do not need to hardcode URIs. Moreover, clients only need to know about possible actions, but not about their business logic[11]. This promotes the separation of concerns. [75]

1.8.4.2 Hypermedia Links Format

RFC 8288 [76] specifies a recommended format for relational links between resources on the Web. Each link should contain the following items:

Link relation type describes the relation between the actual resource and the target resource. We should represent it as the `rel` attribute. [76]

Target URI which we should represent as the `href` attribute. [76]

Target attributes include attributes of a link. We may include attributes such as `title` or `media` or our own attributes. [76]

1.8.4.3 Implementation

We can delivery hypermedia links as a part of a response body or within the `link` HTTP header. In Snippet 1, we can see the HATEOAS links implementation within the response's body in JSON format. [75]

```
{
    "id": 10,
    "title": "Thursday Scheduled Meeting",
    "author": "George",
    "_links": [
        {
            "href": "10/participants",
            "rel": "participants",
            "type" : "GET"
        }
    ]
}
```

Code Snippet 1: HATEOAS Links

[11]For example, clients do not need to consider the user's role when rendering action controls.

1.8.5 Security Essentials

REST APIs are stateless. As a result, request authentication and authorization should not depend on sessions. Instead, each request must contain some form of credentials. [77] Below we outline some of the best practices for REST APIs security:

Keep it simple We should aim for as simple security mechanism as possible. As security becomes more complex, the risk of security breaches increases. [77]

Always use HTTPS In these days, HTTPS uses TLS to secure the communication. It allows us to use token-based security mechanisms. [77]

Never expose information in URL We should never use URL to exchange credentials. [77]

Input parameters validation We should validate request's data as soon as we can. If the validation fails, we should immediately reject the request. [77]

1.8.6 Versioning

Versioning is a useful approach that helps us to manage the impacts of breaking API changes. Breaking changes should result in a change in the REST API's version. [78] They include the following scenarios:

- response's data format changed for one or more resources,
- a data type in the request's or response's data attribute changed,
- some resources were removed from the API. [78]

Below we outline the three most common approaches used for API versioning.

1.8.6.1 URI Versioning

This approach is the most popular and straightforward. The most common way is to specify the version using numeric value or `v\d+` expression. Possible examples are `http://api.example.com/v1` or `http://apiv1.example.com`. [78]

1.8.6.2 Versioning Using Custom Request Header

Although it duplicates the content negotiation implemented by `Accept` header, we may use our custom header to specify the API's version. Possible example is `Accept-version: v1`. [78]

1.8.6.3 Versioning Using “Accept” Header

Using content negotiation for API versioning allows us to keep clean and consistent URLs. However, this approach increases the complexity of API since it becomes responsible for indentifying which version of a resource to return. [78]

1.9 GraphQL

The REST architecture, which we discussed in the previous section, has essentially become a standard in web API design. It has some great features such as statelessness or structured access to resources based on their hierarchy. [79] However, requirements for APIs have changed rapidly since REST was introduced. In many cases, REST APIs have become not flexible enough to keep pace with the rapidly changing requirements of client applications. [80] [79]

There are three main requirements for API development which emerged in recent years:

1. Increasing usage of mobile devices makes it necessary to load data from APIs efficiently. [80]

2. The ability to have multiple client applications that consume a single API makes it difficult to create a single API optimized for all its clients. [80]

3. Changes on the client-side often require changes in consumed data. [80]

GraphQL meets the requirements outlined above since it is designed to provide client applications with a flexible and intuitive way to describe their data needs and interactions. [81] Essentially, it consists of two components:

1. a query language for API,

2. a server-side execution engine. [82]

The GraphQL query language enables us to use requests to specify exactly what data we need from an API, while the execution engine executes queries based on a type system which we define for our data. [81] [80] GraphQL implementation does not require any specific programming language or database. Instead, we use our specific technologies to map the logic and data of our services to GraphQL’s uniform language and type system. [82]

1.9.1 Design Principles

GraphQL follows several design principles which make server-side APIs a friendly environment to build client side applications. We will cover these principles below.

Product-centric GraphQL is driven by the requirements of views and front-end developers. [82]

Hierarchical A GraphQL query is structured hierarchically and the response data has the same structure as the query. This approach is client-side friendly, since the client-side development mostly involves working with view hierarchies. [82]

Strong-typing In GraphQL, we define a type system that is specific to our service. Each operation is then performed with respect to that system. This allows GraphQL tools to verify both the syntax and type validity of each operation before it is executed. In addition, the server-side service is able to guarantee the response shape to a certain level. [82]

Client-specified response A GraphQL service uses its type system to describe the capabilities which are available for its clients to use. The client is than responsible for how it will consume the GraphQL service. It specifies its needs at field-level granularity within the request. [82]

Introspective GraphQL allows us to use its query language to query its type system. This may be useful for various tools or client libraries. [82]

1.9.2 From REST to GraphQL

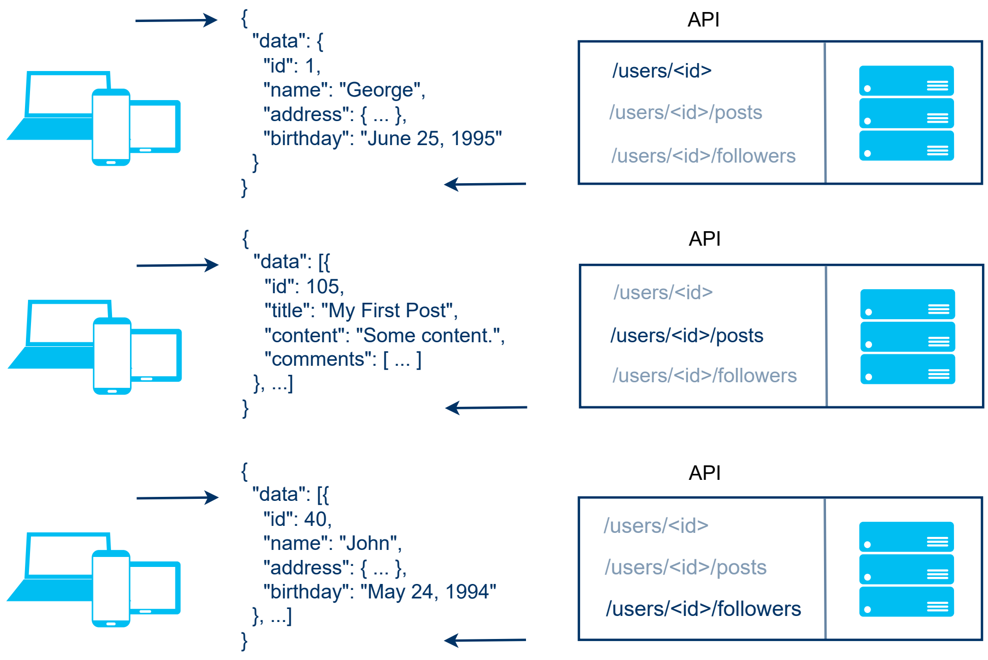
GraphQL is a direct competitor to REST in the context of web APIs. This subsection covers the major differences between these two.

1.9.2.1 Data Fetching

GraphQL provides a declarative data fetching where a client specifies the exact data it needs to get from an API. This brings major differences to data fetching compared to REST. Using a REST API, we typically use multiple endpoints to fetch the required data. In GraphQL, we send a single query to a GraphQL service. This query describes our exact data requirements. [79]

To illustrate the difference mentioned above, let us image a simple scenario. In a blogging application, we have a user's profile screen. The screen displays the basic information about the user, a list of user's posts and the last 5 followers of the user. We need to display a title for each post and a name for each follower. Using REST, we would typically make requests to the endpoints listed below. Figure 1.12 illustrates this scenario.

1. `/users/<id>` to fetch the user's data,
2. `/users/<id>/posts` to fetch all the posts for that user,
3. `/users/<id>/followers` to fetch the followers of the user.



```
{
  "data": {
    "id": 1,
    "name": "George",
    "address": { ... },
    "birthday": "June 25, 1995"
  }
}
```

```
{
  "data": [{
    "id": 105,
    "title": "My First Post",
    "content": "Some content.",
    "comments": [ ... ]
  }, ...]
}
```

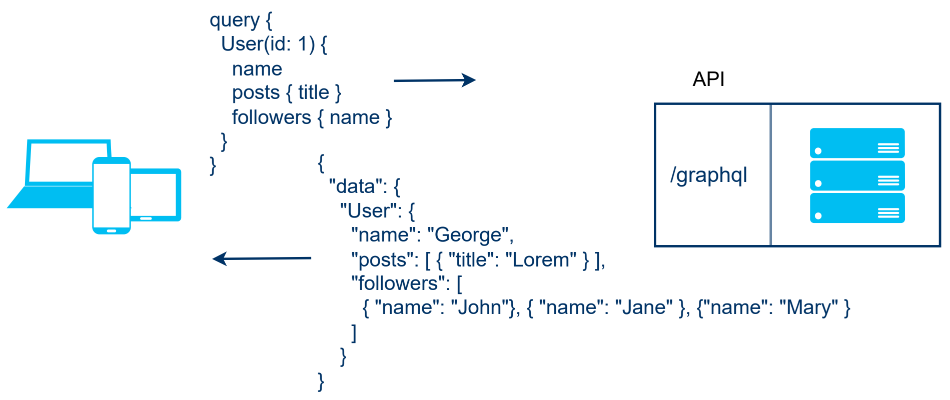
```
{
  "data": [{
    "id": 40,
    "name": "John",
    "address": { ... },
    "birthday": "May 24, 1994"
  }, ...]
}
```

Figure 1.12: Data Fetching With REST

Using GraphQL, we are able to describe our exact data requirements in a single query. We illustrate this scenario in Figure 1.13. In REST, it is hard to fetch data efficiently since endpoints return only fixed data structures. We may customize the data returned by these endpoints, but this approach becomes very difficult especially if the API provides data for many different clients. Because of this, clients often face *overfetching* and *underfetching*. [79]

Overfetching is the situation where a client downloads more information than it needs. We may notice the overfetching in Figure 1.12. The client only needs to display the titles of the posts of a specific user and the names of the last 5 followers of the user. The responses, however, contain additional data which the client do not need. [79]

Underfetching means that an endpoint does not provide enough information to the client. Because of that, the client has to send additional requests to fetch all the required information. Underfetching can lead to the situation where the client has to first send request for the list of resources and then



```
query {
  User(id: 1) {
    name
    posts { title }
    followers { name }
  }
}
```

```
{
  "data": {
    "User": {
      "name": "George",
      "posts": [ { "title": "Lorem" } ],
      "followers": [
        { "name": "John"}, { "name": "Jane" }, {"name": "Mary" }
      ]
    }
}
```

Figure 1.13: GraphQL Data Fetching

send one additional request for each element. This situation is known as the *n+1 problem.* [79]

1.9.2.2 Client-Side Product Iterations

For REST APIs, it is a common practice to customize the data provided by API endpoints to suit the requirements of the client's views. With this approach, the client retrieves all the required data for a particular view in a single request. However, any change in client views may result in a change in the required data. Therefore, the relevant API points must be adjusted to meet the new requirements. [79] Thanks to its flexible query language, GraphQL solves this problem and we can implement changes on the client-side without modifying the API. [79]

1.9.2.3 Insightful Server-Side Analytics

Since each client uses queries to specify its exact data needs, GraphQL allows us to deeply understand how clients use the data provided by API. We can, for example, use the statistics to deprecate the unused fields. [79]

1.9.2.4 Schema and Type System

As we mentioned in Section 1.9.1, GraphQL takes advantage of a service-specific type system. We use the GraphQL Schema Definiton Language (SDL) to create a *schema* which describes all the exposed and consumed types. The created schema than serves as the contract between the GraphQL API and its clients. [79]

1.9.3 Core Concepts

This subsection will briefly summarize the core technical concepts of GraphQL.

1.9.3.1 GraphQL Schema and Types

As we have already mentioned in the previous section, GraphQL has its own type system which is based on a schema. In order to create schema, we use the Schema Definition Language. The most fundamental component of a GraphQL schema is `type`. Type uses fields to describe a shape of an object that is exposed by an API. We can attach arguments to each field. These arguments can later be used to determine the value of the field. In the schema we can also express relations between types. Snippet 2 depicts a simple schema containing `Post` and `Person` types in one-to-many relation. [83]

```
type Post {
   title: String!
   author: Person!
}
```

```
type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}
```

Code Snippet 2: GraphQL Schema Example

Most types in a schema describe data objects, however there are two types with a special role within the schema:

1. `query` type,
2. `mutation` type. [83]

Every valid GraphQL schema must contain at least one `query` type. The `mutation` type is not required. For a comprehensive documentation of GraphQL schema, you may see Schema and Types section of graphql.org[12]. [83]

1.9.3.2 Fetching Data With Queries

Section 1.9.2.1 compared how we fetch data using GraphQL and REST. A REST API exposes multiple endpoints, each of which has a clearly defined structure of provided data. In contrast to REST API, GraphQL API only exposes a single endpoint. The data provided by this endpoint has a dynamic shape and it is the responsibility of client to describe its data needs. In order to describe its data needs, client sends `query` as a part of a request. Queries sent by a client corresponds to queries defined in a GraphQL schema. [84]

[12]https://graphql.org/learn/schema

1.9.3.3 Writing Data With Mutations

Previous section covered how we read data exposed by a GraphQL API. However, a typical web application also needs to make changes to the persistent data stored on the server. With GraphQL, we use mutations to modify the persistent data. Mutations allow us to perform create, update and delete operations on the data. [84]

1.9.3.4 Realtime Updates With Subscriptions

GraphQL offers *subscriptions* as a way to maintain a real-time connection to the server in order to immediately inform the client about events. As the client subscribes to an event, it establishes and keeps a connection to the server. When the subscribed event occurs, the server pushes the specified data to the client. In contrast to request-response model of queries and mutations, subscriptions are essentially a stream of data from the server to the client. [84]

1.9.4 GraphQL Architectures

This subsection provides an overview of the fundamental architectures in which GraphQL server may participate. The three following architectures represent predominant GraphQL use cases and demonstrate the flexibility of GraphQL in the content of usage:

1. server with a connected database,
2. layer that integrates existing services,
3. integration server with a connected database. [85]

1.9.4.1 GraphQL Server with a Connected Database

This is the most common setup for new projects. We have a single web server which implements GraphQL according to the specification. When the GraphQL server receives a query, it parses that query and loads the required data from the database. This process, which is database agnostic, is called resolving the query. When the data is loaded, the server constructs the response according to the specification[13] and sends it back to the client. [85]

1.9.4.2 GraphQL Server That Integrates Existing Services

The second setup integrates multiple existing systems using a single GraphQL API. Imagine a situation where we need to build a modern product that should integrate with a number of legacy or third-party components which expose several different types of APIs. In some cases, integration with these

[13]https://spec.graphql.org/October2021/#sec-Response-Format

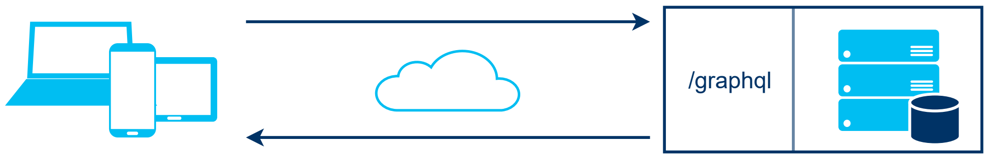


Figure 1.14: GraphQL Server With a Connected Database

components may be too complicated for us. In order to avoid the complexity, we may use a GraphQL server as a middleware that unifies the interaction with these components. This way, any new client application can communicate with the GraphQL server without knowing about the inconsistent infrastructure behind it. [85]

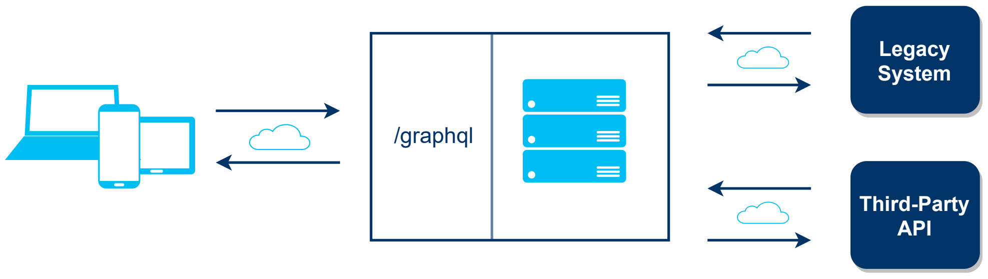


Figure 1.15: GraphQL Server That Integrates Existing Services

In the previous setup, the GraphQL server was independent of the database type. In this setup, it does not require any specific API for integrated components. [85]

1.9.4.3 Integration GraphQL Server With a Connected Database

With GraphQL, we are able to combine both the aforementioned setups and build a server that communicates with other systems and has a connected database. When the server receives a query, it uses either a connected database or some of the integrated service to resolve it. [85]

1.9.5 Resolvers

When we talked about the possible architectures that take advantage of GraphQL, we mentioned the *resolving* process. The way GraphQL performs resolving is the key to its flexibility. [85]

As we mentioned in Section 1.9.3.1, all available queries and mutations are defined as fields of the special `query` and `mutation` types. The GraphQL server implementation maps each of these fields to a specific function that we call a *resolver*. The only purpose of a resolver function is to fetch data corresponding to the filed it is mapped to. [85] When the server receives a query, it calls a resolver for each field specified by that query. As soon as all resolvers return, the server formats the data to match the query structure. [85]

1.9.6 GraphQL Client Libraries

Most client applications that use REST API perform the following steps in order to call an API endpoint and display the result of the call:

1. create and send a request,
2. receive and process the server's response,
3. store received data[14],
4. display the received data. [85]

In GraphQL, there are client libraries such as *Apollo* or *Relay* that abstract the data fetching into the following two steps:

1. declaratively describe requirements for the data,
2. display the received data. [85]

1.9.7 GraphQL Server Implementations

Since Facebook has only released GraphQL as a specification that describes exactly how a GraphQL server should behave [85], many community implementations have emerged. Some implementations have become popular and are nowadays considered proven solutions. Among these proven implementations, we may name the following:

- *Apollo Server* that supports several Node.js frameworks, such as Express, Koa or Fastify,
- *Express GraphQL* for Express,
- *Mercurius* for Fastify.

