BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

WEB APPLICATION FOR MANAGING A COMMUNITY GREENHOUSE

WEBOVÁ APLIKACE PRO SPRÁVU KOMUNITNÍHO SKLENÍKU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **IVAN TSIARESHKIN**
AUTOR PRÁCE

SUPERVISOR **Ing. DAVID BAŽOUT**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

154713

Institut:	Department of Computer Graphics and Multimedia (DCGM)
Student:	**Tsiareshkin Ivan**
Programme:	Information Technology
Title:	**Web Application for Managing a Community Greenhouse**
Category:	Web applications
Academic year:	2023/24

Assignment:

1. Explore the operation of a community greenhouse with smart technology and analyze user requirements for its management.
2. Design a suitable user interface reflecting the requirements from users.
3. Explore existing front-end frameworks, select the appropriate technology, and implement the application.
4. Test the UI on real tasks and make optimizations.
5. Document the results of the work in the form of a poster and a short video.

Literature:

- Steve Krug: Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability, ISBN: 978-0321965516

Requirements for the semestral defence:
Tasks 1, 2, partially task 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Bažout David, Ing.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	7.5.2024

Abstract

This thesis focuses on the development of a web application for managing a community greenhouse. The primary goal is to create management functions supplemented by an intuitive interface that meets the needs of various user types. The work thoroughly describes all the development process steps, including requirement research and analysis, design, user interface evaluation, actual development, and final testing. Various web application development concepts are also discussed, and specific technologies are justified. The entire application is implemented in TypeScript using Express.js and PostgreSQL for the backend and Next.js for the frontend. The result is a tested and optimized application that allows users to manage a community greenhouse comfortably.

Abstrakt

Tato práce se zabývá vývojem webové aplikace pro správu komunitního skleníku. Hlavním cílem je vytvoření manažerských funkcí, které jsou doplněny o intuitivní rozhraní odpovídající potřebám různých typů uživatelů. Práce podrobně popisuje všechny kroky vývojového procesu, včetně průzkumu a analýzy požadavků, návrhu, vyhodnocování uživatelského rozhraní, samotného vývoje a závěrečného testování. Diskutovány jsou také různé koncepty vývoje webových aplikací a je odůvodněn výběr konkrétních technologií. Celá aplikace je implementována v jazyce TypeScript s použitím Express.js a PostgreSQL pro back-end a Next.js pro front-end. Výsledkem je testovaná a optimalizovaná aplikace, která umožňuje uživatelům pohodlnou správu komunitního skleníku.

Keywords

web application, community greenhouse, user interface, TypeScript, Express.js, REST API, PostgreSQL, Next.js, Tailwind CSS

Klíčová slova

webová aplikace, TypeScript, Express.js, REST API, TypeORM, PostgreSQL, Next.js, SSR, Tailwind CSS, Redis

Reference

TSIARESHKIN, Ivan. *Web application for managing a community greenhouse.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. David Bažout

Rozšířený abstrakt

Ve světě globalizace a urbanizace mnoho lidí žije v podmínkách, kde nemají možnost pěstovat vlastní ovoce a zeleninu. Navíc je dnes přístup k přírodním produktům bez chemických úprav a genetických modifikací velmi omezen. Většina produktů na trhu je podrobena různým zpracováním, které mají za cíl zlepšit jejich vzhled a prodloužit trvanlivost, což ztěžuje hledání ekologicky čistých produktů pro současného spotřebitele.

K řešení tohoto problému vyvinula společnost Sensorie s.r.o. koncept chytrého skleníku, který umožňuje pěstovat zdravou domácí zeleninu bez nutnosti věnovat mnoho času sledování skleníku. Jeden takový skleník již byl postaven pro veřejnost a obyvatelé města Brna ho aktivně využívají. Kdokoli může rezervovat místo ve komunitním skleníku a začít pěstovat rostliny bez zbytečných komplikací.

Tato práce je věnována vývoji a realizaci webové aplikace pro zlepšení interakce uživatele s komunitním skleníkem. Cílem práce je vyvinout funkcionality pro management, aby bylo možné automatizovat procesy, které probíhají předtím, než uživatel začne pěstovat své rostliny ve skleníku. Aplikace je také zaměřena na zaměstnanece a správce skleníku a přidává pro ně nezbytné funkce.

S ohledem na potřeby různých uživatelů, začal vývojový proces důkladným průzkumem a analýzou požadavků, což umožnilo identifikaci klíčových funkcionalit, které aplikace musí obsahovat. Na základě těchto informací byl vytvořen návrh uživatelského rozhraní, který byl následně iterativně testován a vyhodnocován s reálnými uživateli, aby se zajistilo, že konečný produkt bude splňovat požadavky.

Z technologického hlediska bylo rozhodnuto využít jazyk TypeScript, framework Express.js a relační databáze PostgreSQL pro back-end, a framework Next.js pro front-end. Tato kombinace technologií byla zvolena na základě jejich výhod, které jsou popsány v teoretické části této práce.

Celý vývojový cyklus zahrnoval iterativní vývoj a závěrečné testování, které zahrnovalo jak funkční, tak uživatelské testování aplikace. Výsledkem je plně funkční webová aplikace, která splňuje požadované funkce pro správu skleníku.

Výsledná aplikace umožňuje zákazníkům prohlížet stávající komunitní skleníky, získávat o nich podrobné informace a provádět rezervace pro pěstování rostlin. Zákazníci si také mohou zakoupit různé pěstitelské pomůcky. V osobním účtu každý zákazník může sledovat své aktivní rezervace a prohlížet historii svých nákupů. Pro zaměstnance byla implementována funkce kontrolního seznamu, který obsahuje všechny nezbytné úkony pro běžnou kontrolu skleníku. Administrátor má plnou kontrolu nad aplikací a může přidávat, upravovat data a přidělovat úkoly zaměstnancům.

Web application for managing a community greenhouse

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. David Bažout. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Ivan Tsiareshkin
May 9, 2024

Acknowledgements

I would like to express my gratitude to my supervisor, Ing. David Bažout, for his guidance, support and exciting topic of this thesis.

Contents

Chapter 1

Introduction

In the current digital era, relying on outdated management practices is neither practical nor effective. This thesis aims to address the issue of insufficient digital infrastructure in a Community Greenhouses and proposes a solution in the form of a web application to manage these greenhouses. By utilizing this application, the pre-planting processes can be automated, and urban residents can effortlessly begin growing healthy, locally sourced produce without any difficulties.

This thesis is structured into seven distinct chapters. Chapter 1, the introduction, outlines the overall structure of the thesis and briefly previews each chapter. Chapter 2 explores the operational aspects of Community Greenhouses. It also includes a requirements analysis and identification of key functionalities for different user types. Chapter 3 delves into the theoretical foundations of web application development. It explains core concepts and compares popular technologies used for development. Chapter 4 outlines the system design, including technology selection, database design, user interface planning, and overall system architecture. Chapter 5 details the practical implementation of the web application, focusing on both backend and frontend components. In Chapter 6, the testing phases, comprising mockup, functional, and user experience testing, are explored, along with the optimizations implemented. In the end, Chapter 7 concludes the project's achievements and discusses possible future enhancements to support Community Greenhouses further.

Chapter 2

Community Greenhouse and requirements analysis

2.1 Overview of the Community Greenhouse

Access to fresh, eco-friendly produce can be a challenge for many people, especially those living in urban areas with limited gardening opportunities. Community Greenhouse from Sensorie Ltd[1] has emerged as a groundbreaking solution to fill this gap. This innovative approach enables individuals to access healthy, organically grown food efficiently and cost-effectively without the extensive time and financial investment typically required for cultivation.

Community Greenhouse operates on a smart principle that allows users to easily monitor and manage the entire growing process via a user-friendly application.

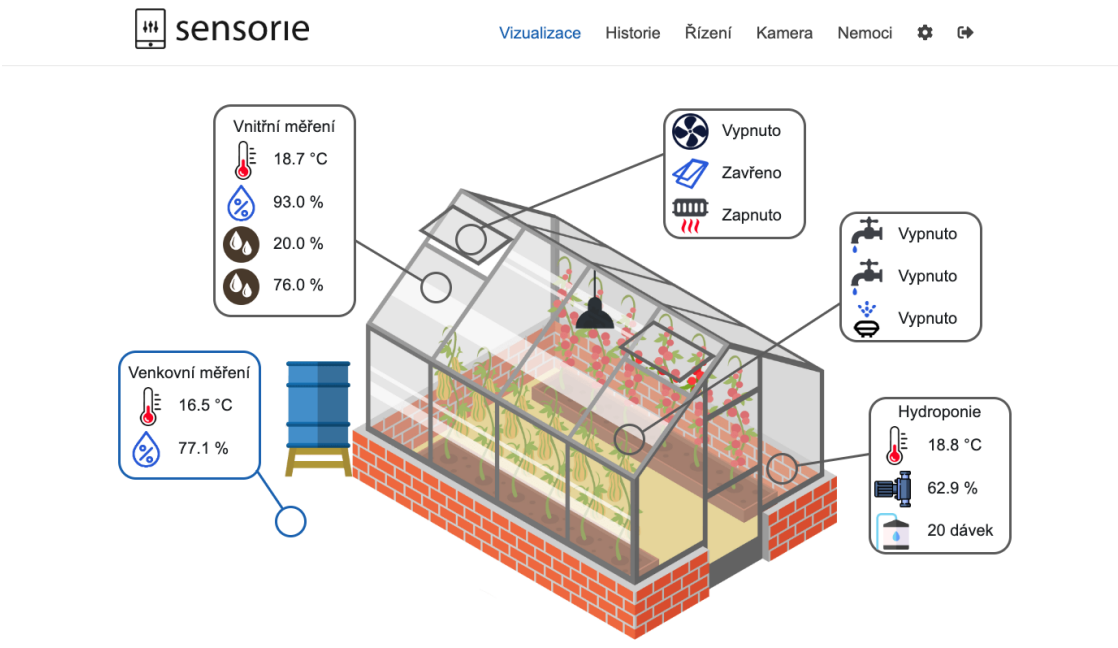


Figure 2.1: Overview of the greenhouse in the application, developed by Sensorie Ltd.

[1] https://sensorie.cz/

This application lets users control various parameters such as irrigation, humidity, and temperature, simplifying and optimizing plant growth. It also offers advanced features for thorough monitoring and management. Users can access a complete record of all measured values from the past, which is especially useful for diagnosing any issues affecting the plants, as the cause can be easily identified.

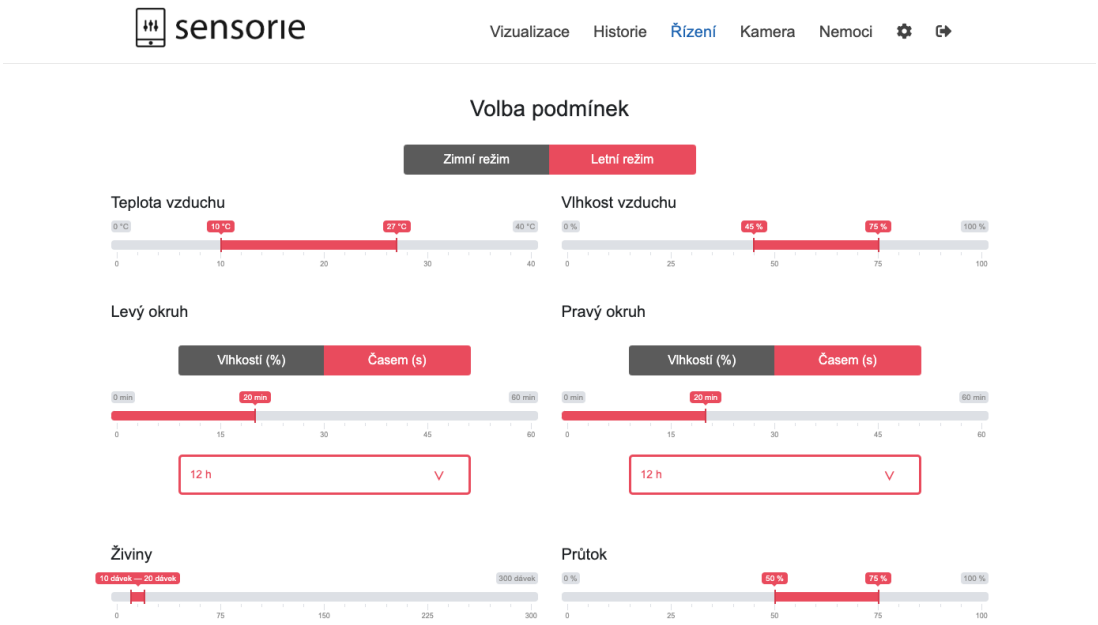


Figure 2.2: Settings management page in the application allows users to configure various parameters of the smart greenhouse.

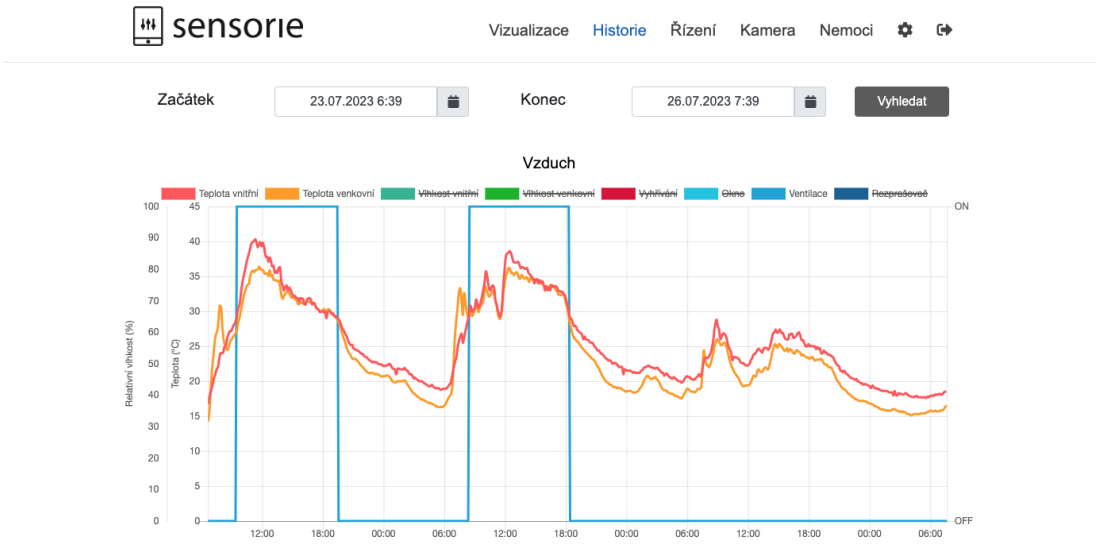


Figure 2.3: The data analysis page in the application displays various charts and metrics collected from the greenhouse.

Thanks to this innovative approach, Sensorie has revolutionized the process of growing vegetables and fruits in greenhouses. Now, there is no need for physical visits to monitor and control the artificial growing conditions; this can all be managed directly from a smartphone. Beyond the convenience, a significant advantage of smart greenhouses is their autonomy. For instance, they utilize solar energy to maintain optimal temperatures and collect rainwater to replenish water supplies. As such, smart greenhouses leverage natural resources to provide the most environmentally friendly solution.

