**“Self-hosted library” application (final project)**

Georgii Frolov, Lyubov Lavrova, Ekaterina Kovrigina(5130903/00301)

**1. WEB Service description**

1.1 Description of the server functionality

The application has the following purpose: it is used to store opinion and some simple data about books, that person read. We found this purpose quite simple and enough useful to serve a role of demonstrative application. To be more specific, opinion of the user is expressed with 2 features: rating from 0 to 10 and comment, which gives user a possibility to express all his feelings about the book. The information about the book is author, author’s country, title of the book and cover image. Every book and author have unique id, each book in books collection it has a link to authors collection. User can perform following actions on books: add, remove, and update. Also, while adding, the information about the book can be found using ISBN of the book, provided by user.

Server has 7 endpoints, which form JSON response using HTTP requests. Their parameters are listed in table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **URL** | **METHOD** | **URL PARAMETERS** | **REQUIRED BODY** |
| getBooks | / | GET | - | - |
| addBook | /:id | POST | + | + |
| deleteBook | /:id | DELETE | + | - |
| updateBook | /:id | PATCH | + | + |
| getAuthors | /authors | GET | - | - |
| getBookByISBN | /isbn/:isbn | GET | + | - |
| getBookByID | /:id | GET | + | - |

1.2 Architecture of the software

The server application was created using NodeJS and MongoDB. The developing process was simplified critically by using MongoDB ORM Mongoose and NodeJS framework express, which is typically used for creation of light APIs.

Database has 2 collections: Authors and Books. Their fields are listed in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| COLLECTION | FIELD NAME | TYPE | REQUIRED |
| Authors | Name | String | + |
| Authors | Country | String | - |
| Books | Author | Object\_id | + |
| Books | Name | String | + |
| Books | Image\_url | String | - |
| Books | Comment | String | - |
| Books | Rating | Number | - |

As we said before, server interacts with database by means of mongoose instances. To be able to correctly retype and fill in DB with data, mongoose requires user-written schemes for each db collection. They are quite demonstrative for better understanding of data structure of the app, so we will provide you with them:

Изображение выглядит как текст, визитная карточка, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, меню, дизайн

Автоматически созданное описание

As you can see, it is exact copy of MongoDB collection schema, but described in means of javascript.

Every API endpoint, all of which were described in previous subparagraph, are brought out to single file.

The main feature of our architecture, which is quite common today, is that we have a server, which can be used by multiple client apps.

In our case we have two “client” apps. One is react frontend app, which will be described in greater detail in paragraph 2. Second is kind of templater, which main goal is to display data as HTML code. Despite that we decided to implement all same functionality in both “clients”.

1.3 Sources of all the servlets/scripts

We weren’t quite sure what we should’ve provide in this subparagraph, because providing all our source code would make the report kind of superfluous and useless, because there is a lot of simple code, which will describe itself clearly. Thus, we will paste one server script here. The rest can be inspected in attached “service” archive.

Here is the code of main server script getBooks, which is used to get info about all books, that are currently stored:



As you can see, it requires model of Book instance, which was described in previous subparagraph. Then it gets all books from collection and populate author field, as by structure of our database it is link to instance of collection Authors.

**2. Interface to the WEB service description**

Let’s describe all endpoints and what they take as input and format of information they return:

* getBooks

URL template: localhost:3000

Method: GET

Request body example: {}

Response body example:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

* deleteBook

URL template: localhost:3000/:id

Method: DELETE

Request body example: {}

Response body example:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

·       getAuthors

URL template: localhost:3000/authors

Method: GET

Request body example: {}

Response body example:

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

• getBook

URL template: localhost:3000/:id

Method: GET

Request body example: {}

Response body example:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

·       getBookByISBN

URL template: localhost:3000/isbn/:isbn

URL example: localhost:3000/isbn/9785389010925

Method: GET

Request body example: {}

Response body example:

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

·       addBook

URL template: localhost:3000/

Method: POST

Request body example:

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Response body example:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

·       updateBook

URL template: localhost:3000/

Method: PATCH

Request body example:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

Response body example:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

**3. Front-End**

So, as we are using React, our front is component oriented. Most of the React-based development features are used, such as hooks, effects. Important to mention, that development is performed by means of typescript, so on each component we have typed props, event handlers and so on. Also, we created custom context to simplify the access to information about current books and authors throughout the app (see contexts/AuthorContext.tsx and contexts/BookContext.tsx).