From npm trends [86], we may notice that Apollo Server and Express GraphQL are much more popular than Mercurius. However, all these implementations follow common pattern. To be able to create a GraphQL server, we must define GraphQL schema and a resolver map first. When instantiating or registering the server, we pass both the schema and the resolver map.

[14]e.g. in order to fetch related data or for caching

1.10 Chapter Summary

This chapter provided an introduction into web application development and a broad overview of the technologies and approaches used in web application development. Our goal was to cover the topic of web development broadly and to get to the topics such as REST, GraphQL and PWA in the most natural way possible.

We first laid out the basic information by explaining what a web application is and how it differs from a classic website. After that, we explained the client-server model and how it is used by web applications. Subsequently, we analyzed the types of web applications from several different perspectives. We followed up with JavaScript as a means of implementing modern types of web applications. We have noted that client-side JavaScript, that is executed by a web browser browser, is an extension to general JavaScript, and moved on to technologies and approaches used in client-side web development. Before we proceeded to server-side web development, we took a detailed look at PWAs. In our analysis of server-side web development, we mentioned an approach called API-first development. We followed up with a detailed analysis of REST and GraphQL.

Starting with the next chapter, we will focus on a selected subset of the analyzed technologies and approaches from a more practical point of view. Specifically, we will focus on building a web application using the decoupled architecture described in Section 1.3.3.2. We will follow mobile-first development and API-first development approaches which we mentioned in Sections 1.5.2.1 and 1.7.5.1. On the server-side we will demonstrate how to build both REST and GraphQL APIs. On the client-side we will demonstrate how to build a PWA and how to communicatie with server using REST and GraphQL APIs.

Chapter 2

Design

In this chapter, we will define the requirements for a prototype of the application that is suitable to be built as PWA. Based on the requirements, we will perform an analysis of the domain model. After that, we will select appropriate server-side technologies and libraries that will allow us to easily implement an extensible and maintainable application that provides both REST and GraphQL APIs. Similarly, we will select a suitable client-side framework whose ecosystem will allow us to easily implement PWA.

Our prototype will focus on content publishing. It should provide users with a way to publish content, such as posts or articles. It will also allow users to create reading lists and to store content into their reading lists. Generally, we will refer to the content as a story, similar to the Medium.com platform. Let us mention that the purpose of the prototype is not to create a production-ready application, but to demonstrate selected technologies and approaches.

2.1 Functional Requirements

This subsection specifies the functional requirements for the prototype.

F1: Access Control and Sign-in Both anonymous and logged-in users can access the application prototype. Anonymous users have access limited to viewing stories and user profiles only. The prototype must allow users to log into their account by entering their email and password.

F2: List Stories The prototype must allow all users to list existing stories. Each story in the list must contain links to its view and author.

F3: Search Stories The prototype must allow all users to search for stories based on the content of their title and description. Each story found must contain links to its view and author.

F4: View Story Detail The prototype must allow all users to view the detail of the story. The detail of the story contains link to its author.

F5: View User Detail The prototype must allow users to view the detail of any user. The detail of the user contains the list of user's stories with links to these stories.

F6: List My Stories The prototype must allow logged-in users to view a list of stories they have created.

F7: Create a Story The prototype must allow logged-in users to create a new story.

F8: Update a Story The prototype must allow logged-in users to update their stories.

F9: Delete a Story The prototype must allow logged-in users to delete their stories.

F10: List My Reading Lists The prototype must allow logged-in users to view a list of their reading lists.

F11: Create a Reading List The prototype must allow logged-in users to create a new reading list.

F12: View My Reading List The prototype must allow logged-in users to view the detail of reading lists they have created. The detail contains a list of stories added into the reading list. Each story has links to its view and author.

F13: Rename Reading List The prototype must allow logged-in users to rename their reading lists.

F14: Delete Reading List The prototype must allow logged-in users to delete the reading lists they have created.

F15: Add Story into Reading List The prototype must allow logged-in users to add stories into their reading lists.

F16: Remove Story from Reading List The prototype must allow logged-in users to remove stories from their reading lists.

F17: Offline Access The prototype must have some features that are essential to PWA. We outlined these features in Section 1.6.1. At the very least, it must be installable and allow users to view content they have already visited without the need for an internet connection.

2.2 Non-Functional Requirements

This subsection specifies non-functional requirements for the prototype.

N1: Architecture The application prototype has to be divided into a client-side application and a server-side application, where the latter will expose an API.

N2: REST and GraphQL Server-side application has to expose both REST and GraphQL APIs with the same capabilities.

N3: Extendability, Sustainability and Testability The result prototype must be extendable, sustainable and testable.

2.3 Domain Model

This section analyzes the domain model of the prototype based on the functional requirements defined in Section 2.1. The purpose of the domain model is to get overview of the entities occurring in the logical layer of the prototype and the relationships between these entities. We will use the domain model in the following section to select the appropriate database technology.

Figure 2.1 depicts the domain model. The functional requirement `F1` implies the need for the existence of `User` entity. Based on several functional requirements, including `F2`, `F4` or `F7`, there is the need for `Story` entity. Similarly, functional requirements such as `F10` or `F11` implies the existence of `Reading List` entity. Furthermore, functional requirements `F6`, `F7`, `F8` and `F9` imply the existence of the relationship between `User` and `Story`. The relationship between `User` and `Reading List` must exist based on functional requirements `F12`, `F13`, and `F14`. Finally, the relationship between `Story` and `Reading List` must exist based on the functional requirements `F15` and `F16`.

2.4 Overall Architecture

After specifying the requirements, we can proceed to the architecture design. Based on the non-functional requirements, the prototype will be divided into a client-side component and a server-side component, where the latter will expose both REST and GraphQL APIs with the same capabilities. In order to demonstrate the communication with server using both REST and GraphQL APIs, we will create two separate client applications. Each of them will communicate with the server using a different API via the HTTP protocol.

Since the server needs to store persistent data, we will create a database for this purpose. The database will run as a separate service, so it will form the third component of the prototype. We may notice that in the context of architectures described in Section 1.9.4, we will implement a GraphQL server

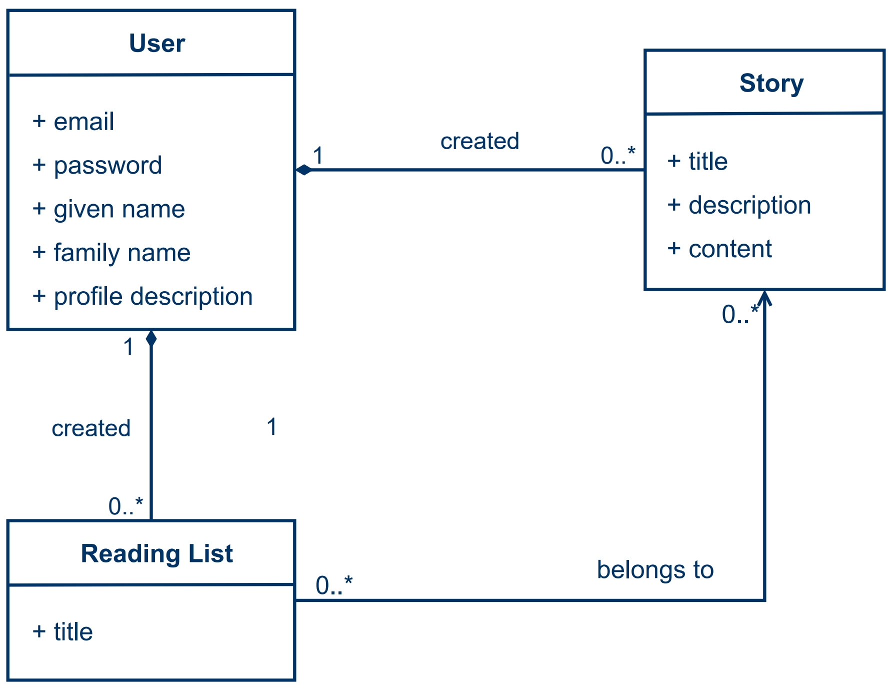


Figure 2.1: Domain Model

with a connected database. Overall, the prototype will implement a three tier architecture as follows:

1. **Presentation tier** – a client-side application that provides UI,
2. **Application tier** – a server-side application that implements business logic,
3. **Data tier** – a separate database service,

2.5 Technologies

This section deals with the selection of suitable technologies to implement the individual components of the prototype.

2.5.1 PostgreSQL

In Figure 2.1, we can see that there is a relation between each two entities. Based on the functional requirements, it is clear that together with an instance of a particular entity, we will often need to work with instances of related entities as well. Therefore, it is advisable to choose a database that supports join operations by default.

We will use *PostgreSQL*[15] for our domain. PostgreSQL is an open source object-relational database system that uses and extends SQL language. It earned the popularity due to its architecture, reliability, extensibility and a robust feature set. [87]

2.5.2 Node.js

We analyzed the popular server-side technologies for web application development in Section 1.7.1. There we covered Node.js and Python technologies. We will use *Node.js* due to its scalability and in order to take advantage of the same technology for both client-side and server-side development.

2.5.3 Server-Side Framework

In this subsection, we first explore the support for creating REST and GraphQL APIs offered by the frameworks described in Section 1.7.4. At the end of this subsection, we will select an appropriate framework based on the information gained.

2.5.3.1 REST With Express.js

Express.js allows us to simply create REST API using routing methods of the Express application object that we create by calling `express()` function exported by the `Express` module. [88] There are essentially two ways how to handle routing in Express:

1. routing methods for specific HTTP methods,
2. chained route handlers with `app.route()`. [88]

In Snippet 3, we can see the example of routing method for HTTP GET method. We are able to specify response status and response headers by modifying the `res` object in the routing method callback.

```
app.get('/user/:id', function (req, res) {
  res.send('user ' + req.params.id)
})
```

Code Snippet 3: Routing Method for Specific HTTP Method

Snippet 4 depicts the example of using `app.route()` method to create routes by chaining path and handlers for a specific HTTP method.

[15]https://www.postgresql.org

```
app.route('/events')
  .all(function (req, res, next) {
    // runs for all HTTP verbs first
  })
  .get(function (req, res, next) { ... })
  .post(function (req, res, next) { ... })
```

Code Snippet 4: Chained Route Handlers With app.route()

2.5.3.2 GraphQL With Express.js

There are several GraphQL server implementations for Express. We will briefly describe Express GraphQL and the Express integration for Apollo Server. As we described in Section 1.9.7, we must define GraphQL schema and a resolver map first to be able to create a GraphQL server using both implementations.

express-graphql [16] is developed and maintained by GraphQL Foundation. It allows us to bind a GraphQL server as middleware to the application object. [89]

apollo-server-express [17] is developed and maintained by the Apollo Community. It is the Express integration of `apollo-server`. In contrast to `express-graphql`, it allows us to bind the Express application as middleware to `ApolloServer` object. [90]

2.5.3.3 REST With Next

In Next, we can use API routes to create RESTful endpoints as part of our application folder structure. *API routes* is a feature of Next which ensures that any file inside the `/pages/api` directory is mapped to corresponding URL prefix with `/api` and will be treated as an API endpoint. Snippet 5 depicts the RESTful route defined in `pages/api/user/[id].js`. [91]

2.5.3.4 GraphQL With Next

We can also use API routes to create GraphQL API. All we have to do is to create a `pages/api/graphql.js` file and to initialize a GraphQL server such as Apollo in the route handler function. Same as with Express, we need to define a GraphQL schema and a resolver map.

[16] https://github.com/graphql/express-graphql
[17] https://www.npmjs.com/package/apollo-server-express

```
export default function userHandler(req, res) {
  const { query: { id, name }, method, } = req;

  switch (method) {
    case 'GET':
      res.status(200).json({ ... });
      break;
    ...
  }
}
```

Code Snippet 5: Next RESTful API Route

2.5.3.5 REST With Nest

In Nest, we can use controllers to create RESTful endpoints. Controllers are created as classes annotated by the `@Controller(<name>)` decorator. We can define a set of methods on each controller. When annotated with the decorator of the corresponding HTTP method, such as `@Get()` or `@Post()`, the method becomes a handler for an API endpoint. Each HTTP method annotation allows us to specify relative route path which can include route parameters. The resulting API endpoint URL consists of the controller name and the relative path of the route method. [92]

On the level of controller methods, we can also use annotations in order to extract data from the request, such as request body, headers, route parameters or query parameters. [92] Listing 6 depicts the RESTful route handlers implemented by Nest controller.

```
@Controller('cats')
export class CatsController {
  @Post()
  @HttpCode(201)
  create(dto: CatDto): string {
    return { ... };
  }

  @Get()
  @HttpCode(200)
  findAll(): string {
    return [ ... ]
  }
}
```

Code Snippet 6: Nest RESTful Routes

2.5.3.6 GraphQL With Nest

Nest provides a built-in `@nestjs/graphql` module. We can configure this module to use Apollo Server for Express or Mercurius for Fastify. [93] There are two ways of building GraphQL API with Nest:

1. code first approach,
2. schema first approach. [93]

In **code first approach**, we use decorators and TypeScript classes to generate the GraphQL schema. This approach is suitable if we prefer to stick to one language, such as TypeScript. [93]

On the other hand, in **schema first approach**, GraphQL SDL files are the source of truth. Since SDL is a language-agnostic, is allows us to share the GraphQL schema between multiple different platforms. We may use Nest to generate TypeScript interfaces or classes from the GraphQL schema in order to avoid boilerplate code. [93]

As we have already learned, to use a GraphQL server implementation, we usually need to define a resolver map. Nest provides the `@nestjs/graphql` package that automatically generates the resolver map. In order to accomplish that, it uses metadata from decorators that we use to annotate classes that implement methods used as resolver functions. [94]

2.5.3.7 Conclusion

We learned that all frameworks provide decent support for building both REST and GraphQL APIs. The information we obtained is summarised in Table 2.1.

Framework	**REST**	**GraphQL**
Express	Uses routing methods.	Directly uses GraphQL server.
Next	Introduces the concept of API routes.	Directly uses GraphQL server.
Nest	Using decorators to create controllers.	Uses @nestjs/graphql module to integrate GraphQL server.

Table 2.1: REST and GraphQL With Node.js Frameworks

From the analyzed frameworks, we choose **Nest** because of its modularity, robust and clean architecture, the possibility to use decorators for declarative development and the existence of a module integrating GraphQL server.

2.5.4 Client-Side Framework

Similarly to the previous subsection, we first explore the support for building PWAs offered by the frameworks described in Section 1.5.3. At the end of this subsection, we select a suitable framework based on the information gained.

2.5.4.1 PWA with React

The React ecosystem includes Create React App[18], a popular library for creating React applications without having to install or configure tools like webpack or Babel. This library provides a number of templates for React applications. These include the following two templates that we may use to make our web application progressive: [95]

- `cra-template-pwa` – the template for a JavaScript PWA,
- `cra-template-pwa-typescript` – the template for a TypeScript PWA.

The templates listed above contains a `src/service-worker.js` file that is based on the Google's *Workbox*[19]. This service worker handles all requests for webpack-generated assets, including navigation requests, using a cache-first strategy. Note that the offline-first behavior is opt-in so that it is up to us to decide if we want to register the service worker or not. [95]

2.5.4.2 PWA with Vue

With Vue, we may use the `@vue/cli-plugin-pwa`[20] plugin for vue cli to add a service worker into our application. We may configure our PWA using the `pwa` property of the `vue.config.js` file. The service worker added by the `@vue/cli-plugin-pwa` is also based on Workbox. [96]

2.5.4.3 Conclusion

We can say that both frameworks are good choices for building PWAs. Based on popularity, we choose **React**.