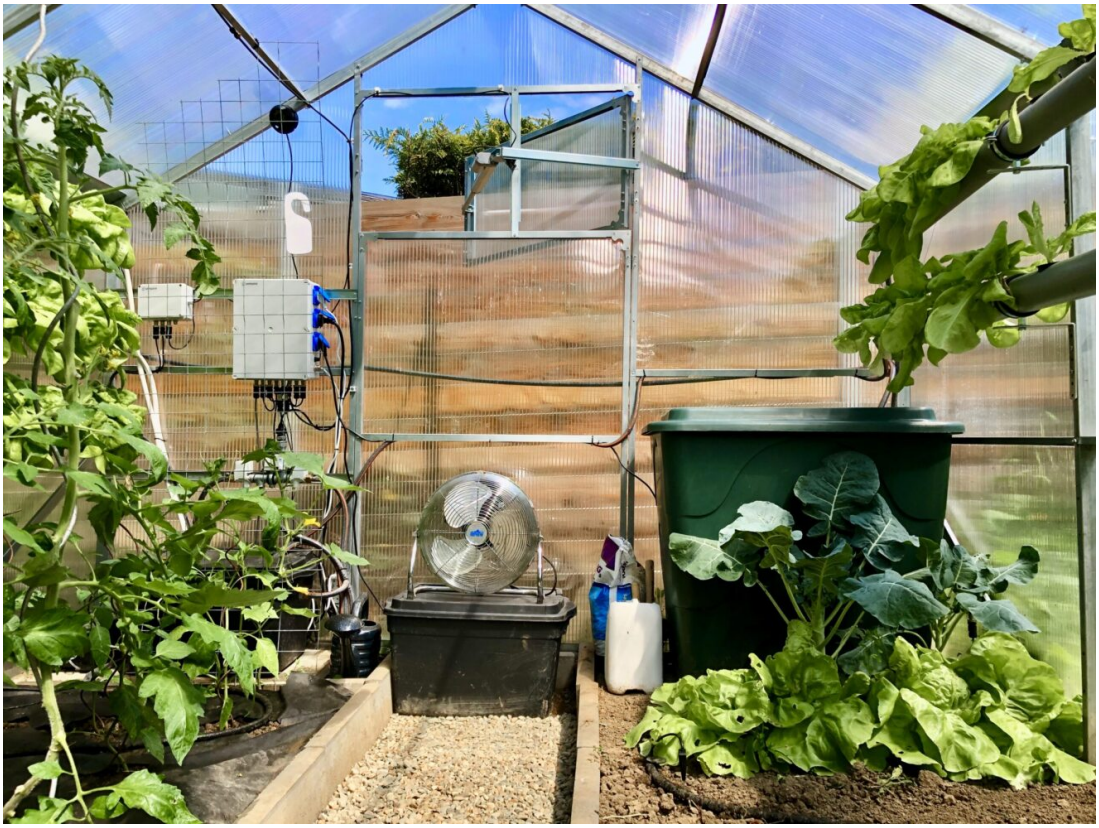


Figure 2.4: Live view inside a smart greenhouse. The image displays two types of planting areas: classic soil on the left and hydroponic cells on the right. Various electronic devices are also visible, which equip the greenhouse with smart capabilities.

However, an essential aspect of greenhouse operations that requires attention is the management of functions related to actions preceding the planting of crops.

Although the application is excellent at aiding the cultivation process, it is crucial to have features that manage the initial stages of greenhouse operations. These features include reserving space, user access management, scheduling and planning for planting, and financial transactions related to space reservation. Efficiently streamlining these pre-planting activities with a robust web application is essential to ensure that users can effectively engage with the greenhouse from the beginning.

The integration of these management features represents a significant advancement in the Community Greenhouse concept, completing the cycle from initial user engagement to produce harvesting. The following sections will detail these functionalities and outline how they can be implemented to provide a holistic and user-friendly greenhouse management experience.

2.2 User roles and interaction in Community Greenhouse

The Community Greenhouse ecosystem is a platform tailored for adults aged 20 and older, where users can interact to create an efficient environment. The core users of the platform are customers who have access to a range of functionalities within the application. They can begin by exploring the existing Community Greenhouses, viewing photographs, reading descriptions, and checking the community blog posts. This interaction is designed to give users a comprehensive understanding of what the greenhouse offers.

Once a user is interested, they can reserve a space based on their desire and needs. The application keeps users informed and engaged by sending notifications about available spaces and welcoming them with an email at the beginning of their reservation period.

The application helps users manage their greenhouse responsibilities as they become active participants. It sends timely notifications for rental payments and offers a convenient shop within the app to purchase essential gardening accessories such as seeds, saplings, and tools. Additionally, the store includes a small selection of fresh products available for purchase. These products are placed in hydroponic baskets, ensuring they remain fresh for several weeks. Each registered user is provided with a personal account, which enables them to view their purchase history and active reservations. Additionally, users can directly renew their reservations from within their personal accounts.

Greenhouse workers play a pivotal role in ensuring the smooth operation of the greenhouse. They are responsible for conducting regular inspections and maintenance, adhering to checklists to ensure everything functions correctly. Additionally, workers have the ability to update the greenhouse blog, such as posting news about closures for maintenance.

The system is managed by an administrator who oversees the entire web application. The administrator's responsibilities include creating and managing new greenhouses and appointing workers. This role is crucial in ensuring that the greenhouse operates efficiently and that users' needs are met. The administrator is also responsible for confirming orders and verifying that payments are made in a timely manner.

Through this well-organized interaction of roles, the Community Greenhouse system ensures a smooth and user-friendly experience, enabling users to obtain fresh, eco-friendly produce efficiently.

2.3 Identifying key features and functions

Analyzing user requirements for managing the Community Greenhouse has identified specific functions and actions for each user role. These functions are vital for the web application's proper functionality and for ensuring that users achieve their intended results.

Unregistered user

1. View information about available Community Greenhouses, including photographs and descriptions.
2. Access each greenhouse's dedicated blog wall with related news.
3. View the products available in greenhouse shops.
4. Register on the platform for expanded features.

Registered user

1. Reserve greenhouse space with different options.
2. View and edit profile, including personal information and preferences.
3. Access and view detailed information on purchases and reservations.
4. View and modify current greenhouse reservations.
5. Purchase accessories such as seeds, saplings, tools, and available fresh products placed in hydroponic baskets.

Greenhouse worker

1. Access and review checklists detailing required maintenance actions for the greenhouse.
2. Mark completed tasks in checklists.
3. View complaints and feedback from tenants.
4. Add and edit blog posts on each greenhouse's dedicated blog wall.

Administrator

1. Complete control over the entire web application.
2. Create and manage new greenhouses.
3. Appoint and supervise greenhouse workers.
4. Adding and editing products in specific greenhouse shops.
5. Manage the receipt of payments from users and maintain records of all orders.

Chapter 3

Web development technologies

The emergence of web applications has transformed the way people utilize digital technology. Web applications have become essential tools for individuals, businesses, and organizations. They offer solutions that are dynamic, accessible, and user-friendly, serving a wide range of purposes from e-commerce to social networking and beyond. This chapter explores the fundamental principles and popular technologies underlying full-stack web development. The complexities of web application development will be examined, which involve client-side(front-end) and server-side(back-end) components, programming languages, frameworks and data management.

3.1 Foundations of web application functionality

At the core of web applications lies the client-server model, a fundamental concept that divides responsibilities between two entities: the client and the server. The client, typically a web browser, sends requests to the server. This is done whenever a user performs an action that requires retrieving data or functionality, such as clicking a link or submitting a form. The server, located remotely, processes these requests and returns the appropriate response [9].

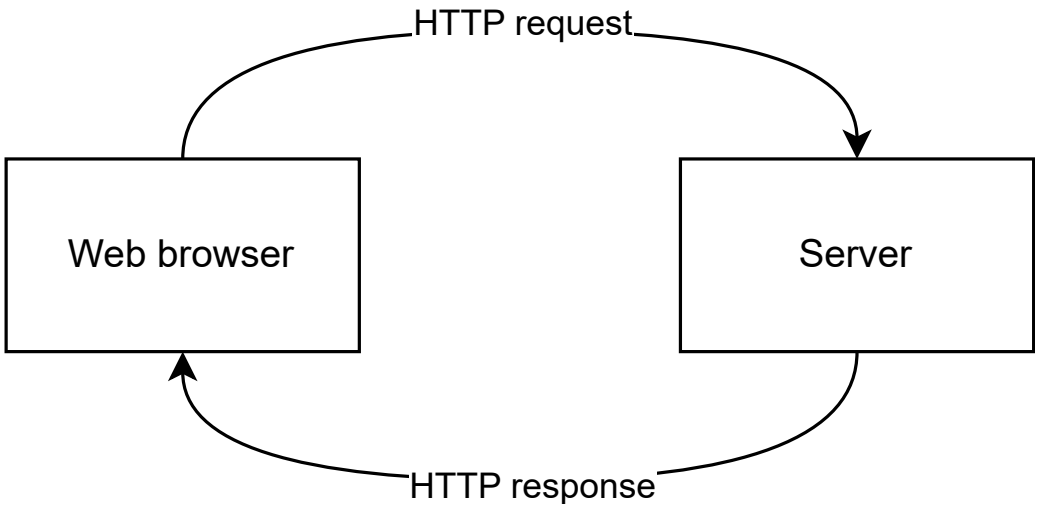


Figure 3.1: Client-server web application architecture diagram.

The communication between client and server in web applications is primarily governed by HTTP (Hypertext Transfer Protocol) or its secure variant, HTTPS (Hypertext Transfer Protocol Secure). These protocols act as a set of rules for exchanging data over the Internet. When a client needs information, it sends an HTTP request to the server. This request includes a specific method (like GET or POST), a URL (Uniform Resource Locator), and possibly additional data or parameters. Upon receiving the request, the server processes it and responds with an HTTP response. This response typically contains a status code, which indicates whether the request was successful or not, and the requested data or content. For example, if a user requests a webpage, the server will respond with the HTML content of that page [**?**].

HTML (HyperText Markup Language), CSS (Cascading Style Sheets), and JavaScript are the three cornerstone technologies of the World Wide Web. HTML is used for structuring and presenting content on the Internet. Every web page is built upon HTML, which dictates the structure and layout of the page using various elements and tags [14].

CSS is used alongside HTML to define the style of the web page. It controls the visual presentation, allowing for customising colours, fonts, layouts, and more. This separation of content (HTML) and style (CSS) makes maintaining and updating websites easier [13].

JavaScript, a programming language, adds interactivity to web pages. While HTML and CSS are static (displaying the same content every time a page is loaded), JavaScript allows for dynamic changes to the content that can occur without having to reload the entire page. This includes interactive forms, animations, and complex web-based applications [15].

Together, HTML, CSS, and JavaScript form the backbone of web application development, enabling the creation of complex, responsive, and interactive web pages that are processed and rendered by the client (the browser) in response to the server's HTTP responses.

3.2 Back-end technologies

This section analyzes various backend technologies and frameworks, focusing on their architecture, built-in features, strengths, and weaknesses.

3.2.1 Python: Django framework

Django is a popular high-level Python web framework emphasising rapid development and clean, pragmatic design.

Django's architecture, which employs a Model-View-Template (MVT) pattern, a variant of the Model-View-Controller (MVC) design, effectively separates data handling (Model), user interface (Template), and control flow (View). This separation enhances the application's clarity and scalability. Django's comprehensive standard library is a significant asset, offering an ORM (Object-Relational Mapping) system for simplified database interactions and built-in support for common web development tasks, such as user authentication and content administration. Its robust security features also safeguard against threats like SQL injection, cross-site scripting and cross-site request forgery.

However, while Django's high level of abstraction is beneficial for rapid development, it can sometimes lead to decreased control over certain aspects of the application, potentially limiting customization for more complex requirements. Though advantageous for large-scale applications, the framework's structure may be overly cumbersome for smaller projects or microservices, where a lighter, more flexible framework could be more appropri-

ate. Also, Django follows a monolithic architecture, which can be less efficient in handling asynchronous tasks, a common requirement in modern web applications.

Furthermore, Django's ORM, while powerful, may not always be the best fit for applications with heavy reliance on database operations, as it might not match the efficiency of raw SQL queries in handling complex data manipulations [7].

3.2.2 Java: Spring Boot

Spring Boot is a popular open-source Java framework that simplifies the process of building and deploying production-ready applications. It is built on top of the Spring Framework, providing an opinionated approach to application development with sensible defaults and autoconfiguration features.

Spring Boot automatically configures various components and dependencies based on the project's classpath and predefined conventions. This approach reduces the need for extensive XML configuration files and boilerplate code, allowing developers to focus on writing business logic.

Spring Boot offers several key features and benefits: It includes embedded servers like Tomcat or Jetty, enabling developers to run applications as stand-alone executables without needing separate server installations. It also includes built-in support for features like health checks, metrics, and externalized configuration, making deploying and monitoring applications in production environments easier. It also provides comprehensive testing utilities, including support for integration tests, making writing and running tests for various application components easier.

Although Spring Boot is a widely used framework, it has some limitations that should be considered. The framework's opinionated approach might not suit all types of projects, particularly those requiring specific customizations. In addition, the magic of autoconfiguration, though beneficial, can sometimes obscure what is happening behind the scenes, leading to challenges in debugging and understanding the application's internal workings. Moreover, the learning curve can be steep for developers new to the Spring ecosystem [21].

3.2.3 PHP: Laravel

Laravel is a PHP framework that uses MVC architecture and provides a structured and modular approach. It simplifies web development tasks such as routing, sessions, caching, and authentication.

This framework offers various tools and features that simplify web development, including Eloquent ORM for interacting with databases, Blade templating engine for creating dynamic views, and Artisan CLI for managing tasks from the command line. Laravel also emphasizes security, with built-in protection against common threats. The framework automatically manages many security aspects, reducing the need to implement these protections manually.

Despite its strengths, Laravel has some drawbacks. Firstly, it has a steeper learning curve compared to simpler frameworks due to its extensive features and the MVC pattern it follows. Laravel's comprehensive features and abstractions can introduce performance overhead compared to lighter-weight frameworks or custom-built solutions. Additionally, keeping up with Laravel's updates and new versions may require refactoring code to maintain compatibility, adding to development time. Furthermore, Laravel's robust features might be unnecessary for smaller projects with simpler requirements and introduce unwanted complexity [1].

3.2.4 JavaScript: Express.js

Express.js is a widely used framework within the Node.js environment. Node.js is a powerful JavaScript runtime built on Google's V8 engine. It enables the execution of JavaScript code server-side, facilitating the development of scalable and efficient web applications and APIs.

Express.js follows a middleware-based approach, where incoming requests pass through a series of functions before reaching the final request handler. This modular structure allows adding, removing, or modifying functionality at different request processing pipeline stages.

One of Express.js's key advantages is its simplicity and flexibility. It provides a minimalistic core that can be extended with various middleware modules and plugins available through NPM, Node.js's package manager. Integration of additional modules enables tailored application development to meet specific needs without obscuring Node.js's native features.

Routing in Express.js is another key feature that maps HTTP methods and URLs to corresponding handler functions. This facilitates organized and maintainable codebases and supports dynamic route parameters for extracting values directly from the URL path.

Since Express.js is built on top of Node.js, it inherits its single-threaded nature, which is efficient in asynchronous I/O operations, but can lead to performance limitations in CPU-intensive tasks, requiring strategic application design to prevent bottlenecks[1].

Another issue is the lack of a strict structure or convention. While this flexibility is often seen as an advantage, it can lead to inconsistencies in project structure. Establishing clear guidelines and best practices is essential to maintain a cohesive codebase [18].

3.2.5 Back-end technologies evaluation

Framework	**Language**	**Pros**	**Cons**
Django	Python	Rapid development; MVT architecture; ORM for simplified database; built-in auth; administration.	High level of abstraction; cumbersome for small apps; less efficient in asynchronous tasks.
Spring Boot	Java	Autoconfiguration; embedded servers; extensive testing utilities.	Opinionated approach; autoconfiguration can obscure details; steep learning curve.
Laravel	PHP	MVC architecture; tools like Eloquent ORM, Blade templating, Artisan CLI; built-in security protections.	Steep learning curve; performance overhead from extensive features; frequent updates may require refactoring.
Express.js	JavaScript	Simple and flexible; middleware-based structure for modularity; minimalistic core extendable with plugins; efficient in handling asynchronous I/O.	Performance limitations in CPU-intensive tasks; lacks strict structure which can lead to inconsistencies in project organization.

Table 3.1: Evaluation table of back-end frameworks.

[1]Point in a system where a process is limited by the capacity or efficiency of one component or stage, slowing down the overall performance

3.3 API

An Application Programming Interface (API) is a crucial component that enables communication between different software systems. It provides a set of protocols and tools for building software and applications and is essential for defining how software components should interact. This section focuses on the two most popular approaches to API design: REST and GraphQL. It will explore their structural complexities, inherent capabilities, advantages, and challenges within the context of web development [6].

3.3.1 REST API

Representational State Transfer (REST) API is a popular architectural style for networked applications that is based on the principles of REST. It uses standard HTTP methods and works on a stateless protocol. This means that each request from a client to a server includes all the essential information to fulfil the request. REST APIs naturally align with CRUD (Create, Read, Update, Delete) operations:

- POST: Utilized for creating a new object. Sending a POST request involves asking the server to accept and store the data in the request's body. This method is typically used for submitting form data or uploading a file.
- GET: Employed for reading or retrieving a resource. GET requests are made to request data from a specified resource. It is the most commonly used method for retrieving web pages and is a read-only operation.
- PUT: Applied for updating or creating an existing object if it does not exist. In PUT requests, the enclosed data is treated as a modified version of the resource stored on the server, as identified by the URI. If the resource does not exist, a new one is created at the specified URI.
- PATCH: Similar to PUT, PATCH updates an existing object with partial changes. It is employed when there is a need to make a partial update to a resource rather than replacing the entire resource as PUT does.
- DELETE: Used for deleting an object. This method removes the resource specified by the URI.

REST API architecture offers scalability and simplicity through its stateless nature and standard HTTP methods, enhancing cacheability and ease of use. However, it can lead to inefficiencies with underfetching or overfetching data and may present performance challenges with large data sets [3].

