

Denis Kuivalainen

Web Application Development

Bachelor's thesis
Information Technology

2021



South-Eastern Finland
University of Applied Sciences

Author (authors)	Degree title	Time
Denis Kuivalainen	Bachelor of Engineering	May 2021
Thesis title		65 pages
Web Application Development		
Commissioned by		
-		
Supervisor		
Timo Mynttinen		
<p>Abstract</p> <p>The goal of the thesis was to make research to obtain knowledge required for the development of a web application. This includes the study of JavaScript, React, Next.js, PostgreSQL and other frameworks and technologies. Another goal was to apply the knowledge gained during the research for creating a full stack web application. The thesis includes a description of the research carried out and a description of the practical solutions taken during the development of the application.</p> <p>The theoretical part describing the research carried out includes a description of the server- and client-side technologies for development on JavaScript. This includes not only a description of the language and its specifications in general, but also a description of the technologies needed to create a web application. A description of the methods of storing data and publishing the application was also given. The practical part of the thesis describes how these technologies have been applied to create a web application. Particular attention in this part was paid to technical problems that arose during the development and their solution.</p> <p>The result of the thesis was a fully functional application for compiling and administering a grocery list. The application was created using technologies studied during the research and described in the thesis.</p>		
Keywords		
javascript, react, node.js, next.js, web application		

CONTENTS

1	INTRODUCTION	5
2	THEORY PART	6
2.1	Introduction to JavaScript programming	7
2.1.1	ECMAScript	8
2.1.2	Programming paradigms.....	12
2.2	Node.js development	14
2.2.1	Next.js.....	16
2.2.2	Ramda	17
2.3	Frontend	18
2.3.1	React	20
2.3.2	Material UI	24
2.3.3	Progressive Web Application	24
2.4	Database	25
2.5	Deployment	27
3	IMPLEMENTATION PART	28
3.1	Application architecture	29
3.2	Server-side development.....	31
3.2.1	Application directory structure.....	32
3.2.2	API planning	33
3.2.3	Database implementation	35
3.2.4	API implementation.....	38
3.3	Client-side development	43
3.3.1	User pages implementation	45
3.3.2	PWA implementation	52
3.4	Deployment on server.....	54

3.5	Demonstration	57
4	CONCLUSIONS	60
	REFERENCES	62

1 INTRODUCTION

Software development is aimed at creating functional solutions that perform a particular task for a variety of users and are able to withstand the load throughout the entire life cycle and success with their task avoiding unexpected errors. Examples of such solutions are mobile applications, desktop applications, web applications, and others. While the use of mobile and desktop applications is limited by devices, their technical requirements, platform, operating system, or other factors, web applications are available on any device or platform. This is due to the fact that the functionality of interaction with the client of such applications occurs in a browser. Thinking of the current trends in web development, it is obvious that capabilities of such applications are comparable to applications developed for a particular platform.

With the help of the internet, computers have evolved from individual workstations to a fully integrated environment of many computers and servers. Deploying applications in this environment improves performance, availability, and traceability, regardless of the device you are using. They make it possible both to promote the business and to increase the simplicity and convenience of using the services by clients. Web applications are able to solve many different tasks, from increasing sales, to simply helping people in everyday situations. The only question is whether it was created and how effectively it was implemented.

The app I decided to create is a grocery list to be used by all members of the group or "family". The application should be convenient and secure, and promptly update the information of all users. The main function of the app is to display a list of necessary groceries that the "family" interacts with. The composition of the "family", as well as the powers of each of its members, are configured in the application interface, which leads to the separation of those who can add and who can only remove items (thinking abstractly, I will assume that this function can be used to teach the children of a real family using this application responsibility for the chores). In addition, there must be an ability to aggregate an item to avoid its completion by several users. Based on the fact that this is the

only function of the application, it is unique among similar applications due to its narrow functionality.

My theoretical part will include the results of research on software development conducted for the implementation of this project. The technologies and libraries that will be used in the implementation will also be described, the theory associated with them and the reason for their choice for development will be described. The technologies and libraries that will be used, the theory associated with them and the reason for their choice for development also will be described.

In order to accomplish my goals I use the following methods:

- Research, gathering skills, and implementation for the server-side code of the application on the Next platform with connection to the configured database.
- Research and implementation of a web application written in React, including the creation of the user interface.
- Exploring possible authorization mechanisms and integrating them into the application.
- Deploy on the cloud platform.

The last section will demonstrate the functionality of the application and an evaluation of the work done.

The application was created with the help of Maria Solovieva (selection of colors for palettes), Aleksandr Shchilkin (logo design), Alina Grigoryan (translation of texts from English into Russian) and Justus Juutilainen (translation of texts from English into Finnish). The research, development of the application structure, all the coding and cloud publishing (everything described within the thesis) was done by the author.

2 THEORY PART

The theoretical research carried out within this work covers all the necessary theoretical background for the implementation of the project and the achievement of its goals. The disclosure of the topic begins with a description of the JavaScript

as a programming language, its standards, technical aspects and features. Then, there is a detailed description of the backend development in this language. This part will describe how the server side of the application works as a whole, what processes are running and how it is maintained. After that, frontend programming is described using as an example React development. The features of the structure, life cycle and other fundamental features of the programming language within this library will be revealed. Finally, the database will be explained.

2.1 Introduction to JavaScript programming

JavaScript is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions (Mozilla 2021d). It is most widely used as a language for scripting web pages, expanding capabilities and user experience. JavaScript is used as client-side programming language by 97.1% of all the websites (W3Techs 2021). Additionally, it is also used for server-side code, desktop applications, AI, and other applications.

JavaScript starts its origins in 1995. Netscape Communications had been promoting its Netscape Navigator browser. Being a strong competitor in the market, they started looking for ways to further develop of web technologies. This led to the birth of JavaScript.

The problem was to create a lightweight scripting language capable of handling the Document Object Model, which was not standardized in those days. The new language was not supposed to target professional programmers, as Java was used for this. Thus, the new scripting language had to target a completely different audience – scripters and designers, or even enthusiasts. All this was supposed to become part of the browser and make the web pages look more dynamic, as well as content creation easy for people not familiar with programming.

Based on these requirements, the Mocha language was created, later renamed LiveScript, influenced by the company's marketing department, and then JavaScript, again, for marketing purposes. At its core, it was unlike any other

language used at the time. It should be like Java, but more dynamic. Its semantics common for huge number of developers used paradigms were inherited. While Java is used by programmers to create new objects and applets, JavaScript is designed for use by HTML page authors and enterprise application developers to dynamically script the behavior of objects running on either the client or the server (Netscape 1999).

JavaScript as a product has been introduced as a scripting language for performing small client tasks in the browser. The first version of JavaScript had all the fundamental features that are still in use. Particularly, its object model and functional features were present in the first release.

2.1.1 ECMAScript

ECMAScript is a standard for scripting programming languages. ECMAScript is based on several originating technologies, the most well-known being JavaScript (Netscape) and JScript (Microsoft) (Ecma International 2020). It defines the basic functionality that a scripting language should provide and how those functionalities should be implemented.

To standardize the JavaScript after it was developed, Netscape approached the European Computer Manufacturers Association (ECMA), which standardizes information technology. That ECMA Standard was submitted to ISO/IEC JTC 1 for adoption under the fast-track procedure and approved as international standard ISO/IEC 16262, in April 1998 (Ecma International 2020). This led to the creation of a new scripting language standard known as ECMAScript, or ES for short. Dialects of this language are JavaScript, ActionScript, SpiderMonkey, V8 and others.

At the moment, there are 11 specifications. In this section only features that had effect on modern JavaScript were described starting with ES2015, since the previous versions are outdated at the moment. Since this release, the decision to update the standard every year was made. Below is a description of the features of the new versions of the standard, which were applied in the development of

the project. All of them have an analogue in a simpler form, but this will make the code larger and more confusing.

The ES2015 edition of the standard introduced changes to JavaScript that made it more flexible and functional. The sixth edition provides the foundation for regular, incremental language and library enhancements (Ecma International 2020). In addition to the new types of variables visible only inside the block, the innovations were destructuring assignments and arrow functions.

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables (Mozilla 2021c). Thus, the feature allows you to read data from the structure with one operator, as shown in Figure 1. This allows you to shorten the code by removing the cumbersome variable assignment constructs.

```

1
2  var a, b, arr;
3
4  // Before
5  arr = [1, 2];
6  a = arr[0];
7  b = arr[1];
8
9
10 // With ES2015
11 [a, b] = [1, 2];
12

```

Figure1. Destructuring assignments example

Arrow functions simplify syntax compared to functional expressions, making the code shorter. As you can see in Figure 2, they cannot be given a name, that is, they are anonymous, but this does not prevent them from being assigned to a variable. Additionally, arrow functions cannot be used as methods. The call, apply and bind methods are not suitable for arrow functions – as they were designed to allow methods to execute within different scopes – because Arrow functions establish "this" based on the scope the Arrow function is defined within (Mozilla 2021a).

```

2  // Before
3  function exampleFunction(...args) {
4  |    // Do something
5  |  }
6
7  // With ES2015
8  (...args) => {
9  |    // Do something
10 |  }
11

```

Figure 2. Arrow function example

ES2015 standard added *Promises*, which are wrappers for callback functions. Their main purpose were the ordering of delayed and asynchronous code fragments. Promise is an object that takes a function with two arguments, *resolve* and *reject*, which is executed immediately, even before the constructor returns the resulting object. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a *promise* to supply the value at some point in the future (Mozilla 2021f).

When a *Promise* is created, it is pending, and then can be fulfilled, returning the result, or rejected, returning the reason for the rejection. There are two methods to handle the return value: *then*, which takes a callback function that takes the result as an argument and returns a new promise or other value, and *catch* is a callback that is called when an exception or other error occurs. The application of these methods is shown in Figure 3.

```

1
2  let promise = new Promise((resolve, reject) => {
3  |    // Async code here
4  |
5  |    resolve("Success!");
6  |    reject("Fail :c");
7  |  });
8
9  // Execute promise
10 promise
11   .then(res => console.log(res))
12   .catch(error => console.warn("Something gone wrong: %s", error));
13

```

Figure 3. Promise example

It became even easier to use promises with the release of the ES2017 standard, with the addition of the *async/await* construct. This makes it possible to "manually" stop the execution of the code using the keyword *await*, waiting for the called fragment to return a value. Await expressions make promise-returning functions behave as though they are synchronous by suspending execution until the returned promise is fulfilled or rejected (Mozilla 2021b).

As stated earlier, *await* will return a *Promise*, which can either succeed or throw an error. For this, a *try...catch* block is used, which has been present since the very first ES1 standard. Thus, the example proposed earlier in Figure 3 can be rewritten into a simpler version, as shown in Figure 4. This form allows the code to be more flexible and functional.

```

8
9  // ES2015
10 promise
11   .then(res => console.log(res))
12   .catch(error => console.warn("Something gone wrong: %s", error));
13
14 // ES2017
15 async () => {
16   try{
17     let res = await promise;
18     console.log(res);
19   } catch(error) {
20     console.warn("Something gone wrong: %s", error);
21   }
22 }
23

```

Figure 4. Async...await construction example

The last described construction is spread syntax, which allows you to simplify iteration and expand the elements available for this. It was introduced in ES2015 and updated in ES2018. This construct is applicable to functions, array literals, and object expressions. Spread syntax can be used when all elements from an object or array need to be included in a list of some kind (Mozilla 2021g). The example shown in Figure 5 describes most of the use cases for the use of spread syntax.

```

1
2  // Array
3  let arr1 = [1, 2, 3];
4  let arr2 = [0, ...arr1]; // [0, 1, 2, 3]
5
6  // Object
7  let obj1 = {key1: "val1", key2: "val2"};
8  let obj2 = {...obj1, key3: "val3"}; // {key1: "val1", key2: "val2", key3: "val3"}
9
10 // Function
11 const fn = (...args) => {
12   // Something
13 }
14
15 fn(1, 2, 3); // args = [1, 2, 3]
16

```

Figure 5. Spread syntax example

To conclude this section, we can say that the new edits that complement the ECMAScript standard make it easier to code. Constructs become simpler and more functional, making JavaScript programming more declarative than imperative. It forces to get deeper into the language, making the code harder to understand, but easier to write. Despite the changes, there is opportunity to write in simple way, as well as language original essence exists unchanged. It should become a scripting language that anyone can use, even without deep programming knowledge.

2.1.2 Programming paradigms

A programming paradigm is a set of ideas and concepts that define the style of coding and the approach to programming in general. By adhering to their program, it turns out structured and ordered, which makes it readable and understandable both for other programmers and for the computer on which it will be executed.

There are two main programming paradigms - imperative and declarative. Analysis and explanation of each of them will be done using a simple example - calculating the length of a circle by diameter. This simple task will help you visualize the difference in paradigms.

The imperative paradigm stems from the origins of programming, when a developer wrote code that was not compiled for a machine, describing every action that a machine must perform. Although programming has become easier nowadays, the essence of the paradigm remains the same - it provides clear instructions for the computer what to do. Figure 6 shows the implementation of the problem using this approach.

```
1
2  const perimeter = (radius) => {
3    |    return radius * 2 * Math.PI;
4    |  }
5
6  perimeter(2); // 12.566370614359172
7
```

Figure 6. Imperative programming example

One of the most common subtypes of imperative programming is object-oriented programming (OOP). It is based on sequential calls of commands that change the data with which the program operates. Thus, it operates on objects, and this is convenient for many applications. From the example in Figure 7, it is obvious that the implementation of the problem in the OOP style is generally the same as the previous example, but with some syntax differences. As a result, the function turned into a class constructor, the local variables into instance variables, and nested functions — in methods (Martin 2018).

```
1
2  class Circle {
3    |    constructor(radius) {
4    |    |    this.perimeter = 2 * radius * Math.PI;
5    |    |    this.square = Math.PI * radius * radius;
6    |    |  }
7    |  }
8
9  new Circle(2).perimeter; // 12.566370614359172
10
```

Figure 7. OOP example.