2.5.5 Prisma

To abstract database operations and bridge the gap between object and relational data representation, we will use *Prisma*[21], an open-source modern Node.js and TypeScript ORM.

[18]https://github.com/facebook/create-react-app
[19]https://developers.google.com/web/tools/workbox
[20]https://github.com/vuejs/vue-cli/tree/dev/packages/%40vue/cli-plugin-pwa
[21]https://www.prisma.io

2.5.5.1 Prisma Concepts

The most important element of Prisma is the *Prisma schema* which allows us to define our application model using its data modelling language. [97] In Prisma schema, we define the following:

data source which accepts an environment variable that specifies our database connection,

generator which contains a language specific provider that generates the *Prisma Client* from the schema,

data model that defines our application model as a collection of `model` objects. [97]

A data model has two major functions. The first is to represent structures of the underlying relational database or MongoDB. The second function is that it serves as a definition file for the Prisma Client which than provides typed queries through *Prisma Client API.* [97]

2.5.5.2 Prisma Workflows

The data model evolves routinely during its lifetime. For a production application, we typically want to keep track of changes made to the schema and to be able to revert those changes if necessary. Moreover, production applications typically have more than one environment and we may want to replicate schema changes in other environments. [98] For this reason, there are two typical workflows when working with Prisma schema:

1. *Prisma Migrate* workflow,
2. workflow with SQL migrations and introspection. [97]

With Prisma's database migration tool, the workflow consists of three step:

1. manual modification of the data model,
2. database migration using `prisma migrate dev` CLI command,
    - migration command generates the migration SQL file, updates database schema and generates *Prisma Client*
3. using *Prisma Client* to access the modified database. [97]

In contrast, when using SQL migration and introspection, the workflow consists of different steps:

1. manual modification of the database without modifying *Prisma schema*,

2. (re-)introspection of our database,
3. (re-)generate *Prisma Client*,
4. using *Prisma Client* to access the modified database. [97]

2.5.5.3 Benefits of Using Prisma

Prisma introduces a new approach to object-relational mapping. It provides a type-safe, data model-specific API for submitting typed database queries that return JavaScript objects with a well-defined structure. This allows us to avoid building SQL queries and complex objects for ORM. Another great feature is that the *Prisma schema* represents a single source of truth both for a database and our application. [99]

2.6 Server

As we mentioned, our server will expose both REST and GraphQL APIs. Through these APIs, the server will consume requests and return responses in JSON format. After receiving the request, the server executes the corresponding business logic and communicates with the database as needed. In this section we will focus on designing the server architecture to be scalable, sustainable and testable.

2.6.0.1 Authentication

To authenticate users, we will implement a simple JWT-based authentication. We will use a configuration file to store the secret for the JWT signature. We will encapsulate all the authentication logic using the corresponding module.

2.6.1 Modularity

To make our server modular and to comply with separation of concerns, we will take advantage of Nest modules. Modules are classes annotated with the `@Module()` decorator. We pass this decorator an object containing the metadata that Nest uses to organize the application. Each Nest application requires at least the root module. [100]

We will use the `AppModule` as a root module for our server. Based on the domain model, we can identify the following domain driven modules that will encapsulate all the logic related to the corresponding entity:

- `User Module`
- `Story Module`
- `ReadingList Module`

In order to encapsulate the logic for authentication, we will create `AuthModule`. Finally, we will create a `PrismaModule` to encapsulate the custom Prisma logic.

2.6.2 Layers

Within each module, we will divide the responsibilities outlined in the beginning of this section into the following three layers:

Presentation Layer that consumes requests and constructs responses.

Business Logic Layer which implements the business logic.

Data Access Layer that communicates with the database.

From a different perspective, we can think of the server architecture as a headless MVC. The three layers listed above comply with the headless MVC architecture as depicted in Figure 2.2. Headless in our case means that the view is not implemented as a UI, but only provides a JSON representation of the resources.

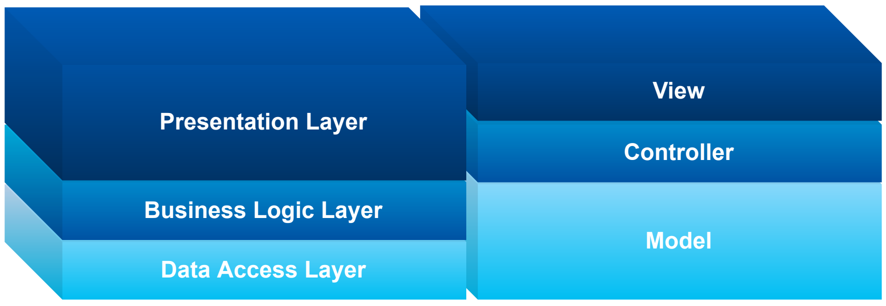


Figure 2.2: Server Architecture Layers to MVC

2.6.2.1 Data Access Layer

In our case, we do not need to manually implement the data access layer, as its implementation will provide the *Prisma Client* generated from the schema. All we need to do is to pass the database connection information to the client. Prisma Client will provide us with a type-safe interface for working with a database based on the data model defined in our schema.

2.6.2.2 Business Logic Layer

In Nest, it is appropriate to implement the business logic layer using services. Services are classes annotated with the `@Injectable()` decorator. This decorator declares that the Nest Inversion of Control container can manage the annotated class. This allows us to leave resolving dependencies to dependency injection. [94]

In order to provide Prisma Client with our database connection information and to make it injectable, we will create the `PrismaService` class as a part of the `PrismaModule`. This class will provide us a connection between the business logic layer and the data access layer. Any service that requires access to the database will inject `PrismaService`. In general, a service can have other services as dependencies.

Services will provide all necessary CRUD operations on the data. Read and delete operations require the necessary data identifiers as input parameters. Create operations require the corresponding DTOs[22] constructed at the presentation layer as their input parameters. Update operations require both.

Where needed, the services return entity classes that we will create as implementations of types generated from our Prisma schema. This will allow us to extend the types generated by Prisma with custom functionality if needed in the future. Additionally, it allows us to annotate these types and, for example, use them to generate OpenAPI documentation for our REST API.

2.6.2.3 Presentation Layer

Our server will provide two versions of headless presentation layer. The first one will provide the representation of resources through the REST API and the second will provide the GraphQL representation of the data.

2.6.3 REST API Design

To implement our REST API, we will use Nest controllers. As we have already mentioned, a Nest controller is a class annotated by the `@Controller()` decorator. Controllers inject the required services as their dependencies.

2.6.3.1 Resource Modelling

A REST API design usually starts with identifying the resources. We have already done this work in Section 2.3 by analyzing the domain model, so we can move on to resource modeling. We follow the theoretical background laid in Section 1.8.2.

In Table 2.2, we can see the URIs design for the identified resources along with the functional requirements that will use the capabilities of the resources. We may notice the modelling of the many-to-many relations between *Reading List* and *Story* entities as well as between *User* and *Reading List* entities.

[22]Data Transfer Object

URI	Archetype	Functional Req.
`/auth/sign-in`	Controller	F1
`/stories`	Collection	F2
`/stories/search`	Controller	F3
`/stories/:id`	Document	F4, F7, F8, F9, F12
`/reading-lists`	Collection	F11
`/reading-lists/:id`	Document	F13, F14
`/reading-lists/:id/stories`	Store	F12
`/reading-lists/:rId/stories/:sId`	Document	F15, F16
`/users/:id`	Document	F4, F12
`/users/:id/stories`	Collection	F5, F6
`/users/:id/reading-lists`	Collection	F10
`/users/:uId/stories/:sId`	Document	F4
`/users/:uId/reading-lists/:rId`	Document	F12

Table 2.2: REST API Resources

2.6.3.2 Design Using HTTP

After modeling the resources, we can proceed to design the interaction with the resources. As we specified in Section 2.4, our clients will communicate with the server using the HTTP protocol. We therefore proceed to designing a REST API using HTTP.

We follow the theoretical background laid in Section 1.8.3. Table 2.3 summarizes our RESTful endpoints with associated HTTP methods. Note that we do not necessarily need an endpoint for the stories list, since we can use the existing search endpoint for that purpose. Let us further note that we use `r-l` as an abbreviation for `reading-lists` to fit on the page. You may notice using the PATCH method for partial updates, using the POST method for the `/auth/sign-in` controller resource or using the PUT method for adding resources into the stores that implement many-to-many relations. Since this feature is not captured in the table, we should mention that each collection resource allows us to specify the maximum number of records returned using the `limit` query parameter of the collection URL.

Method	**URI**	**Description**
POST	`/auth/sign-in`	Sign-in to obtain JWT.
POST	`/stories`	Create a new story.
POST	`/stories/search`	Search for stories.
GET	`/stories/:id`	Get the story.
PATCH	`/stories/:id`	Partially update the story.
DELETE	`/stories/:id`	Remove the story.
POST	`/reading-lists`	Create a new reading list.
PATCH	`/reading-lists/:id`	Update the reading list.
DELETE	`/reading-lists/:id`	Delete the reading list.
GET	`/r-l/:id/stories`	Get stories within the reading list.
PUT	`/r-l/:id/stories/:id`	Add story into the reading list.
DELETE	`/r-l/:id/stories/:id`	Remove story from the reading list.
GET	`/users/:id`	Get the user.
GET	`/users/:id/stories`	Get user's stories.
GET	`/users/:id/r-l`	Get user's reading lists.
GET	`/users/:id/stories/:id`	Get the user's story.
GET	`/users/:id/r-l/:id`	Get the user's reading list.

Table 2.3: REST API Endpoints

The final step in the design of our REST API is to specify the HTTP codes and headers that the API responds with in various situations when clients call the endpoints listed above. We list the used response codes and the corresponding scenarios in Table 2.4.

HTTP Code	**Use Cases**
200 OK	Successful response to read and update operations.
201 Created	Successful response to create operations.
204 No Content	Successful response to delete operations.
400 Bad Request	Invalid request format. Invalid DTO in requests to create, update and controller operations.
401 Unauthorized	Missing valid JWT in requests to secured endpoints.
403 Forbidden	Requests to create, update or delete resources that are not allowed for the user. e.g. delete a story created by another user
404 Not Found	Any request to a non-existing resource.
409 Conflict	Any request that puts the resource in a state that breaks the constraints. e.g. repeatedly adding the story into the reading list
500 Internal Error	Any unexpected error.

Table 2.4: REST API HTTP Status Codes

Now that we have a proposed resource representation and interaction with those resources via HTTP, let us illustrate the design of linking resources using hypertext links. This will ensure that our API is HATEOAS compliant. The linking is depicted in Figure 2.3.

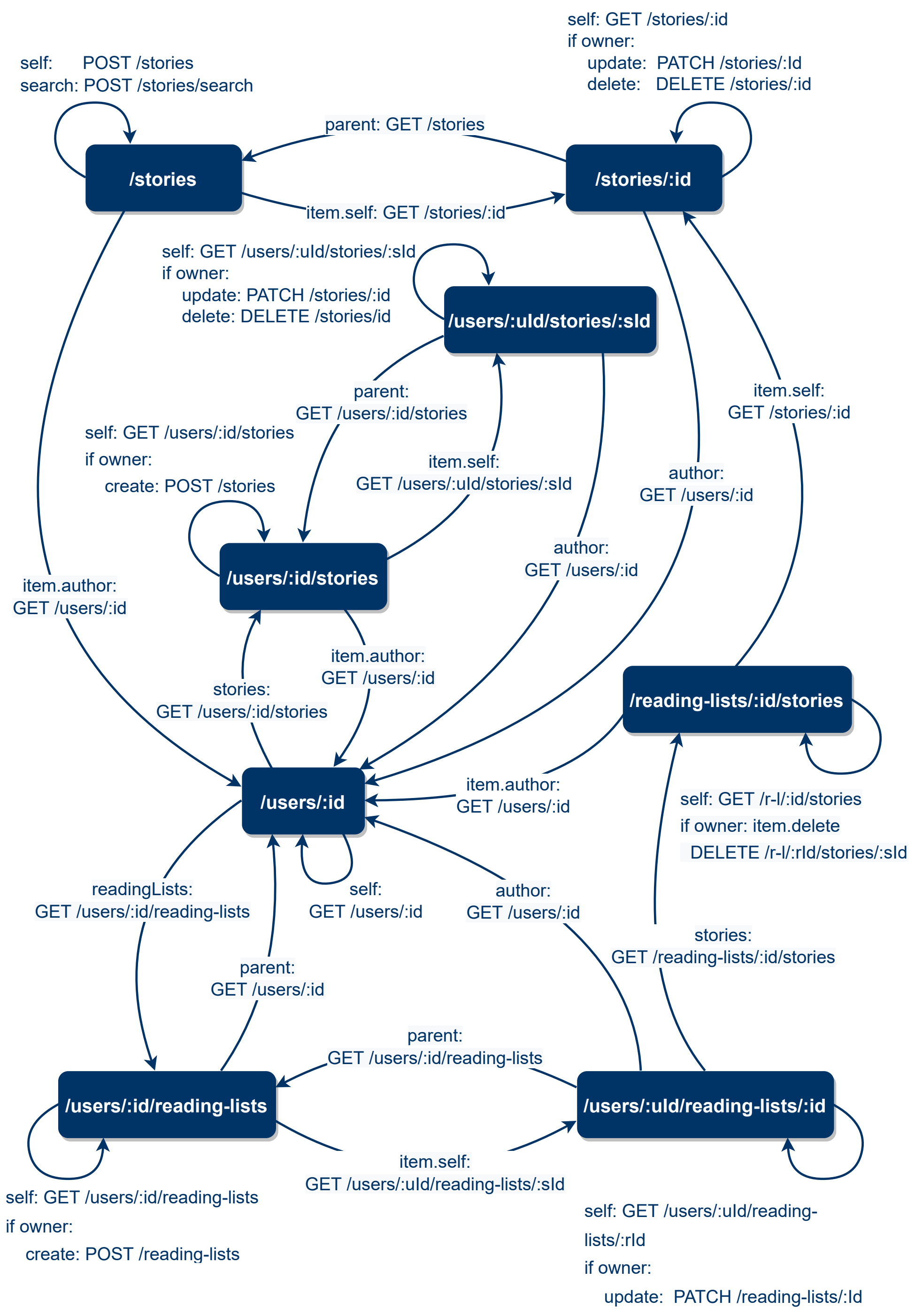

self: POST /stories
search: POST /stories/search
/stories
self: GET /stories/:id
if owner:
update: PATCH /stories/:Id
delete: DELETE /stories/:id
/stories/:id
parent: GET /stories
item.self: GET /stories/:id
self: GET /users/:uId/stories/:sId
if owner:
update: PATCH /stories/:id
delete: DELETE /stories/id
/users/:uId/stories/:sId
parent:
GET /users/:id/stories
item.self:
GET /users/:uId/stories/:sId
self: GET /users/:id/stories
if owner:
create: POST /stories
/users/:id/stories
item.self:
GET /stories/:id
author:
GET /users/:id
author:
GET /users/:id
item.author:
GET /users/:id
item.author:
GET /users/:id
stories:
GET /users/:id/stories
/reading-lists/:id/stories
/users/:id
item.author:
GET /users/:id
self: GET /r-l/:id/stories
if owner: item.delete
DELETE /r-l/:rId/stories/:sId
readingLists:
GET /users/:id/reading-lists
self:
GET /users/:id
author:
GET /users/:id
stories:
GET /reading-lists/:id/stories
parent:
GET /users/:id
parent:
GET /users/:id/reading-lists
/users/:id/reading-lists
/users/:uId/reading-lists/:id
item.self:
GET /users/:uId/reading-lists/:sId
self: GET /users/:id/reading-lists
if owner:
create: POST /reading-lists
self: GET /users/:uId/reading-lists/:rId
if owner:
update: PATCH /reading-lists/:Id

Figure 2.3: Resources Linking in REST

2.6.4 GraphQL API Design

According to the non-functional requirement N2, our server has to expose both REST and GraphQL APIs with the same capabilities. This subsection deals with the design of a GraphQL API that will be equivalent to the proposed REST API.

Because of the flexibility and shareability of the technology-agnostic GraphQL schema, we will apply the schema first approach to build our GraphQL API. Therefore, we will focus on specifying queries and mutations that provide the same capabilities as the REST API proposed in the previous section.

Compared to our REST API, we will replace controllers with *resolvers*. Same as controllers, resolvers inject the required services as their dependencies. Resolvers have a single responsibility – to map GraphQL queries and mutations defined by GraphQL schema to services implemented by the business logic layer of our server. [101]

2.6.4.1 Queries

As we learned in Section 1.9.3.2, GraphQL allows us to define a business-specific set of queries to read the data from the API. Below we describe GraphQL queries that will provide equivalent capabilities to the read-only endpoints we listed in Table 2.3:

1. *User* that returns the `User` type based on its id,
2. *Story* that returns the `User` type based on its id,
3. *Stories* that returns a list of the `Story` type records bases on the search string.

