Техническое описание проекта

Проект **www.webdotg.ru** состоит из трех основных компонентов:

Client: Это фронтендная часть проекта, написанная с использованием Vite, React, TypeScript. Представляет собой сайт-портфолио-магазин без онлайн оплаты.

Содержит страницы: Главная, Портфолио, Магазин, Логин, Регистрация, Доска объявлений, Кабинет.

Server: Бэкендная часть проекта, построенна на ExpressJS. Взаимодействует с базой данных через SQL запросы. Имеет маршруты для пользователей, постов, тегов, корзины.

DB: База данных расположена на удаленном сервере. Для хранения данных используется PostgresSQL с таблицами users, orders, posts, tags. Настройки базы данных (адрес сервера, логин, пароль) хранятся в .ENV файле серверной части проекта.

Для обращения к базе данных используются SQL запросы с использованием библиотеки pg.

Хостинг и домены

Фронтендная часть временно хостится на github.io и привязана к частному домену www.webdotg.ru через gh-pages. В провайдере настроены DNS записи, включая 4 записи A с IP-адресом Git и 1 запись CNAME, перенаправляющая с www... на хост Git.

Перенос проекта на удаленный сервер с использованием Docker

Dockerfile используется для сборки Docker-образа из проекта, который включает серверную и клиентскую части.

Базовый образ Node.js:

Используется образ Node.js в качестве основы для сборки.

Устанавливается рабочая директория /app, где будут размещены все файлы проекта.

```
# Указываю базовый образ для Node.js
FROM node AS build

# Устанавливаю рабочую директорию
WORKDIR /app
```

Серверная часть:

Копируются файлы package.json и package-lock.json для установки зависимостей серверной части.

Запускается команда npm install для установки зависимостей.

```
# Копирую файлы package.json и package-lock.json
COPY package*.json /app

# Устанавливаю зависимости для серверной части
RUN npm install
```

Копирование файлов проекта:

Копируются все файлы проекта в рабочую директорию /app.

Копирую серверную часть
COPY . .

Переход в директорию клиента:

Переходит в директорию клиента /app/CLIENT для установки зависимостей клиентской части.

Перехожу в директорию клиента
WORKDIR /app/CLIENT

Клиентская часть:

Запускается команда npm install для установки зависимостей клиентской части.

Запускается команда npm run build для сборки клиентской части.

Устанавливаю зависимости для клиентской части
RUN npm install

Собираю клиентскую часть
RUN npm run build
Возвращение в корневую директорию:

Переходит обратно в корневую директорию /app.

Возвращаюсь в корневую директорию
WORKDIR /app

Открытие порта 1111:

Используется инструкция EXPOSE 1111, чтобы указать, что контейнер ожидает входящие соединения на порту 1111.

Открываю порт 1111 в контейнере, чтобы он был доступен извне
EXPOSE 1111

Удаление node_modules:

Добавлена инструкция RUN rm -rf /app/CLIENT/node_modules, предназначенная для удаления папки node_modules после сборки клиентской части, чтобы уменьшить размер образа.

Команда для запуска:

Используется команда **CMD ["npm", "run", "deploy"]** для запуска сервера и клиента. Эта команда запустит **npm run deploy**, включает в себя запуск сервера (npm run start) и сборку клиентской части

Команда для запуска сервера и клиента
CMD ["npm", "run", "deploy"]
Запустит NPM RUN START и NPM RUN CLIENT-BUILD
NPM RUN START запустит NODE и папку ./BIN/WWW
NPM RUN CLIENT-BUILD запустит из папки client NPM RUN BUILD а он запустит TSC && VITE BUILD

Инструкции по установке

Client

Перейдите в директорию Client.

Убедитесь, что у вас установлен Node.js и npm.

Запустите команду npm install, чтобы установить все необходимые зависимости.

Для запуска проекта используйте команду

npm run build.

Server

Перейдите в директорию Server.

Убедитесь, что у вас установлен Node.js и npm.

Запустите команду npm install, чтобы установить все необходимые зависимости.

Для запуска сервера используйте команду

npm run start.

Клиентская часть проекта

Описание

Страницы

Главная: Основная страница, представляющая общую информацию о проекте и его возможностях.

Портфолио: Страница, отображающая портфолио с проектами.

Магазин: Страница, позволяющая выбирать товары.

Коммунизм2.0: Страница с доской объявлений где каждый зарегистрированный юзер может предложить свои услуги в замен на другие услуги

ОргиБэнд: Страница сообщества где каждый зарегистрированный юзер может подать заявку на вступление в группу юзера из вне сообщества. Заявка находится на рассмотрении администратором после рассмотрения юзер переходит в категорию “проверенных юзеров” или анкета удаляется из очереди заявок.

Логин: Страница, имеет форму для аутентификации юзера.

Регистрация: Страница, где пользователь может зарегистрировать новый аккаунт.

Кабинет: Личный раздел юзера.

Стилизация

Для стилизации используется файловая структура в формате *.module.scss.

Используемые инструменты и библиотеки

Vite, **React**, **TypeScript**, **Router Dom 6**, **Axios**, **Redux Toolkit**, **famer-motion**, **sass**

Middleware для Axios запросов

Этот фрагмент кода содержит определение Axios-экземпляра с настройками для отправки HTTP-запросов к серверу. Также он включает использование middleware (промежуточного программного обеспечения) для добавления токена авторизации в заголовки запросов.

Настройка Axios:

axios.create() создает экземпляр Axios с базовым URL, указанным как http://localhost:1111/. Этот URL будет использоваться для всех запросов, созданных через этот экземпляр.