3.3.2 GraphQL

GraphQL, developed by Facebook, is a query language for APIs that allows clients to request exactly what they need, contrasting with traditional REST APIs where the server defines the data structure. With GraphQL, clients can specify precise fields on objects, potentially reducing data fetching. Unlike REST, which uses multiple endpoints for different resources, it operates through a single endpoint.

However, this approach has its downsides. The flexibility of GraphQL can lead to complex queries, especially when dealing with large schemas or deep nested structures. This

complexity can make it harder for developers to optimize query performance. Additionally, the single-endpoint structure of GraphQL can be a challenge for caching mechanisms, which are more straightforward in REST APIs with multiple endpoints. Despite efficient data retrieval in specific scenarios, GraphQL might require more effort in query planning and server-side optimization [10].

3.4 Managing data in web applications

This section delves into the core aspects of managing data within web applications, emphasizing the pivotal role that both relational and NoSQL database systems play in the architecture and functionality of modern web applications. Examining relational and NoSQL database systems highlights their distinct architectures and roles in back-end data handling.

3.4.1 NoSQL databases

NoSQL (not only SQL) databases, known for their flexibility and scalability, have become increasingly popular in handling large volumes of unstructured or semi-structured data. Unlike traditional SQL databases with a structured query language and a predefined schema, NoSQL databases allow for a more dynamic data storage approach. This makes them particularly well-suited for big data applications and real-time web applications.

One of the key features of NoSQL databases is their schema-less data model, which provides significant flexibility in handling changes and variations in data. This is especially beneficial in agile development environments where requirements and data models can evolve rapidly. NoSQL databases come in various forms, including document-oriented (like MongoDB), key-value pairs (like Redis[2]), wide-column stores (like Cassandra[3]), and graph databases (like Neo4j[4]), each optimized for specific types of data and access patterns.

Another significant advantage of NoSQL databases is their scalability. Many NoSQL systems are designed to scale out by distributing data across multiple servers, often in a distributed and fault-tolerant manner. This can lead to improved performance for large-scale applications and enable handling high throughput and massive volumes of data.

However, NoSQL databases also have certain limitations. The lack of a standardized query language may pose challenges in complex querying scenarios. Furthermore, the flexible schema design can sometimes result in data consistency issues, especially in distributed environments. This necessitates careful consideration of data modelling and consistency requirements in NoSQL database design.

In conclusion, NoSQL databases offer a modern approach to data management, providing flexibility, scalability, and efficient handling of diverse and voluminous data. Their use is particularly advantageous when data structure is not rigidly defined and can change over time. However, considerations regarding data consistency, querying capabilities, and the specific data model appropriate for the application's needs are critical in leveraging the full potential of NoSQL databases in web development projects [16].

[2] https://redis.io/
[3] https://cassandra.apache.org/
[4] https://neo4j.com/

3.4.2 Relational databases

Relational databases, utilizing Structured Query Language (SQL), represent a classical approach to data management and are fundamental in numerous web applications. These databases are based on a structured schema, where data is stored in tables with predefined relationships between them. This structured approach enables complex queries and transactions, making relational databases highly suitable for applications that require strict data integrity and consistency.

One of the main strengths of relational databases such as PostgreSQL, MySQL, and Oracle is their robust transactional integrity. They follow the ACID[5] principles, ensuring reliable data handling despite errors or system crashes. This consistency is crucial for applications like banking systems, where data accuracy and reliability are paramount.

SQL, the language used for querying and manipulating data in relational databases, is well-standardized and widely understood. It offers rich data retrieval and manipulation capabilities.

However, relational databases also have certain limitations. Their fixed schema design can make them less adaptable to changes in data models, requiring schema modifications and migrations that can be both time-consuming and complex. Additionally, while they are typically very efficient for transactional operations, relational databases can face performance challenges when scaling horizontally, particularly with very large volumes of data or high-throughput applications.

In conclusion, relational databases offer a time-tested solution for data management, excelling in scenarios requiring complex querying, data accuracy, and consistency. However, the rigidity of their schema and potential scalability challenges in horizontal scaling need to be considered when choosing a database solution for web development projects. Balancing these considerations is key to harnessing the strengths of relational databases in supporting structured and reliable data management [19].

3.4.3 Data management evaluation

Database Type	**Pros**	**Cons**
NoSQL	Efficiently handles large volumes of data; flexibility; scalability.	Complicate complex querying; data consistency issues in distributed environments.
Relational	Complex queries and transactions with integrity; ACID principles; rich data manipulation capabilities with SQL.	Fixed schema design is restrictive and adapt poorly to changes, requiring complex migrations; horizontal scaling impacts performance with large data volumes.

Table 3.2: Evaluation of two database types.

[5] https://www.geeksforgeeks.org/acid-properties-in-dbms/

3.5 Front-end technologies

This section focuses on front-end technologies that are widely used to create user interfaces (UI), integral components of the user's interaction with web applications. It explores various front-end frameworks and libraries, examining their features, architecture, strengths, and weaknesses.

3.5.1 Angular

Initially released in 2010, Angular was one of the first frameworks to popularize the concept of Single-Page Applications (SPAs). SPAs are web applications that load a single HTML page and dynamically update the content as the user interacts with the application without requiring a full page reload. This approach provides a more fluid and responsive user experience like a desktop application.

Angular follows a component-based architecture, allowing for modular and reusable code. It uses TypeScript[6], a superset of JavaScript that adds static typing and enhanced tooling support. The framework provides robust features, including dependency injection, two-way data binding, and a rich ecosystem of libraries and tools.

At the core of Angular's structure lies the concept of components. Components are self-contained units of functionality that encapsulate the template, style, and behaviour of a portion of the user interface. They communicate with each other through inputs and outputs, promoting a clear separation of concerns. Angular also utilizes services, which are injectable classes that provide shared functionality across components, such as data retrieval or authentication.

However, Angular has a steeper learning curve than other JavaScript frameworks. Its complex ecosystem and concepts, such as dependency injection and decorators, may require more time to master. Additionally, Angular applications' initial load time and performance can be a concern, especially if not appropriately optimized for production. Another consideration is the size of Angular applications. The framework's comprehensive nature often leads to larger bundle sizes, which might affect application load times and performance. Careful management and optimization strategies are essential to mitigate these issues [4].

3.5.2 Vue.js

Vue.js is a JavaScript framework used for building UIs and SPAs. Vue.js aims to be approachable, versatile, and performant. It adopts a component-based architecture, allowing developers to create reusable and self-contained application building blocks.

One of Vue.js's key features is its reactivity system. Vue.js automatically tracks dependencies and updates the view whenever the underlying data changes. This reactivity is achieved through a system that monitors data changes and a virtual DOM (Document Object Model[7]). The virtual DOM is a lightweight copy of the actual DOM, which allows Vue.js to efficiently determine the minimal changes needed and apply them to the actual DOM, resulting in fast updates and improved performance.

Vue.js components are written using a single-file component (SFC) format, encapsulating a component's template, logic, and styles in a single file with a .vue extension. This SFC approach promotes a clear separation of concerns and enhances code maintainability.

[6] https://www.typescriptlang.org/

[7] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

Components can communicate with each other through props (for parent-to-child communication) and custom events (for child-to-parent communication).

Vue.js is highly flexible and can be incrementally adopted into existing projects. It can be a lightweight library for simple applications or a full-featured framework for complex SPAs. Vue.js integrates well with other libraries and tools in the JavaScript ecosystem, such as state managers, client-side routing, and various build tools.

However, Vue.js may have a smaller ecosystem compared to some other popular frameworks like React or Angular. While the Vue.js community is active and growing, the availability of third-party libraries, tools, and resources might be comparatively limited [8].

3.5.3 React

React is the most popular JavaScript library for building UIs. Developed by Facebook, React has gained significant traction in the web development community due to its efficiency and flexibility. Like Vue.js, React follows a component-based architecture, allowing the creation of reusable UI components and efficiently updating the view when the underlying data changes.

At the core of React's architecture lies the concept of components. Components are self-contained pieces of code that encapsulate the structure, logic, and styling of a part of the user interface. React components can be either functional or class-based, with functional components being more lightweight and easier to understand. Components receive input data through props (short for properties) and can maintain their own internal state using the useState hook or class-based state.

Similar to Vue.js, React utilizes a virtual DOM to update the user interface efficiently. When a component's state or props change, React creates a new virtual DOM tree and compares it with the previous one. This process, called reconciliation, identifies the minimal modifications required to update the actual DOM, resulting in fast and efficient rendering.

React's component-based architecture promotes code reusability and modularity. Components can be easily composed and nested to create complex user interfaces. This modular approach allows for better code organization, maintainability, and collaboration among development teams. React also supports the use of JSX[8], a syntax extension that enables developers to write HTML-like code within their JavaScript files, making the code more readable and intuitive.

One of React's strengths is its extensive ecosystem. React benefits from a large and active community that contributes to a wide range of libraries, tools, and frameworks, including state management solutions, routing libraries, and various UI component libraries.

Another notable advantage of React is its performance. Its virtual DOM and efficient reconciliation algorithm minimize the number of direct manipulations to the actual DOM, resulting in faster rendering and improved performance. React's ability to efficiently update only the necessary components, rather than re-rendering the entire page, contributes to its overall speed and responsiveness.

However, while React's core concepts are relatively simple, mastering advanced topics such as state management, higher-order components, and the component lifecycle can take time. React's ecosystem can also be overwhelming for newcomers, with numerous libraries and tools to choose from [5].

[8] https://react.dev/learn/writing-markup-with-jsx

3.5.4 Next.js

Next.js is a popular React framework that extends React's capabilities and provides a powerful set of features for building server-rendered React applications. By abstracting away many of the complex configurations and setups required in a traditional React application, Next.js aims to simplify the process of creating scalable and performant web applications.

At its core, Next.js is built on top of React, and it introduces several key enhancements and abstractions that set it apart from plain React. One of the primary advantages of Next.js is its built-in server-side rendering (SSR) capabilities. With Next.js, developers can easily create applications that render on the server, providing improved performance, better search engine optimization (SEO), and faster initial page loads [17].

Next.js follows a file-based routing system, where each file in the „pages“ or „app“ directory automatically becomes a route. This convention-based approach eliminates the need for manually configuring routes, making it intuitive and easy to manage the application's navigation. Next.js also supports dynamic routes, allowing for the creation of pages with dynamic content based on parameters passed in the URL.

One of Next.js's key features is its automatic code splitting and optimized bundle generation. Next.js intelligently splits the application's code into smaller chunks, ensuring that only the necessary code is loaded for each route. This optimization improves the application's performance by reducing the initial bundle size and enabling faster page loads. Next.js also supports prefetching, which allows for the preloading of linked pages in the background, resulting in near-instant navigation.

However, Next.js introduces an additional layer of complexity compared to plain React. It is necessary to understand concepts such as server-side rendering, data fetching, and the Next.js-specific API to utilize the framework effectively. Additionally, Next.js may not be suitable for all types of applications, particularly those that heavily rely on client-side interactivity and real-time updates [2].

3.5.5 Front-end technologies evaluation

Framework	**Language**	**Pros**	**Cons**
Angular	TypeScript	Component-based architecture; dependency injection; rich ecosystem.	Steep learning curve; slow initial load times; large bundle sizes that affect performance.
Vue.js	JavaScript	Easy to integrate; reactivity system for efficient updates; supports single-file components.	Smaller ecosystem compared to React and Angular, which may limit available third-party resources and tools.
React	JavaScript	Large community and ecosystem; highly performant; supports JSX.	Learning advanced concepts can be challenging; overwhelming huge ecosystem.
Next.js	JavaScript	Built-in SSR for performance and SEO; automatic code splitting; file-based routing system for simplified navigation management.	Adds complexity with SSR and specific Next.js APIs; may not suit applications that prioritize client-side interactivity.

Table 3.3: Evaluation of front-end frameworks and libraries.

3.6 Web UIs styling - CSS

3.6.1 Sass

Sass is a preprocessor[9] scripting language that is interpreted or compiled into CSS. Sass simplifies CSS development by introducing programming concepts, resulting in more modular and maintainable code. It offers features like variables for consistent values, mixins for reusable CSS rules, nesting for organized code structure, and functions for complex calculations.

Sass promotes best practices such as separating concerns and organizing code modules for code reusability, adherence to the design system, and concise style sheets.

However, the added complexity of Sass introduces a learning curve. It also requires an additional compilation step, potentially impacting build times and development workflows. Furthermore, while Sass promotes modular code, it does not inherently prevent the accumulation of unused CSS, a problem that utility-first frameworks like Tailwind CSS aim to mitigate [22].

3.6.2 Tailwind CSS

Tailwind CSS, on the other hand, adopts a utility-first approach, offering low-level utility classes that can be composed to build custom designs. Unlike Bootstrap[10], Tailwind does not provide pre-designed components; instead, it allows developers to create bespoke designs by combining utility classes directly in HTML.

This approach provides greater control over the design process, enabling developers to create unique, tailored user interfaces. Tailwind CSS is highly appreciated for its flexibility and minimalistic output, as it generates CSS that is specific to the components being used, reducing unused CSS and optimizing load times.

However, Tailwind's utility-first paradigm can lead to verbose HTML with numerous classes, which may be daunting and less intuitive for newcomers. It requires a good understanding of CSS and design principles to utilize its full potential effectively [12].

3.6.3 Evaluation table

Technology	Pros	Cons
Sass	Programming concepts, modular and maintainable; variables, mixins, nesting, and functions; promotes best practices for code reusability.	Adds complexity with a learning curve and requires an additional compilation step, which may affect build times. Does not inherently prevent unused CSS.
Tailwind CSS	Utility-first approach that provides precise control over design with low-level utility classes; generates minimalistic CSS, reducing unused code and optimizing load times.	Can lead to verbose HTML with extensive class usage; requires a solid understanding of CSS and design principles.

Table 3.4: Comparison of CSS Technologies: Sass and Tailwind CSS.

[9] https://developer.mozilla.org/en-US/docs/Glossary/CSS_preprocessor
[10] https://getbootstrap.com/

Chapter 4

System design

The following chapter describes the application's system design. First, the technologies described in Chapter 3 and selected based on the requirements outlined in section 2.3 are shown. Then, the database structure is described, including explanations and diagram schema. Next, the structure of the API and the server part is defined, after which the user interface wireframes are shown, including detailed mockups of some pages.

4.1 Used technologies

4.1.1 Back-end

Following an analysis of requirements and an examination of web technologies, Express.js and Node.js were selected for the server-side implementation of the application. These technologies were chosen primarily for their capacity to support lightweight server configurations that adeptly facilitate the design of APIs. Specifically, the REST API architectural style was selected to structure the communication between the client and server, leveraging its stateless, cacheable, and uniform interface to enhance scalability and performance. The choice of Express.js was further influenced by its ability to expedite development, user-friendly interface, and extensive ecosystem of available packages. Additionally, the typical drawbacks associated with Express.js, described in section 3.2.4, are expected to have minimal impact on the application, considering the project's specific requirements and defined scope.

4.1.2 Front-end

Given the application's requirement for interactive modules and functions, Next.js was selected to develop a React-based application enhanced by superior server-side rendering, routing, and various optimizations. This choice combines React's component-driven architecture with Next.js to leverage its automatic code splitting and optimized page rendering, significantly improving performance and user experience. Next.js also facilitates easier data fetching, integral for maintaining a smooth and responsive user interface.

4.1.3 Other techonlogies

TypeScript

TypeScript was selected as the primary programming language for both the frontend and backend development to ensure consistency across the application's architecture. The choice of JavaScript frameworks for these layers – specifically Next.js for the front end and Express.js for the back end – complements TypeScript by enhancing type safety and scalability. TypeScript, a superset of JavaScript, introduces static typing, enabling developers to catch errors at compile time rather than at runtime. This leads to more reliable code, easier maintenance, and improved productivity.

Redis

Redis[1], paired with the express-session[2] middleware, was utilized for session management. Known for its rapid data access capabilities, Redis was selected as the session store. This integration facilitates efficient session data handling, particularly for authentication and authorization purposes. For further details, refer to the subsection 5.1.3.

AWS S3

Amazon Web Services (AWS) S3[3] was utilized for storing all media files of the application. AWS S3, renowned for its scalability and high durability, was selected as the storage solution. The integration of AWS S3 ensures that media files are easily retrievable and maintained with high availability, crucial for the application's scalability and potential handling of large volumes of media data in the future.

Tailwind CSS

Tailwind CSS was selected over Sass due to its utility-first design approach, which provides complete control over styling. This selection facilitated rapid development, enabling the creation of a user-friendly and responsive design. As a result, the application's interface can be visually distinct and highly functional.