The three queries above provide us with all the querying power we need. This is thanks to the *Client-specified response* design principle of the GraphQL that we mentioned in Section 1.9.1. We learned its consequences on data fetching with GraphQL in Section 1.9.2.1. In summary, we can simply query the associated data using resolvers.

Because of that, *User* query covers all `GET /users/:id/*` endpoints. Since we will use `GET /reading-lists/:id/stories` only to fetch data related to the, it covers this endpoint too. *Story* query covers the `GET /stories/:id` query. With taking advantage of the GraphQL query arugments, *Stories* query covers `GET /stories` and `POST /stories/search` endpoints.

2.6.4.2 Mutations

For write operations, GraphQL allows us to define a set of mutations specific for our API. Below we describe GraphQL mutations that will provide equivalent capabilities to the write endpoints we listed in Table 2.5:

GraphQL Mutation	**RESTful API endpoint**
SignIn	POST `/auth/sign-in`
CreateStory	POST `/stories`
UpdateStory	PATCH `/stories/:id`
DeleteStory	DELETE `/stories/:id`
CreateReadingList	POST `/reading-lists`
UpdateReadingList	PATCH `/reading-lists/:id`
DeleteReadingList	DELETE `/reading-lists/:id`
AddStoryIntoReadingList	PUT `/reading-lists/:rId/stories/:sId`
RemoveStoryFromReadingList	DELETE `/reading-lists/:rId/stories/:sId`

Table 2.5: GraphQL Mutations to REST API Endpoints Mapping

2.6.5 Server Architecture Summary

To summarize the previous subsections, Figure 2.4 provides the fundamental overview of the design of the server components. It captures the services, controllers and resolvers in the aforementioned modules. Note that we do not show dependencies between components across domain modules, especially dependencies of controllers and resolvers on services from other domains, because this is implementation dependent.

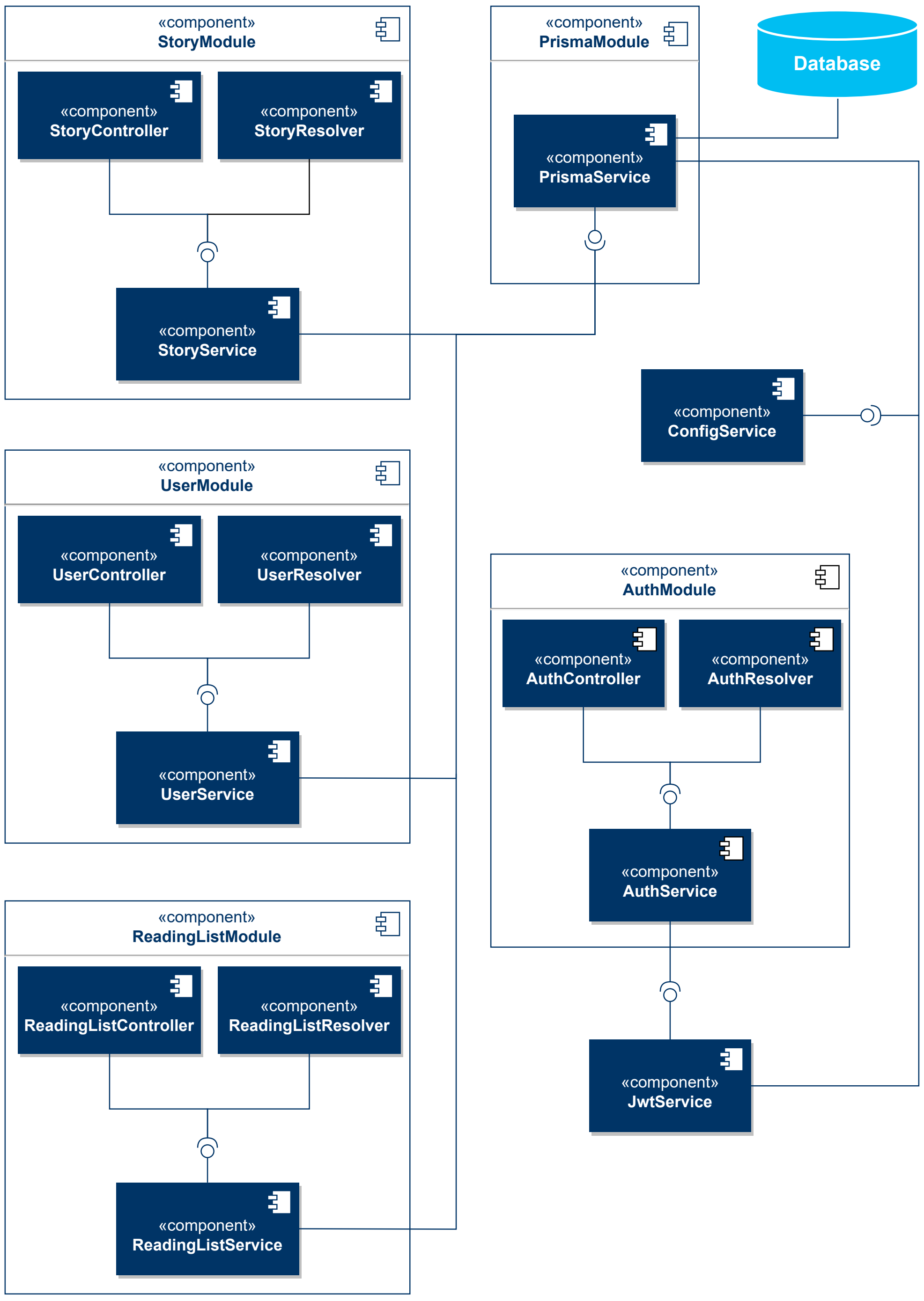


Figure 2.4: Server Architecture Components

2.7 REST Client

The first client will demonstrate the integration of the client-side application with the REST API exposed by our server. As we decided in Section 2.5.4, we will implement both client applications using *React*. Since functional components nowadays provide practically the same possibilities as class components, we will use functional components that are more lightweight.

2.7.1 State Management

A common problem in React development is the state management. In a regular React app, we need to manage the following types of state:

- Local state,
- Global state,
- Server state,
- URL state. [102]

2.7.1.1 Local State

The local state is the most basic type of state. It represents the data that we manage within a single component. React functional components provide the great way to manage this type of state in the form of the `useState` hook. We will use this approach in our implementation. [102]

2.7.1.2 Global State

Essentially, the global state represents data that we need to share across multiple components. A common example of this state is the logged-in user state, which is also encountered in our client application. The most basic approach to solving this problem is called *lifting state up*. This approach is based on lifting a shared state to the nearest ancestor in the component tree whose descendants still require that shared state. We then pass the shared state to any descendants that require it via props. [103]

While this approach is suitable for very simple applications, it is unsustainable for larger applications. React nowadays offers a *Context API* to solve this problem, but it is not a solution for state management. It only allows us to read global state, not change it. [102] Because of this, there are a number of third-party libraries for state management, such as *Redux*[23], *Recoil*[24], or *Zustand*[25]. Because of its simplicity and flexibility, we will use *Zustand* in our clients.

[23]https://redux.js.org
[24]https://recoiljs.org
[25]https://github.com/pmndrs/zustand

2.7.1.3 Server State

Server state represents the data stored on the remote server that we have to interpret in our UI. This problem becomes more complicated if we want to deal with displaying the status of queries, for example the application shell on load. [102] To make the server state management as simple as possible, we will use the *React Query*[26] library.

2.7.1.4 URL State

We will use the URL state to hierarchically identify our resources. It will also contain identifiers of specific entities.

2.7.2 Architecture Design

Figure 2.5 depicts the architecture design for the REST Client. It illustrates the basic components of the architecture, how they communicate with each other and how they send requests to the REST API exposed by the server.

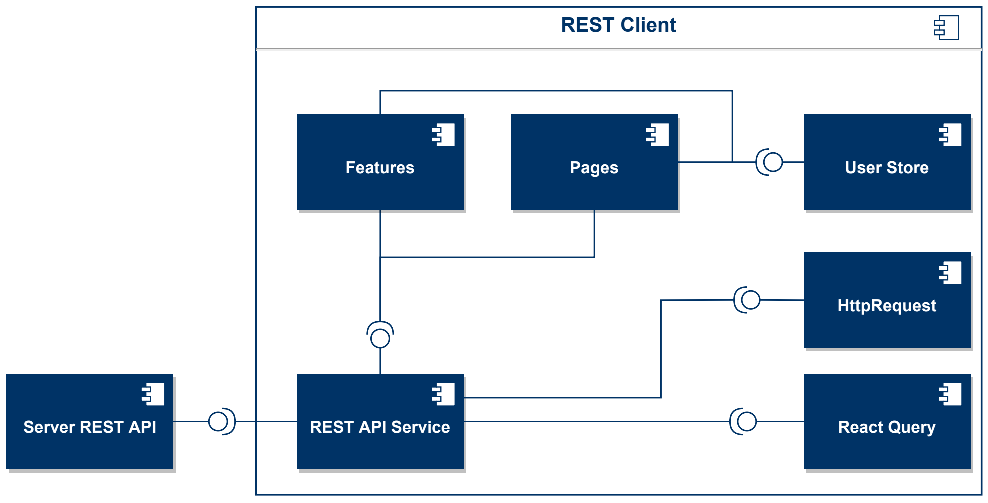


Figure 2.5: REST Client Architecture Design

2.7.2.1 HttpRequest

This component provides a wrapper over the JavaScript `fetch` function.

2.7.2.2 React Query

This component represents *React Query* library that handles our API requests

[26]https://react-query.tanstack.com

2.7.2.3 REST API Service

This component provides us with functions for querying REST API endpoints. It passes the requests created by the *HttpRequest* to *React Query*.

2.7.2.4 User Store

This component will use Zustand state management library to provide *Features* and *Pages* with the store for logged-in user.

2.7.2.5 Pages

The *Pages* component groups all screens of the client application. It may use *REST API Service* to load the data based on the URL state.

2.7.2.6 Features

Features component organizes all React components by their domain. These components can use *REST API Client* to interact with the REST API without leveraging HATEOAS links, or they can use *React Query* to send queries constructed from the links.

2.8 GraphQL Client

The second client application will demonstrate the integration of the client-side application with the GraphQL API exposed by our server. Since this client is identical to the first one in terms of functional requirements and will be implemented using the same framework, this section only provides an overview of the changes compared to the REST Client.

2.8.1 Server State Management

Since this client will communicate with the GraphQL API instead of the REST API, we will use a different library to simplify working with the server state. Specifically, we will use the *Apollo Client*[27] integration for React.

2.8.2 Architecture Design

Figure 2.6 depicts the architecture design for the GraphQL client. It illustrates the basic components of the architecture, how they communicate with each other and how they send queries and mutations to the GraphQL API exposed by the server.

[27]https://www.apollographql.com/docs/react

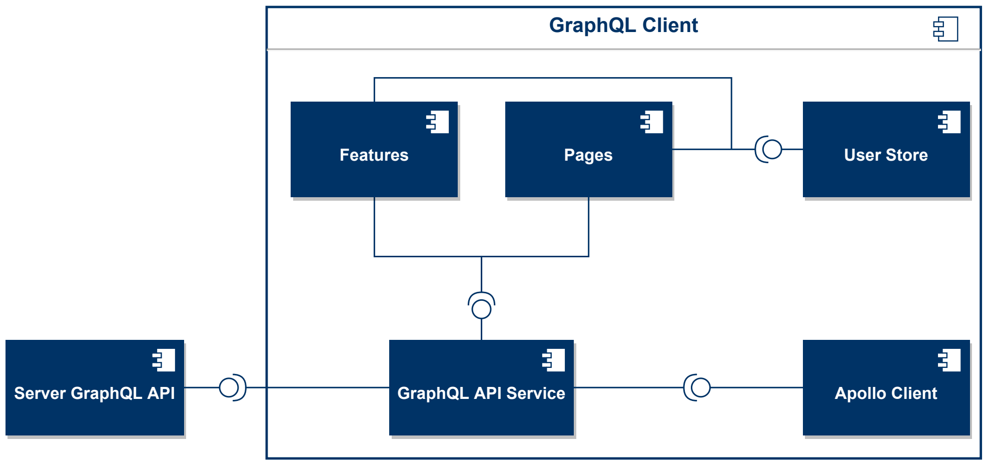


Figure 2.6: GraphQL Client Architecture Design

2.9 Chapter Summary

This chapter defined the requirements for the prototype that will serve as the practical output of this thesis and designed the prototype. Based on the requirements, we performed the analysis of the domain model. We designed both REST and GraphQL APIs that the server component of the prototype will expose. After that, we selected PostgreSQL, Prisma and the Nest framework to implement our server. To implement our REST Client, we selected React together with React Query and Zustand libraries. For REST Client, we replaced React Query with the Apollo Client library. After selecting the technologies and libraries, we designed the server and both clients.

CHAPTER 3

Implementation

This chapter focuses on the implementation of the prototype for which we designed the architecture and selected the appropriate technologies in the previous chapter. It first describes the configuration of the environment in which the project is run. Then, it describes the implementation of the database service and the process of initializing the database. Finally, it documents the implementation of the server and both clients.

3.1 Docker Setup

Since the implementation consists of several isolated components, we decided to use *Docker*[28] in order to provide the easy way to build and run the project as a whole. Using Docker, we created two separate environments for the project. These environments are defined by the following docker-compose files:

1. `docker-compose.dev.yml` which builds and runs the project for development purposes. We will refer to this environment as *development.*

2. `docker-compose.prod.yml` which builds and runs the project in order to present the practical output of the thesis. We will refer to this environment as *production.*

Table 3.1 depicts all services created by both docker environments. In addition to the services corresponding to the components listed in the previous chapter, both environments also include the `mwd-prisma` service. The highlighted service `mwd-prisma-studio` is created only for the *development* environment and provides us with a database client for Prisma.

[28]https://www.docker.com

Service	**Outer Port**	**Inner Port**
mwd-postgres	5438	5432
mwd-prisma-db-init	-	-
mwd-prisma-studio	*5555*	*5555*
mwd-server-side	8080	3000
mwd-client-side-rest	3100	3010
mwd-client-side-graphql	3200	3020

Table 3.1: Docker Services

3.2 PostgreSQL Service

Our database is provided by the `mwd-postgres` service listed in Table 3.1. We used the official docker image[29] for this service. Snippet 7 illustrates the configuration of PostgreSQL service in a docker compose file. We may notice the credentials definition under the `environment` key.

```
mwd-postgres:
  image: postgres:12.10
  container_name: mwd-postgres
  restart: always
  environment:
    - POSTGRES_USER=mwd_u
    - POSTGRES_PASSWORD=mwd_p
    - POSTGRES_DB=mwd_db
  ports:
    - '5438:5432'
```

Code Snippet 7: PostgreSQL Service Configuration

3.3 Database Initialization

Database initialization is the responsibility of the `mwd-db-init` service. All logic and data required by the `mwd-db-init` service is located within the `prisma-tools/` directory. This service starts when the `mwd-postgres` container is created and shuts down after it completes the following tasks:

1. database schema creation,

[29]https://hub.docker.com/_/postgres

2. database seeding.

Before we describe the implementation of the tasks listed above, let us briefly introduce the organization of the `prisma-tools/` directory. Important here is the `prisma/` subdirectory. It contains the following files:

1. `schema.prisma` which defines the Prisma schema,
2. `generate-data.ts` that generates random data for the schema,
3. `seed.ts` which inserts the generated data into the database.

3.3.1 Database Schema Creation

The service uses *Prisma CLI* to create a database schema based on the data model defined by our Prisma schema. In Snippet 8, we may see the model for `ReadingList` entity defined in the `prisma.schema` file.

```
model ReadingList {
    id Int @id @default(autoincrement())
    createdAt DateTime @default(now())
    title String
    author User @relation(fields: [authorId], references: [id])
    authorId Int
    stories StoriesOnReadingLists[]
    @@map("reading_lists")
}
```

Code Snippet 8: Prisma Schema Snippet

Our goal is to achieve the desired final state of the schema without tracking the steps leading to that state. Because of this, we do not follow the workflows described in Section 2.5.5.2. Instead, we use the `prisma db push` command with the `--force-reset` flag in order to reset the database and create the schema each time the service starts. [98]

3.3.2 Database Seeding

The service uses the `prisma db seed` command to initialize the database with pre-generated data from the `data/` subdirectory.

In the `package.json` file, we prepared the `yarn data:generate` command that generates the content of the `data/` sub-directory using `faker-js/faker`[30], a popular library for generating test data. The service does not call the `yarn data:generate` command at startup for optimization reasons. If necessary, we can use it to manually regenerate the data at any time.

[30]https://github.com/faker-js/faker

3.4 Server

Our server implementation is located within the `server-side/` directory and is run by the `mwd-server-side` service. As we decided in Section 2.5.3, we implemented the server using the *Nest* framework. We followed the default configuration and used the Express web server framework as the underlying platform. Before we continue, let us briefly introduce the organization of the `server-side/` directory:

- `config/` contains the configuration files for the server application,
- `graphql/` contains scripts for generating TypeScript classes and interfaces from the GraphQL schema,
- `prisma/` contains the same `prisma.schema` as the one used by the `mwd-prisma-db-init` service,
- `src/` contains the application logic of the server,
- `test/` contains end-to-end tests.