As we use typescript, we also need to use some kind of instance for typed objects, which come from servers. We use interfaces listed below:

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

Again, it would be superfluous to provide all js code here, as report would take 100 pages. You can see all js code in archive called “front-end”.

Now let’s look at main component called BookList, which is rendered in default url.

import { useEffect, useState } from "react";

import BookCard from "./BookCard";

import Book from "../interfaces/book";

import Author from "../interfaces/author";

import UpdateBookForm from "./UpdateBookForm";

import { useBooks } from "../contexts/BookContext";

import { useAuthors } from "../contexts/AuthorContext";

interface RawBookData {

    \_id: string;

    image\_url?: string;

    name: string;

    author: object;

    rating: number;

    comment: string;

}

interface RawAuthorData {

    \_id: string;

    name: string;

    country: string;

}

const BookList = () => {

    const { books, setBooks } = useBooks();

    const { setAuthors } = useAuthors();

    const [editingBook, setEditingBook] = useState<Book | null>(null);

    const handleEditBook = (book: Book) => {

        setEditingBook(book);

    };

    const handleUpdateSubmit = async (updatedBookData: Book) => {

        setEditingBook(null);

        try {

            const response = await fetch(

                `http://localhost:3000/${updatedBookData.id}`,

                {

                    method: "PATCH",

                    headers: {

                        "Content-Type": "application/json",

                    },

                    body: JSON.stringify({

                        name: updatedBookData.name,

                        image\_url: updatedBookData.imageUrl,

                        rating: updatedBookData.rating,

                        comment: updatedBookData.comment,

                        author: updatedBookData.author.name,

                    }),

                }

            );

            if (!response.ok) {

                throw new Error("Failed to update the book");

            }

            const result = await response.json();

            console.log("Book successfully updated:", result);

            setBooks([

                ...books.filter(

                    (currentBook) => currentBook.id != updatedBookData.id

                ),

                updatedBookData,

            ]);

        } catch (error) {

            console.error("Error updating book:", error);

        }

    };

    useEffect(() => {

        const fetchBooks = async () => {

            try {

                const response = await fetch("http://localhost:3000/");

                if (!response.ok) throw new Error("Something went wrong!");

                const rawData = await response.json();

                const books: Book[] = rawData.map((rawBook: RawBookData) => ({

                    id: rawBook.\_id,

                    imageUrl: rawBook.image\_url ?? null,

                    name: rawBook.name,

                    author: rawBook.author,

                    comment: rawBook.comment,

                    rating: rawBook.rating,

                }));

                setBooks(books);

                console.log(books);

            } catch (error) {

                console.log(error);

            }

        };

        const fetchAuthors = async () => {

            try {

                const response = await fetch("http://localhost:3000/authors");

                if (!response.ok) throw new Error("Something went wrong!");

                const rawData = await response.json();

                const authors: Author[] = rawData.map(

                    (rawAuthor: RawAuthorData) => ({

                        id: rawAuthor.\_id,

                        name: rawAuthor.name,

                        country: rawAuthor.country,

                    })

                );

                setAuthors(authors);

                console.log(authors);

            } catch (error) {

                console.log(error);

            }

        };

        fetchBooks();

        fetchAuthors();

    }, [setAuthors, setBooks, handleUpdateSubmit]);

    return (

        <div

            style={{

                maxWidth: "1280px",

                display: "flex",

                justifyContent: "space-between",

                alignContent: "flex-start",

                marginTop: "20px",

                flexWrap: "wrap",

            }}

        >

            {books.map((book: Book) => (

                <BookCard

                    key={book.id}

                    book={book}

                    setBooks={setBooks}

                    onEdit={() => handleEditBook(book)}

                />

            ))}

            {editingBook && (

                <UpdateBookForm

                    initialBookData={editingBook}

                    onSubmit={handleUpdateSubmit}

                    setBook={setEditingBook}

                />

            )}

        </div>

    );

};

export default BookList;

So, here we use 2 contexts and 2 effects each for books and authors. Effects trigger when book added or updated. Inside effects we fetch information with server. Then we process it(convert it to object with types of our interfaces and then contexts are updated).