Radix UI

Radix UI[4] was selected for the project to provide a foundation for building accessible, customizable, and low-level UI components. This choice was driven by the need to create a highly tailored user experience where each component could be adjusted to meet the application's specific design and functionality requirements. Further information is provided in the subsection 5.2.3.

[1] https://redis.io/
[2] https://www.npmjs.com/package/express-session
[3] https://aws.amazon.com/s3/
[4] https://www.radix-ui.com/

4.2 Database Design

PostgreSQL (PSQL) was selected as the relational database for this project, coupled with TypeORM as the Object-relational mapping (ORM) tool. The preference for a relational database over a NoSQL option was influenced by the project's specific needs for structured data management without the necessity for high scalability or the flexibility to handle unstructured data.

TypeORM facilitates communication between the application and the PSQL database. As an advanced ORM, it provides a powerful yet straightforward way to define and manage entities and their relationships, enhancing the application's ability to handle complex queries efficiently. This integration allows the application to benefit from PSQL's powerful data management capabilities, while using TypeORM's advanced features to simplify database operations and maintain a clear and maintainable codebase.

The database comprises **11** different tables. The role and function of each table are explained in detail in the following subsections.

„users“ and „user_greenhouses“ tables

The „users“ table stores essential information about each user. It includes fields such as email and password used for authentication. Additionally, this table holds other vital user details, including name and role. The role field determines access levels within the system and is defined by a string enum of standard, worker, and admin types of users. Additionally, this table contains a uuid[5] token field and a boolean flag, which is used to verify the authenticity of the user's email address during the signup process. This verification process ensures that emails are sent to a verified email address, enhancing the security and reliability of the user's experience within the application. Users with administrator rights can assign worker users to specific greenhouses through the „user_greenhouses“ table. This table establishes a many-to-many relationship between users and greenhouses, granting worker users additional rights and capabilities within those assigned greenhouses.

„cart“ and „cart_item“ tables

These tables are essential for the application's e-commerce functionality. Upon user registration, an entry is automatically generated in the „cart“ table, enabling users to begin adding products or reservations. The „cart_item“ table categorizes each item by type, distinguishing between products and reservations. It stores details such as quantity, product_id, and price for products. For reservations, it captures reservation start and end dates, duration in months, selected areas in the greenhouse, and the total price.

„orders“ table

After a purchase or reservation is made, a corresponding entry is recorded in the „orders“ table. This entry includes the transaction date, payment status, a QR code for the payment, and details of the items included in the order. This system ensures that all transaction details are accurately captured and accessible for both management and customer reference.

[5] https://www.cockroachlabs.com/blog/what-is-a-uuid/

„tasks“ table

A user with administrator rights can assign tasks to worker users. Each task is associated with a specific greenhouse and includes essential details such as a title, description, and a completion deadline. Additionally, each task features a completion flag, updated when a worker marks the task as completed. This functionality facilitates effective task management and tracking within the application.

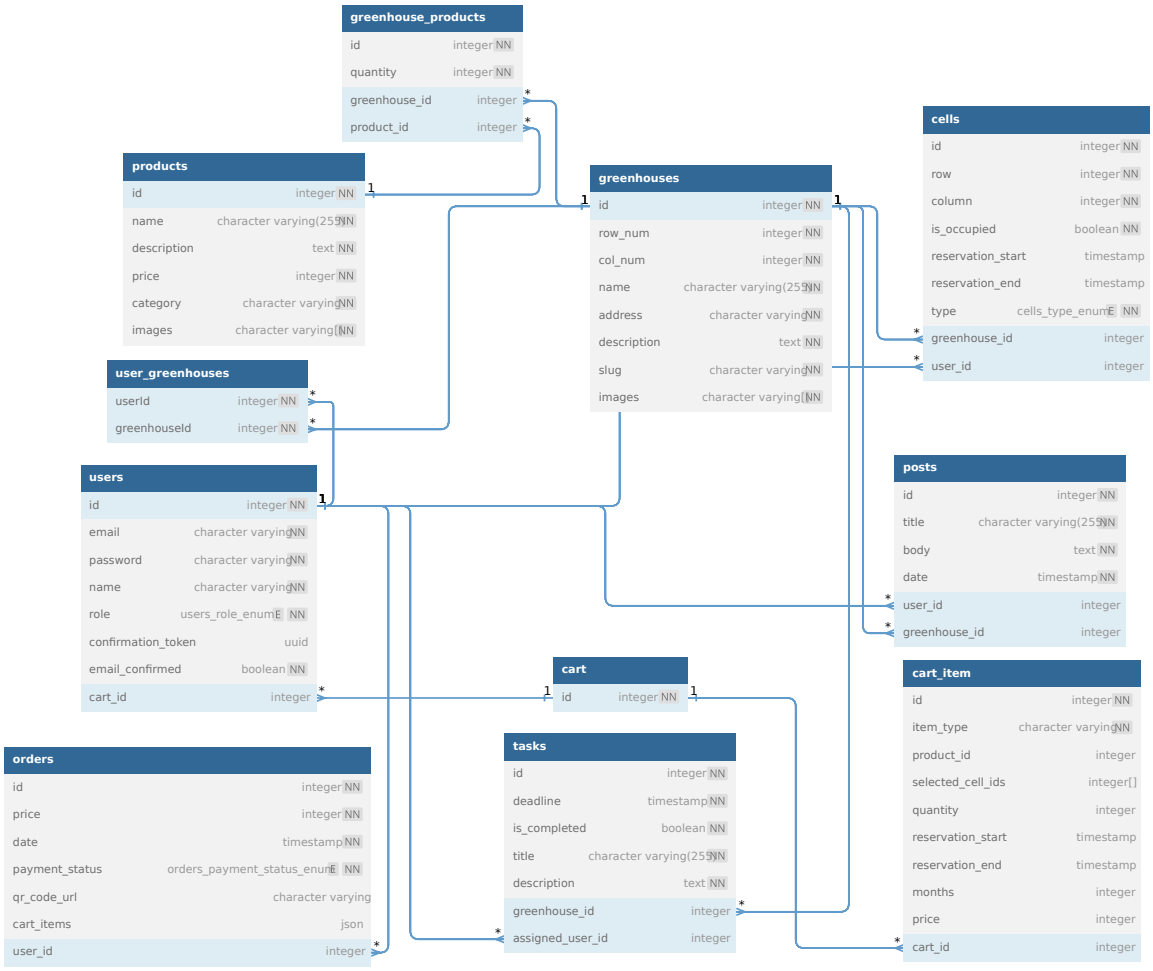


Figure 4.1: Final database schema.

„greenhouse“ table

The „greenhouse“ table is a central component of the application's database. Each greenhouse is represented as a grid, defined by its number of rows and columns. Additionally, this table includes the greenhouse's name, address, and a detailed description. Photographs of each greenhouse are stored as an array of strings in the 'images' field, with each string being a link to a photo hosted on AWS S3.

„cells“ table

Given that each greenhouse is structured as a grid, the „cells“ table represents individual locations within the greenhouse. Each cell is defined by two values: its row and its column within the grid. Cells are also categorized by type, either „soil“ or „hydroponic“. Each cell includes a flag indicating whether it is occupied to manage reservations. If a cell is occupied, the table also records the start and end dates of the reservation, as well as the identity of the user who made the reservation.

„posts“ table

The „posts“ table is used to implement a news feed within each greenhouse. This table stores essential information about each post, including the title, text, date it was published, and the user who authored it. This functionality effectively communicates news and events specific to each greenhouse, such as scheduled maintenance closures or other important updates, enhancing user engagement and operational awareness.

„products“ and „greenhouse_products“ table

The „products“ table represents each product within the application, detailing the product's name, description, category, price, and images. Those images are stored as an array of strings, similar to the method used in the greenhouse table. A corresponding „greenhouse_products“ table is used to accommodate products available in multiple greenhouses. This table implements a many-to-many relationship between the „products“ and „greenhouses“ tables, effectively linking products to the various greenhouses where they are available.

4.3 UI layout

Proper UI layout is an essential part of web application development. In his book *Don't Make Me Think*, Steve Krug emphasizes the importance of keeping interfaces simple. The design should be self-evident, making it obvious which elements are clickable and how to navigate the site. He also highlights the significance of effectively using size, colour, and layout, which can direct attention to the most critical aspects of a page. These principles have profoundly influenced the development of the user interface for this project, particularly with an emphasis on simplicity and clarity [11].

After analyzing the target audience, it became apparent that the application would be used by various age groups, from young adults to seniors. This diversity underscores the need for a simple and understandable user interface, critical to ensuring the application is accessible and easy for everyone. This section will describe the design of the user interface layout and highlight the main steps.

4.3.1 Wireframes

Typically, UI design starts with prototyping and wireframing. Wireframes are low-fidelity visual representations that outline a digital interface's fundamental structure and functional elements. They provide a blueprint for the layout and functionality, focusing on what interface elements will exist on key pages [11].

Following these principles, the main interface pages were first highlighted - a `list of greenhouses`, a `greenhouse page`, a `greenhouse reservation module`, and `checklists` for workers.

Wireframe: List of greenhouses

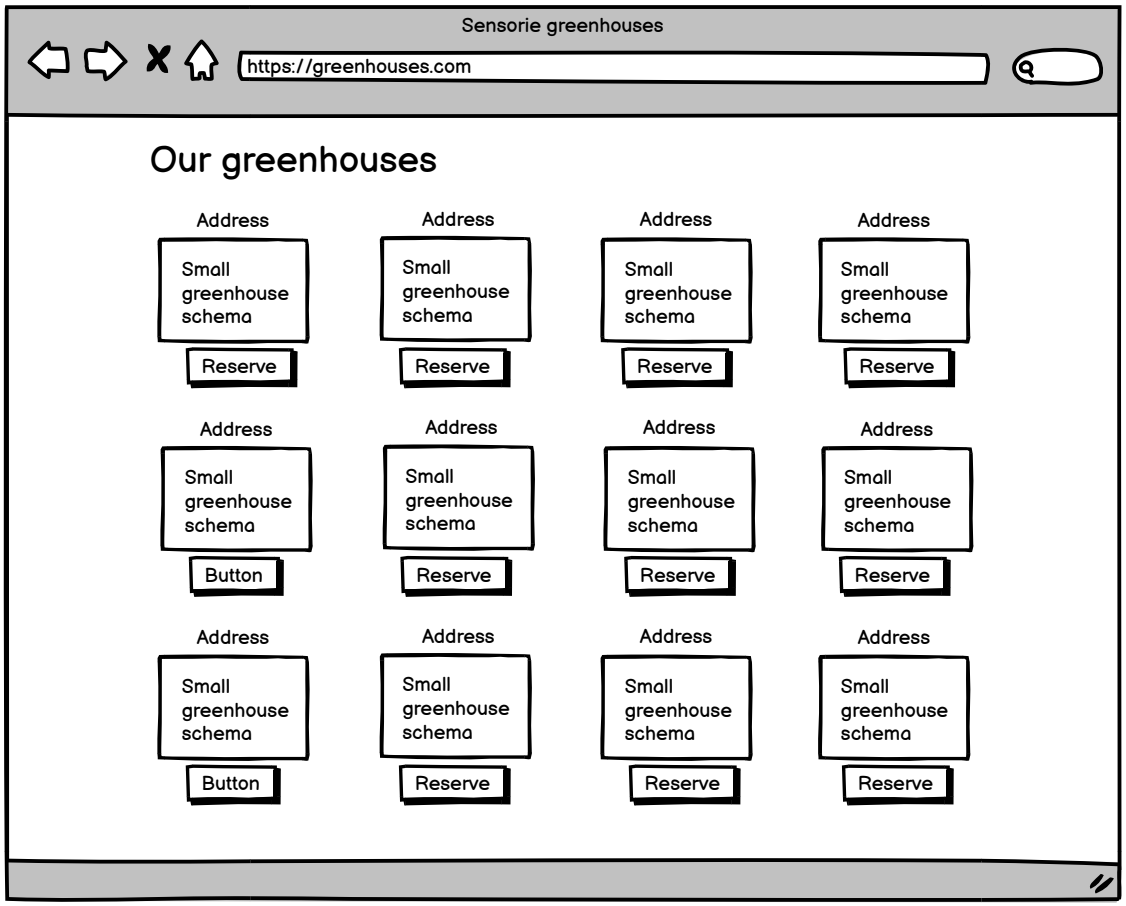


Figure 4.2: List of available greenhouses.

Since users' primary task is to reserve space and plant crops in the greenhouse, developing a clear listing page was essential. This page allows users to select the desired greenhouse by displaying critical information such as the address and current occupancy status. From here, users can easily navigate and make a reservation.

Wireframe: Greenhouse reservation

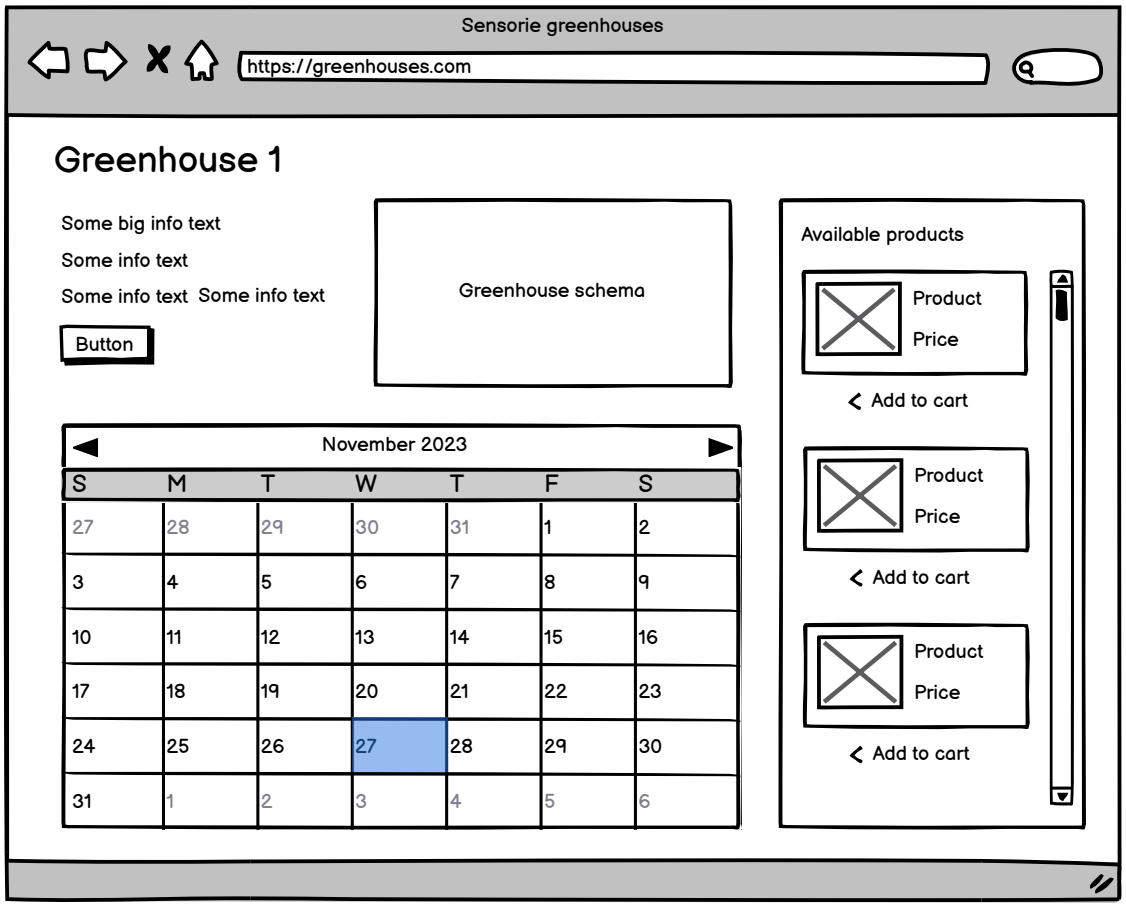


Figure 4.3: Greenhouse reservation page.

After selecting a greenhouse, the user is redirected to the registration page. In this section, additional information about the selected greenhouse is displayed, and the user can choose the desired starting date and specific cells for their planting needs. Users can not only reserve space in the greenhouse, but also purchase a range of products to aid their planting activities. The corresponding module is integrated into the Reservation page with available products in a greenhouse.