Now that we understand the structure of the `server-side/` directory, let us describe what happens at the startup of the `mwd-service-side` service:

1. it uses the `prisma generate` command to generate the Prisma Client from the `prisma/prisma.schema` file,
2. it uses `graphql:generate-classes` command from `package.json` to generate TypeScript classes based on the GraphQL schema.

3.4.1 Modularity

Our implementation follows the design proposed in the Section 2.6.1 and uses Nest modules to achieve modularity. The `AppModule` serves as the root module of the application. In order to build the application graph, it imports all the necessary modules. Each of these modules encapsulates a closely related set of capabilities. Snippet 9 depicts how the `AppModule` imports other modules.

The `@Module()` decorator takes an input object that may contain the following properties to describe the module:

providers which specifies the providers that the Nest injector will instantiate. These can be shared across the actual module or exported and used across other modules.

controllers specifies the controllers defined in the module that have to be instantiated.

```
@Module({
  imports: [
    ConfigModule.forRoot({ ... }),
    GraphQLModule.forRoot({ ... }),
    AuthModule,
    UserModule,
    StoryModule,
    PrismaModule,
    ReadingListModule,
  ],
  controllers: [AppController],
})
export class AppModule {}
```

Code Snippet 9: Server's Root Module

imports takes the list of modules that export providers required by the actual module.

exports specifies the subset of providers defined by the actual module that are available for the modules which import the actual module.

3.4.2 Prisma Module

The implementation of this module is trivial. It contains `PrismaService` class that extends `PrismaClient` class generated by the `mwd-server-side` service at startup and enum `constants.ts` which provides us with constants for Prisma error codes. Snippet 10 depicts the decorators used to describe the module. We may notice that we used the `@Global()` decorator. This ensures that we only need to import the module once in `AppModule` and it will be automatically imported into all other modules.

```
@Global()
@Module({
  providers: [PrismaService],
  exports: [PrismaService],
})
```

Code Snippet 10: Prisma Module Decorators

3.4.3 Domain Driven Modules

By domain driven modules we mean modules dedicated to individual entities captured by the domain model depicted in Figure 2.1. As we proposed in Section 2.6.1, we implemented the following three domain driven modules:

- `UserModule`
- `StoryModule`
- `ReadingListModule`

These modules share the structure that follows the architecture proposed in Section 2.6:

- `dto/` JS module contains DTO classes for operations implemented by the module.
- `entities/` JS module contains classes that implement types generated as the part of the Prisma Client.
- `envelopes/` JS module contains classes that represent REST API responses. These classes are based on the classes from the `entities/` JS module.
- `<name>Module` class that defines the module.
- `<name>Service` class which implements the business logic layer of the module.
- `<name>Controller` class that implements RESTful endpoints for the corresponding resource.
- `<name>Resolver` class that implements resolver for GraphQL.

3.4.4 Auth Module

The `AuthModule` follows the basic structure of domain driven modules described above. However, it serves a special purpose, as it implements user authentication for client applications. We describe the authentication flow below.

3.4.4.1 Authentication Flow

As we proposed in Section 2.6.0.1, this module implements a simple JWT-based authentication. The flow described below stays same for both RESTful and GraphQL APIs:

1. Client sends a sign-in request that contains user's email address and password.
2. The server validates the credentials from the previous step.
   - If they are valid, the server fetches the data of the corresponding user.
   - If they are invalid, the server returns error and the flow ends.
3. The server signs the JWT with the payload derived from the user's data and sends the JWT to the client as the part of the response.
4. The client receives the response and sends the JWT in subsequent requests as a bearer token within the HTTP `Authorization` header.
5. The server authenticates the used based on the JWT.

The following subsection presents a standard way to implement authentication using Nest.

3.4.4.2 Passport Library in Nest

For the authentication purposes, Nest integrates *Passport*[31], a popular authentication library for Node.js, using the `@nestjs/passport` module. [104]

To better understand the implementation of this module, let us describe how the Passport library works and how Nest integrates it. Essentially, *Passport* performs the three following steps:

1. Authenticate a user based on the credentials, such as username-password pair or JWT. [104]
2. Manage the authenticated state representation, such as JWT or Express session. [104]
3. Attach the authenticated user object to the `Request` object. This allows us to access the user within the route handlers. [104]

The Passport library can be considered a framework to some extent. It provides us with an Inversion of Control and abstracts the authentication into a set of strategy-specific steps that we customize. It has a rich ecosystem of implemented strategies that we configure by supplying a configuration object and callback functions that Passport calls based on its inner logic. [104] With `@nestjs/passport`, we perform the following steps in order to configure a strategy:

[31] https://github.com/jaredhanson/passport

1. We extend a `PassportStrategy` class from the `@nestjs/passport` module. [104]

2. Optionally, we provide configuration specific for that strategy in the `super()` method. [104]

3. We provide the verify callback by implementing the `validate()` method. In this method, we specify how Passport verifies the credentials. [104]

When we extend a Passport strategy, the Passport library provides us with an `AuthGuard`, a special type of guard that invokes the corresponding Passport strategy.

In Nest, guards are classes that implement the `CanActivate` interface and are annotated with the `@Injectable()` decorator. They have a single responsibility, and that is to decide whether the request will be processed by the route handler or not. We usually refer to this process as authentication and authorization. In Express application, it is typically handled by middleware. Nest also provides middleware that we can use for this purpose. However, guards provide benefits such as context knowledge. We will demonstrate the usage of the context in Section 3.4.8.4 when describing the GraphQL authentication. [105]

Now that we know our authentication flow and how Passport works, we can understand why this module contains `guard` and `strategy` JS modules.

3.4.4.3 Strategies

For the purpose of our authentication flow, the server implements the following Passport strategies:

- `LocalStrategy` that extends the `passport-local` strategy provided by Passport,

- `JwtStrategy` that extends the `passport-jwt` strategy provided by Passport.

We use `LocalStrategy` in the second step of the authentication flow. We have configured the strategy to lookup for the username under the `email` field of the request data. The `validate()` method uses `AuthService` to validate the credentials.

`JwtStrategy` implements the fifth step of our flow. We have configured the strategy to fetch the JWT as a bearer token from the `Authorization` header and to retrieve the JWT secret using the Nest built-in `ConfigService`. Furthermore, since we implemented a prototype, we ignored the expiration of the token. Snippet 11 depicts the configuration of `JwtStrategy`.

```
constructor(
  private readonly userService: UserService,
  readonly configService: ConfigService
) {
  super({
    jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
    ignoreExpiration: true,
    secretOrKey: configService.get('JWT_SECRET'),
  });
}
```

Code Snippet 11: JWT Strategy Configuration

3.4.4.4 Guards

In order to follow the DRY principle, our server provides guard classes that extend the `AuthGuard` class for both `local` and `jwt` strategies. These guards differs for REST and GraphQL APIs.

3.4.5 Data Transfer Objects Validation

In order to validate input DTOs, our server uses the *class-validator*[32] library. This library allows us to define validation rules over specific attributes of DTO classes using decorators. Snippet 12 provides an example of using class-validator decorators. To enable the class-validator, we registered a global validation pipe in the `src/main.ts` file. We configured the class-validator to remove all non-whitelisted properties from the input data. A non-whitelisted property is a property that the class-validator cannot map to any decorated property of DTO.

3.4.6 Common Logic

When implementing software, we typically encounter repetitive tasks. To ensure that our application is DRY compliant, we either implement custom solutions for recurring issues or use Nest features such as `ParameterDecorator` or `NestInterceptor`. All this logic is located in the `/src/common/` directory.

3.4.7 REST API

As we proposed in Section 2.6.3, we used Nest controllers as a building block for the REST API exposed by our server. We have also implemented the linking proposed at the end of Section 2.6.3.2.

[32]https://github.com/typestack/class-validator

```
export class CreateStoryDto {
  @IsString()
  @IsNotEmpty()
  title: string;

  @IsString()
  @IsOptional()
  description?: string | null;

  ...
}
```

Code Snippet 12: Class-Validator Decorators

3.4.7.1 Controllers

As we described in the previous sections, domain driven modules together with the `AuthModule` contain controller classes. We followed the principles described in Section 2.5.3.5. For each module, we created a controller using the `@Contoller()` decorator. To specify the base path of the controller, we used the `path` property of the `@Controller()` decorator input object.

For each controller, we implemented methods that cover the endpoints proposed in Section 2.6.3.2. To make each controller method a RESTful endpoint handler, we annotated that method with the decorator corresponding to the HTTP method of the handled endpoint. Where necessary, we specified the path relative to the base path of the controller as a decorator parameter. This path may contain route parameters used, for example, to specify the resource identifier. If necessary, we also specified a default HTTP status code of the response using the `@HttpCode()` decorator. This was necessary, for example, in the following cases:

- for POST `/stories/search` since this endpoint retrieves the data and Nest returns 201 OK code by default in responses to POST requests,
- for delete endpoints, where it was necessary to set HTTP code 204 No Content.

3.4.7.2 Endpoint Handlers

Each endpoint handler uses the services injected by the controller. Within endpoint handlers, we often need to work with route parameters, user object or response object. To get these values, we use the following Nest decorators in the signature of route handlers:

- Built-in `@Param()` decorator that extracts the route parameter by its key specified in the HTTP method decorator. We can bind pipes to this decorator in order to transform or validate the parameter. We typically use the `ParseIntPipe` to transform string parameter to integer.
- Built-in `@Res()` decorator to access the request object of the underlying platform.
- Custom `@User()` decorator to extract the user object from the request.
- Custom `@Limit()` decorator to extract the `limit` query parameter.
- Custom `@Jwt()` decorator to extract JWT from the `Authorization` header of the HTTP request.

3.4.7.3 Access Restriction

In order to restrict access to specific endpoints, we annotated the corresponding handlers using the `@UseGuards()` decorator. This decorator accepts a guard as an input parameter. In the case of the POST `/user/sign-in` endpoint handler, we used `LocalAuthGuard` as the input parameter of the `@UseGuards()` decorator. This guard internally uses `LocalStrategy`. For other restricted actions, we passed `JwtAuthGuard` as an input parameter of the `@UseGuards()` decorator.

3.4.7.4 HATEOAS

We have implemented an envelope system to incorporate links into the response data. At the end of the endpoint handler, we wrap the data into an envelop and assign the appropriate links. We assign links based on the context. So we are able, for example, to assign extra links in the case of a request that contains the JWT of the resource owner. For example, these can be links to update or delete this resource.

Links are assigned to the `_links` property of the envelope. In order to have typed links available, we created the `HateoasLink` class that complies to the hypermedia links format recommended by RFC 822 that we mentioned in Section 1.8.4.2.

To simplify link assignment, we created a simple factory function `createLink` and the `addLinks` helper function that assigns links to the `_links` property of the input object. These functions are available within the common logic.

3.4.7.5 Versioning

In order to enable versioning of our REST API, we call `app.enableVersioning()` in the `bootstrap()` function defined in the `src/main.ts` file. We specify the version of the API at the controller level using the `version` property of the `@Controller()` decorator.

3.4.7.6 OpenAPI and Swagger

Once the REST API implementation was complete, we annotated the controllers using decorators from the `@nestjs/swagger` module. This module provides Nest integration of *Swagger*[33], a tool based on the *OpenAPI* specification that allows us to generate interactive API documentation.

The documentations is available under the `/api` route.

3.4.8 GraphQL API

As proposed in Section 2.6.4, we implemented our GraphQL API using the schema-first approach.

3.4.8.1 GraphQLModule Registration

As we learned in Section 2.5.3.6, Nest provides a built-in `@nestjs/graphql` module. In order to use this module, we need to import it into `AppModule` and configure it.

We can see the configuration of this module in Snippet 13. Since we used Express as the underlying platform, we may notice that we configured the `GraphQLModule` to use the *Apollo Server*. Next, we can notice the value of the `typePaths` property which tells the module to search for GraphQL schema definitions recursively throughout the `src/` directory. The module will merge all found schemas into one. In our case, however, there is only one GraphQL schema. Finally, the `definitions` property tells the module to generate TypeScript classes based on the schema into the `src/graphql/graphql.ts` file.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  typePaths: ['./src/**/*.graphqls'],
  definitions: {
    path: join(process.cwd(), 'src', 'graphql', 'graphql.ts'),
    outputAs: 'class',
  },
}),
```

Code Snippet 13: GraphQLModule Configuration

3.4.8.2 GraphQL Schema

The schema for the GraphQL API exposed by our server can be found in the `src/graphql/schema.graphqls` file. It has the following logical structure:

[33] https://swagger.io

1. Firstly, we list the scalar types used.
2. Then we define `Entity` and `NamedEntity` interfaces.
3. After the interfaces, we define the JWT wrapper type and the input data for sign-in.
4. Then we define the types that represent the entities and the input data for mutations of these entities. Each entity implements either the `Entity` or `NamedEntity` interface.
5. Finally, we define the available queries and mutations.

Snippet 14 depicts the definition of selected GraphQL mutations.

```
type Mutation {
    signIn(content: SignInContent!): AuthPayload
    createStory(content: CreateStoryContent!): Story
    updateStory(id: Int! content: UpdateStoryContent!): Story
    deleteStory(id: Int!): Story
    deleteReadingList(id: Int!): ReadingList
    ...
}
```

Code Snippet 14: GraphQL Mutations Definition

3.4.8.3 Resolvers

As we explained in Section 1.9.5, resolvers instruct a GraphQL server how to transform a GraphQL operation defined by a GraphQL schema into data. Resolvers return data in a shape that matches the definition in the GraphQL schema. In the case of Node.js, this data is returned either synchronously or asynchronously as a promise that resolves to data of the corresponding shape.

In Section 2.5.3.6, we learned that Nest provides the `@nestjs/graphql` package that generates a resolver map automatically using the metadata that we provide by using specific decorators. We use these decorators to annotate `<name>Resolver` classes of our modules.

To turn class into a resolver, we use the `@Resolver()` decorator. This decorator accepts an optional string argument that specifies the name of the resolver. It becomes required if we use the `@ResolveField()` decorator within the resolver. The `@ResolveField()` decorator informs Nest that the decorated method is bound to the parent type. Such methods are applied as resolvers for fields that represent queries for related data. Snippet 15 illustrates how query fields in the GraphQL schema and resolver methods decorated by

`@ResolveField()` are related. We can notice the use of the `@Parent()` decorator to get the parent type of the resolved field – in this case we get the object corresponding to the `User` type from the GraphQL schema.

```
// src/graphql/schema.graphqls
type User implements Entity {
    ...
    stories(limit: Int): [Story!]!
    ...
}

// src/user/user.resolver.ts
@ResolveField('stories')
async getStories(@Parent() user, @Args('limit') limit: number) {
    const { id } = user;
    return this.storyService.findManyByAuthor(id, limit);
}
```

Code Snippet 15: GraphQL Related Data Fields and Resolvers

To map resolver class methods to individual queries, we decorate these methods using the `@Query()` decorator. Similarly, we use the `@Mutation()` decorator to map methods to mutations.

3.4.8.4 Access Restriction

We restricted the resource access using the `@UseGuards()` decorator similar to the REST API. However, the implementation differs in two ways:

1. we restrict access to actions at another stage of processing,
2. guards must obtain request data differently.

In contrast to the REST API, we do not decorate methods serving directly as route handlers, but methods serving as resolvers. Since guards are designed to secure common HTTP-based APIs, such as REST APIs, they cannot automatically retrieve the necessary data from a GraphQL query or mutation. Because of that, we need to create parallel guard classes to the classes used by our REST API. These guards retrieve the necessary request data using the `GqlExecutionContext` factory method. Here we take advantage of Nest guards that we discussed in Section 3.4.4.2.

3.5 REST Client

The implementation of our REST Client can be found in the `client-side-rest/` directory. The client is run by the `mwd-client-side-rest` service. As we decided in Section 2.5.4, we implemented the client using *React*.

3.5.1 Production Environment Setup

The PWA template provided by *Create React App* supplies us with a robust service worker registration script that ensures that it is only installed in the production build of the application. To avoid the frustration of loading cached assets instead of the latest changes, the documentation does not recommend modifying the registration script to register the service worker in the development build. [95] For this reason, the production docker environment contains a production client build.

For security reasons, browsers only allow service workers to be installed via HTTPS. This does not apply to localhost, which browsers consider to be a secure origin and do not require a secured connection to install a service worker from localhost. [106] To serve our client from localhost, we configured the production container to host the production build using the *Nginx* web server. Snippet 16 shows a fragment of the corresponding Dockerfile.