In a declarative paradigm, the developer only describes the problem and the expected result. Instructions, variables or states are not applicable here. This is clearly seen in functional programming, which states that a program is created as a tool that solves a specific problem and returns the desired result. We can treat

functions like any other data type and there is nothing particularly special about them - they may be stored in arrays, passed around as function parameters, assigned to variables, and what have you (Lonsdorf 2018).

The advantage of functional programming is ease of development and clean code. As it can be seen in Figure 8, how easy it is to read and understand what the program is doing. This is based on the principle of pure function, ignoring everything outside itself. A pure function is a function that, given the same input, will always return the same output and does not have any observable side effect (Lonsdorf 2018).

```
1
2  let getPerimeter = compose(multiply(Math.PI), multiply(2));
3
4  getPerimeter(2); // 12.566370614359172
5
```

Figure 8. Functional programming example

The problem is that the computer does not initially know how to fulfill the user's requirements. That is, in order to write functional code, you first need to write the functions on which it will rely. In addition, the possibilities of functional programming are limited by the capabilities of functions, which again leads to the use of imperative programming to write new functions.

All described concepts will be used for the development of this project. Since it is not a specific task that is being implemented, but an application is created from scratch, the base will be written imperatively, and the structure will be built declaratively. This will simplify the development of modules and make the application skeleton extensible.

2.2 Node.js development

JavaScript was designed as a language for browsers. But its simplicity and ease for computers attracted something more out of it. In 1996, the creator of the Netscape language tried to turn it into a server-side language. Server-side JavaScript (SSJS) extends the core language by supplying objects relevant to

running JavaScript on a server (Netscape Communications Corporation 1998). But the technology was not successfully adopted by the developers and further development was abandoned.

The SSJS model was overhauled in 2009 when the thread-based concurrency model was abandoned in favor of event-driven systems. An American engineer Ryan Dahl took this important step for programming in JavaScript creating Node.js. The created platform was warmly accepted by programmers and, over time, became very widespread. According to Stack Overflow Developer Survey (2020b), for the second year in a row, Node.js takes the top spot, as it is used by half of the respondents.

As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications (OpenJS Foundation 2020). Due to the chosen ideology, the platform can execute real-time applications, handling a large number of requests, fully avoiding any deadlocks. This allows systems developed in Node.js to be lightweight and efficient in development and in use.

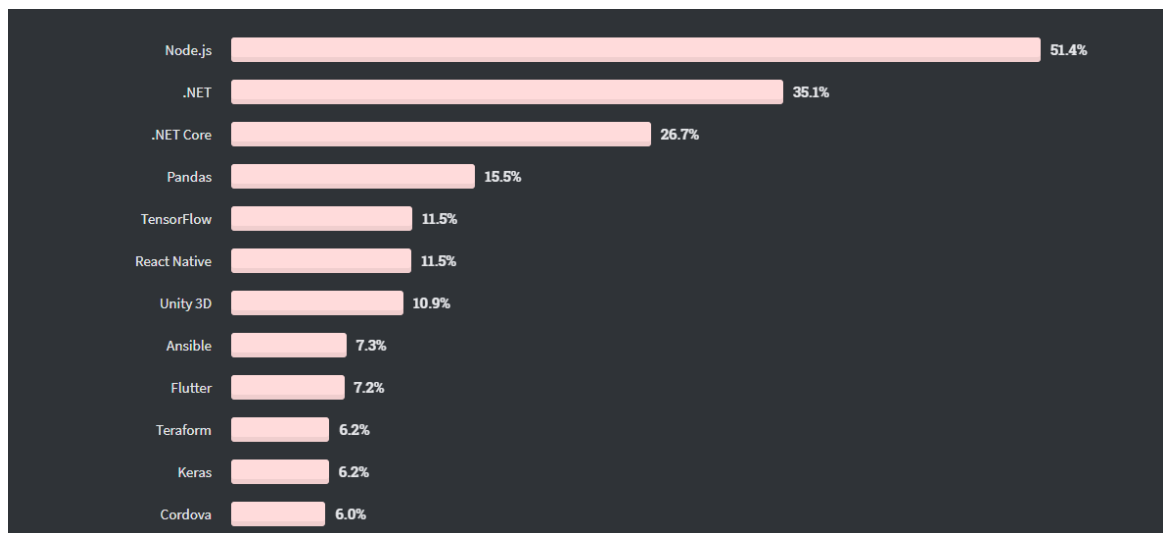


Figure 9. Most popular web technologies (Stack Overflow 2020c)

Node.js allows to develop and use libraries and frameworks using Node Packet Manager (npm). The npm Registry is a public collection of packages of open-source code for Node.js, front-end web apps, mobile apps, robots, routers, and

countless other needs of the JavaScript community (npm, Inc. 2021). This leads to the following statement - the use of this platform leads to the development efficiency. This is due to the use of one language for the entire application and the possibility of code reuse.

2.2.1 Next.js

For the development of the project, the Next.js platform was chosen. Since the main requirements for the application are fast processing of API requests and support for the front-end React library, this platform is the best solution. Next.js gives the best developer experience with all the features you needed for production: hybrid static & server rendering, TypeScript support, smart bundling, route pre-fetching, and more (Vercel, Inc. 2021e).

A feature of the platform is server-side rendering. This helps reduce the load on the device by performing most of the operations on the server. By default, Next.js pre-renders every page (Vercel, Inc. 2021d). This concept breaks the main React principle - deploying the entire application on a single page (this will be described in Section 2.3.1 React). However, this concept helps to improve Search Engine Optimization, as the main content of the application will only be loaded when the user visits the page they want.

Despite the fact that the platform is aimed at simplifying the configuration and easier use, the ability to customize your own API paths is available. They are server-side only bundles and won't increase the client-side bundle size (Vercel, Inc. 2021a). Also, this solution provides the ability to work with request and response objects, as shown in Figure 10.

```
const handler = async (req, res) => {  
  const cookies = new Cookies(req, res);  
  const session = cookies.get("session");  
  
  message(res, await switchMethod(req.method)(req, session));  
  res.end();  
}  
  
export default handler;
```

Figure 9. API function example

Next.js has flexibility for development - it allows you to create an application by fully customizing it to your needs. The platform allows you to change the server, the React application, set your own routing, change the rules for working with files and API. It allows you to feel free to develop in accordance with application objectives.

2.2.2 Ramda

Ramda is used to implement the functional paradigm in project development. This is a library designed specifically for a functional programming style, one that makes it easy to create functional pipelines, one that never mutates user data (Ramda 2021). This is perhaps one of the most critical requirements in the ongoing work with data.

Written in pure JavaScript, this library provides ease of integration. This allows to use it with any library or framework written in JavaScript after installing through the package manager. This maintains and improves the advantage of Node.js - increased productivity, ease of reading and writing, and reusability of code.

Another important feature of Ramda is the currying of all function. In other words, when not all required arguments are passed to a function, it will return a function requiring the missing ones. This allows you to easily build up new functions from old ones simply by not supplying the final parameters (Ramda 2021). The meaning of this functional programming principles is illustrated in Figure 11.

```

1  import R from "ramda";
2
3
4  const sum = x => y => z => x + y + z;
5
6  let add = sum(1)(3); // 1 + 3 + z
7
8  add(2); // 1 + 3 + 2
9
10 // SAME AS
11
12 const sumCur = R.curry(
13   (x, y, z) => x + y + z
14 );
15
16 let addCur = sumCur(5, 8); // 5 + 8 + z
17
18 addCur(3); // 5 + 8 + 3
19

```

Figure 11. Currying example

Last but not least, Ramda strives for performance (Ramda 2021). Contrary to the trend that functional programming makes code harder to deploy and compile, the developers of this library focused on performance. This makes the code written using this library more flexible and less resource-demanding in terms of hardware and operation time.

2.3 Frontend

In the previous section, server-side development was considered, thus it should be described what front-end development is, which, in fact, is the creation of the user interface. This includes working with 3 main technologies – HTML, CCS, and a programming language stands for functionality. HyperText Markup Language (HTML) is a standardized markup language for web documents. HTML has a wide array of extensibility mechanisms that can be used for adding semantics in a safe manne (van Kesteren 2021b). This makes it possible to inject the document with style and scripts. Styles and page design on the web are implemented using Cascading Style Sheets. CSS is the language for describing the presentation of Web pages, including colors, layout, and fonts (W3C 2016). Scripting languages bring functionality, logic, animation, and other interactivity that can attract, retain and make it convenient to use for the end users. This allows to distribute some of the load from the server on the computers of end

users, and also simplifies development, because there is a separation of which and where processes should take place. As mentioned earlier, JavaScript is the most popular web development language, based on this statement, only it will be described below.

Recently, developers have moved away from described development model as it is outdated nowadays. It is much more convenient to manipulate the document, rather than its individual parts, which is possible due to the Document Object Model. DOM is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document (van Kesteren 2021a). This standard uses HTML tags as objects, describing the dependencies between them, which in fact is a tree, shown on Figure 12. These tags are available for access and change with JavaScript, which allows to dynamically change the structure of the page.

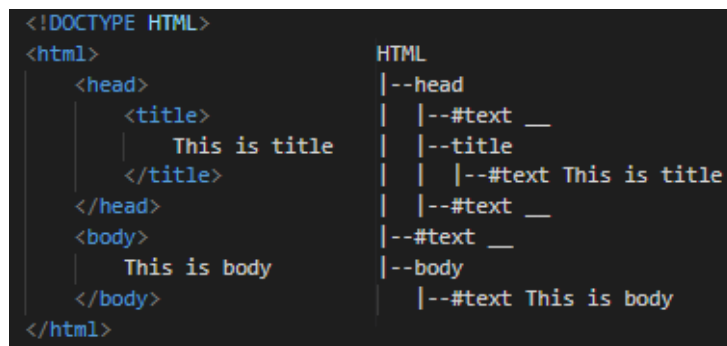


Figure 12. DOM basic structure

Nowadays, many libraries have been developed that can work with DOM. Below is a description of several popular ones:

- Angular is an application design framework and development platform for creating efficient and sophisticated single-page apps (Google 2021). It implements a strong component architecture that makes possible the component reuse. But there are problems with SEO and migration due to the resource requirements of the developed applications to facilitate component reuse. But there are problems with SEO and migration due to the resource requirements of the developed applications.
- Ember.js is a productive, battle-tested JavaScript framework for building modern web applications (Tilde inc. 2021). This library is great for building scalable single page applications. Unlike most similar libraries, this one

builds on the Model-View-ViewModel architecture, which simplifies the interaction of UI and business logic.

- React.js is a library for building web interfaces. As of beginning of 2021 it is the most widely used among similar libraries. This popularity is due to fast rendering as well as design variability. React has been designed from the start for gradual adoption, and you can use as little or as much React as you need (Facebook Inc. 2021a).

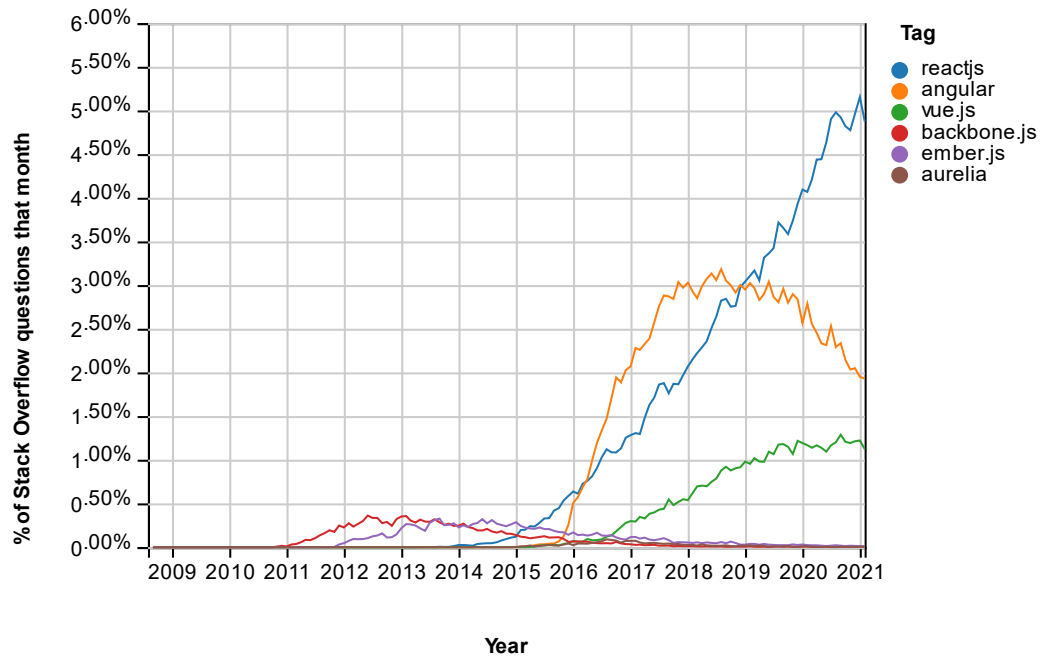


Figure 13. Most popular frontend libraries (Stack Overflow 2021)

Based on the previous description, the choice of a library to develop the user interface of the application was made in favor of React.js. It was chosen due to its popularity, large and responsive community, as well as ease of integration with the chosen development platform (Next.js).

2.3.1 React

React is a JavaScript library for building user interfaces (Facebook Inc. 2021d). This library was developed by Facebook in 2013 for the internal needs of the company. Later the company decided to change the policy regarding the project and make it open source. Backed by Facebook and other companies such as Netflix and Airbnb, React has become the most popular front-end development library by 2016.