Wireframe: Greenhouse details

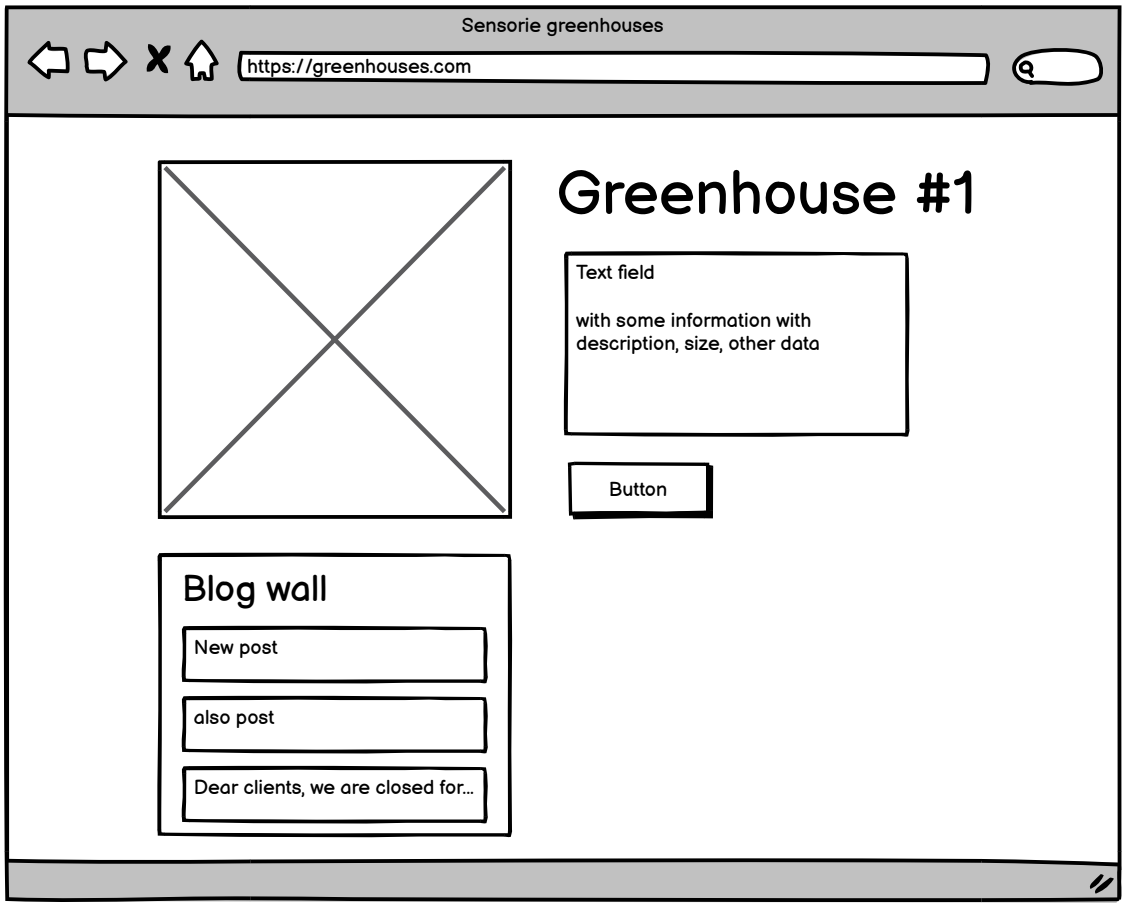


Figure 4.4: Greenhouse details page.

From the reservation page, users can conveniently access a detailed page about the greenhouse. This page contains a wealth of information, including photographs, a full description, and a news blog where users can see all the latest updates related to the greenhouse.

Wireframe: Checklist with tasks

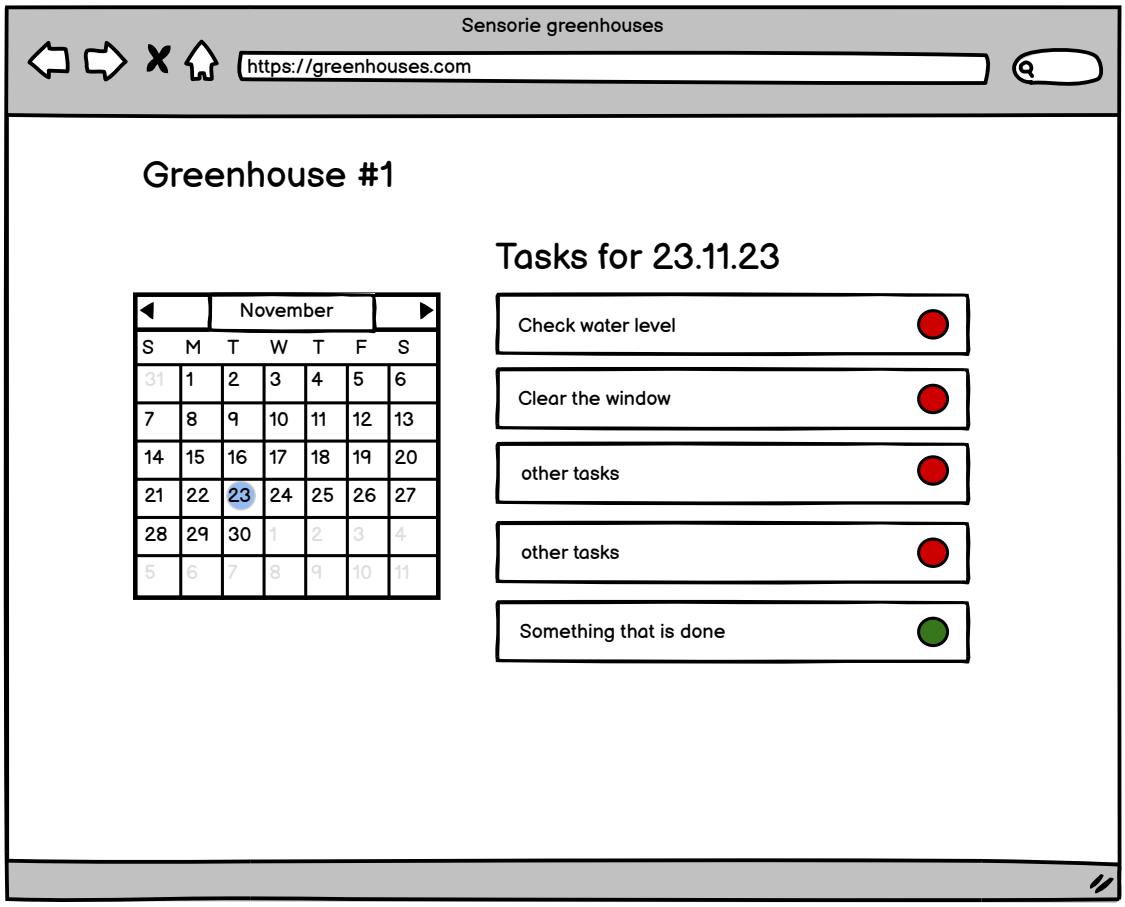


Figure 4.5: Checklist with tasks page.

For greenhouse workers, the primary interface is focused on checklists and tasks. This section features a simple list layout with a date selector, allowing workers to view tasks assigned for a specific day. Each task clearly outlines its objectives and includes a distinct indicator to show whether it has been completed.

4.3.2 Detailed mockup

After the wireframes were developed, detailed mockups were designed for two key pages of the interface using Figma[6]. Although full layout mockups were not created for every other page, this phase was initiated to further refine the visual aspects of the main pages and enhance the user experience by providing a more precise representation of the final project. By focusing on two critical pages, the design process could be optimized to ensure that the core elements of the interface were given due attention. The creation of detailed mockups for these pages served as an advanced stage of design, where colour schemes, typography, and UI elements were selected and crafted to align with the project's aesthetic and functional goals. These mockups would serve as a solid foundation and style guide for the subsequent design of the remaining pages, ensuring consistency and coherence throughout the interface.

Mockup: List of greenhouses

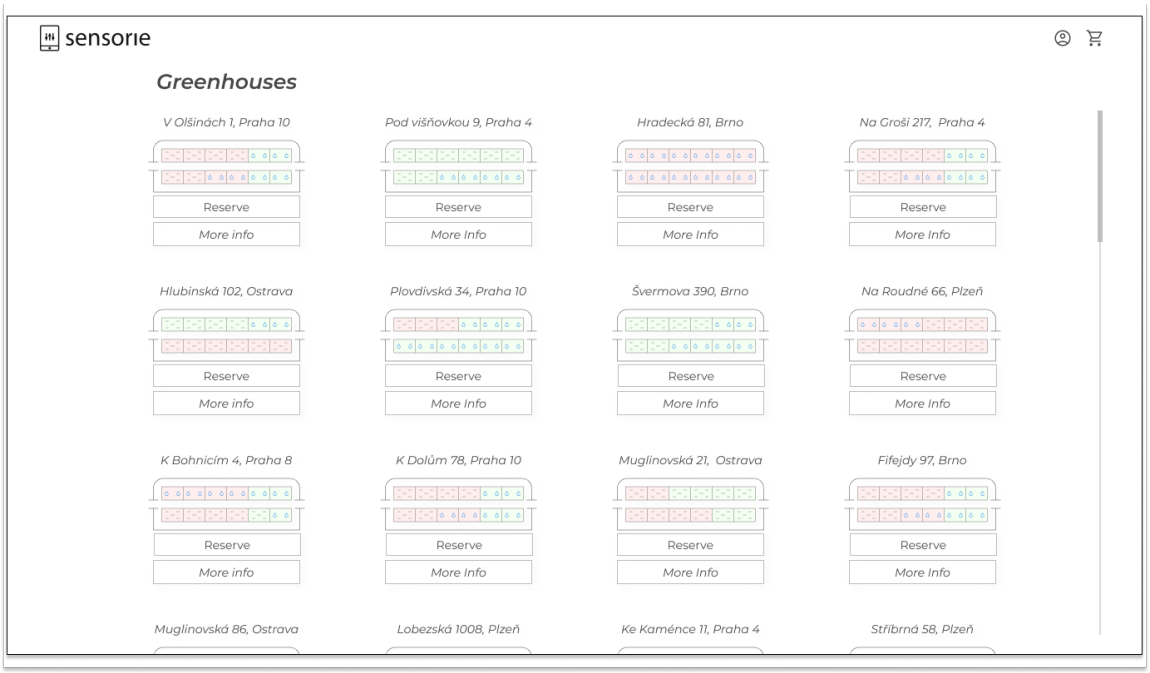


Figure 4.6: Detailed greenhouse list mockup.

As described earlier, the page for the list of available greenhouses is one of the key ones. In the detailed design, each greenhouse will have two buttons: one will allow the user to proceed with reservation, and the second will show detailed information about the greenhouse. The main element of the page is a small schema that shows the occupancy of the greenhouse (using red colour), as well as the location of the individual cells available in the greenhouse. The user can easily understand what types of cells are available depending on the icon. The primary colours of the entire design are minimalistic white, black and grey.

[6] https://www.figma.com/

Mockup: Greenhouse reservation

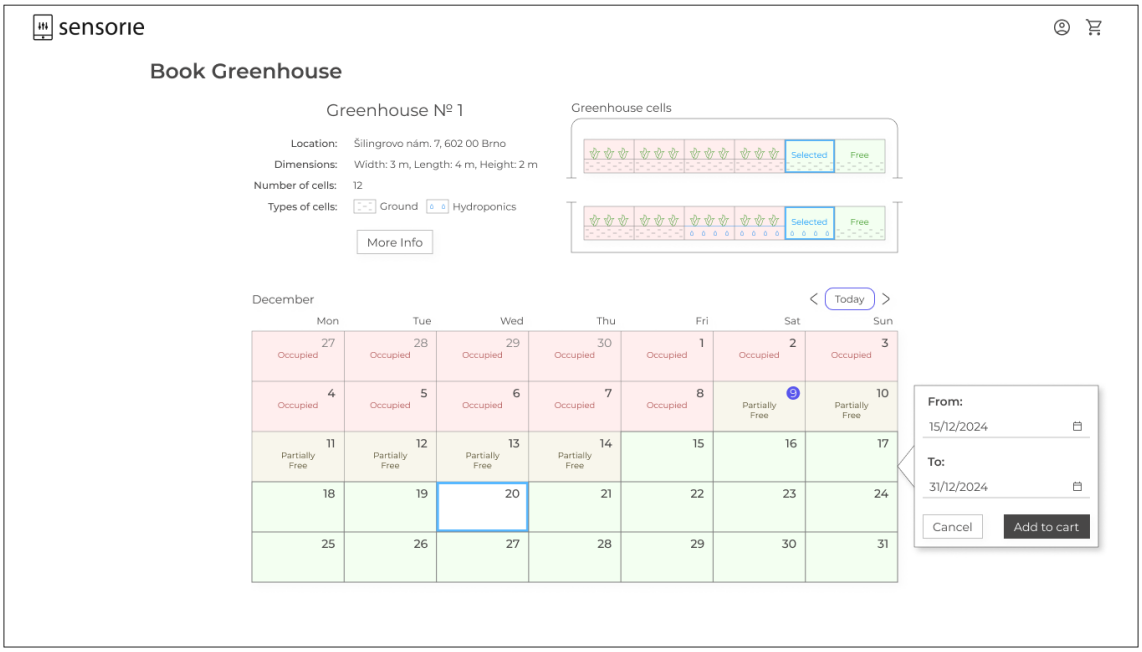


Figure 4.7: Detailed reservation page mockup.

The reservation page maintains the colour scheme specified earlier, ensuring visual continuity from the previous page. It presents concise greenhouse information alongside a button directing users to a page with detailed information. The mockup here illustrates the reservation process rather than the page's initial state. It features a scheme of the greenhouse showing occupancy, allowing users to select their desired cells from those available; in this example, two cells are selected. A calendar lets the user choose the start date of the reservation. Following this selection, a modal window appears, offering the user options to determine the duration of their reservation.

4.4 Overall system architecture

The schema below illustrates the application's final architecture, offering a minimalist view of how different services communicate with each other.

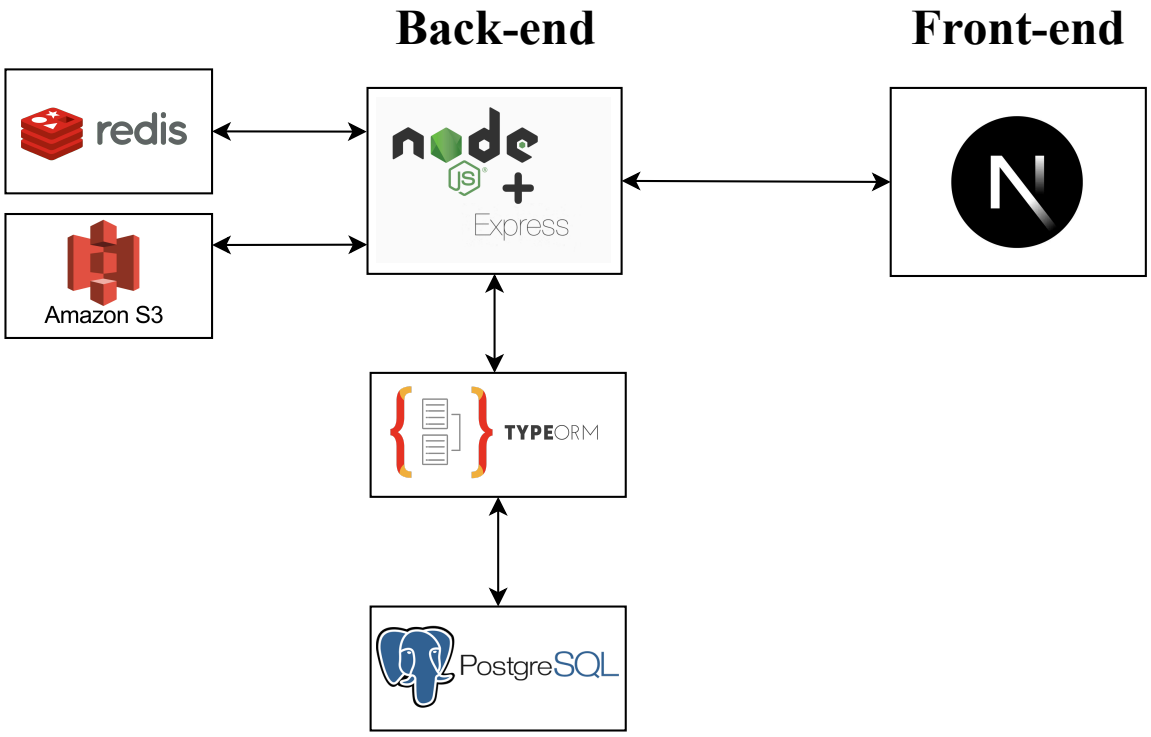


Figure 4.8: System architecture.