```
FROM nginx:1.21.0-alpine AS production

COPY --from=builder ... ...
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 3010
CMD ["nginx", "-g", "daemon off;"]
```

Code Snippet 16: Nginx in Docker

3.5.2 Source Code Organization

Essentially, the organization of the client source code follows the architecture presented in Section 2.7.2. The important contents of the `src/` directory are as follows:

- `features/` organizes all *React* components into JavaScript modules according to their domains. In addition, it includes the `core` module with shared, domain-independent components.

- `helpers/` provides a JavaScript module of globally applicable functions that simplify certain operations.

- `pages/` contains a JavaScript module providing the components of individual pages of the application.
- `services/` which generally bundles the service modules used by the client. In our case, it contains a service that integrates the client with the REST API.
- `stores/` contains a module that groups stores for sharing the global state. In our case, it contains the storage of the logged in user.
- `types/` provides a module of globally applicable types.
- `validations/` provides a module that encapsulates the functions used to validate user input.
- `Router.tsx` that uses *React Router v6*[34] to perform routing.
- `service-worker.ts` which implements basic offline experience required by functional requirement F17 described in Section 2.1.

3.5.3 State Management

This section presents the implementation of the state management that we proposed in Section 2.7.1. We omit URL state management, as it was already sufficiently covered in the previous chapter. We also omit server state management as it will be covered in Section3.5.4.

3.5.3.1 Local State Management

As we mentioned in Section 2.7.1.1, React functional components provide the `useState` hook to manage the local state. Snippet 17 depicts the use of the `useState` to manage the local state of the `ReadingListForm` component.

3.5.3.2 Global State Management

As we proposed in Section 2.7.1.2, we implemented global state management using the *Zustand* library. Using Zustand, we implemented the `useUserStore` hook within the stores module. This hook uses `localStorage` to store both user's JWT token and the decoded data contained in the JWT. It stores this data in the `jwt` and `user` properties and provides the `setUser` and `removeUser` functions to manipulate it.

Zustand allows us to access elements defined within the `useUserStore` hook using selector functions. Snippet 18 illustrates the use of the `useUserStore` hook and selector functions.

[34] https://reactrouter.com

```
const [state, setState] = useState<ReadingListFormState>({
  dto: {
    title: readingList ? readingList.title : '',
  },
  validation: { title: !!readingList },
  enabled: !!readingList,
});
...
setState({
  ...state,
  ...{
    dto: newDto,
    validation: newValidation,
  },
  enabled: valid,
});
```

Code Snippet 17: Local State Management With useState

```
const jwt = useUserStore(state => state.jwt);
const removeUser = useUserStore(state => state.removeUser);
```

Code Snippet 18: Manage Logged User With Zustand

3.5.4 RESTful API Integration

As we proposed in Section 2.7.2, we implemented the `rest-api-service` module that integrates the REST client with the server using its REST API. To provide end-to-end type safety to some extent, it provides clones of DTOs, entities, and envelopes implemented by the server.

The service implementation uses *React Query* which is built around the following three core concepts:

1. queries,
2. mutations,
3. query invalidation. [107]

In order for our client to use React Query, we provided our application with an object of type `QueryClient`. To achieve this, we wrapped our application in the `QueryClientProvider` component.

3.5.4.1 Queries

In React Query, queries are essentially a declarative dependency on an asynchronous request for data which are identifies with a unique key. In order to subscribe to the query, we use the `useQuery` hook with the following arguments [108]:

- a unique query key,
- a function that returns a promise that resolves the data or throws an error. [108]

The query result returned by this hook provides us with an easy access to the various states of the query, such as response data, error or loading status. [108]

Our REST API service provides a module that implements queries using the aforementioned `useQuery` hook. As the resolving function mentioned above, we use our `HttpRequest` function that wraps the `fetch` function of the client-side JavaScript. In Snippet 19, we can see the implementation of the query that fetches a story from the server.

```
export const useStory = (id: number) => {
  return useQuery<StoryEnvelope>(
    ['story', id], () =>
      HttpRequest<StoryEnvelope>(`/stories/${id}`)
  );
}
```

Code Snippet 19: Query With useQuery Hook

3.5.4.2 Mutations

Unlike loading data, React Query uses mutations to perform server-side effects such as creating, updating, or deleting data. To implement a mutation, we use the `useMutation` hook exported by React Query. This hook returns a result object that provides a mutate() method to call when needed. The result object also provides indicators of the status of the mutation request. The `useMutation` also provides us with helper options that allow us to specify side effects for mutation states, such as a callback after the mutation is successfully completed. [109] Snippet 20 shows the implementation of the mutation for sign-in.

3.5.4.3 Query Invalidation

We use query invalidation after update mutation over stories and reading lists and after assigning or removing a story from the reading list. To invalidate

```
export const useSignIn
  = (dto: SignInDto, successCallback: (data: JwtEnvelope) => void)
    => {
      return useMutation<JwtEnvelope, ErrorMessage>('signIn',
        () => HttpRequest<JwtEnvelope, SignInDto>(
          '/auth/sign-in', 'POST', dto),
          {
            onSuccess: successCallback
          }
      );
    };
...
const signIn = useSignIn(dto, signInSuccessCallback);
```

Code Snippet 20: Mutation With useMutation Hook

the query, we retrieve the client using the `useQueryClient` hook. Then we call the `invalidateQueries` method of the client. As an argument, we specify the key of the query we want to invalidate.

3.5.4.4 Using HATEOAS Links

In our REST client, we use HATEOAS links for the following purposes:

1. Retrieve links to related resources from the current resource representation and then retrieve related resources from the server and possibly creating links to navigate to them.
2. Detection of available operations for a given resource. Based on the detected operations, we render the controls in the UI.

To send requests to our REST API based on found links, we prepared a generic query `useLinkQuery` and a generic mutation `useLinkMutation`.

3.5.5 Turning REST Client Into a PWA

The Create React App template for the PWA we used to create the REST client provided us with the PWA implementation that consists of the following elements:

- `public/manifest.json` file that makes our client installable,
- `src/serviceWorkerRegistration.ts` script for conditional service worker registration,
- `src/service-worker.ts` that caches Application Shell for offline access.

To meet the functional requirement F17 defined in Section 2.1, we extended the service worker to cache content already loaded from the API for offline access. Using the *Workbox* library integrated by the PWA template, we registered the caching of API queries using the *StaleWhileRevalidate* caching strategy. In Snippet 21, we can see setting up the caching.

```
registerRoute(
  ({url}) => url.origin === ApiConfig.host,
  new StaleWhileRevalidate({
    cacheName: 'rest-api-cache',
    plugins : [
      new CacheableResponsePlugin({
        statuses: [0, 200]
      }),
    ]
  })
);
```

Code Snippet 21: Caching REST API Responses for Offline Access

3.6 GraphQL Client

The GraphQL client implementation is essentially a modified clone of the REST client. For this reason, we will only describe the parts of the implementation where they differ.

3.6.1 GraphQL API Integration

As we proposes in Section 2.8.2, we implemented the `graphql-api-service` module that integrates the GraphQL client with the server using its GraphQL API. This module exports schema-based interfaces from the server-side generated `graphql-typings.ts`. These interfaces provide end-to-end type safety to some extent.

The GraphQL API service implementation uses *Apollo Client* that allows us to declaratively fetch or manipulate server state via a GraphQL API. Another advantage of Apollo Client is that it provides caching of responses out of the box. [110] It provides us with hooks for all input types of GraphQL API:

- queries,
- mutations,
- subscriptions. [110]

In order to use Apollo Client, we created the object of type `ApolloClient` and provided our application with that object by wrapping the App component in the `ApolloProvider` component.

3.6.1.1 Queries

To execute GraphQL queries, Apollo Client supplies us with the `useQuery` hook. To run a query, we pass a GraphQL query string to this hook. At component render, the `useQuery` hook returns a result object that contains data, loading and error properties.

Using the `useQuery` hook, we implemented a custom hook for each query defined in the GraphQL schema within the `queries` module of the service. Snippet 22 depicts the implementation of the custom hook `useStoriesQuery` that fetches stories.

```
const STORIES_QUERY = gql`
  query Stories($searchString: String!, $limit: Int) {
    stories(searchString: $searchString, limit: $limit) {
      id
      createdAt
      title
      description
      author {
        id
        givenName
        familyName
      }
    }
  }
`;
...
export function useStoriesQuery(_variables: StoriesVars) {
  return useQuery<StoriesData, StoriesVars>(STORIES_QUERY, {
    variables: _variables,
  });
}
```

Code Snippet 22: GraphQL Query With Apollo Client

3.6.1.2 Mutations

In order to perform mutations, Apollo Client supplies us with the `useMutation` hook. To execute a mutation, we first need to call this hook with a mutation string as argument. [111] At component render, the `useMutation` returns a tuple that includes the following:

- a mutate function that we call when anytime we desire,
- a representation of the current status of the mutation execution similar to the result object of the `useQuery` hook. [111]

Similar to queries, we implemented a custom hook for each mutation defined in the GraphQL schema within the `mutations` module of the service.

3.6.1.3 Query Invalidation

To invalidate queries, we call the `refetchQueries` method of the Apollo Client object retrieved using the `useApolloClient` hook.

3.6.2 Turning GraphQL Client Into a PWA

Since WorkBox uses the `CacheStorage` API under the hood, which only allows caching responses to GET requests, we could not use the `registerRoute` function provided by WorkBox.

To achieve the same functionality as the REST client, we implemented custom caching using the `IndexedDB` API.

3.7 Chapter Summary

In this chapter, we described the implementation of a prototype demonstrating the use of selected client-side and server-side technologies and approaches. Especially API-first development, REST, GraphQL, Nest, React and client integration with REST and GraphQL APIs.

Chapter 4

Testing

To demonstrate the possibilities of testing the technologies used, we finished the implementation with automated tests covering selected parts of the server and the REST client. This section first describes the different types of automated tests. Then, it explores testing capabilities of the Nest framework. Finally, it briefly introduces the tools used to implement the tests and provides specific examples of implemented tests for both the server and the client.

4.1 Types of Automated Tests

There are several different types of automated tests, such as unit tests, integration tests or end-to-end (e2e) tests. Each of these tests differs in the amount of functionality tested.

Unit tests represent the lowest level of software testing. They focus on testing individual units of software. A unit is the smallest section of code that can be logically isolated. In functional programming, it is usually a function. In object-oriented programming, it is typically a method of a concrete class. The purpose of unit tests is to verify that each unit of software behaves as expected. If the unit requires interaction with other isolated units, we use so-called *mock objects* to simulate these units. [112]

Unlike unit tests, in *integration tests* we integrate software modules and test them as a whole. The purpose of integration tests is to verify that the components interact with each other as expected. [113]

End-to-end testing validates the behavior of the software as a whole, including interaction with external services such as databases. [114]

4.2 Testing the Nest Server

Our goal in testing the server implementation was to demonstrate the ability to test parts of the implementation that are crucial to both the REST and

GraphQL APIs. Before we proceed to the description of the actual implementation of the tests, let us get acquainted with the possibilities of testing Nest applications.

Although testing has undeniable benefits, it can be challenging to set up. To facilitate testing, Nest particularly:

- provides a test runner that constructs an isolated loader for modules and application,
- supplies us with dependency injection in the test environment for easy mocking,
- integrates Jest and Supertest out-of-the-box. [115]

With the features mentioned above, Nest provides the means for all of the aforementioned types of automated tests out-of-the-box.

4.2.1 Used Tools

This section briefly introduces the tools used to implement the tests, specifically the Jest[35] testing framework and the SuperTest[36] library.

Fundamentally, the *Jest* framework provides us with resources particularly suitable for writing unit tests. It allows us to create isolated tests and structure the related tests into blocks. The most basic tests using Jest involve using matchers to test values. Jest provides a variety of different matchers. [116] It also supplies us with functions to setup the environment before each test, as well as to teardown the environment after each test. [117] It is also very suitable for mocking and subsequent inspection of the mock state. [118]

The *SuperTest* library serves a different purpose compared to Jest. SuperTest allows us to test the interaction with software components by simulating HTTP requests.

4.2.2 Unit Tests

When writing unit tests, we focused on the Story domain. Specifically, we covered the controller and resolver classes in the Story Module.

To simulate the data, we prepared fixture arrays containing items of the desired types. Since both `StoryController` and `StoryResolver` classes require service classes from the application logic layer as a dependency, we created mocks for these services using the Jest framework. In Snippet 23, we can see calling the `jest.fn()` function to mock `findMany` and `update` methods of the `StoryService` class.

[35]https://github.com/facebook/jest
[36]https://github.com/visionmedia/supertest

```
  const storyService = {
    findMany: jest.fn().mockResolvedValue(storiesFixture),
    ...
    update: jest.fn(async (_id: number, dto: UpdateStoryDto) => {
      const story = storiesFixture.find((s) => s.id === _id);
      if (story) {
        return {...story, ...dto};
      }
      throw new NotFoundException();
    }),
    ...
  };
```

Code Snippet 23: Mocking the StoryService Class

We use the `beforeEach` function provided by Jest to setup the environment before starting each test. We use the testing utilities provided by the `Test` class of the `@nestjs/testing` package to setup the environment. Snippet 24 depicts the usage of the `Test` class to mock the full Nest runtime in order to test the `StoryResolver` class in isolation.

```
const moduleRef: TestingModule = await Test.createTestingModule({
  providers: [StoryResolver],
})
  .useMocker((token) => {
    if (token === StoryService) {
      return storyService;
    }
    if (token === UserService) {
      return userService;
    }
    ...
  })
  .compile();

storyResolver = moduleRef.get(StoryResolver);
```

Code Snippet 24: Using the Test Class to Mock the Nest Runtime

Since we introspect mock object statistics in the tests, we reset these statistics after each test using the `afterEach` function. We organize the tests using the `describe` function in groups according to which method they belong to. In our tests, we use `toBe` and `toStrictEqual` matchers to compare expected and received values.

To test the interaction of the tested method with mock objects, we use the `mock` attribute of mocked methods. This attribute contains an array of calls of the method and an array of arguments for each call. Snippet 25 depicts the implementation of two unit tests for `getStory` method of the `StoryResolver` class.

```
describe('Testing the getStory() method.', () => {
  test('Should return story entity.', async () => {
    expect(await storyResolver.getStory(1))
      .toBe(storiesFixture.find((s) => s.id === 1));
    expect(storyService.findOneById.mock.calls.length).toBe(1);
    expect(storyService.findOneById.mock.calls[0][0]).toBe(1);
  });
  test('Should throw NotfoundException.', async () => {
    await expect(storyResolver.getStory(100))
      .rejects
      .toStrictEqual(new NotFoundException());
    expect(storyService.findOneById.mock.calls.length).toBe(1);
    expect(storyService.findOneById.mock.calls[0][0]).toBe(100);
  });
});
```

Code Snippet 25: Jest Unit Tests

4.2.3 Integration Tests

After we covered `StoryController` and `StoryResolver` with unit tests, we also covered the domain with integration tests.

As in the case of the unit tests, we prepared fixture arrays to simulate the data for the integration tests. In contrast to unit tests, we did not create mocks for individual services that the tested classes require as dependencies. Instead, we mocked the `PrismaService` class on which the mentioned services depend.

For the purpose of setting up the environment for individual tests, we again take advantage of the `beforeEach` function provided by Jest.

Compared to unit tests, we mock the full Nest runtime with our entire application, not just the tested class. When mocking the runtime, we specify which providers we want to replace with the prepared mock objects. Finally, we create and run the Nest application. We can see this process in Snippet 26.

We organize the tests using the `describe` function in groups according to which RESTful endpoint or GraphQL query/mutation they belong to. To test the RESTful enpoint, we perform the following steps:

1. We define the expected response data.

```
    const moduleFixture: TestingModule = await Test
      .createTestingModule({
          imports: [AppModule],
      })
      .overrideProvider(PrismaService)
      .useValue(prismaService)
      .overrideProvider(ConfigService)
      .useValue(configService)
      .compile();

    app = moduleFixture.createNestApplication();
    await app.init();
```

Code Snippet 26: Mocking the Nest Runtime for Integration Tests

2. We decorate the underlying Express HTTP server of the Nest application using the `request` function of the SuperTest library.

3. We fluently describe the test request using the SuperTest methods.

4. We use the `expect` method of the SuperTest library to compare the expected and the received response in terms of the HTTP status code and the body.

Snippet 27 provides an example of the integration test for the RESTful endpoint that provides the create story operation.