React is a library that provides the ability to work with independent encapsulated elements for rendering HTML. Simple use of the library is shown in Figure 14. Thus, this library is designed to perform DOM manipulations using markup language as the output. It is because of this concept that the flexibility and convenience of React are achieved - all states, data and functions are outside the DOM. According to Facebook Inc. (2021d), all programming in react comes down to design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes. React philosophy is based on adhering to the concept of SPA – a single page application. This means that the entire application will be placed on one web page, and any changes that the user sees on the screen are just a change in the tree of elements of this single page. Further, a more detailed description of the functions and features of the library necessary for the implementation of the project will be provided.

```
1 | import React from 'react';  
2 |  
3 | const componentExample = () => <div>this is div</div>  
4 |
```

Figure 14. React function

As stated above, there are several programming paradigms, React makes it possible to work with all of them. But due to the predominantly use of a functional style, working with classes will not be considered within the React description. Using classes or functions to write code does not imply significant differences other than how the code will look, but the functionality and features will be the same.

The main difference between the implementation of functions and classes is the use of hooks. Hooks are functions that let “hook into” React state and lifecycle features from function components (Facebook Inc. 2021b). This allows you to use the functionality of classes in functions without losing capabilities and efficiency. Despite the ease of use, as will be shown below, hooks have a number of limitations, not observing which you can break the logic of React: use

is possible only in functions and at their top level. This means that hooks cannot be used in classes, loops, conditions, or nested functions.

To work with mutable data in react, there is a concept called state. It allows you to store data that will dynamically change during rendering. The difference between *state* and a simple variable is that *state* can affect the DOM and cause re-rendering, while a variable declared via `let`, for example, has no effect. Functional components use the `useState` hook to implement this concept, its use is shown in Figure 15.

```
1
2 | import React, { useState } from 'react';
3
4 | const stateExample = () => {
5 |   // Declare state
6 |   const [number, setNumber] = useState(0);
7
8 |   // Log number
9 |   console.log(number); // 0
10
11 |   // Update state
12 |   setNumber(3);
13
14 |   // Log updated number
15 |   console.log(number); // 3
```

Figure 15. `useState` hook example

It is worth clarifying how the rendering process works, and in general how the entire life cycle of a component proceeds, since this will cost the work of the client part of the application being developed (for example, updating data). Conventionally, the entire life cycle can be divided into 3 stages: mounting, updating and unmounting. Mounting involves reading initial data from the component's constructor (if any), getting data from parent components, rendering the DOM for the first time, and completing this first rendering. The update stage can be caused by a change in state inside this component, or by data coming from the parents. This will trigger a re-render which will update the DOM. Unmounting includes the process of removing a component from the DOM. The clearly described life cycle is depicted in Figure 16.

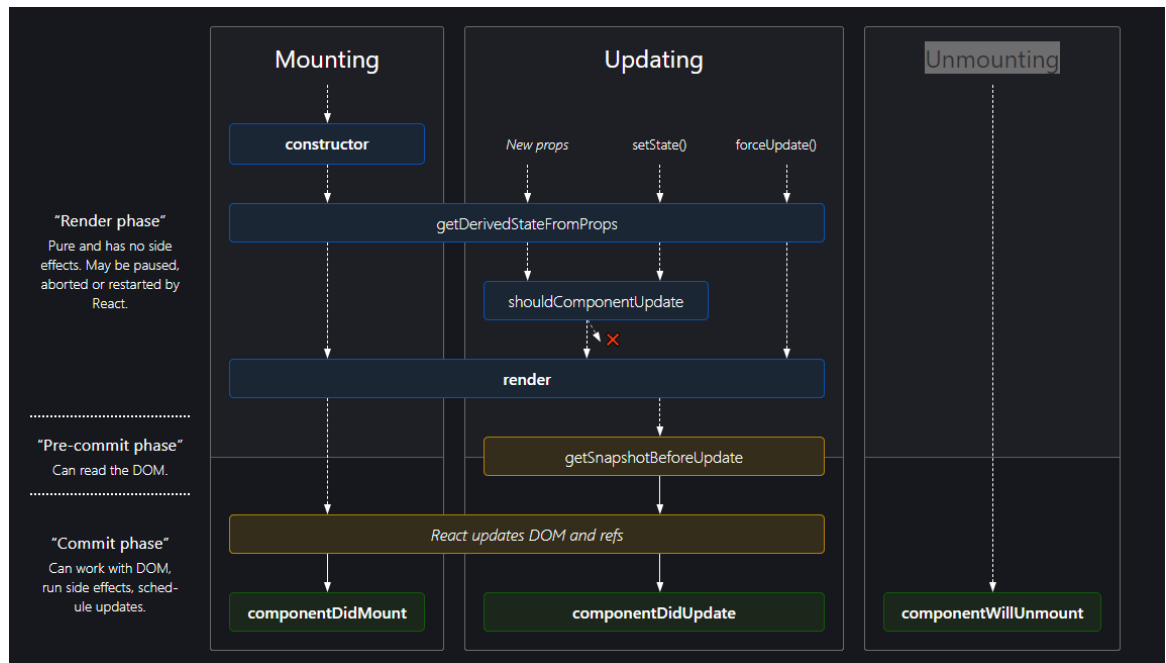


Figure 16. React lifecycle methods (Facebook Inc. 2021c)

During any stage of the life cycle, it is possible to perform any operations. Figure 16 shows the methods for interacting with the rendering process in the classes. In functions this is simplified to two main effect hooks - *useEffect* and *useLayoutEffect*. Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects (Facebook Inc. 2021e). Thus, if a change in state or DOM happened, the callback nested in these methods will be executed. A simple example of using hook instead *componentDidMount* method is shown in Figure 17.

```

1 | import React, { useState, useEffect } from 'react';
2 |
3 | const effectExample = () => {
4 |   const [rendered, setRendered] = useState(false);
5 |
6 |   // Will be only called after first rendering
7 |   useEffect(() => {
8 |     setRendered(true);
9 |   }, []);

```

Figure 17. useEffect hook example

React provides convenient and flexible ways of working with data and rendering, allowing you to create an intuitive and responsive interface using a minimum amount of computer power and making it easy to write code on it.

2.3.2 Material UI

One of the components of the client-side of the application is the user interface (UI). It is important to provide users with access to all the tools in the application by creating interactive elements. But even a functional interface can cause rejection due to overloading of elements or poor design. Paying attention to user experience (UX) solves this problem. The Material-UI library was chosen to create a friendly UI/UX. According to Material-UI documentation, this library provides React components for faster and easier web development (Material-UI 2021). An example of the library component is shown in picture 18.

```
<Button
  style={{marginTop: 10}}
  onClick={() => router.push(url)}
  startIcon={<ArrowBackIosRounded />}
>
  {language(38)}
</Button>
```

Figure 18. Material-UI Button component

The Material-UI philosophy is about providing developers with a convenient and simple toolkit for creating user interfaces using React. The main focus of the developers is on a pleasant design and a number of functional elements. In addition to using the standard set of elements, the library provides the ability to customize any elements using the methods included in the library.

Since the development of the design work will be done by one person and the volume of work is large, the design development will take a lot of time and resources. Using Material-UI will speed up the development of the user interface, make it simpler and more functional.

2.3.3 Progressive Web Application

The application being developed must be fast and multi-platform to improve usability and accessibility. The main directions for using are mobile devices and

browsers. To avoid writing additional software, this project will be a Progressive Web Applications (PWA).

Progressive Web Apps are web apps that use emerging web browser APIs and features along with traditional progressive enhancement strategy to bring a native app-like user experience to cross-platform web applications (Mozilla 2021e). In other words, PWA does not differ in any way from a regular site externally, but with higher performance and loading speed. Despite the front-end similarity of such applications with the native one, generally, it is only a cosmetic solution, which is for ease of use. In terms of technology, these types of applications are made identical - as the operating system launches native applications, so does the browser provide the PWA.

The heart of PWA is Service Worker - an additional layer between the front-end and the back-end, through which all requests pass. A Service Worker is a script that allows intercepting and control of how a web browser handles its network requests and asset caching (Thomas 2020). This additional layer turns the web browser into a virtual machine with its own file system and database. Its code defines how the browser should handle requests coming from the client and responses from the server.

Such applications should be reliable, fast, and engaging. The quality of the connection does not affect the loading of the application - it is shown immediately, since the application data is cached on the first load. Therefore, only a small part of the data needs to be downloaded from the server, which makes data exchange fast over the network. This makes it possible to devote more resources to a responsive user interface, which makes the user experience of using the application more comfortable and enjoyable.

2.4 Database

Most applications work with large amounts of data to achieve their goals. During the processes performed in the application, various types of variables are used: numbers, strings, dates, and others. For convenient work with them, databases

are used. Database (DB) is an organized structure intended for storing, changing and processing interrelated information, mainly large volumes.

Generally, a database is just a repository of the necessary information, without defining the structure and principles of working with data. For this, database management systems (DBMS) are created. DBMS is software complex for creating and managing databases. The main functions of the DBMS are support for scripting languages, data management of disk spaces, data management in memory using a cache, as well as administration methods such as logging changes, backing up and restoring the database after failures. Based on the interest of users according to the Stack Overflow survey shown in Figure 19, the most used in developers community DBMS are PostgreSQL, MySQL and Microsoft SQL Server.

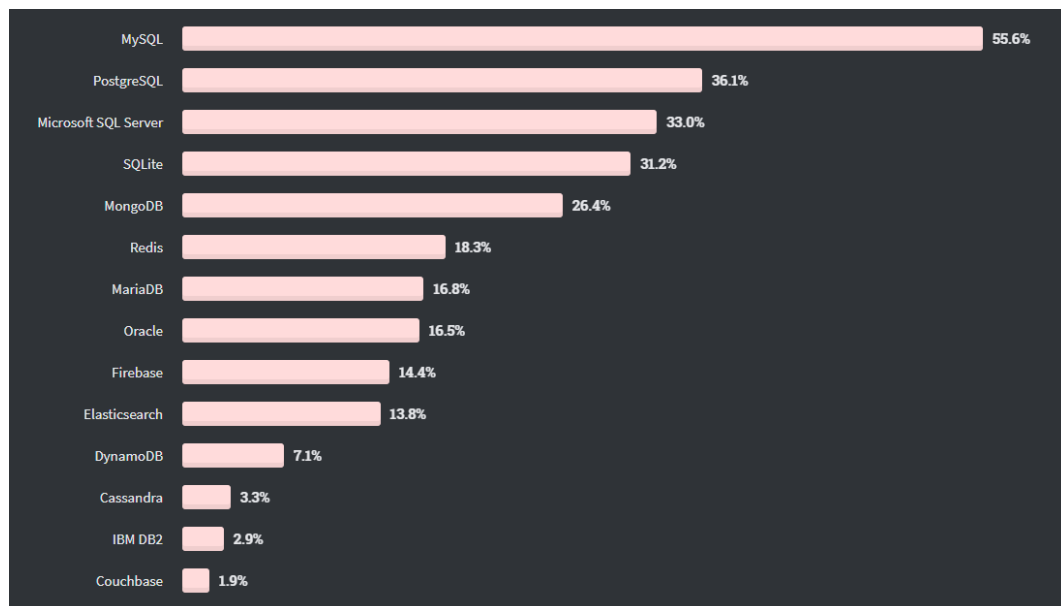


Figure 19. Most popular databases (Stack Overflow 2020a)

PostgreSQL was chosen as the DBMS for the developed application. It is an open-source object-relational database. PostgreSQL comes with many features aimed to help developers build applications, administrators to protect data integrity and build fault-tolerant environments, and help you manage your data no matter how big or small the dataset (The PostgreSQL Global Development Group

2021c). This choice was made due to the availability of hosting (more details in section 2.5) and extensive functionality.

The main advantage is the data model. PostgreSQL has a rich set of native data types available to users (The PostgreSQL Global Development Group 2021a). In addition, this DBMS supports custom objects, including custom data types, functions and indexes, which makes administrating the database flexible and reliable. Also, this DBMS supports multidimensional arrays, which can simplify data storage and structure. It is also worth mentioning the limitations of PostgreSQL regarding the size of the implemented databases. In this DBMS, the maximum database size is unlimited, that is, the available disk space on the server serves as a limiter. The number of possible entries in the table is also unlimited. These characteristics play an important role in the choice of a DBMS, since the work will be carried out with a large amount of data, which may entail constant expansion of the database.

The PostgreSQL toolkit also provides the PL/pgSQL procedural language. PL/pgSQL is a loadable procedural language for the PostgreSQL database system (The PostgreSQL Global Development Group 2021b). This will reduce the number of database queries by moving some of the backend logic to the database. Simple operations such as conditional SELECT queries or loops can be performed on the DBMS side, thereby reducing the number of API requests between the application server and the data server.

The database chosen for use in the implementation of the developed project is PostgreSQL. This convenient and flexible solution will maximize the efficiency of access and work with stored data, and the popularity and community support will help in solving problems arising during the development and in future maintenance.

2.5 Deployment

For the finished application to be used by clients, you must provide them with access to it. In web development, this is done by placing an application on an

application server that is publicly accessible over the internet. But using a private server is expensive and resource-intensive in terms of hosting a single application on it. Therefore, it was decided to use Heroku for these purposes.

Heroku is a container-based cloud Platform as a Service (PaaS) (Salesforce 2021). This means that users are provided with space on the cloud server to host their applications. The server will automatically balance the load on its ports, where the applications of different users are located. By setting asleep or waking up ports, the server reduces the load placed on it, increases the processing time of requests due to the fact that only the working ports are awake, and thereby reduces the cost of server maintenance.

In addition to all of the above, the platform provides access to a large number of addons that provide various functions. The Heroku experience provides services, tools, workflows, and polyglot support — all designed to enhance developer productivity (Salesforce 2021). Within the framework of this project, an addon will be used that provides access to the PostgreSQL database. Thus, hosting and applications and databases will happen on the same platform.

3 IMPLEMENTATION PART

This section will describe the stages of application development. The concepts and technologies described in the theoretical part will be applied in practice. Despite the presence of research in this subject area and detailed documentation on all technologies used, development is not limited to coding. There are many architectural, logical, and technical problems that need to be solved in order to create an application. This is a process of analyzing, evaluating and solving problems that will be described in the implementation part.

Application development will begin by describing the architecture of the application being developed. After that, the installation and creation of the database and application server will be described. Next, we will describe how the application code was created that connected the database, the server and the

user's browser. In conclusion, the installation of the application on a cloud server will be described and its functionality will be shown.