Detailed architectural description

- **Back-end:** Express.js serves as a backend framework and primarily manages API routes and business logic. Additionally, it handles the setup of middleware for request processing, authentication/authorization, and error handling.
- **Database and ORM:** The backend, using TypeORM, retrieves and stores persistent data in the PostgreSQL database.
- **Redis:** Employed as a session store, Redis efficiently manages and persists session data accessible via express-session middleware.
- **AWS S3:** The backend leverages AWS S3 to store static files such as images. By transferring the storage responsibilities to S3, the local server is unburdened from directly managing and serving these files, enhancing overall system efficiency.
- **Front-end:** As a front-end framework, Next.js communicates with the backend through API calls to fetch or send data.

Chapter 5

Implementation

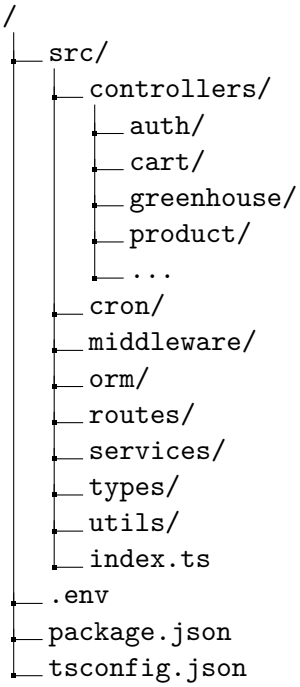
This chapter will delve into the practical aspects of implementing the application, focusing on both backend and frontend functionalities. The implementation details are divided into structured subsections to provide a comprehensive overview of server and client-side components.

5.1 Server

This section details the implementation of the server component of the application, encompassing both the API and the business logic. It begins with a description of the organization of the file structure, followed by a discussion of the resulting API endpoints. Additionally, this section covers the integration and utilization of NPM packages and external services.

5.1.1 Project structure

The application's structure is organized into distinct folders inside the `src` directory, each containing files that perform specific functions. Given that Express.js does not mandate a specific architectural pattern and is not inherently an object-oriented programming (OOP) or model-view-controller (MVC) framework, a two-tier architecture was adopted for this project. This architecture is a simplified version of the conventional three-tier design. The typical three-tier architecture defines distinct levels for the Router, Controller, and Data Access. However, in the two-tier model used here, employing an ORM, the Controller level directly interacts with the database without an intermediary abstraction layer. This streamlined approach was chosen considering the project's scope and the understanding that even substantial future expansions would not necessitate such abstractions.



Detailed description

- **controllers:** This directory contains files for handling the application's business logic, organized into subfolders corresponding to specific functionalities such as authentication, cart/product management, greenhouse operations, etc. Within these subfolders, controllers are tasked with processing incoming data, data validation, and directly interacting with the database using ORM. After processing the data, controllers generate the appropriate responses to send back to the client.
- **cron:** Includes scripts configured to schedule periodic tasks through cron, such as removing expired reservations and clearing redundant error logs.
- **middleware:** Contains functions that execute before requests reach final handlers (controllers) and are used for authentication/authorization and file uploading.
- **orm:** Folder with a configuration file for connecting to the database and entity definitions.
- **routes:** Defines API endpoints, mapping incoming HTTP requests to specific controllers.
- **services:** This folder contains configuration files for various services, such as Redis, AWS S3, logger, mailer, etc.
- **types:** Stores TypeScript files that define custom types and interfaces.
- **utils:** Provides utility functions used across the application for tasks such as error handling, QR code generation, email sending functions or validations.
- **.env:** A file in which configuration settings, environment variables, and sensitive information are securely stored and can be loaded into the process.env, a global variable added by Node.js at runtime.

5.1.2 API endpoints

As previously mentioned, REST (Representational State Transfer) was selected as the architectural style for this project's application programming interface (API). The subsequent sections will detail the specific endpoints provided by the API and briefly describe their functionalities.

Authentication

Method	Path prefix	Name	Description
POST	/v1/auth	/register /login /logout	Handle basic authentication operations.
POST	/v1/auth	/change-password	Changes the user's password.
POST	/v1/auth	/forgot-password	Sends a password reset link to the user's email if the account exists.
POST	/v1/auth	/update-password	Updates a user's password using a verified token from a reset email.
GET	/v1/auth	/confirm-email/:token	Sends a link with a token to confirm email.
GET	/v1/auth	/resend-confirmation	Allows to resend confirmation link. Is limited to 3 links per hour.

Table 5.1: API endpoints for user authentication.

Users

Method	Path prefix	Name	Description
GET	/v1/users	/workers	Shows all users with Worker role.
GET	/v1/users	/session	Returns data of the user who is currently logged in (whose userID is present in the session cookie)

Table 5.2: API endpoints for retrieving user information.

Cart

Method	Path prefix	Name	Description
GET	/v1/cart	/items	Shows all items in the user's cart.
POST	/v1/cart	/add /remove	Adds or removes items from the cart.
PATCH	/v1/cart	/product/increment /product/decrement	Increments or decrements the quantity of product in the cart.

Table 5.3: API endpoints for managing shopping cart.

Products

Method	Path prefix	Name	Description
GET	/v1/products	/:id	Shows product by ID.
GET	/v1/products	/greenhouse/:id	Shows all products in the specified greenhouse.
POST	/v1/products	/create	Creates a new product.
PATCH	/v1/products	/:id	Edits the specified product.
DELETE	/v1/products	/:id	Deletes the specified product.

Table 5.4: API endpoints for managing orders.

Greenhouses

Method	Path prefix	Name	Description
GET	/v1/greenhouses	/	Shows all greenhouses.
GET	/v1/greenhouses	/:slug	Shows the specified greenhouse.
POST	/v1/greenhouses	/create	Creates a new greenhouse.
PATCH	/v1/greenhouses	/edit	Edits the specified greenhouse.
DELETE	/v1/greenhouses	/:slug	Deletes the specified greenhouse.

Table 5.5: API endpoints for managing greenhouses.

Tasks

Method	Path prefix	Name	Description
GET	/v1/tasks	/greenhouse/:id	Shows all tasks in the specified greenhouse.
POST	/v1/tasks	/greenhouse/:id/create	Creates a new task in the specified greenhouse.
PATCH	/v1/tasks	/:id/complete	Marks the specified task as completed.
DELETE	/v1/tasks	/:id	Deletes the specified task.

Table 5.6: API endpoints for managing tasks for workers.

Posts

Method	Path prefix	Name	Description
GET	/v1/posts	/greenhouse/:id	Shows all posts in the specified greenhouse.
POST	/v1/posts	/create	Creates a new post.
PATCH	/v1/posts	/:id	Edits the specified post.
DELETE	/v1/posts	/:id	Deletes the specified post.

Table 5.7: API endpoints for managing posts.

Reservations

Method	Path prefix	Name	Description
GET	/v1/reservations	/all	Shows all active reservations.
POST	/v1/reservations	/create	Creates a new reservation.
PATCH	/v1/reservations	/extend	Extends the specified reservation.
DELETE	/v1/reservations	/end	Ends the specified reservation.

Table 5.8: API endpoints for managing reservations.

Orders

Method	Path prefix	Name	Description
GET	/v1/orders	/	Shows all user orders.
GET	/v1/orders	/latest	Shows the user's last order.
POST	/v1/orders	/create	Creates a new order with items in the user's cart.

Table 5.9: API endpoints for managing orders.

5.1.3 Key aspects of back-end implementation

Authentication

For this project, a session-based authentication strategy was chosen. The approach operates as follows:

1. After the user logs in, the server generates a `sessionId`, which is signed using a securely stored secret key.
2. This `sessionId` is:
   - stored in a session store, in the case of this project – Redis, and
   - sent to the client's browser as a secure HTTP-only cookie.
3. Upon receiving the cookie, the browser stores it within its cookie storage.
4. For every subsequent request made to the server, the browser includes this cookie.
5. When the server receives a request with a cookie, it verifies the `sessionId` from the cookie against the session information stored in the store.
6. If the `sessionId` is a valid session, the middleware populates `request.session` with the session data. The user's ID is accessible via `request.session.userId`.

This authentication approach is implemented using the `express-session` npm package, which provides middleware that is executed before processing each request. To store session data in the Redis store, the `connect-redis`[1] package is utilized. The session cookie is configured with the `HttpOnly` flag, which ensures that the cookie cannot be accessed or manipulated via client-side scripts, such as JavaScript. This helps protect the cookie's data from potential cross-site scripting (XSS) attacks[2], as the cookie is not exposed to the client-side environment. Additionally, the `Secure` flag ensures that the cookie is sent only over HTTPS connections, thereby preventing threats such as 'man-in-the-middle' (MITM) attacks[3]. To help prevent cross-site request forgery (CSRF)[4], the `SameSite` attribute is set to `Lax`, allowing the cookie to be sent with top-level navigations to the server from external sites.

Authorization

Authorization is implemented using middleware that executes after the `express-session` middleware has been completed. This authorization middleware is invoked for each protected route. The code snippet below shows how only users with the „ADMIN“ role can delete a greenhouse.

```
router.delete('/:slug', checkAccess(['ADMIN']), deleteGreenhouse);
```

The process employed by the `checkAccess` middleware is straightforward: it uses the session to identify the requesting user, retrieves their details from the database, and verifies their role against the predefined list of roles allowed for the action.

[1] https://www.npmjs.com/package/connect-redis
[2] https://owasp.org/www-community/attacks/xss/
[3] https://www.imperva.com/learn/application-security/man-in-the-middle-attack-mitm/
[4] https://owasp.org/www-community/attacks/csrf

Managing reservations

In the reservation module of the web application, a specific policy has been implemented to control the start date of reservations. This policy is designed to prevent a greenhouse cell from remaining unused for an extended period between reservations. For instance, to prevent a situation where today's date is January 1st, and a cell appears unoccupied, but it has already been reserved from February 3rd. Users are restricted from setting a reservation start date too far in advance. Specifically, they can only book a start date one week ahead. This limitation ensures that the cells do not sit idle for long periods, such as a whole month, which would be impractical given that the growth period for plants cannot be precisely predicted and is often longer than a month.

As was previously mentioned, a reservation is initiated by sending a `POST` request to the `/v1/reservation/create` endpoint. The request payload must include an array of the selected cell IDs and the reservation's end date. To prevent the cells from remaining unoccupied for a few days before the user-set „reservation start" date, this date is automatically set to the current day. When the request is processed, the start and end dates are assigned to the selected cells, and the `isOccupied` boolean flag is set to true, indicating that the cells are no longer available for reservation.

Reservations can also be extended or ended. In the case of an extension, the application updates the existing reservation to reflect a new end date for the same selected cells. For business reasons, reservation extensions are offered to users only within three days following the conclusion of the original reservation period. Should the user choose not to renew, the system will interpret this as an intent to end the reservation. This policy allows users to delay payment for the subsequent reservation period by offering a brief grace period to secure an extension.

Emailing implementation

The application's emailing functionality was implemented using node-mailer[5], an npm module. The configuration was set up to send emails through Gmail's Simple Mail Transfer Protocol (SMTP) server rather than a private, dedicated SMTP server. This approach was chosen for its simplicity and ease of integration during the initial development stages.

Emails are dispatched in response to specific back-end endpoint triggers. Key events initiating an email include user registration, order confirmation or password update. Each action activates a predefined email template that is automatically customized with user-specific information and sent to the intended recipient.

While the current system effectively meets the immediate needs for user communication, using Gmail's SMTP server represents an area for potential future enhancement. Transitioning to a private SMTP server could offer greater control over the emailing processes and improve email deliverability and security.

Payment management

For convenient payment integration, the `qrcode`[6] npm package was utilized to generate various QR codes. Utilizing the Short Payment Descriptor (SPAYD)[7] format, the system generates a QR code containing a specific variable symbol, price, and necessary bank in-

[5] https://www.npmjs.com/package/nodemailer
[6] https://www.npmjs.com/package/qrcode
[7] https://qr-platba.cz/pro-vyvojare/specifikace-formatu//

formation. This approach was chosen to avoid the higher costs associated with integrating a full payment system. Consequently, QR codes serve as a more convenient alternative to traditional bank transfers, which remain available for users preferring that method.

This work does not handle automatic payment confirmation and receipt or interact with banking APIs. Instead, payment confirmations are manually verified within the administrator's personal account. This aspect is identified as a potential area for improvement in Chapter 7.

5.2 Client

This section outlines the implementation of the front-end part of the application, covering the project's structure, routing, and key aspects. It explores the critical aspects of the front end, including the use of UI libraries, communication with API, handling forms and authentication.

5.2.1 Project structure

Next.js is unopinionated about project structure and file conventions, apart from routing, which primarily depends on the framework's version. This project utilizes version 14, which features the app router - a dynamic routing feature that replaced the previous page-based routing approach since version 13. The app router allows for more flexible and efficient routing, simplifying the creation of complex application structures. This project follows the guidelines in the Next.js 14 documentation, fully leveraging app router features and capabilities. As a framework for React that does not impose specific file organization conventions, this project employs a popular method of grouping files by type. The following subsections detail the project's structure and explain the roles and functions of specific directories and files.

```
/
├── public/
├── src/
│   ├── api/
│   │   └── greenhouse.ts
│   ├── app/
│   │   └── ...
│   ├── components/
│   │   ├── forms/
│   │   ├── icons/
│   │   ├── layout/
│   │   ├── shop/
│   │   ├── ui/
│   │   └── ...
│   ├── utils.ts
│   └── middleware.ts
├── .env
├── package.json
├── next.config.mjs
├── tailwind.config.ts
└── tsconfig.json
```

Detailed description

- **public:** The directory contains static assets like images.
- **api:** This folder contains functions for interfacing with API endpoints. A detailed description can be found in section 5.2.3.
- **app:** The directory, which structures the application's routing. Further details are provided in section 5.2.2.
- **components:** Contains reusable React components, like forms, icons, layout components, page-specific components or basic UI components.
- **utils:** A file with utils functions that are used globally throughout the app. In the case of this project, it only contains `cn` function to merge Tailwind CSS styles properly.
- **middleware:** A file with custom middleware that is used primarily for authorization and route protection.
- **tailwind.config.ts:** The Tailwind CSS configuration file, which contains custom colour, margins, etc.
- **.env:** A file in which configuration settings, environment variables, and sensitive information are securely stored and can be loaded into the process.env, a global variable added by Node.js at runtime.
- **package.json:** This file defines the project's dependencies, scripts, and other configurations necessary to run and manage the project.

5.2.2 Routing

As was mentioned earlier in section 3.5.4, Next.js uses a file-system-based router where:

- Folders are used to define routes. A route is a single path of nested folders, following the file-system hierarchy from the root folder down to a final leaf folder that includes a page.tsx file.
- Files are used to create UI that is shown for a route segment.

Routes in Next.js can be nested[8] or dynamic[9]. Dynamic routing is utilized when segments of the URL, such as an `[id]` or `[slug]`, are not known ahead of time. More details can be found in the official Next.js documentation [20].

The directory tree below outlines all the primary pages implemented in the `app` folder, defining the navigation structure of the application:

```
/
└── app/
    ├── auth/
    │   ├── register/
    │   │   └── page.tsx/
    │   └── ...
    ├── greenhouses/
    │   ├── [slug]/
    │   │   ├── checklist/
    │   │   │   └── page.tsx
    │   │   ├── reserve/
    │   │   │   └── page.tsx
    │   │   └── page.tsx
    │   └── page.tsx
    ├── order/
    │   ├── [id]/
    │   │   └── page.tsx
    ├── profile/
    │   ├── my-details/
    │   │   └── page.tsx
    │   ├── orders/
    │   │   └── page.tsx
    │   └── page.tsx
    ├── shop/
    │   ├── [slug]/
    │   │   └── page.tsx
    │   └── page.tsx
    ├── actions.ts
    ├── globals.css
    ├── layout.tsx
    └── page.tsx
```

[8] https://nextjs.org/docs/app/building-your-application/routing#nested-routes
[9] https://nextjs.org/docs/app/building-your-application/routing/dynamic-routes

5.2.3 Key aspects of front-end implementation

Communication with API

Communication with the API is managed through distinct functions housed in the `/api` folder. Each function is designed to construct a request for a specific endpoint, utilizing the web fetch API, which Next.js extend[10]. This extension allows each request to define its own persistent caching semantics. Furthermore, these API-call functions are responsible for setting the appropriate headers, including cookies as necessary, preparing the required payload, and processing the responses received.

Authentication and authorization

As previously discussed in section 5.1.3, authentication in the whole app is managed using the session-based approach, while authorization is handled by additional middleware that checks the user's role. When the front end needs to perform an action, it sends a request to the server, including credentials (cookie). The server then processes or denies the request based on the user's permissions.

The application's client side determines which pages or specific sections are accessible to a user. For instance, standard users cannot access checklists, and only administrators have permission to add new greenhouses or products. Next.js supports the creation of middleware similar to those used in Express.js, allowing the application to check a user's rights before the page is rendered and hydrated. Route protection was implemented using this feature.