```
test('Should respond with 403 Forbidden.', async () => {
  return request(app.getHttpServer())
    .post('/stories')
    .send(createDto)
    .set('Accept', 'application/json')
    .set('Authorization', 'Bearer ' + user4Jwt)
    .expect(403)
    .expect({ statusCode: 403, message: 'Forbidden' });
});
```

Code Snippet 27: RESTful Endpoint Integration Test

Integration tests for GraphQL queries and mutations differ as follows:

- We always test the POST `/graphql` request.

- We always set the request body to the appropriate query or mutation.

4.3 Testing the REST Client

After we finished testing the server, we implemented tests for the REST client. Our goal in testing the client implementation was to demonstrate the DOM testing for React components. Since React components are isolated units, as we described in Section 1.5.3.1, we test them using unit tests.

For testing we used the *Testing Library*[37] implementation for React[38] in combination with the Jest framework described in Section 4.2.1. The React Testing Library allows us to use its render function to render a React component into the container over which the library operates. The result object of this function call contains the container itself and a number of hooks to query the contents of the container.

To demonstrate the DOM testing, we limited ourselves to testing the rendering of the save button in the `FullscreenDialog` component. This button can be in the following three different states:

1. unrendered if the dialog is not open,
2. rendered and disabled if the dialog content is not in a validated state,
3. rendered and enabled if the dialog content is in a validated state.

Snippet 28 depicts the unit test that implements DOM testing for the `FullscreenDialog`. We can notice using the `queryByText` query hook provided by the React Testing Library. We can also notice the use of Jest matchers `toBeDefined`, `toBeVisible` and `toBeDisabled` to evaluate the button state.

```
test('Test the disabled save button.', () => {
  const {queryByText} = render(
    <FullscreenDialog
      isOpened={true}
      actionEnabled={false}
    />
  );
  const saveBtn = queryByText('save');
  expect(saveBtn).toBeDefined();
  expect(saveBtn).toBeVisible();
  expect(saveBtn).toBeDisabled();
});
```

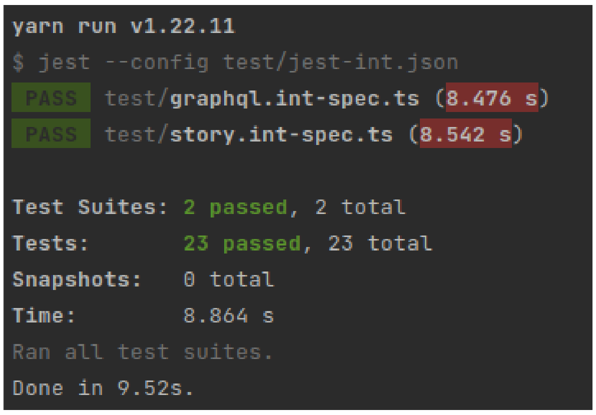
Code Snippet 28: React DOM Testing

[37]https://github.com/testing-library
[38]https://github.com/testing-library/react-testing-library

4.4 Chapter Summary

This section explored the testing capabilities of the technologies used, described the tools used for testing and finally provided an overview and examples of the implemented tests.

In the server implementation, we covered `StoryController` and `StoryResolver` with unit and integration tests. `StoryController` exposes five RESTful endpoints for CRUD operations and searching over the `Story` entity. In comparison, `StoryResolver` provides six resolver methods, since it additionally implements the resolving of the story author. Over these two classes, we implemented a total of 28 unit tests and 23 integration tests. The tests verify the correct behavior of both classes and can easily detect errors caused by possible modifications to the implementations. Figure 4.1 illustrates the results of the implemented integration tests.



```
yarn run v1.22.11
$ jest --config test/jest-int.json
 PASS  test/graphql.int-spec.ts (8.476 s)
 PASS  test/story.int-spec.ts (8.542 s)