3.1 Application architecture

The main task of this application is the ability to create and manage a shopping list for user groups. This means that you need not just create an application that can remember the data entered by the user, but that has the ability to provide many users with the same data, giving them the opportunity to operate with it. This requires intelligently planning the architecture of the application in order to avoid possible complexities in development and to have a clear idea of what needs to be done.

To understand what an application should be able to do, it is necessary to make a list of requirements for the final product. The following list describes all the functions and features that should be developed and their description:

- Grocery list management. Users should be able to maintain a single grocery list for a particular group. This includes the ability to add, reserve and remove items, as well as display information about who added a given item and whether it is reserved.
- User system. Users should be able to create an account for personal use. All data entered by users must be reliably protected so that no one else can get them.
- User administration. For greater isolation of each group, allow only one user to create and manage a group. This will allow for more flexible management of the group, giving and taking away opportunities for users from only one main account.
- Language support. To attract and create more comfort of use, it is necessary to provide the application interface in several languages. For the start, English, Finnish and Russian must be added, but their number must be easy to increase at any time.
- Customization. For a better user experience, you just need to be able to customize the color scheme yourself. Therefore, several color themes should be offered to users for use, with the possibility of increasing their number in the future.
- UI/UX. It should be easy and understandable for users to use the application so that they want to continue using it.

To implement an application with the declared functionality, it will be divided into 4 parts: a database, an application server, a service worker and UI. This

architecture is shown in Figure 20. The database will contain all the data used in the application, such as information about users, groups (families), open sessions, and more. In addition, some of the data validation operations will be carried out on the database, which will reduce the number of requests, thereby reducing the queue and speeding up the operations performed. The application server will be responsible for processing requests coming from the client-side and for accessing data in the DB. Since the selected development platform (Next.js) provides server-side rendering by default, this is where pre-rendering will take place and which files should be sent to the client. The service worker is necessary to convert the application to PWA, so its main task is to determine the data for caching and save it using the browser in the device's memory. Also, in its essence, this is the basis of the client-side, since all the functionality is implemented here, leaving the UI to be just a cosmetic shell of the application. The UI will be directly used by the client, so it is a set of elements for displaying and interacting with data in a simple way that anyone can understand. When the user interacts with the interface, requests will be sent to the service worker for processing and determining what actions need to be taken.

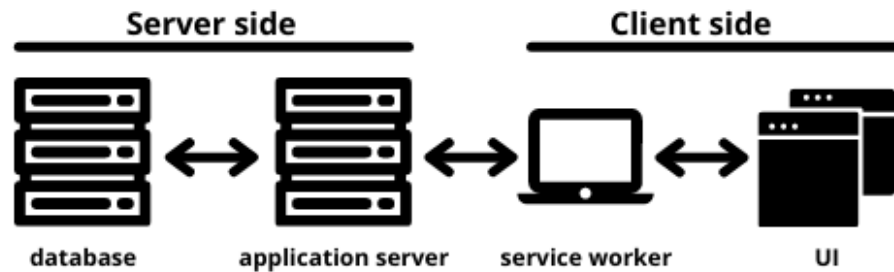


Figure 20. Application architecture

Such an architecture will allow not only to separate the processes performed by the application, making it easier to understand the work, but also will make it possible to expand the functionality in the future. When components are isolated from one another, you know that you can change one without having to worry about the rest (Thomas 2020). In other words, if the stated requirements will be followed, the application will be developed in a flexible and easy-to-understand environment.

The described architecture will allow you to achieve all the declared requirements for the developed application. In addition, such a structure will facilitate writing code and linking components into a single system, as well as avoiding hidden errors due to a clear division of system functions between its components.

3.2 Server-side development

The structure of the project is considered, thus, the development of each of its parts can be started. The central part of the project is the Next.js platform, which will allow you to implement the logic of the application and become the link between all its parts. In addition, this platform already provides a ready-made solution for setting up a progressive web application.

To start development, it is needed to install the default server template provided by the platform developers. The installation process is shown in Figure 21. The downloaded example is extended with the npm package *next-pwa* by shadowwalker. This will avoid the manual configuration of the service worker, which will speed up the development process and eliminate unnecessary tasks within the project.

```
C:\Users\Di Kuivalainen\Documents\GitHub>npx create-next-app --example progressive-web-app grocery
Creating a new Next.js app in C:\Users\Di Kuivalainen\Documents\GitHub\grocery.

Downloading files for example progressive-web-app. This might take a moment.

Installing packages. This might take a couple of minutes.

yarn install v1.22.5
info No lockfile found.
[1/4] Resolving packages...
warning next-pwa > workbox-webpack-plugin > workbox-build > @hapi/joi@15.1.1: Switch to 'npm install joi'
warning next-pwa > workbox-webpack-plugin > workbox-build > @hapi/joi > @hapi/hoek@8.5.1: This version has been deprecated and is no longer supported or maintained
warning next-pwa > workbox-webpack-plugin > workbox-build > rollup-plugin-babel@4.4.0: This package has been deprecated and is no longer maintained. Please use @rollup/plugin-babel.
warning next-pwa > workbox-webpack-plugin > workbox-build > @hapi/joi > @hapi/address@2.1.4: Moved to 'npm install @sideway/address'
warning next-pwa > workbox-webpack-plugin > workbox-build > @hapi/joi > @hapi/bourne@1.3.2: This version has been deprecated and is no longer supported or maintained
warning next-pwa > workbox-webpack-plugin > workbox-build > @hapi/joi > @hapi/topo@3.1.6: This version has been deprecated and is no longer supported or maintained
warning next-pwa > workbox-webpack-plugin > workbox-build > @hapi/joi > @hapi/topo > @hapi/hoek@8.5.1: This version has been deprecated and is no longer supported or maintained
warning next-pwa > workbox-webpack-plugin > workbox-build > strip-comments > babel-plugin-transform-object-rest-spread > babel-runtime > core-js@2.6.12: core-js@<3 is no longer maintained and not recommended for usage due to the number of issues. Please, upgrade your dependencies to the actual version of core-js@3.
[2/4] Fetching packages...
info fsevents@2.3.2: The platform "win32" is incompatible with this module.
info "fsevents@2.3.2" is an optional dependency and failed compatibility check. Excluding it from installation.
[3/4] Linking dependencies...
warning " > next-pwa@3.1.5" has unmet peer dependency "webpack@>=4.0.0".
warning "next-pwa > clean-webpack-plugin@3.0.0" has unmet peer dependency "webpack@*".
warning "next-pwa > workbox-webpack-plugin@5.1.4" has unmet peer dependency "webpack@^4.0.0".
[4/4] Building fresh packages...
success Saved lockfile.
Done in 36.41s.

Initialized a git repository.

Success! Created grocery at C:\Users\Di Kuivalainen\Documents\GitHub\grocery
```

Figure 21. Next.js with PWA support basic project installation

The Next.js platform allows to dynamically apply changes to the server without restarting it. But the service worker is assembled based on the current functionality of the software, which means that if changes are made, it will be overwritten. In the context of this platform, this will significantly slow down development, as a result of which it will be temporarily disabled.

The initial installation is complete, so the development of the server-side of the application will be described below. This includes creating a database and scripts associated with them, developing API and creating authentication mechanisms.

3.2.1 Application directory structure

To start developing your application, you need to sketch what needs to be implemented in the code. The architecture was described above, but now we need to disassemble the architecture into smaller parts. As part of the development of the server-side of the application, it is necessary to pay attention to the description of the structure of the project directories and API paths.

When the project was created by the console command, a basic directory structure was created. The initial project structure is extremely simple and contains only pages, stylesheets and public folders. Thus, it is needed to add some folders to store additional files for more flexibility and convenience in working with project files. Table 1 shows the resulting directory structure for the application.

Table 1. Application directory structure

Path	Description
/pages	Contains React documents to be displayed as web pages in the application. Each page is associated with a route based on its file name (Vercel, Inc. 2021d). All contents of this directory will be available at <i>[url]/[path]</i> , where <i>path</i> is the name of a file.
/pages/api	All requests to the server returning json files are described in the files of this directory. As well as the pages, contents of this

	directory will be available at <i>[url]/api/[path]</i> , where <i>path</i> is the name of a file.
/public	Contains files used to generate a web application in the user's browser: pictures, icons, manifest, service worker, and others.
/sql	The contents of the folder are not used anywhere in the project and serve only as a repository of SQL scripts for restoring the database.
/src	When writing functional code, many intermediate functions are created, many of which are used multiple times during assembly. Therefore, it is more rational to take all these functions into separate files, which will be stored in this directory.
/styles	This directory stores the cascading style sheets used in the application. Style files when working with the Next.js platform should be requested in the root React file and loaded during rendering on the server. In production, all CSS files will be automatically concatenated into a single minified “.css” file (Vercel, Inc. 2021b).

Such a project structure divides files according to their purpose, which leads to clarity about the purpose and methods of using certain code fragments contained in them. This will make it easier to understand the code, both if it is necessary to add functionality or refactoring, and to third-party developers who have shown interest in the project.

3.2.2 API planning

In order the client part of the application to be able to interact with the server, it is necessary to create application programming interfaces. In the implementation on the next.js platform, their creation is the placement of functions that can work with request and response in the */pages/api* folder. As stated earlier, the advantage of the platform is its ease of use, as many things, such as API paths, do not require additional configuration to use.

Since the API support technology is based on the http protocol, it is possible to obtain its method from the request, which will allow several functions to be placed on one path at once. To adhere to this concept, it is necessary to clearly understand which interfaces should be created on the server, what methods to interact with them and what data they accept for processing. Based on the requirements for the application described in 3.1 Application architecture, a detailed description of all CUI paths that need to be created was compiled, which is displayed in Table 2.

Table 2. API paths

Path	Method	Input	Description
/api/items	GET	family_id	Displays a list of items for given family.
/api/items	POST	item, desc	Adds a new item to the user's family list based on the name and description.
/api/items	PUT	id	Reserves or cancels reservation of an item by ID.
/api/items	DELETE	id	Removes an item from the database by its ID.
/api/cookies	GET	name, value	Sets a cookie with the name and value obtained from the request. This cookie has a maximum age equals to about 50 years and is available for reading in a browser.
/api/login/auth	GET	user, pass	Sets a cookie for the session if the user has entered the correct username and password. This cookie with a maximum age equals to about 50 years and it is used (even if it is not set) to check user data on all pages and for all client requests.
/api/login/logout	GET	---	Removes session information from the database and a cookie from the client's device.

/api/login/register	GET	user, pass, name	Creates a user using the entered nickname, username and password.
/api/login/user_pass	PUT	user, pass	Allows the user to add a password to the account created by the head of the group.
/api/login/user_pass	DELETE	user	Allows the head of the group to reset the password for a user belonging to the same group.
/api/login/user	GET	---	Displays other members of the user's group.
/api/login/user	POST	user, name, add, reserve	Allows the head of the group to create a new user, specifying a nickname, login and privileges to use the application.
/api/login/user	PUT	id, add, reserve	Allows the head of the group to change the privileges of using the application for the user by id.
/api/login/user	DELETE	id	Allows the head of the group to delete a user account by ID.

The described specification defines a rigid framework for what functionality each API should have. When developing an application, this documentation must be fully implemented, since it is compiled based on the requirements and covers all the declared functionality of the application.

3.2.3 Database implementation

To provide constant access to the stored data used in the application, you need to create and configure a database. This process can be divided into 3 stages: database design, table creation, and scripting.

As mentioned above, the cloud platform provides a PostgreSQL database for use. This solution has several use plans, but a free plan is also suitable for the development process of this application. Figure 22 displays the main database indicators included in the usage plan.

Hobby Dev	Free	Postgres Extensions	✓
Hobby Basic	\$9/mo	RAM	0 Bytes
Standard 0	\$50/mo	Database Forks	
Standard 2	\$200/mo	Direct SQL access	✓
Premium 0	\$200/mo	Row Limit	10,000
Private 0	\$300/mo	Storage Capacity	1 GB
Shield 0	\$350/mo	Database Followers	
Premium 2	\$350/mo	Dataclips	✓
Standard 3	\$400/mo	Continuous Protection	✓
Private 2	\$600/mo	Connection Limit	20
Premium 3	\$750/mo	High Availability	
Standard 4	\$750/mo	Rollback	
Shield 2	\$750/mo	Rollback	0 Seconds
Private 3	\$1000/mo	Encryption at Rest	
Premium 4	\$1200/mo	PostGIS	✓
Shield 3	\$1200/mo	PGBackups	✓
Standard 5	\$1400/mo	PGBackups Retained	2 Backups

Figure 22. Heroku PostgreSQL subscription plans

To implement all the functions, we need to create 4 tables containing data about users, groups to which they belong, a list of items and open sessions. The content and use of the tables are described below:

- **Users.** This table displays basic information about the user, including his name, login, access rights to the list of items and belonging to a group (family).
- **Families.** Is the core of the data connection in the database. Despite the fact that it contains only the group ID, it is used to link users to a family and individual items into a single sheet.
- **Items.** Displays information about the item: what it is, who added it, who reserved it and whose family list it belongs.
- **Sessions.** This table stores information for checking the user for authorization. Each record must contain a session identifier, an indication of the user who opened the session and the date of the last session use to check the activity and the possibility of deleting unused for a long time sessions.

Based on this description, you can draw up a database schema, which is shown in Figure 23.