Function, which handles book updating is forwarded to component UpdateBookForm, which is quite common way to handle such situations.

Information about books is forwarded to BookCard components, which are then rendered.

Here we will provide some screenshots of the interface:

Изображение выглядит как текст, Человеческое лицо, снимок экрана, человек

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Мультимедийное программное обеспечение, дизайн

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Мультимедийное программное обеспечение, программное обеспечение

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Мультимедийное программное обеспечение, программное обеспечение

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, плакат, багет

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Мультимедийное программное обеспечение, программное обеспечение

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Мультимедийное программное обеспечение, программное обеспечение

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

**4. WEB client**

The web client is written on php. It fetches data with our server using curl\_init, curl\_exec, curl\_set and curl\_close built-in functions. The main page of web client performs a role of data templater. Let’s take a look at it:

<?php

include 'fetch.php';

include 'utils.php';

$data = get\_books();

$page\_content = render('templates/main.php', ['books' => $data]);

$layout\_content = render(

    'templates/layout.php',

    ['content' => $page\_content, 'title' => 'Книги']

);

print ($layout\_content);

So, first we fetch books, then forward php array with data about books into main template.

Let’s see, how we perform fetching of books:

function get\_books()

{

    $curl = curl\_init("http://localhost:3000");

    curl\_setopt($curl, CURLOPT\_RETURNTRANSFER, true);

    curl\_setopt($curl, CURLOPT\_HTTPGET, true);

    $response = curl\_exec($curl);

    $data = "";

    if ($response === false) {

        $error = curl\_error($curl);

        echo "cURL Error: $error";

    } else {

        $data = json\_decode($response, true);

    }

    curl\_close($curl);

    return $data;

}

So, we set a url, init a connection with it, then set property, that allow us to get response and then set method of request, which is GET in this case. Then we execute the request. If we get positive response, we convert is from json to php array and return it. Otherwise, we print error and return empty data. Every fetching in project is performed very similar way.

<nav>

    <a href="add.php"><button class="addBook">Добавить книгу</button></a>

</nav>

<main>

    <div class="books">

        <?php foreach ($books as $book): ?>

                    <div class="book" onclick="location.href='/update.php?id=<?= $book['\_id'] ?>'">

                        <img src="

                        <?= $book['image\_url'] ?? "" ?>" alt="">

                        <h2> <?= $book['name'] ?> </h2>

                        <h4> <?= $book['author']['name'] ?> </h4> <?php if (isset($book['rating'])): ?> <p> <?= $book['rating'] ?> / 10 </p> <?php endif; ?>         <?php if (isset($book['comment'])): ?> <p class="comment"> <?= $book['comment'] ?> </p> <?php endif; ?> <a href="delete.php?id=

                            <?= $book['\_id'] ?>" onclick="event.stopPropagation();">

                            <svg viewBox="0 0 24 24" fill="none" xmlns="http://www.w3.org/2000/svg">

                            <g id="SVGRepo\_bgCarrier" stroke-width="0" />

                            <g id="SVGRepo\_tracerCarrier" stroke-linecap="round" stroke-linejoin="round" />

                            <g id="SVGRepo\_iconCarrier">

                                <path d="M10 11V17" stroke="#fff" stroke-width="2" stroke-linecap="round" stroke-linejoin="round" />

                                <path d="M14 11V17" stroke="#fff" stroke-width="2" stroke-linecap="round" stroke-linejoin="round" />

                                <path d="M4 7H20" stroke="#fff" stroke-width="2" stroke-linecap="round" stroke-linejoin="round" />

                                <path d="M6 7H12H18V18C18 19.6569 16.6569 21 15 21H9C7.34315 21 6 19.6569 6 18V7Z" stroke="#fff" stroke-width="2" stroke-linecap="round" stroke-linejoin="round" />

                                <path d="M9 5C9 3.89543 9.89543 3 11 3H13C14.1046 3 15 3.89543 15 5V7H9V5Z" stroke="#fff" stroke-width="2" stroke-linecap="round" stroke-linejoin="round" />