Additionally, some parts of the front-end require information about the user whose session is currently active, and a separate endpoint `/v1/users/session` is used for this, as described in section 5.1.2.

Server side rendering

In Next.js, server-side rendering (SSR) is a key feature that enhances performance by rendering JavaScript content on the server before sending it to the client's browser. SSR pre-renders each page at the requested time, allowing the server to deliver fully rendered HTML in response to user actions. This is especially beneficial for users on slower connections as it improves load times. By default, pages in Next.js render on the server without requiring additional configuration.

While SSR offers significant advantages for performance and SEO, there may be scenarios where client-side rendering (CSR) is preferable, such as when real-time data updates are required or when utilizing client-specific interactivities like button clicks. In Next.js version 14, transitioning from SSR to CSR within the new App Router environment is streamlined by adding a „use client“ directive above the imports at the top of a component file.

It's important to use the client keyword judiciously – typically within individual components rather than entire pages or large parent components – to maintain the benefits of SSR for initial page loads while leveraging CSR for specific interactive elements. Such an approach is commonly used during implementation. In the code, a simple button may often be singled out into a separate component to make it interactive. This hybrid approach ensures optimized performance and responsiveness, providing a seamless user experience that balances SSR and CSR benefits.

[10] https://nextjs.org/docs/app/api-reference/functions/fetch

Headless components

At its core, React lets build applications by breaking down an application into reusable UI components. Creating these components can be approached in several ways. One common method involves utilizing libraries such as Bootstrap[11], Material-UI[12], or Chakra UI[13]. These libraries provide a wide range of pre-built components with comprehensive styles, making creating a visually appealing user interface quick and easy. However, the downside is that these components have poor customization and control. An alternative option is to use so-called „headless“ libraries, which provide the underlying functionality and accessibility features of UI components without any visual styling or layout. This approach gives maximum flexibility to create custom-styled components that fit any specific design needs. Opting for headless libraries can combine the benefits of rapid development with extensive control over style and customization.

Radix UI was chosen as this project's headless library for creating UI components. It provides the logic and behaviour for complex UI elements such as dialogues, accordions, and drop-down menus. In conjunction with Radix UI, the shadcn/ui[14] collection of wrappers has been used. This collection provides default styling and an additional abstraction layer, serving as the foundational reference for developing a component library tailored to the project's design specifications.

Forms

To implement the forms in this project, the `react-hook-form` library was used alongside the `zod` package for validation purposes. `React-hook-form` is designed to minimize re-renders and optimize form performance by leveraging uncontrolled components. This approach significantly reduces the amount of boilerplate code needed and enhances form responsiveness.

Initially, a schema for each form is created using `zod`, specifying the necessary value types and constraints. For example, the schema below is used in `ProfileFormEmail.tsx`, which changes user email.

```
const formEmailSchema = z.object({
  email: z.string().email(),
});
```

Here, the input in the email field must be a string that conforms to a valid email format, ensuring data validation at the schema level. This method simplifies validation logic, making it both more readable and maintainable.

The user interface for the forms was then constructed utilizing hooks provided by `react-hook-form` and components from Radix UI.

Furthermore, `react-hook-form` offers a defaultValue property for implementing editable forms. This property can populate the form with existing data from the API, facilitating the display and subsequent modification of pre-existing values within the form.

[11] https://getbootstrap.com/
[12] https://mui.com/
[13] https://v2.chakra-ui.com/
[14] https://ui.shadcn.com/docs

Chapter 6

Testing and optimization

6.1 System design testing

During the development process, it's imperative to ensure that the user interface (UI) and user experience (UX) meet the expectations of the end-users. Mockup testing is critical to validate design choices and gather feedback from potential users before committing significant resources to full-scale implementation.

The design was tested at various stages of development, from initial wireframes to detailed mockups. This iterative process helped make numerous improvements and refinements to the final design. Besides, it allowed for correcting errors during the layout phase, thus enhancing the overall quality and effectiveness of the user interface.

The mockup testing was conducted through one-on-one calls with **four** respondents from the target audience group. The testing process was divided into two stages:

6.1.1 Early wireframe testing

Process

In the initial phase of the design process, basic wireframes were introduced to the respondents. This stage was geared primarily towards evaluating the designs' intuitiveness and user-friendliness. The objective was to assess the overall layout, navigation flow, and content organization.

During these sessions, the functionality and concepts of specific pages and modules were explored. Participants shared their views on features they found appealing and areas they felt needed enhancement.

Results

Based on the feedback gathered, adjustments were made to certain aspects of the information architecture to improve data positioning and user interaction:

1. **Greenhouse list:** Now, each greenhouse in the list will have an additional button to quickly navigate to its detailed information.

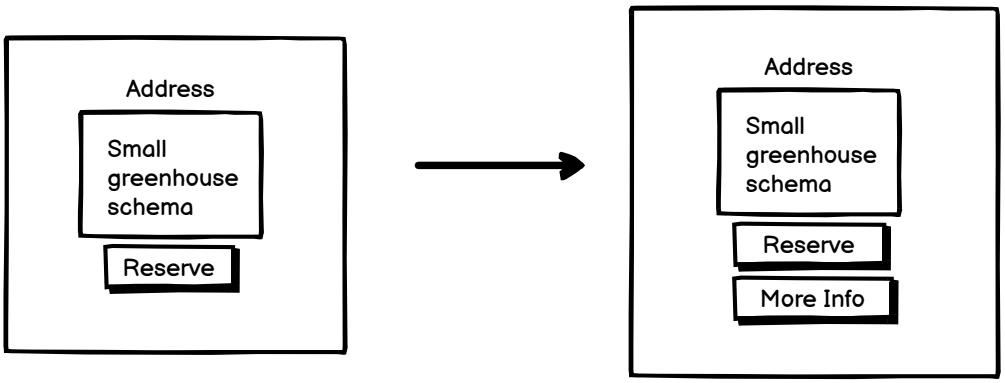


Figure 6.1: Greenhouse list item wireframe comparison: old vs. new.

2. **Greenhouse reservation:** Initially, the design combined the shop with the reservation system for convenience, enabling users to add necessary products with just a few clicks. However, feedback highlighted that this crowded the page and conflicted with the single responsibility principle. To improve clarity and usability, the design was revised to split these modules into two separate pages.

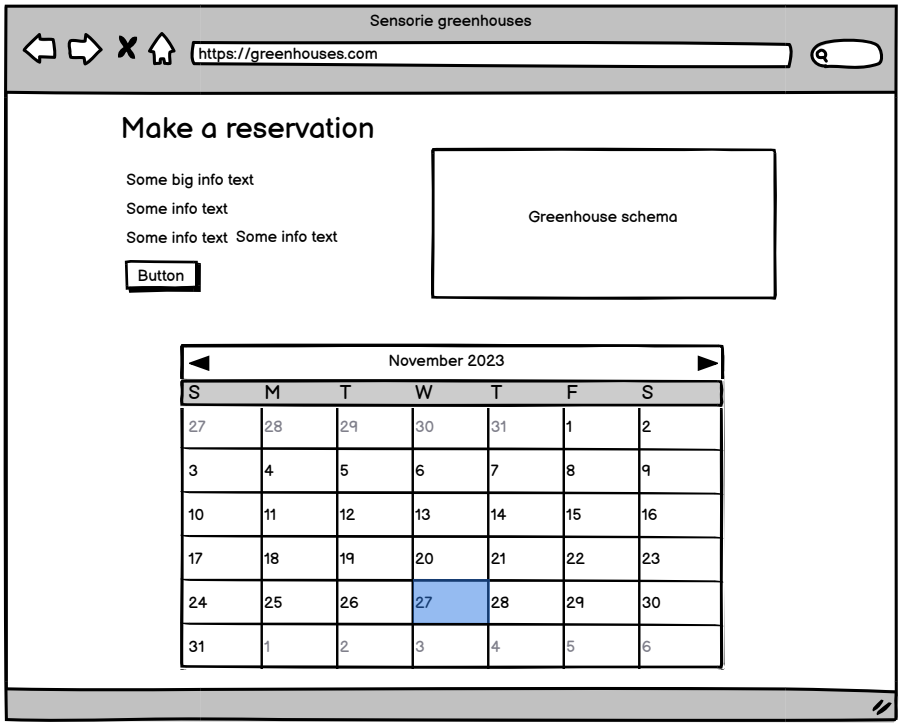


Figure 6.2: New greenhouse reservation page.

3. **Greenhouse shop:** Following the initiated separation, a distinct shop page was designed. The separation has enabled the shop to offer full functionality, including filters and the convenient viewing of available products. This two-point improvement enhanced user experience by simplifying navigation and focusing on core tasks within each module.

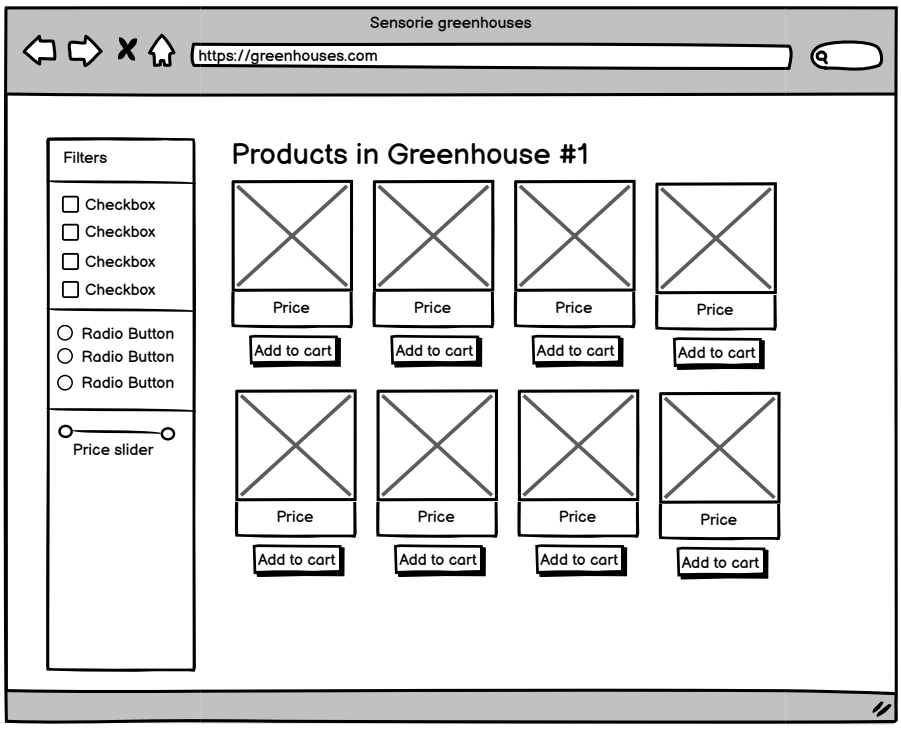


Figure 6.3: New greenhouse shop page.

6.1.2 Detailed mockup testing

Process

Following the development of these mockups, which were based on improvements identified during the early wireframe testing, the design process advanced to the detailed mockup testing phase. In this stage, high-detailed mockups featuring accurate typography, colour schemes, and imagery were presented to participants. This phase is designed to simulate the actual user experience closely, allowing for a more comprehensive evaluation of the application's aesthetics and interaction dynamics.

During these sessions, the feedback focused more on the user's emotional response to the design and its ease of use in realistic scenarios. The goal of this phase is to validate the design decisions made earlier and identify any potential usability issues before moving into development, ultimately leading to a more user-friendly and engaging application.

Results

The reservation module has undergone several design modifications to improve user interaction and efficiency. The following section outlines the final implemented design that incorporates these changes:

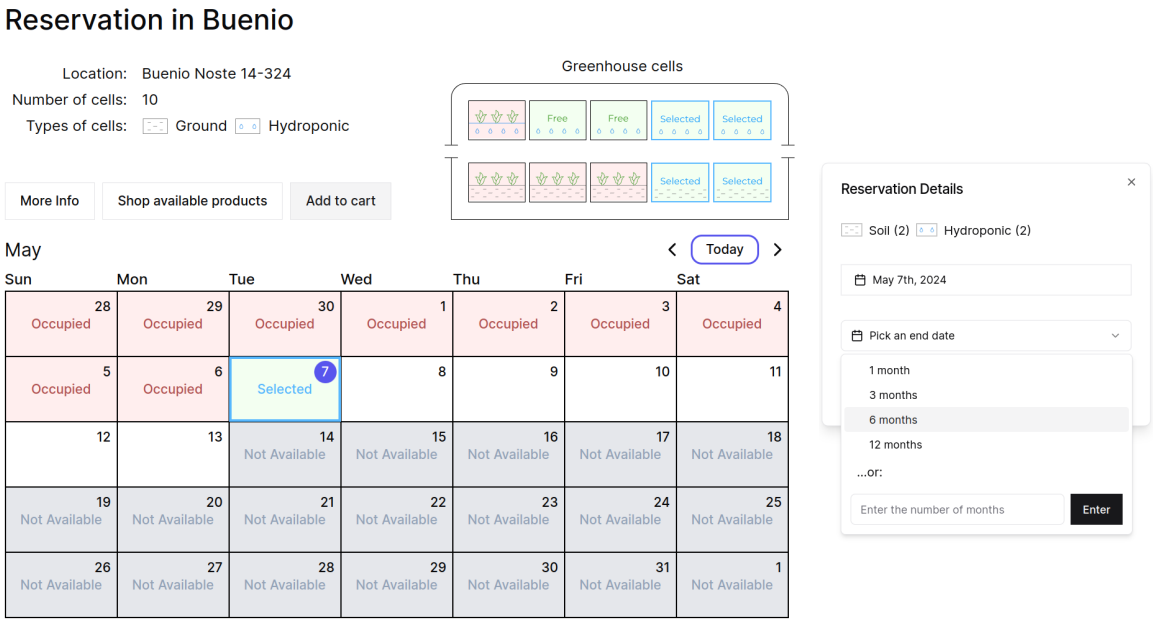


Figure 6.4: Improved reservation module. It contains brief greenhouse information, navigation buttons, a calendar, and greenhouse cells. A redesigned modal window on the right includes selected cells, the reservation start date, and an input field for the reservation duration in months.

- **Direct shop access button:** A new button has been added to the reservation page, allowing users to navigate directly to the shop page of that specific greenhouse. This button is clearly labelled and prominently positioned to ensure easy navigation.
- **Improved date selection visuals:** The calendar interface has been updated to indicate available and unavailable dates better. Unavailable dates are now distinctly marked. The visual representation of the selected date has also been enhanced to stand out, aiding users in tracking their current selection without confusion.
- **Updated 'add to cart' modal window:** The 'Add to cart' button appears on the screen after selecting the start date and the desired greenhouse cells. Once clicked, the modal window will appear and show the selection details. It also prompts users to choose the reservation duration in months. This approach allows users to review their selections and make any necessary adjustments before adding the reservation to their cart for payment.

6.2 Functional Testing

6.2.1 Back-end API

For the API testing phase of this project, Postman[1] was utilized as the primary tool to conduct manual tests on the various endpoints. This software enabled a detailed examination of request responses, ensuring the API behaved as expected under different scenarios. Through Postman, each API function was verified, from the correct handling of input data to the expected output and error-handling capabilities. The testing and debugging of the program were conducted iteratively at each stage of the development process. This approach allowed continuous improvements and refinements, ensuring potential issues were addressed promptly and effectively before the final deployment.

6.2.2 Front-end

Manual testing ensured the application's front-end met the required functionality, usability, and responsiveness standards. The primary focus was on various aspects of the user experience, including layout, design consistency, and interactive elements. Special attention was given to how these features functioned on multiple platforms and browsers to guarantee seamless operation across different devices.

The manual testing process comprehensively examined each component's behaviour under typical user interactions, such as clicking, scrolling, and entering data. React Developer Tools[2], a browser extension, proved invaluable during this process. It allowed for a thorough inspection of the application's component hierarchy, state, and props, making identifying and resolving any issues related to the React components' rendering or performance easier.

Manual testing was conducted on various devices, mainly focusing on different sizes of smartphones or desktops using browser emulators, but it was also tested on two real phones by connecting to the web app locally. This approach allowed for assessing the application's responsiveness and compatibility and identifying potential layout inconsistencies, touch input issues, or other device-specific problems that could impact the user experience across different real-world scenarios.

6.3 User experience testing

This section describes the user experience testing conducted on the developed web application. The testing evaluates the application's usability, functionality, and interactivity for both customers and greenhouse workers. The primary objective is to identify any interface issues and functional deficiencies that could be improved to enhance user interaction with the system.

6.3.1 Test preparation

Several preparatory steps were taken to ensure feedback reliability and validity before testing began. This subsection outlines each of these steps.