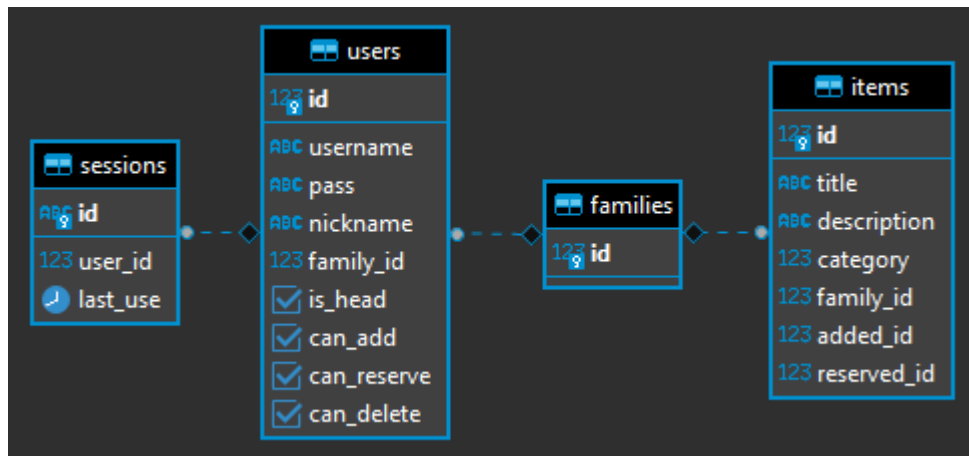


Figure 23. Database structure

As noted earlier, PostgreSQL allows for the use of more data and other methods of working with them than in SQL. PostgreSQL tries to conform with the SQL standard where such conformance does not contradict traditional features or could lead to poor architectural decisions (The PostgreSQL Global Development Group 2021c). There are differences between the languages of these DBMSs, but they are not significant when working with simple databases and data structures, as used in the developed application. Figure 24 shows the creation of the items table.

```

27 create table items (
28     id int primary key,
29     title char(50) not null,
30     description char(150) not null,
31     category int not null,
32     family_id int,
33     added_id int not null,
34     reserved_id int,
35
36     foreign key(family_id) references families(id)
37 );

```

Figure 24. Table creation query

To avoid many database queries, scripting functions will be created that combine many database operations into one for the application server. An example of such a function is shown in Figure 25. Calling any function will be the same as

when writing a database query. Functions created with PL/pgSQL can be used anywhere that built-in functions could be used (The PostgreSQL Global Development Group 2021b).

```

1 CREATE OR REPLACE FUNCTION public.new_head_user(username text, pass text, nickname text)
2 RETURNS integer
3 LANGUAGE plpgsql
4 AS $function$
5 declare
6     family_id integer;
7     user_id integer;
8 begin
9     select max(id) into family_id from families;
10    insert into families values (family_id + 1);
11    select max(id) into user_id from users;
12    insert into users values (user_id + 1, username, pass, nickname, family_id + 1, true, true, true, true);
13    return user_id + 1;
14 end;
15 $function$
16 ;

```

Figure 25. Function to create a new head of a group user

The database supports only one user connection at a time, because of which all incoming requests are queued waiting for their time to receive data from the DB. Without using the function, it would be necessary to make several requests to obtain the desired result (for example, the function in Figure 25 performs 4 requests). This will increase the queue several times, which in turn will increase the load on the data server. In addition, processing access to data within the database is faster than sending multiple requests over the network from the application server. Thus, the functions will speed up the processing of queries and lighten the load on the database.

Following the described specification, a database and functions were created, that is, the store of operable information for the application is ready and can be used to communicate with the application server.

3.2.4 API implementation

As shown in Table 2, a single API path can provide multiple methods for communicating with the server at once. In order not to check for the method of requesting the interface, it was decided to create a single mechanism that can work with the request object to determine further operations that need to be

performed. Figure 26 shows a function that accepts request and response objects and an array of functions, and returns a response to the client.

```

23  ✓ const userMethodSwitch = async (req, res, results) => {
24      const cookies = new Cookies(req, res);
25
26  ✓      const getSuccess = () => {
27          // Convert args to functions
28  ✓          const toFunctions = (n, d) => compose(
29              map(arg => !!arg ? arg : function() {return false;}),
30              take(n),
31              concat(__, repeat(d, n))
32          );
33
34          let fns = toFunctions(5, undefined)(results);
35  ✓          switch(req.method) {
36  ✓              case "GET" :
37                  return fns[0];
38  ✓              case "POST" :
39                  return fns[1];
40  ✓              case "PUT":
41                  return fns[2];
42  ✓              case "DELETE":
43                  return fns[3];
44                  default: fns[4];
45          }
46      }
47
48      const success = await getSuccess()(req, cookies);
49      message(res, success);
50      res.end();
51  }

```

Figure 26. API status switch function

This part of the code is an illustrative example of functional style. It does not interact with the external environment, but uses only its own arguments, which leads to a certain set of results that are obvious. Calling a function once is the same as calling it twice and discarding the result of the first call (Milewski 2014). All the arguments entered are checked against types, and any errors will lead to the same results, which ensures consistency.

Using this method of processing requests allows you to exclude the repetition of multiple repetitions of the same parts of the code. Instead, some of the necessary operations for the request have already been completed and you can focus on the main ones: the authorization mechanism, access to the database, and so on.

Most of the functionality requires working with data, it is necessary to create a clear and permanent mechanism for working with the database. Since the used

DBMS is PostgreSQL, the npm package *pg* by brianm was chosen. It has support for callbacks, promises, async/await, connection pooling, prepared statements, cursors, streaming results, C/C++ bindings, rich type parsing, and more (Carlson 2014). Figure 27 shows the implementation of the algorithm for accessing the database using this library. In addition to making a request, the function will retrieve from the returned object an array containing the requested data.

```

1  const { Client } = require('pg');
2  const { assoc, curry } = require('ramda');
3  const { composeP, viewOnPath } = require('ramda-godlike');
4  const { config } = require('../config');
5
6  const credentials = url => assoc("connectionString", url, {ssl: { rejectUnauthorized: false }});
7
8  const newClient = (creds) => new Client(creds);
9
10 const connectClient = client => client.connect();
11 const queryClient = (query, values, client) => client.query(query, values);
12 const endClient = client => client.end();
13 const accessDb = async (query, values, client) => {
14   await connectClient(client);
15   let result = await queryClient(query, values, client);
16   await endClient(client);
17
18   return result;
19 }
20
21 // getData:: a -> [a] -> [{k: v}]
22 const db = (query, values) => composeP(
23   viewOnPath(["rows"]),
24   curry(accessDb)(query, values),
25   newClient,
26   credentials,
27   config
28 )("db");
29
30 module.exports = { db };

```

Figure 27. Database access function

The created method will be interacted with throughout the application. The example shown in Figure 28 implements the logout process. Using the *db* function, the session obtained from the client's cookies will be deleted from the database. This means that the user will no longer be able to use his account on this device until the new authorization, even if the cookie with the session data is not removed from the device.

```

5  const handler = async (req, res) => {
6      const cookies = new Cookies(req, res);
7      const session = parseInt(cookies.get("session"));
8
9      await db("delete from sessions where id = $1", [session]);
10     cookies.set("session", "", {maxAge: 1000});
11
12     redirect(res, "/auth/login");
13
14     res.end();
15 }

```

Figure 28. Delete session function

As was explained how the session is deleted, it is necessary to explain the authentication process as a whole. The term authentication refers to an electronic process that allows for the electronic identification of a natural or legal person (Dawn 2016). In a development application, this is a user authentication to determine the authority he has in his group.

In order to have access to the application and start using it, you need to go through the user registration process, which algorithm is shown in Figure 29, during which the user data will be checked for uniqueness and correctness and recorded in the database. It should be noted that the username should have at least 8 characters length and the password should have at least 8 characters length, include numbers, both uppercase and lowercase letters and at least one special character (! @ # ?]). These security measures will protect accounts from being hacked.

```

8  const addNewUser = async (res, ...args) => {
9      await db("select * from new_head_user($1::text, $2::text, $3::text)", args);
10
11     redirect(res, "/auth/success");
12 }
13
14 const handler = async (req, res) => {
15
16     const getValue = (param) => JSON.parse(req.body)[param];
17
18     let username = getValue("user");
19     let pass = await composeP(hash, getValue)("pass");
20     let name = getValue("name");
21
22     await checkUsername(username) ?
23         await addNewUser(res, username, pass, name) :
24         message(res, "Username exists. Please, choose another one.");
25
26     res.end();
27 }

```

Figure 29. New user registration function

For greater security of the data stored in the DB, it is necessary to protect user data in the event of a hacker attack on the database. The most critical among the credentials is the password, since it is exactly the code word for logging into the account. Passwords are usually stored in a hashed form in a server's database (Biryukov 2016). In Figure 29, line 19 is a call to the password hash function. It was written using the library *bcrypt* by kelektiv. The function for conversion of the password to hash on the application server is shown in Figure 30.

```

1  const bcrypt = require("bcrypt");
2
3  const hash = async (pass) => {
4    const salt = await bcrypt.genSalt(10);
5    return await bcrypt.hash(pass, salt);
6  }
7
8  const compare = bcrypt.compare;
9
10 module.exports = { hash, compare };

```

Figure 30. Password hashing function

To enter the system, the user enters his username and password. The server retrieves the hash from the database by username and compares it with the entered password. If there is a match, a new record about the open session is created in the DB, and a cookie is sent to the user's device containing the unique identifier of the new session.

Based on the fact that all pages are pre-rendered on the server, we can check whether the user has access to this page or not until a response has been sent. For this, the next.js platform provides the *getServerSideProps* method. It allows you to process the data before the first rendering of the page on the server. Since we are talking about the backend, we will only consider the implementation of the check shown in Figure 31, and not its application.

As stated earlier, if the session is not in the database, the session cookies are meaningless as long as they are deleted. The implementation of this is shown in line 44 in Figure 31. Logging out is done in the same way - the record is deleted from the DB and the user loses all rights, even to use the application.

```

27 const checkPrivileges = async (ctx) => {
28   const cookies = new Cookies(ctx.req, ctx.res);
29   let url = ctx.resolvedUrl;
30
31   let session = cookies.get("session");
32
33   let userData = await db("select id, nickname, is_head, can_add, can_reserve from users where id = (select user_id from sessions where id = $1::text)");
34   let user = userData[0];
35
36   !!user && await db("update sessions set last_use = now() where id = $1::text", [session]);
37   await db("delete from sessions where extract(epoch from now()) - last_use > 86400 * 20");
38
39   // get redirect url
40   let redirect = !!user ? isAuthenticated(url, user) : notAuthenticated(url);
41
42   //remove unnecessary cookies
43   const deleteCookies = c => cookies.set(c, 0, {maxAge: 1000});
44   !!user && forEach(deleteCookies, config("authCookies"));
45
46   //create return object containing only props
47   let returnObj = {props: (!!user ? user : {})};
48
49   //if redirect needed, add redirect to return
50   return !!redirect ? assoc("redirect", redirect, returnObj) : returnObj;
51 }

```

Figure 31. Access control function

All API paths described in section 3.2.2 API planning have been implemented using these design principles. All interfaces are ready to work with the client-side: work with a request and response, connect to a database, check access rights. Everything that was done as part of the development of the server part of the application meets the stated requirements and implements all the announced functions of the developed application.

3.3 Client-side development

The client-side of the application should serve as a tool with which users can interact with the application server through the created earlier API paths. To achieve this, it is needed to create an interface that provides access to use all the functionality of the application.

To avoid the complexity of the interface, it is necessary to separate and group all the frontend functions, placing each on a separate page. A similar procedure was carried out when planning the API paths. Table demonstrates description of all the pages that will be created, their purpose and what actions they provide to the user.

Table 3. Pages routes

Path	Description
/	The welcome page of the application. Contains general information about the functionality and the necessary links to get started.
/404	The error page to which the user will be redirected if the server could not find and could find the page requested by the user.
/500	The error page to which the user will be redirected if an error occurred on the server while processing the user's request.
/auth/register	A page containing a registration form for a new user as a head of the group.
/auth/login	A page that provides a user with a login functionality.
/auth/password	After the head of the group creates a new user, he needs to create a password to start using the application. It can be done on this page.
/auth/success	If the user has successfully registered or created a password, he will be redirected to this page informing about the success.
/app	The main page of the application. It displays a list of items in the group, as well as contains links to go to the page for adding an item. The domain branch <i>/app</i> is available only to registered users; without authorization, a redirect to the welcome page must go through.
/app/help	Contains a guide on how to use the application.
/app/add	This page provides an option to add a new item to the list of topics for users who have permission to do so.
/app/settings	Users can change the color scheme, enter the user administration section or view information about the application on this page.
/app/admin	Displays a list of users included in the group and gives functionality to manage them. Only the head of the family has access to the domain branch <i>/app/admin</i> , the rest will be redirected to the main page.

/app/admin/add	Provides the ability for the head of the group to add a new user.
/app/admin/edit	Provides the ability for the head of the group to edit the user's privileges.

The following will describe the design principles used to create the client-side of the application. This includes creating pages, their interface and functionality (in accordance with Table 3), as well as connecting a service worker to change the type of application into a progressive web application.

3.3.1 User pages implementation

As mentioned earlier, the main task of the client-side is to provide user interaction with the application and having access to all of its functionality. The development of this part will be based on what the user can do, and not what features the application provides. In other words, the development will begin with the implementation of the user interface, to which the functionality will then be attached.

React assumes that functions will return DOM elements by applying code to them. Such an example is shown in Figure 32. But when using the functional programming paradigm, such elements will be used only as the top elements of the DOM.

```
const changeTheme = (isDay, lang, handleClick) => {
  const daytime = () => isDay ?
    <Brightness5Rounded /> :
    <Brightness4Rounded />;

  return (
    <Button
      onClick={e => handleClick(isDay)}
    >
      {daytime()}
    </Button>
  );
}
```

Figure 32. Change theme button function

But this brings many problems that need to be addressed. Hence, another difficulty follows - the resources of the user's device are limited, therefore, solving the problems that have arisen should be easy and effective. The following are the most important decisions made in the development of the client-side of the application.

Functional elements

When writing the code, a functional programming style was used. Instead of writing a single DOM structure, many smaller components were created that combined like bricks to create a page. Figure 33 shows the code written for how that page (*/app/help*); arrows show which functions return which elements. This allowed us to reduce the amount of written code by reusing it.