Test Suites: 2 passed, 2 total
Tests:       23 passed, 23 total
Snapshots:   0 total
Time:        8.864 s
Ran all test suites.
Done in 9.52s.
```

Figure 4.1: Integration Tests Results

In the REST Client implementation, we demonstrated DOM testing using the unit tests for `FullscreenDialog` component. In total, we implemented 3 tests that verify the rendering of the button to trigger the dialog action. This verified that the user cannot run an action under conditions where it should not be allowed.

CHAPTER 5

Evaluation

The last chapter evaluates the practical outputs of the thesis in terms of implementation, extendability, sustainability, and testability.

5.1 Server

In this section, we evaluate the implemented server from the aforementioned perspectives.

5.1.1 Implementation

Right at the beginning of the implementation, the Nest framework provided us with a basic TypeScript application with a robust architecture. The implementation is object-oriented, modular and makes heavy use of JavaScript decorators. The modules encapsulate the implementation of the associated logic. Nest allows us to import other modules into this module as part of the module declaration and also specify what specific logic the module exposes to other modules.

Despite the fact that Nest is based on a number of design patterns and often uses advanced Node.js constructs, we found the learning curve surprisingly moderate. We owe this to the fact that the framework is both opinionated and well documented. The use of a range of well documented built-in decorators and integration modules for the technologies used helped to improve our productivity.

Let us note that we spent more effort implementing authentication for GraphQL than for REST because we had to retrieve the request data using the GraphQL context.

5.1.2 Extendability

As the previous section suggests, the framework leads us to create extensible applications primarily using modularization and OOP principles such as DRY or separation of concerns. The proof of Extendability is that the initial implementation of the server exposed only the REST API and the GraphQL API was implemented later. This was achieved by simply registering the built-in module and creating new classes that did not require a single modification to the existing solution.

5.1.3 Sustainability

By following OOP principles and using TypeScript, the implementation is sustainable. In particular, TypeScript and the implemented tests contribute to a faster understanding of existing solutions and safer modification of the implementation.

5.1.4 Testability

We have demonstrated through implemented tests that our server can be easily tested with unit and integration tests. Furthermore, the server could be tested using end-to-end tests if we modified the integration tests to use a test database.

5.2 REST API

This section evaluates the implementation of the REST API that is exposed by the server evaluated in the previous section.

5.2.1 Implementation

The REST API implementation takes advantage of Nest controllers and available built-in decorators for declaring endpoints, route parameters, HTTP methods and status codes. We also make use of the built-in decorator to obtain request body where needed. For the needs of our REST API and to comply with the DRY principle, we implemented our own decorators to get the authenticated user from the request, get the JWT and also the query parameter to limit the number of records in the response that contains a collection of resources.

Overall, we implemented most of the basic form of the REST API declaratively using the built-in Nest decorators. To comply with HATEOAS, we implemented our own solution for linking resources. This cost us much more effort and overhead.

5.2.2 Extendability

We can easily extend the existing controllers with additional decorated route handlers, or we can create and integrate a new module containing a controller into the application.

5.2.3 Sustainability

The sustainability of a REST API implementation is a consequence of the sustainability of the server implementation that exposes it.

5.2.4 Testability

Since the server tests we implemented also covered a RESTful controller in terms of both unit and integration tests, we demonstrated that our REST API is easy to test.

5.3 GraphQL API

This section evaluates the implementation of the GraphQL API exposed by our server.

5.3.1 Implementation

Since we used Express as the underlying platform for Nest, the GraphQL API implementation takes advantage of Nest's integration of the Apollo Server. Since we chose a schema first approach for the implementation, we had to learn how to use GraphQL Schema Definiton Language to define the schema of our API. However, defining a GraphQL schema is easy to learn.

Since Nest provides a built-in package for GraphQL integration, all we had to do after defining the schema was to configure the GraphQL module to use this schema. We also configured this module to generate TypeScript classes corresponding to the schema at the server start. Thanks to this, we got free typing. For the implementation of resolvers, we again used the mentioned Nest package. The package allows us to use decorators to declaratively map methods of resolver classes to the corresponding queries, mutations and query fields of types.

5.3.2 Extendability

We can easily extend the implementation by modifying the GraphQL schema and then modifying existing resolvers, or by implementing new resolvers mapped to new elements of the GraphQL schema. The automatic generation of TypeScript classes ensures that we automatically have typing available for new elements.

5.3.3 Sustainability

The sustainability of the resolver implementation is a consequence of the sustainability of the server implementation.

Furthermore, the GraphQL schema provides us with easy-to-understand documentation for the GraphQL API. In addition, the schema serves as a contract between back-end and front-end developers. This makes it easier and more efficient to integrate client applications with any changes to the GraphQL API.

5.3.4 Testability

The implemented unit tests cover the selected resolver class. Furthermore, the implemented integration tests cover queries and mutations that use the methods of the class covered by the unit tests for the resolving process. This demonstrated the testability of our GraphQL API.

5.4 Clients

This section evaluates the implementation of both REST and GraphQL clients that demonstrate the integration of client-side applications with our server.

5.4.1 Implementation

We used the React library to implement the clients. React made it easy to split the application into isolated components. React is unopinionated. Because of this, we had trouble figuring out how to organize the application in a way that made sense and allowed for easy and meaningful integration of any additional components and their tests. In addition, we spent a long time deciding which query library to use to integrate the REST client with the REST API. However, after overcoming the initial problems, we implemented logically organized client applications.

5.4.1.1 REST Client

For communication with the REST API, we decided to use the React Query library, which simplifies handling the request status and implements automatic refetching when the user returns to the browser tab with the application.

We encountered the typical problem of underfetching and overfetching. In order to obtain all required data, we send requests for the following data:

1. basic reading list data,

2. data about the author,

3. list of contained stories,

4. basic data for each contained story.

We also encountered the need to implement a low-level HTTP requests construct using the fetch API. The implementation of the offline experience according to the F17 functional requirement was trivial in this case thanks to the Workbox library.

5.4.1.2 GraphQL Client

When integrating with the GraphQL API using the Apollo Client, we did not encounter the problem of underfetching or overfetching due to the nature of GraphQL. Furthermore, we did not have to deal with the HTTP requests construction.

Since we communicate with the GraphQL API exclusively via POST requests, it was problematic to implement caching for offline access, since the Cache API used by the Workbox library allows only GET requests to be cached.

5.4.2 Extendability

Both client applications are extensible because they are built with isolated components that interact with each other through defined interfaces. When extending with complex components requiring global state management, we recommend implementing additional stores using the Zustand library.

5.4.3 Sustainability

Since we used TypeScript for the implementation, we significantly increased the maintainability compared to a React application developed with JavaScript.

5.4.4 Testability

By implementing unit tests we demonstrated the testability of individual application components. The application could be further tested using integration tests, where we would test multiple components as a whole.

Conclusion

This thesis aimed to study and analyze web application development in client-side and server-side domains with a particular focus on REST, GraphQL and PWA. Furthermore, the thesis aimed to design and implement prototype web application demonstrating the use of appropriate technologies and approaches in the aforementioned domains. A secondary goal was to explore the types of web applications from different perspectives.

We started with an introduction to web application development, exploring types of web applications from four different perspectives. We also created a diagram showing how they are related. Subsequently, we provided a broad theoretical analysis of technologies and approaches used in client-side and server-side web development. We gradually progressed from general topics to REST, GraphQL and PWA, which we analyzed thoroughly.

We then focused on designing a prototype web application to demonstrate the use of some of the analyzed technologies and approaches. We defined functional and non-functional requirements so that the prototype could demonstrate the features of both the REST and GraphQL APIs and at the same time the clients had to implement the functionality using the service worker. During the design process, we focused on selecting a suitable server-side framework that allows easy implementation of both REST and GraphQL APIs. We also focused on choosing a client-side framework that would allow us to easily create a PWA. We also emphasized that the resulting implementation should be scalable and sustainable. Finally, we designed a specific solution for the server and client applications.

The design was directly followed by the implementation part of the thesis. The design was directly followed by the implementation part of the thesis, where we described the Docker environment created to run the project and the implementation of the server and clients. The implementation part of the thesis directly followed the design. We described the Docker environment we created to run the project, as well as the implementation of the server and clients.

Then, we subjected selected parts of the prototype components to automated tests at the level of unit tests and integration tests.

Finally, we evaluated the individual components of the prototype, and thus the technology used to implement them, in terms of quality and difficulty of implementation, extendability, sustainability and testability. Based on the evaluation, we came to the conclusion that the implemented prototype is extensible, sustainable and testable.

We have shown by implementation that it is easy to implement identical APIs using both REST and GraphQL. Both approaches have advantages and disadvantages. We decided to implement REST, including HATEOAS, which cost us an extra effort. However, we have shown that links, as part of a resource representation, can move business logic related to, for example, evaluating a user's permissions to actions on a server.

We have also demonstrated that GraphQL gives clients more flexibility and actually eliminates both underfetching and overfetching. However, we have found, for example, that caching for offline access is more difficult when using the GraphQL API.

The main practical output of this thesis, ie a prototype of a web application, could be further expanded. For example, we could implement user registration, paging for collections, or a subscription and notification system, for example. At the same time, it would be appropriate to maintain the prototype in the future to keep pace with the development of the technologies used.

Bibliography

[1] Sacha Greif, with help from a team of open-source contributors and consultants. The State of JS 2021 [online]. 2022, [2022-03-05]. Available from: https://2021.stateofjs.com

[2] Sacha Greif, with help from a team of open-source contributors and consultants. The State of JS 2021: Back-end Frameworks [online]. 2022, [2022-03-05]. Available from: https://2021.stateofjs.com/en-US/libraries/back-end-frameworks

[3] Sacha Greif, with help from a team of open-source contributors and consultants. The State of JS 2021: Front-end Frameworks [online]. 2022, [2022-03-05]. Available from: https://2021.stateofjs.com/en-US/libraries/front-end-frameworks

[4] Indeed Editorial Team. Website vs. Web Application (App): What's the Difference? [online]. August 2021, [Cited 2022-04-09]. Available from: https://www.indeed.com/career-advice/career-development/website-vs-web-application

[5] Indeed Editorial Team. What Is a Web Application? How It Works, Benefits and Examples [online]. November 2021, [Cited 2022-01-31]. Available from: https://www.indeed.com/career-advice/career-development/what-is-web-application

[6] Mozilla and individual contributors. Introduction to the server side - Learn web development [online]. February 2022, [Cited 2022-02-03]. Available from: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction

[7] Singh, E. Client-Server Architecture [online]. August 2021, [Cited 2022-02-09]. Available from: https://medium.com/codex/client-server-architecture-5e103aa0106d

[8] Mendez, N. Understanding Rendering in Web Apps: Intro [online]. January 2021, [Cited 2022-02-12]. Available from: `https://dev.to/snickdx/understanding-rendering-in-web-apps-intro-21cl`

[9] Mendez, N. Understanding Rendering in Web Apps: SSR [online]. January 2021, [Cited 2022-02-12]. Available from: `https://dev.to/snickdx/understanding-rendering-in-web-apps-ssr-1h83`

[10] Mendez, N. Understanding Rendering in Web Apps: CSR [online]. January 2021, [Cited 2022-02-12]. Available from: `https://dev.to/snickdx/understanding-rendering-in-web-apps-csr-354d`

[11] Khalifa, Z. Multi-page, Single-page, or a Hybrid? [online]. June 2020, [Cited 2022-02-12]. Available from: `https://medium.com/swlh/spa-mpa-or-a-hybrid-42fdf6b3415c`

[12] Mendez, N. Understanding Rendering in Web Apps: SPA vs MPA [online]. January 2021, [Cited 2022-02-12]. Available from: `https://dev.to/snickdx/understanding-rendering-in-web-apps-spa-vs-mpa-49ef`

[13] Ltd, A. B. Single Page Application (SPA) vs Multi Page Application (MPA) – Two Development Approaches [online]. November 2019, [Cited 2022-02-12]. Available from: `https://asperbrothers.com/blog/spa-vs-mpa/`

[14] Akhtar, J. Microservices Introduction (Monolithic vs. Microservice Architecture) - DZone Microservices [online]. December 2018, [Cited 2022-02-13]. Available from: `https://dzone.com/articles/microservices-1-introduction-monolithic-vs-microse`

[15] Molnar, Z. Decoupled architecture: how to modernise your frontend [online]. April 2020, [Cited 2022-02-13]. Available from: `https://inviqa.com/blog/decoupled-architecture-how-modernise-your-frontend`

[16] Geers, M. Micro Frontends - extending the microservice idea to frontend development [online]. [Cited 2022-02-16]. Available from: `https://micro-frontends.org`

[17] Budiu, R. Mobile Websites: Mobile-Dedicated, Responsive, Adaptive, or Desktop Site? [online]. February 2016, [Cited 2022-02-16]. Available from: `https://www.nngroup.com/articles/mobile-vs-responsive`

[18] Mozilla and individual contributors. Introduction to progressive web apps - Progressive web apps (PWAs) [online]. February 2022, [Cited 2022-02-23]. Available from: `https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction`

[19] Sacha Greif, with help from a team of open-source contributors and consultants. The State of JS 2021: Features [online]. 2022, [2022-04-10]. Available from: https://2021.stateofjs.com/en-US/features/other-features

[20] W3Techs. Usage Statistics of JavaScript as Client-side Programming Language on Websites, March 2022 [online]. [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Glossary/CSS_preprocessor

[21] Stack Overflow. Stack Overflow Developer Survey 2021 [online]. 2021, [Cited 2022-03-06]. Available from: https://insights.stackoverflow.com/survey/2021

[22] Sacha Greif, with help from a team of open-source contributors and consultants. The State of JS 2021: Other Tools [online]. 2022, [2022-04-10]. Available from: https://2021.stateofjs.com/en-US/other-tools

[23] Mozilla and individual contributors. About JavaScript [online]. July 2021, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript

[24] Mozilla and individual contributors. Introduction [online]. February 2022, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction

[25] Mozilla and individual contributors. JavaScript [online]. July 2021, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[26] Mozilla and individual contributors. JavaScript technologies overview [online]. January 2022, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/JavaScript_technologies_overview

[27] Branscombe, M. JavaScript 6 Offers Big Changes, and Kicks Off an Expedited Timetable [online]. 2016, [Cited 2022-03-06]. Available from: https://thenewstack.io/ecmascript-6-biggest-update-javascript-yet-start-rolling-annual-improvements

[28] Wang, S. The State of JS 2021: Conclusion [online]. 2022, [2022-04-10]. Available from: https://2021.stateofjs.com/en-US/conclusion

[29] Microsoft and TypeScript Community. TypeScript: Documentation - TypeScript for the New Programmer [online]. 2022, [Cited 2022-04-10].

Available from: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html

[30] Mozilla and individual contributors. General asynchronous programming concepts [online]. January 2022, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Concepts

[31] Mozilla and individual contributors. Introducing asynchronous JavaScript [online]. February 2022, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing

[32] Mozilla and individual contributors. Getting started with HTML [online]. February 2022, [Cited 2022-02-23]. Available from: https://medium.com/@dennis.pintilie.alexandru/separation-of-concerns-soc-fd72b0191b1f

[33] Mozilla and individual contributors. What is CSS? [online]. February 2022, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS

[34] Mozilla and individual contributors. CSS preprocessor [online]. October 2021, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Glossary/CSS_preprocessor

[35] Mozilla and individual contributors. Introduction to web APIs [online]. February 2022, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction

[36] Mozilla and individual contributors. Document Object Model (DOM) [online]. December 2021, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

[37] Mozilla and individual contributors. Fetch API [online]. February 2022, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

[38] Mozilla and individual contributors. Web Storage API [online]. February 2022, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

[39] Mozilla and individual contributors. The event loop [online]. February 2022, [Cited 2022-02-23]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

[40] Wiredelta. 10 Most Popular Web Development Trends of 2022 [online]. 2022, [Cited 2022-03-05]. Available from: https://wiredelta.com/10-most-popular-web-development-trends-of-2022

[41] Sulikowska, M. 22 Web Development Trends for 2022 [online]. 2022, [Cited 2022-03-05]. Available from: https://naturaily.com/blog/22-web-development-trends-for-2022

[42] Global Media Insight. 23 Latest Web Development Trends to Follow in 2022 [online]. 2021, [Cited 2022-03-05]. Available from: https://www.globalmediainsight.com/blog/web-development-trends

[43] StatCounter. Desktop vs Mobile vs Tablet Market Share Worldwide [online]. [Cited 2022-03-05]. Available from: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet

[44] Mozilla and individual contributors. Mobile first - Progressive web apps (PWAs) [online]. March 2022, [Cited 2022-03-05]. Available from: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Responsive/Mobile_first

[45] Mozilla and individual contributors. Introduction to client-side frameworks [online]. February 2022, [Cited 2022-02-28]. Available from: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction

[46] You, Y. vuejs/vue: Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web. [online]. 2022, [Cited 2022-04-10]. Available from: https://github.com/vuejs/vue

[47] Meta Platforms, Inc. React – A JavaScript library for building user interfaces [online]. 2022, [Cited 2022-02-28]. Available from: https://reactjs.org

[48] Meta Platforms, Inc. Introducing JSX [online]. 2022, [Cited 2022-02-28]. Available from: https://reactjs.org/docs/introducing-jsx.html

[49] GeeksforGeeks and individual contributors. ReactJS Data Binding [online]. May 2021, [Cited 2022-03-01]. Available from: https://www.geeksforgeeks.org/reactjs-data-binding

[50] Lvova, E. Best JavaScript Framework in 2021: React vs. Vue [online]. March 2021, [Cited 2022-02-28]. Available from: https://dzone.com/articles/react-vs-vue-in-2021-best-javascript-framework

[51] Mozilla and individual contributors. Getting started with React [online]. February 2022, [Cited 2022-02-28]. Available from: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started

[52] Evan You and the team. Introduction [online]. [Cited 2022-02-28]. Available from: https://vuejs.org/guide/introduction.html

[53] Mozilla and individual contributors. Getting started with Vue [online]. February 2022, [Cited 2022-02-28]. Available from: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Vue_getting_started

[54] Tatwa, T. N. Vue.js Two Way Binding Model [online]. March 2021, [Cited 2022-03-01]. Available from: https://www.geeksforgeeks.org/vue-js-two-way-binding-model

[55] Evan You and the team. Components Basics [online]. [Cited 2022-02-28]. Available from: https://vuejs.org/guide/essentials/component-basics.html

[56] Martin, S. Mobile App Vs. Mobile Website: Which is the Best Choice in 2021? [online]. June 2021, [Cited 2022-02-16]. Available from: https://javascript.plainenglish.io/mobile-app-vs-mobile-website-which-is-the-best-choice-in-2021-25cb9a53ec47

[57] Google Web Fundamentals. Introduction to Progressive Web App Architectures [online]. February 2021, [Cited 2022-02-24]. Available from: https://developers.google.com/web/ilt/pwa/introduction-to-progressive-web-app-architectures

[58] Chrome Developers and Contributors. Service worker overview [online]. September 2021, [Cited 2022-02-24]. Available from: https://developer.chrome.com/docs/workbox/service-worker-overview

[59] Chrome Developers and Contributors. A service worker's life [online]. September 2021, [Cited 2022-04-11]. Available from: https://developer.chrome.com/docs/workbox/service-worker-lifecycle

[60] Mozilla and individual contributors. How to make PWAs installable - Progressive web apps (PWAs) [online]. February 2022, [Cited 2022-02-24]. Available from: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Installable_PWAs

[61] OpenJS Foundation and Contributors. Introduction to Node.js [online]. [Cited 2022-03-07]. Available from: https://nodejs.dev/learn/introduction-to-nodejs

[62] OpenJS Foundation. The Node.js Event Loop, Timers, and process.nextTick() [online]. [Cited 2022-03-07]. Available from: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick

[63] Megida, D. A deep dive into queues in Node.js [online]. June 2020, [Cited 2022-04-12]. Available from: https://blog.logrocket.com/a-deep-dive-into-queues-in-node-js

[64] Python Software Foundation. 1. Whetting Your Appetite [online]. April 2022, [Cited 2022-04-12]. Available from: https://docs.python.org/3/tutorial/appetite.html

[65] Korduba, Y. NodeJS vs Python: Choosing the Best Technology to Develop Back-End of Your Web App [online]. October 2021, [Cited 2022-04-12]. Available from: https://keenethics.com/blog/nodejs-vs-python

[66] Potter, J. @nestjs/core vs fastify vs next vs nuxt vs strapi [online]. [Cited 2022-04-12]. Available from: https://www.npmtrends.com/@nestjs/core-vs-fastify-vs-next-vs-nuxt-vs-strapi

[67] Mysliwiec, K. Documentation [online]. [Cited 2022-03-08]. Available from: https://docs.nestjs.com

[68] Patel, R. Top Node.js Frameworks to use in 2022 [online]. March 2021, [Cited 2022-03-08]. Available from: https://javascript.plainenglish.io/top-node-js-frameworks-to-use-in-2021-4951ee5940b8

[69] MuleSoft LLC. Types of APIs and how to determine which to build [online]. 2022, [Cited 2022-04-12]. Available from: https://www.mulesoft.com/resources/api/types-of-apis

[70] Massé, M. *REST API Design Rulebook*. Sebastopol: O'Reilly Media, 2012, ISBN 978-144-9310-509.

[71] Gupta, L. What is REST - REST API Tutorial [online]. April 2022, [Cited 2022-04-12]. Available from: https://restfulapi.net

[72] Scharhag, M. HTTP methods: Idempotency and Safety [online]. February 2020, [Cited 2022-04-12]. Available from: https://www.mscharhag.com/api-design/http-idempotent-safe

[73] Gupta, L. HTTP Content Negotiation in REST APIs [online]. September 2021, [Cited 2022-03-13]. Available from: https://restfulapi.net/content-negotiation

[74] Gupta, L. Caching REST API Response [online]. January 2022, [Cited 2022-03-13]. Available from: https://restfulapi.net/content-negotiation

[75] Gupta, L. HATEOAS Driven REST APIs [online]. October 2021, [Cited 2022-03-13]. Available from: https://restfulapi.net/hateoas

[76] Nottingham, M. RFC 8288 - Web Linking [online]. October 2017, [Cited 2022-04-13]. Available from: https://datatracker.ietf.org/doc/html/rfc8288

[77] Gupta, L. REST API Security Essentials [online]. October 2021, [Cited 2022-03-14]. Available from: https://restfulapi.net/security-essentials

[78] Gupta, L. What is API Versioning in REST? [online]. October 2021, [Cited 2022-03-14]. Available from: https://restfulapi.net/versioning

[79] Prisma and the GraphQL community. GraphQL vs REST - A comparison [online]. [Cited 2022-04-14]. Available from: https://www.howtographql.com/basics/1-graphql-is-the-better-rest

[80] Prisma and the GraphQL community. Learn GraphQL Fundamentals with Fullstack Tutorial [online]. [Cited 2022-04-14]. Available from: https://www.howtographql.com/basics/0-introduction

[81] The GraphQL Foundation. Introduction to GraphQL [online]. 2022, [Cited 2022-04-14]. Available from: https://www.howtographql.com/basics/1-graphql-is-the-better-rest

[82] Facebook, Inc. and GraphQL contributors. GraphQL [online]. October 2021, [Cited 2022-04-14]. Available from: https://spec.graphql.org/October2021

[83] The GraphQL Foundation. Schemas and Types [online]. 2022, [Cited 2022-04-15]. Available from: https://graphql.org/learn/schema

[84] Prisma and the GraphQL community. GraphQL Core Concepts Tutorial [online]. [Cited 2022-04-16]. Available from: https://www.howtographql.com/basics/2-core-concepts

[85] Prisma and the GraphQL community. GraphQL Architecture & Big Picture [online]. [Cited 2022-04-18]. Available from: https://www.howtographql.com/basics/3-big-picture

[86] Potter, J. apollo-server vs express-graphql vs mercurius [online]. April 2022, [Cited 2022-04-20]. Available from: https://www.npmtrends.com/apollo-server-vs-express-graphql-vs-mercurius

[87] The PostgreSQL Global Development Group. PostgreSQL: About [online]. 2022, [Cited 2022-04-19]. Available from: https://www.postgresql.org/about

[88] StrongLoop, IBM, and other expressjs.com contributors. Express routing [online]. 2017, [Cited 2022-04-19]. Available from: https://expressjs.com/en/guide/routing.html

[89] The GraphQL Foundation. Running an Express GraphQL Server [online]. 2022, [Cited 2022-04-20]. Available from: https://graphql.org/graphql-js/running-an-express-graphql-server

[90] Apollo Graph Inc. Choosing an Apollo Server package [online]. [Cited 2022-04-20]. Available from: https://www.apollographql.com/docs/apollo-server/integrations/middleware

[91] Vercel, Inc. and Contributors. API Routes: Introduction [online]. 2022, [Cited 2022-04-20]. Available from: https://nextjs.org/docs/api-routes/introduction

[92] Mysliwiec, K. Controllers [online]. [Cited 2022-04-20]. Available from: https://docs.nestjs.com/controllers

[93] Mysliwiec, K. GraphQL + TypeScript [online]. [Cited 2022-04-20]. Available from: https://docs.nestjs.com/graphql/quick-start

[94] Mysliwiec, K. Providers [online]. [Cited 2022-04-23]. Available from: https://docs.nestjs.com/providers

[95] Facebook, Inc. Making a Progressive Web App [online]. 2022, [Cited 2022-04-20]. Available from: https://create-react-app.dev/docs/making-a-progressive-web-app

[96] Evan You and the team. @vue/cli-plugin-pwa [online]. February 2022, [Cited 2022-04-21]. Available from: https://cli.vuejs.org/core-plugins/pwa.html

[97] Prisma and Contributors. What is Prisma? (Overview) [online]. April 2022, [Cited 2022-04-21]. Available from: https://www.prisma.io/docs/concepts/overview/what-is-prisma

[98] Prisma and Contributors. Schema Prototyping [online]. April 2022, [Cited 2022-04-27]. Available from: https://www.prisma.io/docs/concepts/components/prisma-migrate/db-push

[99] Prisma and Contributors. Why Prisma? Comparison with SQL query builders & ORMs [online]. August 2021, [Cited 2022-04-22]. Available from: https://www.prisma.io/docs/concepts/overview/why-prisma

[100] Mysliwiec, K. Modules [online]. [Cited 2022-04-23]. Available from: `https://docs.nestjs.com/modules`

[101] Mysliwiec, K. GraphQL + TypeScript - Resolvers [online]. [Cited 2022-04-23]. Available from: `https://docs.nestjs.com/graphql/resolvers`

[102] Barger, R. How to Manage State in Your React Apps [online]. February 2022, [Cited 2022-04-25]. Available from: `https://www.freecodecamp.org/news/how-to-manage-state-in-your-react-apps`

[103] Meta Platforms, Inc. Lifting State Up [online]. 2022, [Cited 2022-04-25]. Available from: `https://reactjs.org/docs/lifting-state-up.html`

[104] Mysliwiec, K. Authentication [online]. [Cited 2022-04-27]. Available from: `https://docs.nestjs.com/security/authentication`

[105] Mysliwiec, K. Guards [online]. [Cited 2022-04-27]. Available from: `https://docs.nestjs.com/guards`

[106] Mozilla and individual contributors. Using Service Workers [online]. April 2022, [Cited 2022-05-02]. Available from: `https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers`

[107] Tanner Linsley and Contributors. Quick Start [online]. [Cited 2022-04-30]. Available from: `https://react-query.tanstack.com/quick-start`

[108] Tanner Linsley and Contributors. Queries [online]. [Cited 2022-04-30]. Available from: `https://react-query.tanstack.com/guides/queries`

[109] Tanner Linsley and Contributors. Mutations [online]. [Cited 2022-04-30]. Available from: `https://react-query.tanstack.com/guides/mutations`

[110] Apollo Graph Inc. Why Apollo Client? [online]. [Cited 2022-04-30]. Available from: `https://www.apollographql.com/docs/react/why-apollo`

[111] Apollo Graph Inc. Mutations in Apollo Client [online]. [Cited 2022-04-30]. Available from: `https://www.apollographql.com/docs/react/data/mutations`

[112] Hamilton, T. Unit Testing Tutorial: What is, Types, Tools & Test EXAMPLE [online]. April 2022, [Cited 2022-05-02]. Available from: `https://www.guru99.com/unit-testing-guide.html`

[113] Hamilton, T. Integration Testing: What is, Types, Top Down & Bottom Up Example [online]. April 2022, [Cited 2022-05-02]. Available from: `https://www.guru99.com/integration-testing.html`

[114] Hamilton, T. END-To-END Testing Tutorial: What is E2E Testing with Example [online]. April 2022, [Cited 2022-05-02]. Available from: `https://www.guru99.com/end-to-end-testing.html`

[115] Mysliwiec, K. Testing [online]. [Cited 2022-05-02]. Available from: `https://docs.nestjs.com/fundamentals/testing`

[116] Facebook, Inc. and Contributors. Using Matchers [online]. April 2022, [Cited 2022-05-02]. Available from: `https://jestjs.io/docs/using-matchers`

[117] Facebook, Inc. and Contributors. Setup and Teardown [online]. April 2022, [Cited 2022-05-02]. Available from: `https://jestjs.io/docs/setup-teardown`

[118] Facebook, Inc. and Contributors. Mock Functions [online]. April 2022, [Cited 2022-05-02]. Available from: `https://jestjs.io/docs/mock-functions`

Appendix A

Acronyms

REST Representational State Transfer

PWA Progressive Web Application

CSS Cascading Style Sheets

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

URL Uniform Resource Locator

URI Uniform Resource Identifier

SSR Server-Side Rendering

CSR Client-Side Rendering

MPA Multi-Page Application

SPA Single-Page Application

JS JavaScript

API Application Programming Interface

DOM Document Object Model

UI User Interface

JSX JavaScript XML

JSON JavaScript Object Notation

OS Operating System

I/O Input/Output

FIFO First In, First Out

TCP Transmission Control Protocol

TLS Transport Layer Security

SEO Search Engine Optimization

RPC Remote Procedure Call

CRUD Create, Read, Update, Delete

MDN Mozilla Developer Network

HATEOAS Hypermedia as the Engine of Application State

SDL Schema Definition Language

CLI Command Line Interface

SQL Structured Query Language

ORM Object-Relational Mapping

MVC Model-view-controller

DTO Data Transfer Object

OOP Object Oriented Programming

JWT JSON Web Token

DRY Don’t Repeat Yourself

Appendix **B**

Contents of CD

```
 readme.txt ........................ the file with CD contents description
 src ....................................... the directory of source codes
   implementation ...................... implementation source codes
   thesis ............... the directory of LaTeX source codes of the thesis
 text ........................................ the thesis text directory
   thesis.pdf ....................... the Diploma thesis in PDF format
```