[1] https://www.postman.com/
[2] https://react.dev/learn/react-developer-tools

Selection of participants

This initial step involved selecting participants to represent the two distinct user groups identified for testing:

1. **Clients:** Five users between the ages of 20 and 74 were chosen, aligning with the Community Greenhouse's direct target audience. This group included two individuals who had participated in previous design testing, ensuring a mix of new and experienced users. This diversity ensures that the application is intuitive and functional for users encountering it for the first time.

2. **Greenhouse workers:** Two users, both of whom had previously engaged in mockup testing, were selected to represent greenhouse workers. To adequately prepare them, a detailed orientation on the typical processes and operations of a Community Greenhouse was conducted. This simulation mirrors the introduction typically given to a new worker, clarifying the purpose and responsibilities of their role.

Preparation of test scenarios

The next step was the creation of detailed usage scenarios that cover all key functionalities of the application.

Test Scenarios for the Clients:

1. Searching for a specific greenhouse, its description and photographs:
    - While on the home page, locate the appropriate tab in the navigation menu and navigate to the list of available greenhouses.
    - While on the list of all greenhouses, locate the desired greenhouse and go to its detail page.
    - Find the corresponding description and photographs.
2. Searching for a specific blog post in a greenhouse details page:
    - While on the detailed page of the greenhouse, find the blog posts module and the relevant post with a specific date.
3. Greenhouse reservation:
    - While on the list page of greenhouses, select the desired greenhouse and proceed to the reservation page.
    - Make a reservation by selecting the next day as the reservation start date and selecting the two cells: the soil cell in the lower right corner and the hydroponic cell in the upper right corner.
    - Specify the length as five months (enter the number of months in the form input) and add the reservation to the cart.
4. Purchasing necessary products:
    - While on the reservation page, navigate to the shop page to find products available in the corresponding greenhouse.

- Find a specific product by using the category filter.
- Add this product in a quantity of three to the cart.

5. Confirmation and payment of order:
    - While on any application page, open a cart and change the item quantity in the cart to two units.
    - Proceed to check out.
    - Confirm the order and complete payment via QR code.
6. Managing Personal Account:
    - Access the personal account and review recent orders and personal information.
    - Locate a specific order from the purchase history and view its detailed information.
    - View active reservations.
    - Locate the recently created reservation and extend it by an additional month.

Test Scenarios for the Greenhouse workers:

1. Adding and editing a blog post:
    - While on the list page of greenhouses, select the specific greenhouse and proceed to the details page.
    - Find the button to create a new blog post and add a blog post with a specific title and content.
    - After adding, find the edit button for the blog post and change the title.
2. Viewing active tasks in the greenhouse:
    - While on any application page, navigate to the checklists module and select the specific greenhouse from the list to view all active tasks.
    - Using the calendar, find tasks that must be completed on a specific date.
    - View the detailed information about a specific task.
    - Pretend the task is completed and mark it as completed in the system.
3. Creating and editing information about a greenhouse:
    - On the list page of greenhouses, click the button and create a new greenhouse with specified parameters.
    - The greenhouse should have two rows and five columns. Meanwhile, the bottom row must be marked as hydroponic-type cells.
    - Create a greenhouse and navigate to its details.
    - Edit the greenhouse information by entering a new address and description.

6.3.2 Conducting the Testing

The testing process was divided into several successive stages, each conducted through personal online meetings:

1. **Orientation session:** This initial session involved a brief meeting with each participant to explain the purpose of the testing, introduce the context of Community Greenhouses in general, and outline the tasks to be completed.

2. **Testing session:** During this phase, each participant interacted with the application individually. They were given tasks to complete based on predefined scenarios, and the environment was controlled and supervised to ensure consistent and reliable feedback collection. Participants were encouraged to express their thoughts and actions out loud while engaging with the application.

3. **Discussion of each task:** Once the participant finished his tasks, he participated in a short discussion. This discussion aimed to obtain prompt feedback on any difficulties faced during the tasks and to identify any particularly interesting or problematic aspects of the interface. The participants were asked to share any points of confusion or difficulty and highlight helpful features.

6.3.3 Testing results and evaluation

This section presents the test results and outlines the metrics used to evaluate them. It details the outcomes of specific test scenarios, providing a basis for the subsequent optimization of the application.

Evaluation parametrs

Three key parameters were established to evaluate the user testing of the web application, which directly affects the quality of the final application.

- **Effectiveness:** This parameter measures the users' ability to complete tasks using the application successfully. It is quantified by the percentage of tasks completed correctly and the overall capability of the application to meet users' goals.

- **Efficiency:** This metric evaluates the time and effort required for users to complete tasks. It is crucial as it directly impacts the user experience, reflecting how intuitively and quickly users can navigate the application and achieve their objectives without undue delay or confusion.

- **Satisfaction:** User satisfaction assesses users' subjective responses regarding their interactions with the application. This parameter is evaluated based on direct user feedback and satisfaction surveys regarding the application's use. It was decided to measure it on a scale from 1 to 5.

Together, these parameters form the basis for assessing the quality of the user interface, ensuring that the application is not only functional but also user-focused and effective. Evaluation based on these parameters can identify areas for optimization and improvement, enhancing the overall user experience.

Client testing results

The table below presents the evaluation results for each user task based on the three established scoring parameters: Effectiveness, Efficiency, and Satisfaction. These approximate values quantitatively measure the user experience during the conducted testing.

Name	Effectiveness	Efficiency (Time)	Satisfaction (Rating)
Greenhouse searching	100%	1.5 min	4.8/5
Blog post searching	100%	1 min	5/5
Greenhouse reservation	80%	3.5 min	4.4/5
Purchasing products	80%	2.5 min	4.6/5
Confirmation and payment of order	100%	1.5 min	4.6/5
Managing personal account	80%	2 min	4.6/5

Table 6.1: Evaluation of client testing scenarios.

The results of the client testing highlighted specific challenges for users interacting with the web application for the first time. These users generally required additional time to familiarize themselves with the application and occasionally failed to complete some tasks successfully without assistance. Despite these initial hurdles, all respondents were able to perform navigation and information retrieval tasks. However, more complex operations such as reservations or purchases posed more significant difficulties. Nevertheless, the overall evaluations of the user interface were quite favourable. Drawing on the extensive feedback gathered, the following chapter will detail the improvements and optimizations implemented to enhance the user experience.

Greenhouse workers testing results

Consistent with the prior table, the following presents the evaluation results for each task performed by greenhouse workers. These results quantitatively assess the user experience, utilizing the established scoring parameters of Effectiveness, Efficiency, and Satisfaction.

Task	Effectiveness	Efficiency (Time)	Satisfaction (Rating)
Adding and editing a blog post	100%	2 min	5/5
Viewing active tasks	100%	2.5 min	4.5/5
Creating and editing greenhouse information	50%	3 min	4/5

Table 6.2: Evaluation of greenhouse workers testing scenarios.

Evaluations of testing results for greenhouse workers indicated that users familiar with greenhouse operations could easily complete the assigned tasks. The only difficulty encountered was by one respondent who was confused about how to modify the cell types in the form used for creating or editing greenhouse information. This specific issue has been addressed in the following optimization section, where solutions to enhance the user interface have been implemented.

6.3.4 Optimization

Due to the limited number of individuals involved in the testing process, it is impossible to make absolute conclusions about the application's quality. However, the feedback received is precious and has helped significantly improve the user interface. This section will detail the enhancements implemented based on the feedback received.

1. **Notification that the action was successful:** Some users reported confusion after interacting with forms in modal windows within the application, particularly noting a lack of clear feedback after actions like adding items to their cart. A „toast" notification has been introduced to resolve this issue and enhance user clarity. Now, when a user completes an action, a toast notification briefly appears to confirm the success of their action.

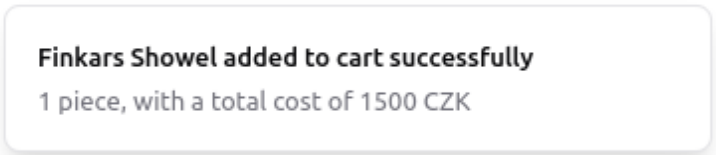


Figure 6.5: A "Toast" notification appears at the bottom of the screen, confirming that the product has been successfully added to the cart.

2. **Clickable reservation elements:** The reservations module has been improved to provide users with a better experience when interacting with clickable elements such as calendar dates and greenhouse cells. To clarify that these objects are interactive, the button styles have been modified so that the cursor appropriately reflects the action possible. For instance, when hovering over occupied cells or unavailable dates, the cursor changes to a not-allowed symbol, indicating these are not selectable. Conversely, elements that can be selected now trigger a pointer cursor, signalling to users that these items are interactive.

Figure 6.6: The cursor changes according to whether it is hovering over a clickable object or not.

3. **Bug in the „Add to cart“ modal window:** During testing, users identified a bug in the application where they could select any quantity of an item, even if only one was available in the greenhouse. For instance, a user could enter a quantity of five and subsequently encounter an error at checkout. The modal window has been modified to restrict the quantity selection to the available stock to address this issue. Additionally, the current availability is now clearly displayed in the modal window, ensuring users know the quantity they can order successfully.

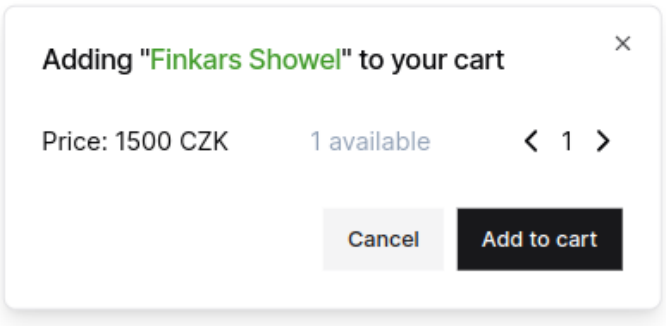


Figure 6.7: A modified modal window displaying the available quantity of the product.

4. **Greenhouse form:** One respondent experienced difficulty changing the cell types during user testing while creating or editing the greenhouse. A label has now been added to the form to resolve this issue and improve clarity, explicitly instructing users on how to modify the cell type.

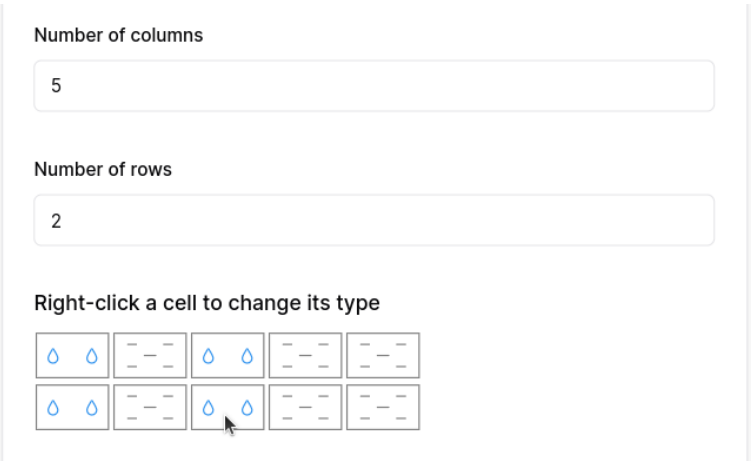


Figure 6.8: The image shows part of the form where the dimensions of the greenhouse are entered, and the types of specific cells are selected.

Chapter 7

Conclusion

The main objective of this bachelor's thesis was to develop a web application for efficiently managing a Community Greenhouse. The web application should help automate processes and improve the user experience for Community Greenhouse clients and workers.

This goal was approached through research on modern web technologies, which guided the application's development. Since the application was being developed for an existing Community Greenhouse with active participants, user requirements were analyzed, and critical aspects of the future application were identified. Following the requirement analysis, the next phase involved the design process, which involved the creation of various user interface elements essential for subsequent development stages. Throughout this phase, multiple iterative tests were conducted to refine and enhance the design.

The application was implemented in Typescript using the Express.js framework for the back-end and Next.js for the front-end. Key functionalities were successfully implemented, including creating reservations, purchasing products directly through the app, managing checklists for greenhouse workers, handling various content management tasks, and streamlining the greenhouse's day-to-day management.

As a result, the application has significantly improved interaction between users and the Community Greenhouse, providing a user-friendly platform for management tasks. The system's design and functionality have effectively met users' needs, as evidenced by positive feedback gained through user experience testing.

The application serves as a solid basis for further expansion of functionality. For instance, integrating an existing smart greenhouse sensor monitoring application could provide real-time data to optimize greenhouse management further. Additionally, developing advanced administrative dashboards would offer deeper insights and control. Finally, incorporating a payment system would streamline transactions, making the platform more convenient for users.

Bibliography

[1] Armel, J. *Web application development with Laravel PHP Framework version 4*. Helsinki, FI, 2014. Bachelor's thesis. Helsinki Metropolia University of Applied Sciences. Available at: https://www.theseus.fi/bitstream/handle/10024/74052/Author.pdf.

[2] Ballamudi, V., Lal, K., Desamsetti, H. and Dekkati, S. Getting Started Modern Web Development with Next.js: An Indispensable React Framework. march 2021, vol. 1, p. 1–11.

[3] Biehl, M. *API Architecture*. CreateSpace Independent Publishing Platform, 2015. API-University Series. ISBN 9781508676645. Available at: https://books.google.cz/books?id=6D64DwAAQBAJ.

[4] Branas, R. *AngularJS Essentials: Design and Construct Reusable, Maintainable, and Modul Web Applications with AngularJS*. Packt Publishing, 2014. Community experience distilled. ISBN 9781783980086. Available at: https://books.google.cz/books?id=koTWngEACAAJ.

[5] Chen, S., Thaduri, U. R. and Ballamudi, V. Front-End Development in React: An Overview. *Engineering International*. december 2019, vol. 7, p. 117–126.

[6] Davidse, J. *API fundamentals* [online]. IBM, december 2020. Available at: https://developer.ibm.com/articles/api-fundamentals/.

[7] Forcier, J., Bissex, P. and Chun, W. *Python Web Development with Django*. Pearson Education, 2008. Developer's Library. ISBN 9780132701815. Available at: https://books.google.cz/books?id=M2D5nnYlmZoC.

[8] Franklin, J., Wanyoike, M., Bouchefra, A., Silas, K., Campbell, C. et al. *Working with Vue.js*. SitePoint, 2019. ISBN 9781492071440. Available at: https://books.google.cz/books?id=u8zDEAAAQBAJ.

[9] Hanson, M. D. The client/server architecture. In: *Server Management*. Auerbach Publications, 2000, p. 17–28.

[10] Hat, R. *What is GraphQL?* [online]. Red Hat, Inc, january 2019. Available at: https://www.redhat.com/en/topics/api/what-is-graphql.

[11] Krug, S. *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability*. Pearson Education, 2013. Voices That Matter. ISBN 9780133597264. Available at: https://books.google.cz/books?id=QlduAgAAQBAJ.

[12] Kumar, D. *Tailwind CSS* [online]. December 2023. Available at: https://www.geeksforgeeks.org/tailwind-css/.

[13] MDN contributors. *CSS: Cascading Style Sheets* [online]. MDN Web Docs, september 2020. Available at: https://developer.mozilla.org/en-US/docs/Web/CSS.

[14] MDN contributors. *HTML: HyperText Markup Language* [online]. MDN Web Docs, september 2020. Available at: https://developer.mozilla.org/en-US/docs/Web/HTML.

[15] MDN contributors. *JavaScript* [online]. MDN Web Docs, september 2020. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[16] Pagán, J., Sánchez Cuadrado, J. and Molina, J. A repository for scalable model management. *Software & Systems Modeling*. february 2013, vol. 14, p. 2–121.

[17] Patel, V. Analyzing the Impact of Next.JS on Site Performance and SEO. *International Journal of Computer Applications Technology and Research*. october 2023, vol. 12, p. 24–27.

[18] Peters, C. *Building Rich Internet Applications with Node.js and Express.js* [online]. Carl von Ossietzky University of Oldenburg, Germany, february 2017. Available at: https://uol.de/f/2/dept/informatik/ag/svs/download/reader/reader-seminar-ws2016.pdf#page=18.

[19] Solarwinds. *What is a SQL Database?* [online]. November 2021. Available at: https://www.solarwinds.com/resources/it-glossary/sql-database.

[20] Vercel Inc.. *Next.js Documentation* [online]. January 2024. Available at: https://nextjs.org/docs.

[21] Walls, C. *Spring in Action, Sixth Edition*. Manning, 2022. ISBN 9781617297571. Available at: https://books.google.cz/books?id=2zVbEAAAQBAJ.

[22] Watts, L. *Mastering Sass*. Packt Publishing, 2016. ISBN 9781785889578. Available at: https://books.google.cz/books?id=pUrWDQAAQBAJ.