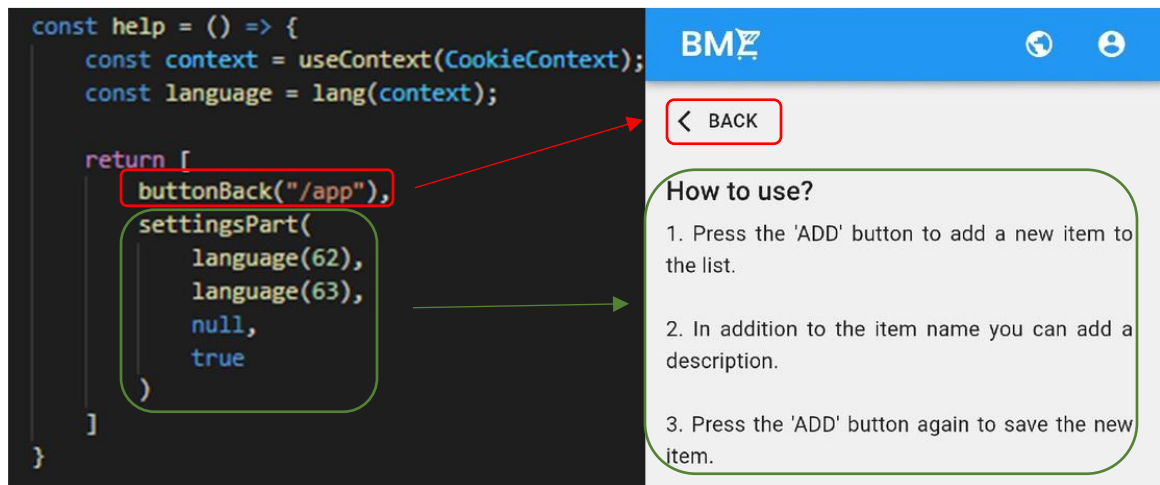


Figure 33. Building interface out of elements

In some components, such as the *settingsPart()* component shown in Figure 33, it was necessary to add customization parameters for more flexible use. Figure 34 shows (marked with red) the use cases for this component. Using additional parameters, it is possible to get a different display on the page, which allows you to avoid creating similar elements with slight differences. Instead, one is created, and the differences act as part of the customization.

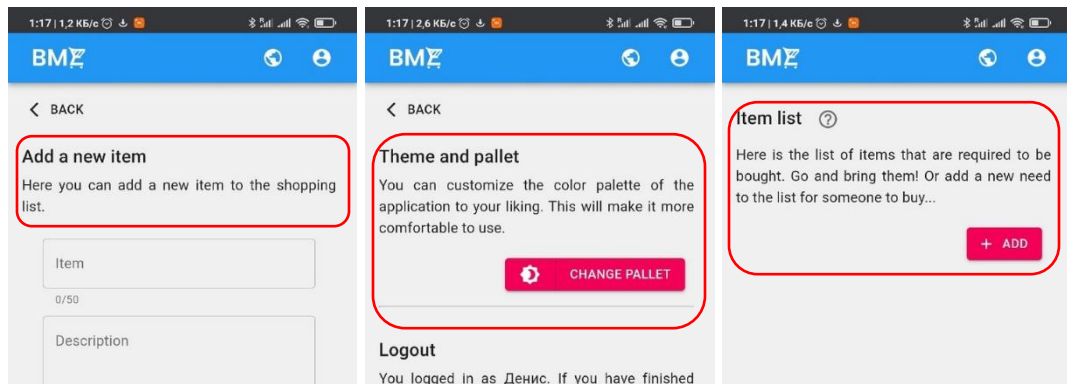


Figure 34. Usage of settingsPart element

The component implementation is shown in Figure 35. Based on this code fragment, an explanation will be given of how it functions and how the customization is implemented.

```

94  const settingsPart = (title, description, content, dropLine) => compose(
95    arr => dropLine ? arr : append(<Line />, arr),
96    map(gridContainer)
97  )(
98    [
99      <Typography variant="h6" style={styles.settingsTitle}><{title}</Typography>,
100      <Typography variant="body1" style={{whiteSpace: 'pre-line'}}><{description}</Typography>,
101      <p style={{marginTop: 1}} />,
102      content
103    ]
104  );

```

Figure 35. settingsPart function

This component takes as arguments a title, description, additional components and information about whether to display the separator. It is worth noting that the possibilities of using MaterialUI also allow to effectively use the functions, since, according to Material-UI (2020), the Typography can accept other elements as children, that is, not limited to text. Thus, when arguments are passed to this component, they essentially become the top elements of the DOM, being passed from the function to its child. This principle is shown in Figure 36 on the example of a part of the application page (*/app*).



Figure 36. Interface composed with functional programming paradigm

Most of the functional elements of the application were created according to the described concept. This allowed many pieces of code to be reused, resulting in a reduction in development time.

Theming

One of the problems that arose during the development was the implementation of the theme change. When the server receives the request, it renders the required component(s) into an HTML string, and then sends it as a response to the client (47). But when using the Next.js platform, all rendering takes place on the server, in other words, in this context, this means that the styles are applied once at load and cannot be changed until the next reload of the web page. It is worth noting that we are talking about a web page, and not about application pages.

To solve this problem, the code of the root element of the application `_document.js` was changed. A custom Document is commonly used to augment your application's `<html>` and `<body>` tags (Vercel, Inc. 2021c). Figure 37 shows the modified code, the new function of which is the dynamic change of style sheets.

```

export default class MyDocument extends Document {
  render() {
    return (
      <Html lang="en">
        <Head style={{maxWidth: "100vw"}}>
          <link
            rel="stylesheet"
            href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap"
          />
        </Head>
        <body>
          <Main />
          <NextScript />
        </body>
      </Html>
    );
  }
}

MyDocument.getInitialProps = async (ctx) => {
  // Render app and page and get the context of the page with collected side effects.
  const sheets = new ServerStyleSheets();
  const originalRenderPage = ctx.renderPage;

  ctx.renderPage = () => {
    originalRenderPage({
      enhanceApp: (App) => (props) => sheets.collect(<App {...props} />),
    });
  };

  const initialProps = await Document.getInitialProps(ctx);

  return {
    ...initialProps,
    // Styles fragment is rendered after the app and page rendering finish.
    styles: [
      ...React.Children.toArray(initialProps.styles),
      sheets.getStyleElement(),
    ],
  };
};

```

Figure 37. Custom Document for processing dynamic theme change

Having solved this problem, the implementation of a quick change of design without reloading the page was introduced. Figure 38 shows how this was done.

```

144   return parseCookies(cookies);
145 }
146
147
148 const Wrapper = ({ children }) => {
149   const clientHeight = useClientHeight(56);
150   const cookies = useCookies();
151
152   return (
153     <CookieContext.Provider value={cookies}>
154       <ThemeProvider theme={theme(cookies)}>
155         <div context={cookies}>
156           <Paper
157             square
158             variant="elevation"
159             elevation={0}
160             style={mergeLeft(styles.paperContainer, {minHeight: clientHeight})}
161           >
162             <Grid
163               container
164               spacing={1}
165               direction="column"
166               alignItems="center"
167               style={styles.gridContainer}
168             >
169               {children}
170             </Grid>
171             <div style={{height: 60}}/
172           </Paper>
173         </ThemeProvider>
174       </CookieContext.Provider>
175     );
176   );
177 }
178 export default Wrapper;

```

```

6  const theme = ((theme, palette) => {
7    const checkArg = (arg) => !!arg ? parseInt(arg) : 0;
8
9    theme = checkArg(theme); // 0 stands for light
10   palette = checkArg(palette); // 0 stands for default theme
11
12   let type = theme ? "dark" : "light";
13
14   let color = (n) => {
15     let c = viewOnPath([palette, type, n], palettes);
16     return !!c ? c : viewOnPath([0, type, n], palettes);
17   }
18   // COLOR: [p.main, s.main, p.text, s.text, error, [bg, paper]]
19
20   return createMuiTheme({
21     palette: {
22       type: type,
23       primary: {
24         main: color(6),
25         contrastText: color(2)
26       },
27       secondary: {
28         main: color(1),
29         contrastText: color(3)
30       },
31       error: {
32         main: color(4),
33       },
34       background: {
35         paper: color(6),
36         default: color(5)
37       },
38     },
39   });
40 }

```

Figure 38. Theme change implementation

Thus, the change of colors and palettes in the developed application is implemented.

Multiple language support

The need to support fast language change has also become a problem in development. The essence of the problem - how to store translations in different languages to be able to quickly change the language without reloading the application.

The solution was to use a json file containing the same modules with the same translation mapping. An example of this file is shown in Figure 39.

```

1 {
2   "lang": "English",
3   "content": [
4     "Enter username and password to login and use the app!",
5     "Login",
6     "Name should have at least 5 characters length.",
7     "Name",
8     "Username should have at least 8 characters length.",
9     "Username",
10    "Password should have at least 8 characters length, include numbers, both uppercase and lowercase letters and at least one special character (! @ # ? ).",
11    "Password",
12    "Enter name, username and password to create an account.",
13    "Passwords do not match.",
14    "Repeat password",
15    "You have successfully created a new account! Log in and get in :)",
16    "Register",
17    "Create a password to use the BRINGM3 app.",
18    "Change pallet",
19    "Theme and pallet",
20    "You can customize the color palette of the application to your liking. This will make it more comfortable to use.",
21  ]
22 }
```

Figure 39. Language map json file

The implementation of changing language (as well as the palette) is implemented through reading cookies. When the user changes the language (or palette), a request for the address is sent to the server. Because of this, a cookie is sent to the client on the device, which contains information about the user's choice. This is shown in Figure 40.

Name	Value	D	P...	E...	S...	HttpOnly	S..	S..	S..	P..
lang	1	b..	/	2...	5					M..

Figure 40. Language cookie

Since the problem is that the application is rendered on the server, this does not allow us to quickly change data without rebooting, then a check for changing user settings is performed every second. The implementation of this is shown in Figure 41. At startup, the application starts and a timer that reads the user's cookies every second.

```
const useCookies = () => {
  const [cookies, setCookies] = useState("");

  const parseCookies = compose(
    mergeAll,
    map(arr => {return {[arr[0]]: arr[1]}}),
    map(split("=")),
    split("; ")
  );

  setInterval(() => {
    // As next.js provides SSR, it is required to check that code
    // tries to access document in browser
    let newCookies = process.browser ? document.cookie : "";
    newCookies !== cookies && setCookies(newCookies);
  }, 1000);

  return parseCookies(cookies);
}
```

Figure 41. Client-side cookie reader function

The function shown in Figure 41 is used at the root level of the application, which allows the state to propagate the values of cookies to all components. When the language function shown in Figure 42 receives data about the selected language by the user, it looks for records in this language in the json file and selects the required one by the number from the function argument.

```
22  const lang = ({ lang }) => (opt) => {
23    let ln = lang ? parseInt(lang) : 0;
24    let phrase = viewOnPath([ln, "content", opt], languages);
25    return !!phrase ? phrase : "ERROR! Contact support."
26  }
```

Figure 42. Phrase returning function

The implementation of this function was stated in the requirements for the usability of the application. It was necessary to do this without spending a lot of resources on the user's device, which was achieved.

3.3.2 PWA implementation

When creating the server, a template was used that provides PWA functionality. But to save computer resources and development time, it was decided to disable this option until the application is fully ready. Now that the application is completely ready, you need to make settings to return to using the service worker. To do this, it is needed to create a manifest, edit the header of the web page document, and add the configuration to the next.config.js file.

A manifest is a file that describes which files will be saved in the client's browser. As it can be seen from the configuration made for the project, shown in Figure 43, the path to the icons, the starting URL, and other information for using as a desktop application are set there. When an HTTP request is received, test if those files are requested, then return those static files (Wang 2021).

```

1  {
2    "name": "BRINGM3 - grocery list",
3    "short_name": "BRINGM3",
4    "theme_color": "#2196f3",
5    "background_color": "#ffffff",
6    "display": "standalone",
7    "orientation": "portrait",
8    "scope": "/",
9    "start_url": "/",
10   "icons": [
11     {
12       "src": "icons/icon-72x72.png",
13       "sizes": "72x72",
14       "type": "image/png"
15     },
16     {
17       "src": "icons/icon-96x96.png",
18       "sizes": "96x96",
19       "type": "image/png"
20     },

```

Figure 43. Manifest file for PWA

For use of the created manifest, it is necessary to create a link to it in the header of the HTML page. Since the project is developed on the next.js platform, you must use the special component `<Head />`. As shown in Figure 44, inside it is written where the manifest, icons and other settings of the web page are stored

on the server. The `<Head />` component used here should only be used for any `<head>` code that is common for all pages (Vercel, Inc. 2021c).

```
<Head>
  <meta charset="utf-8" />
  <meta httpEquiv="X-UA-Compatible" content="IE=edge" />
  <meta name="description" content="Description" />
  <meta name="keywords" content="Keywords" />
  <title>BRINGM3</title>
  <meta name="viewport" content="initial-scale=1, width=device-width" />
  <link rel="shortcut icon" type="image/x-icon" href="favicon.ico" />
  <link rel="manifest" href="/manifest.json" />
  <link
    href="/icons/favicon-16x16.png"
    rel="icon"
    type="image/png"
    sizes="16x16"
  />
  <link
    href="/icons/favicon-32x32.png"
    rel="icon"
    type="image/png"
    sizes="32x32"
  />
  <meta name="theme-color" content="#000" />
</Head>
```

Figure 44. Custom Head of the application

At the conclusion of the PWA configuration, it is necessary to set up in the platform configuration that the npm package *next-pwa* should be used, which converts a simple web application into a progressive one. It is also necessary to specify which folder will be public, that is, from which folder the server will take files available to the client. Figure 45 shows this configuration.

```
1 | const withPWA = require('next-pwa');
2 | const runtimeCaching = require('next-pwa/cache');
3 |
4 | module.exports = withPWA({
5 |   pwa: {
6 |     dest: 'public',
7 |     runtimeCaching,
8 |   },
9 | })
```

Figure 45. next.config configurations

By opening the browser developer panel, you can find information about the running service worker. Figure 46 shows the Google Chrome browser developer

panel. It is clear from the status that the application is now functioning as a progressive web application.