                            </g>

                            </svg>

                        </a>

                    </div>

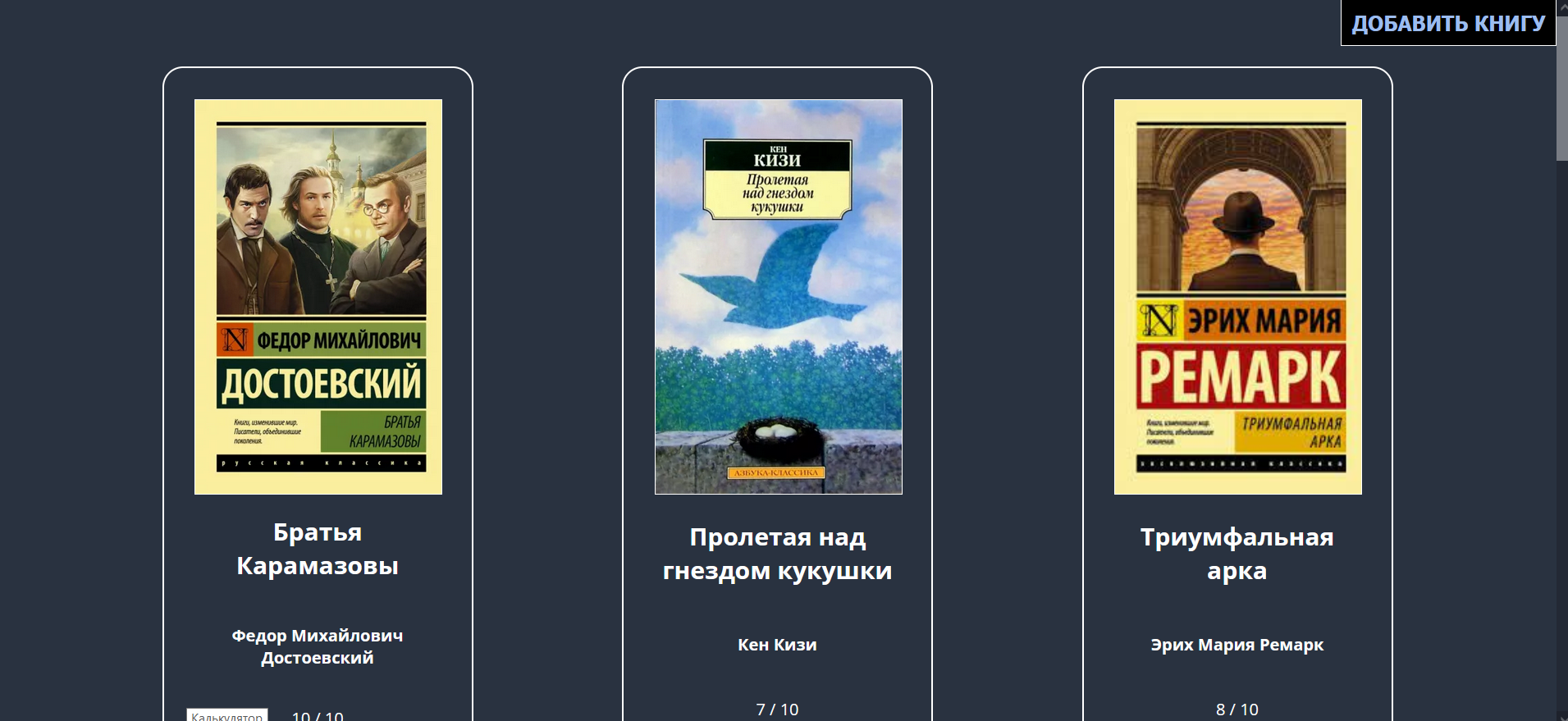
        <?php endforeach; ?>

    </div>

</main>

Then in main template we render each book in for cycle. Full information about books is available in separate html tag.

Let us provide you with screenshots of the interface:



Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Изображение выглядит как текст, Рекламный проспект, плакат, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Изображение выглядит как текст, плакат, Рекламный проспект, снимок экрана

Автоматически созданное описание