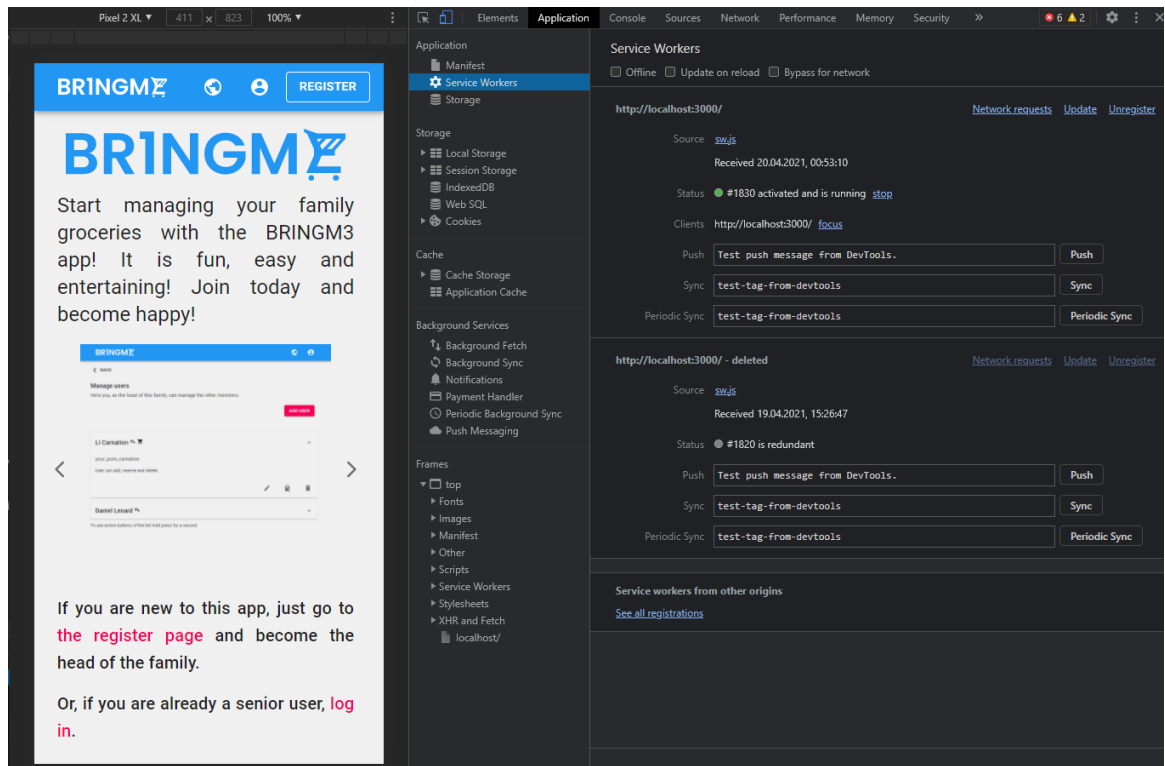


Figure 46. Service worker usage in browser

The performed settings turned the created application into a PWA. This means that the download speed of content will increase, and users will have access to the application, even if the server is offline. Thus, one more requirement stated earlier for the developed application is completed.

3.4 Deployment on server

An account was created for publishing on Heroku. By default, all accounts receive a free subscription with minimal features such as longer start-up times, limited hours of work per week, little memory, and more. More details about the restrictions can be found in the Figure 47. But for the developed application, such conditions are acceptable.

	Free & Hobby Try Heroku with no commitment.	Standard & Performance Run business apps in production.	Private Build secure, private apps.	Shield High-compliance apps.
	< Free		Hobby	>
RAM	512MB		512MB	
Deploy from Git	•		•	
Automated OS patching	•		•	
Unified logs	•		•	
Number of process types	2		10	
Always on	Sleeps after 30 mins of inactivity, otherwise always on depending on your remaining monthly free dyno hours.		•	
Custom domains	•		•	
Free SSL on custom domains			•	
Automated Certificate Management on custom domains			•	

Figure 47. Heroku hosting pricing (Salesforce 2021)

To upload files to the server, you need to create an application, as shown in Figure 48. The name of the application will become a domain name, that is, by entering "bringm3" the application will be published at *<https://bringm3.herokuapp.com/>*. It is also worth paying attention to the selected region because the waiting time for the response depends on this.


Create New App

App name

✓

bringm3 is available

Choose a region

 Europe
 ↕

Add to pipeline...

Create app

Figure 48. Heroku application creation

Uploading to the cloud application server will be done through the GitHub repository. For the convenience of development, version control was carried out, therefore, for publication, you must select the desired version of the application, as shown in Figure 49. It will be automatically uploaded to the Heroku server.

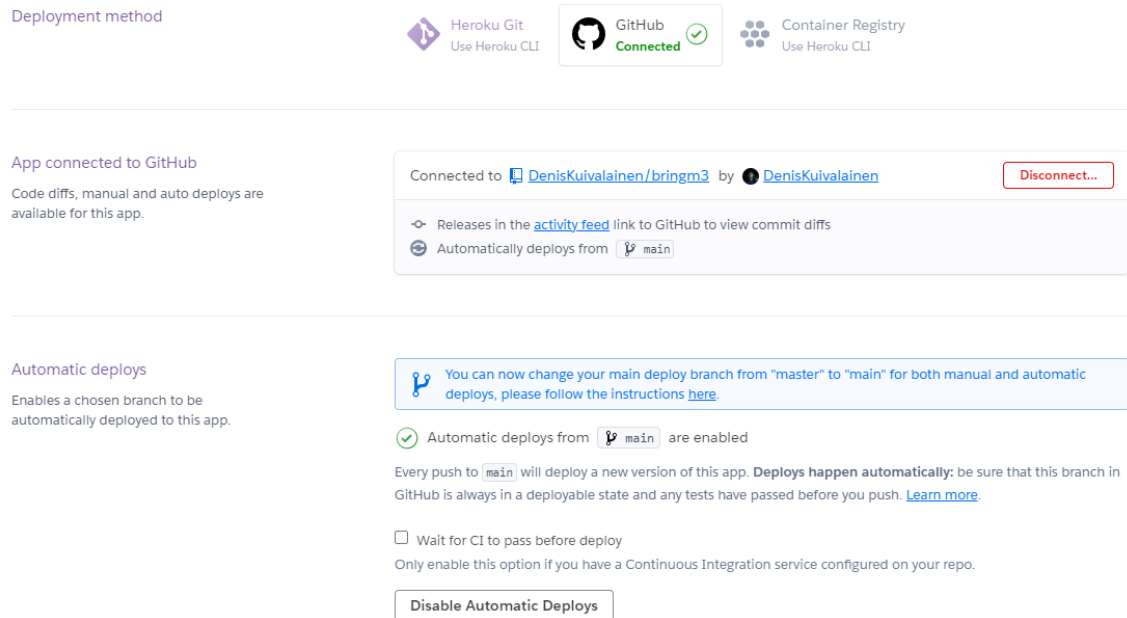


Figure 49. Heroku deployment configuration

The installation process is shown in Figure 50. It should be noted that when the platform is launched, the command *yarn run build* is launched, and when the application is opened *yarn run build*.



Figure 50. Application installation to the Heroku server process

The application is uploaded to the server and ready to use. This means that the application is available for use over the internet.

3.5 Demonstration

For clarity of the functionality of the application, a demonstration of its work will be made. The course will show you creating an account, logging in, adding an item, changing a style, and adding a new user. These are the main functions that the application must support according to the stated requirements.

To create a new account, you need to open the application in a browser on <https://bringm3.herokuapp.com/>. On the navigation panel there is a button "REGISTRATION", by clicking on it, the user gets to the page with the registration form. After entering the data and pressing the "REGISTER" button, in case of success, it will be redirected to the page with information that the account has been created. It is shown in Figure 51.

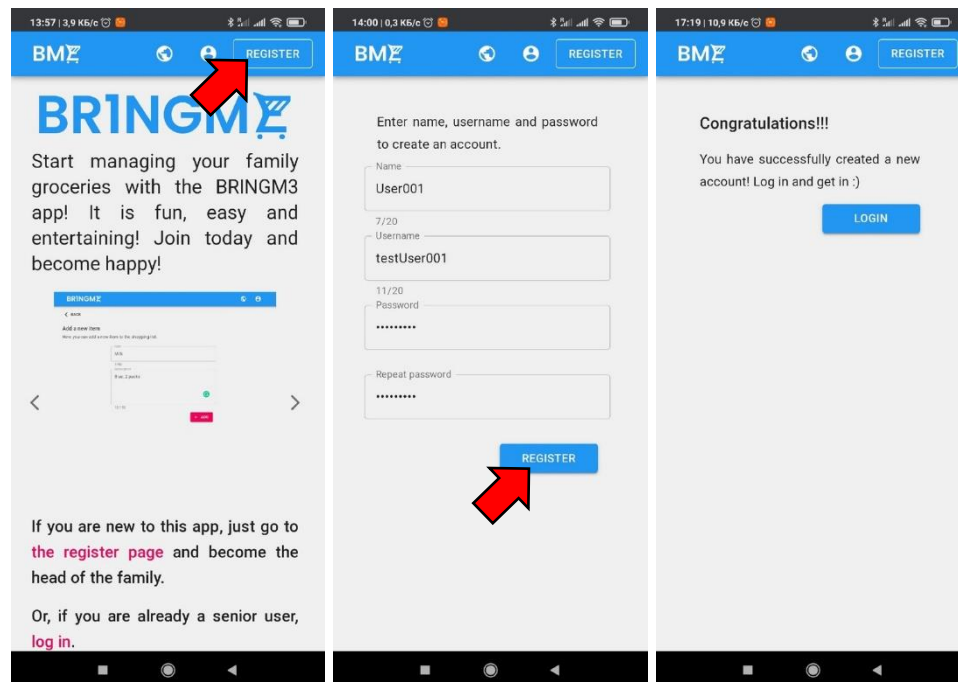


Figure 51. Registration of a new user process

After successfully creating an account, you must be logged in to work with the application. As shown in Figure 52, you must enter the previously created

username and password, and if successful, you will be redirected to the main page of the application.

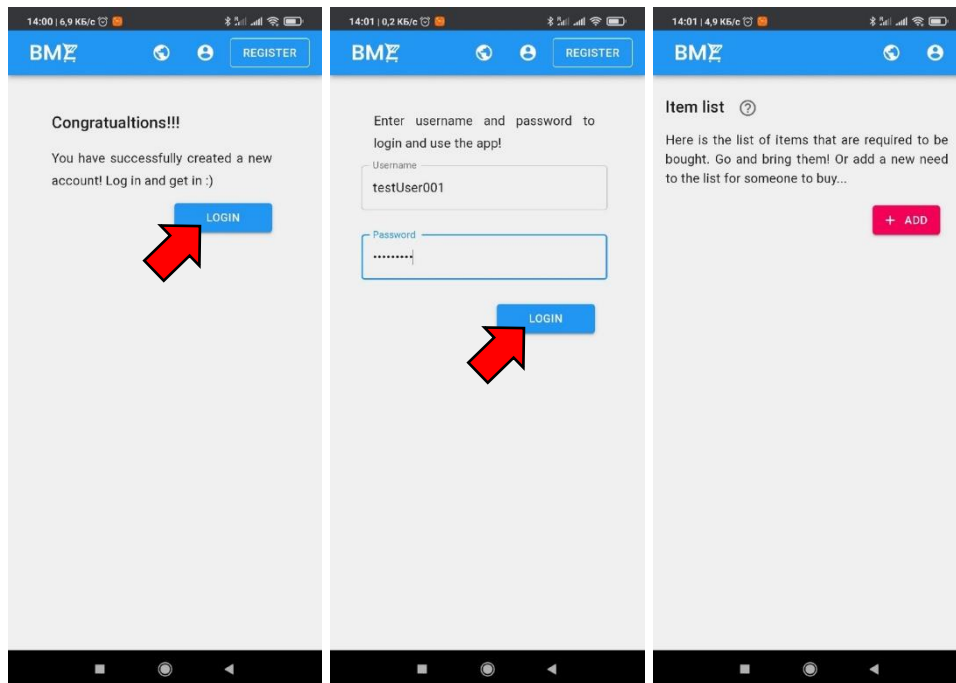


Figure 52. Logging in process

The application provides the functionality to administer the grocery list. Figure 53 shows the steps you need to take to add a new item to the list.

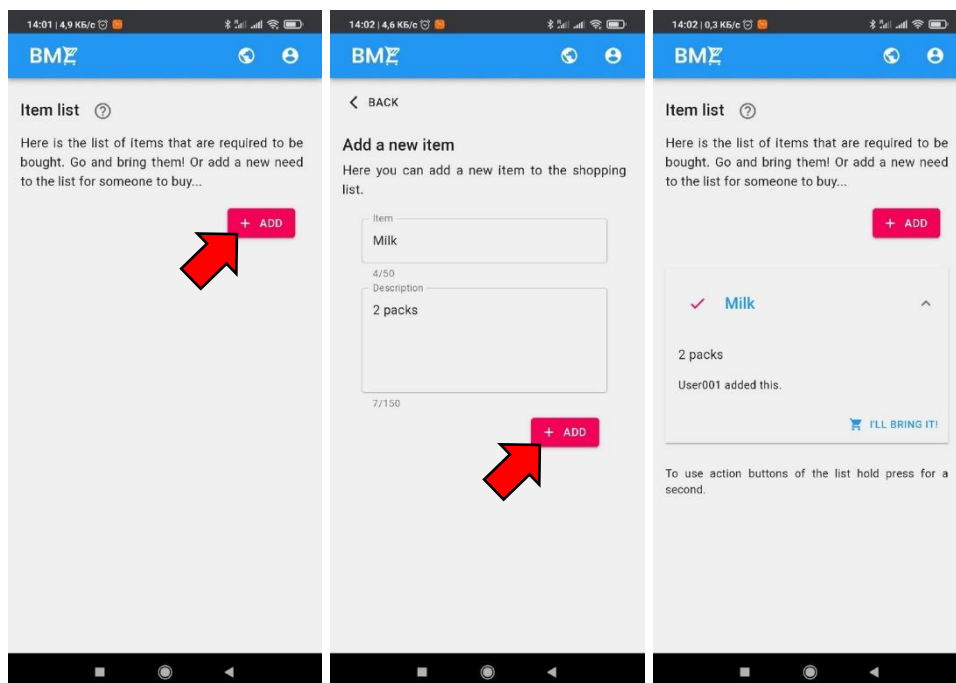


Figure 53. New item adding process

To change the design, you need to go to the settings menu and select the desired palette in the required section. This is clearly shown in Figure 54.

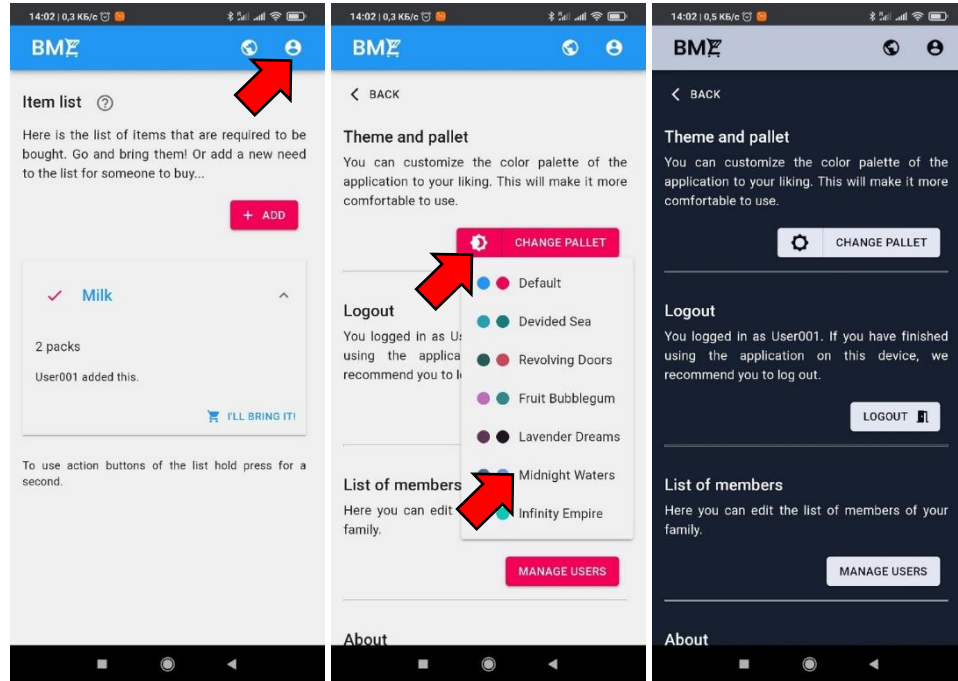


Figure 54. Changing theme process

Adding a user is available only to the head of the group through the page "Manage users". The newly created user can be given privileges according to which he will interact with the list. The process for adding a new user is shown in Figure 55.

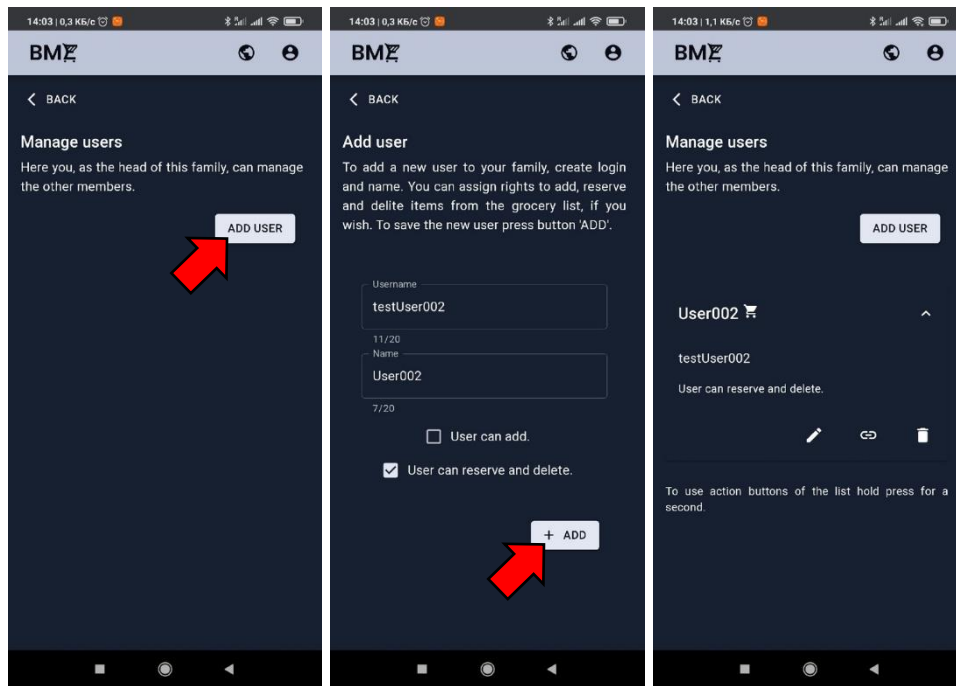


Figure 55. New user adding process

The examples above show only part of the application, but this reflects that the application is functional. The examples provided demonstrate that the application meets all the stated requirements described in section 3.1 Application architecture and all essential functionality according to initial idea was implemented.

4 CONCLUSIONS

The purpose of the thesis was to develop an application and conduct research to implement this. The requirements for the application were described, the technology stack was determined, and a plan was drawn up to study the necessary technologies.

Because an action plan and how the application should look like were not drawn up, but the requirements for its functionality were only known, flexibility in development was achieved. This means that all the time was devoted to achieving specific goals, and not following the plan, which allowed changing minor aspects depending on the current situation.

However, the app is still under development. In the future, some changes will be added related to registration and logging in via email, adding categories to items (this function is implemented on the server side, but not on the client side), adding subscription plans and other features.

To implement the project, a study was conducted on trends in web development. This included a deeper understanding of the JavaScript programming language, the study of libraries that make it easier to write code, the study of the basics of the PostgreSQL database, as well as the mechanics of creating and connecting the client and server parts of the application. All the knowledge gained was tested in practice during the development of the application.

In the boundaries of thesis, an application was developed and created for maintaining a grocery list for a group or family. In addition to the main purpose, the created application had to support accounting for working with groups of users, the ability to flexibly customize to improve the user experience and have high technical characteristics, such as the speed of processing user requests. All these requirements were completed during development.

REFERENCES

- Biryukov, A., Dinu, D., Khovratovich, D. 2016. Argon2: new generation of memory-hard functions for password hashing and other applications. Available at: <https://orbilu.uni.lu/bitstream/10993/31652/1/Argon2ESP.pdf> [Accessed 9 April 2021]
- Carlson, B. 2014. Documentation. WWW document. Available at: <https://node-postgres.com/> [Accessed 7 April 2021]
- Dawn, M. 2016. Digital Authentication - the basics. WWW document. Available at: <https://www.cryptomathic.com/news-events/blog/digital-authentication-the-basics> [Accessed 7 April 2021]
- Ecma International. 2020. ECMAScript 2020 Language Specification. WWW document. Available at: <https://262.ecma-international.org/11.0/#sec-intro> [Accessed 20 March 2021]
- Facebook Inc. 2021a. Getting Started. WWW document. Available at: <https://reactjs.org/docs/getting-started.html> [Accessed 31 March 2021]
- Facebook Inc. 2021b. Hooks at a Glance. WWW document. Available at: <https://reactjs.org/docs/hooks-overview.html> [Accessed 2 April 2021]
- Facebook Inc. 2021c. React. WWW document. Available at: <https://reactjs.org/> [Accessed 2 April 2021]
- Facebook Inc. 2021d. Using the Effect Hook. WWW document. Available at: <https://reactjs.org/docs/hooks-effect.html> [Accessed 2 April 2021]
- Google. 2021. Introduction to the Angular Docs. WWW document. Available at: <https://angular.io/docs> [Accessed 31 March 2021]
- Lonsdorf, B. 2018. Professor Frisby's Mostly Adequate Guide to Functional Programming. Ebook. Available at: <https://mostly-adequate.gitbook.io/mostly-adequate-guide/> [Accessed 26 March 2021]
- Martin, R. 2018. Clean Architecture. Hoboken, New Jersey: Prentice Hall.
- Material-UI. 2019. Typography API. Server Rendering. Available at: <https://material-ui.com/guides/server-rendering/#the-theme> [Accessed 21 April 2021]
- Material-UI. 2020. Typography API. WWW document. Available at: <https://material-ui.com/api/typography/> [Accessed 11 April 2021]
- Material-UI. 2021. Installation. WWW document. Available at: <https://material-ui.com/getting-started/installation/#npm> [Accessed 2 April 2021]

Milewski, B. 2014. Pure Functions, Laziness, I/O, and Monads. WWW document. Available at:

<https://web.archive.org/web/20161027145455/https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io>

[Accessed 7 April 2021]

Mozilla. 2021a. Arrow function expressions. WWW document. Available at:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions [Accessed 23 March 2021]

Mozilla. 2021b. Async function. WWW document. Available at:

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/async_function [Accessed 23 March 2021]

Mozilla. 2021c. Destructuring assignment. WWW document. Available at:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

[Accessed 23 March 2021]

Mozilla. 2021d. JavaScript. WWW document. Available at:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript> [Accessed 19 March 2021]

Mozilla. 2021e. Progressive web apps (PWAs). WWW document. Available at:

https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps [Accessed 6 April 2021]

Mozilla. 2021f. Promise. WWW document. Available at:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise [Accessed 23 March 2021]

Mozilla. 2021g. Spread syntax (...). WWW document. Available at:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax [Accessed 23 March 2021]

Netscape Communications Corporation. 1998. JavaScript Overview. WWW document. Available at:

<https://docs.oracle.com/cd/E19957-01/816-6411-10/intro.html> [Accessed 27 March 2021]

Netscape. 1999. Netscape and Sun Announce Javascript, the Open, Cross-platform Object Scripting Language for Enterprise Networks and the Internet. WWW document. Available at:

<https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html> [Accessed 19 March 2021]

npm, Inc. 2021. About npm. WWW document. Available at:

<https://www.npmjs.com/about> [Accessed 28 March 2021]

OpenJS Foundation. 2020. About Node.js. WWW document. Available at: <https://nodejs.org/en/about/> [Accessed 28 March 2021]

Ramda. 2021. Ramda. WWW document. Available at: <https://github.com/ramda/ramda> [Accessed 28 March 2021]

Salesforce. 2021. Heroku Pricing. WWW document. Available at: <https://www.heroku.com/pricing> [Accessed 11 April 2021]

Stack Overflow. 2020a. Databases. Picture. Available at: <https://insights.stackoverflow.com/survey/2020#technology-databases> [Accessed 27 March 2021]

Stack Overflow. 2020b. Developer Survey. WWW document. Available at: <https://insights.stackoverflow.com/survey/2020> [Accessed 27 March 2021]

Stack Overflow. 2020c. Web Frameworks. Picture. Available at: <https://insights.stackoverflow.com/survey/2020#technology-web-frameworks-all-respondents2> [Accessed 27 March 2021]

Stack Overflow. 2021. Stack Overflow Trends. Picture. Available at: <https://insights.stackoverflow.com/trends?tags=reactjs%2Cangular%2Cvue.js%2Cbackbone.js%2Cember.js%2Cvue.js%2Ccaurelia> [Accessed 31 March 2021]

The PostgreSQL Global Development Group. 2021a. Data Types. WWW document. Available at: <https://www.postgresql.org/docs/current/datatype.html> [Accessed 6 April 2021]

The PostgreSQL Global Development Group. 2021b. Overview. WWW document. Available at: <https://www.postgresql.org/docs/9.6/plpgsql-overview.html> [Accessed 6 April 2021]

The PostgreSQL Global Development Group. 2021c. What is PostgreSQL? WWW document. Available at: <https://www.postgresql.org/about/> [Accessed 6 April 2021]

Thomas, D., Hunt, A. 2020. The Pragmatic Programmer. Boston, Massachusetts: Addison-Wesley.

Tilde inc. 2021. What is Ember?. WWW document. Available at: <https://guides.emberjs.com/release/getting-started/> [Accessed 31 March 2021]

Anne van Kesteren. 2021a. DOM. WWW document. Available at: <https://dom.spec.whatwg.org/#infrastructure> [Accessed 30 March 2021]

Anne van Kesteren. 2021b. HTML Standart. WWW document. Available at: <https://html.spec.whatwg.org/multipage/> [Accessed 30 March 2021]

Vercel, Inc. 2021a. API Routes. WWW document. Available at: <https://nextjs.org/docs/api-routes/introduction> [Accessed 28 March 2021]

Vercel, Inc. 2021b. Built-In CSS Support. WWW document. Available at: <https://nextjs.org/docs/basic-features/built-in-css-support> [Accessed 7 April 2021]

Vercel, Inc. 2021c. Custom Document. WWW document. Available at: <https://nextjs.org/docs/advanced-features/custom-document> [Accessed 23 April 2021]

Vercel, Inc. 2021d. Pages. WWW document. Available at: <https://nextjs.org/docs/basic-features/pages> [Accessed 7 April 2021]

Vercel, Inc. 2021e. Why Next.js. WWW document. Available at: <https://nextjs.org/> [Accessed 28 March 2021]

W3C. 2016. HTML & CSS. WWW document. Available at: <https://www.w3.org/standards/webdesign/htmlcss#whatcss> [Accessed 30 March 2021]

W3Techs. 2021. Usage statistics of JavaScript as client-side programming language on websites. WWW document. Available at: <https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html> [Accessed 19 March 2021]

Wang, W. 2021. Zero Config PWA Plugin for Next.js. Available at: <https://github.com/shadowwalker/next-pwa> [Accessed 9 April 2021]