Middleware для запросов:

instance.interceptors.request.use() устанавливает middleware для всех исходящих запросов. Это позволяет добавить токен авторизации в заголовки каждого запроса перед его отправкой на сервер.

В данном случае, при каждом запросе, происходит проверка наличия токена в локальном хранилище window.localStorage.getItem('token'). Если токен существует, он добавляется в заголовок Authorization для авторизации пользователя на сервере.

Этот Axios-экземпляр позволяет облегчить отправку запросов к серверу, автоматически добавляя токен авторизации ко всем запросам, если он доступен в локальном хранилище.

Конфигурация Redux Store

Этот код относится к настройке хранилища Redux в приложении с использованием TypeScript. Он использует библиотеку Redux Toolkit для создания хранилища, которое содержит состояние приложения.

Настройка reducers:

authReducer, postsReducer, и cartReducer представляют собой reducer, отвечающие за изменение состояния хранилища для аутентификации, постов и корзины соответственно.

Функция загрузки состояния:

loadState() проверяет наличие данных в localStorage и, если они присутствуют, извлекает их для использования.

Применение предварительно загруженного состояния:

preloadedState используется для предварительной загрузки данных из localStorage в хранилище до его создания.

configureStore() создает Redux Store, объединяя reducers и применяя предварительно загруженное состояние для раздела корзины.

Если данные были загружены из localStorage, выполняется инициализация состояния корзины

Slice аутентификации Auth Slice

Описание

Этот код представляет собой часть Redux-логики, организующей управление состоянием аутентификации в приложении. Используются createSlice и createAsyncThunk из библиотеки Redux Toolkit для создания slice состояния, асинхронных thunk-функций для выполнения асинхронных запросов на сервер.

Функции Thunk для работы с сервером

fetchLogin: Выполняет запрос на сервер для аутентификации юзера по email и паролю.

fetchRegister: Выполняет запрос на сервер для регистрации нового юзера.

fetchAuth: Выполняет запрос на сервер для получения данных текущего аутентифицированного юзера.

Исходное состояние initialState

Объект initialState содержит начальное состояние для раздела аутентификации, где данные юзера data установлены в null, а статус status - в 'loading'.

Обработка результата запросов к серверу extraReducers

extraReducers определяет, как изменится состояние приложения после выполнения асинхронных операций:

fetchLogin: Устанавливает статус 'loading' при отправке запроса, 'loaded' при успешной аутентификации, 'error' в случае ошибки.

fetchRegister: Аналогично обрабатывает состояние при регистрации нового пользователя.

fetchAuth: Обновляет состояние при получении данных текущего юзера.

Селекторы для получения данных из состояния

selectIsAuth: Позволяет проверить, аутентифицирован ли юзер на основе данных в состоянии.

selectUserName: Получает имя юзера из данных состояния, если юзер аутентифицирован.

Reducer и экспорт

authReducer: представляет reducer, который обрабатывает action, созданные с помощью authSlice.actions. Кроме того, logOut - это action для выхода юзера из системы.

Slice для работы с постами Posts Slice

Описание

Этот код представляет собой часть Redux-логики, обеспечивающую управление состоянием постов в приложении. Используются createSlice и createAsyncThunk из библиотеки Redux Toolkit для создания slice состояния и асинхронных thunk-функций для выполнения асинхронных запросов на сервер и управления состоянием.

Thunk-функции для работы с сервером

fetchPosts: Запрос на сервер для получения списка постов.

fetchTags: Запрос на сервер для получения списка тегов.

fetchRemovePost: Запрос на сервер для удаления поста по его id.

Исходное состояние initialState

Объект initialState содержит начальное состояние для раздела постов и тегов, где:

posts содержит список постов и их статус загрузки.

tags содержит список тегов и их статус загрузки.

deletePostMessage и **deletePostData** представляют сообщение об удалении поста и данные удаленного поста

Обработка результатов запросов к серверу extraReducers

extraReducers определяет, как изменится состояние после выполнения асинхронных операций:

Для **fetchPosts** и **fetchTags** обрабатывается статус загрузки и полученные данные.

Для **fetchRemovePost** обрабатывается статус загрузки, сообщение об удалении и обновление списка постов после удаления.

Reducer and export

postsReducer представляет reducer, который обрабатывает action, созданные с помощью postsSlice.actions.

Slices для работы с корзиной Cart Slice

Описание

Этот код представляет собой часть Redux-логики, обеспечивающую управление состоянием корзины в приложении. Используется createSlice из библиотеки Redux Toolkit для создания slice состояния и определения reducers для изменения состояния корзины.

Исходное состояние initialState

Объект initialState содержит начальное состояние для корзины, где selectedItems представляет собой массив

объектов CartItem, представляющих выбранные элементы в корзине.

Reducers

addToCart: Добавляет элемент в корзину. Обновляет состояние корзины, добавляя переданный элемент в массив selectedItems. Также обновляет данные в localStorage, сохраняя текущее состояние корзины.

clearCart: Очищает содержимое корзины, устанавливая selectedItems в пустой массив. Удаляет данные о корзине из localStorage.

deleteItem: Удаляет выбранный элемент из корзины по его itemId. Обновляет состояние корзины, фильтруя элементы для удаления из selectedItems. Обновляет данные в localStorage после удаления элемента.

loadCartState: Загружает состояние корзины из переданных данных, устанавливая selectedItems в переданный массив CartItem.

Экспорт

Экспортируются созданные reducers (addToCart, clearCart, deleteItem, loadCartState) и сам reducer cartSlice.reducer.

API-документация

Middleware для **Auth** (аутентификации) в приложении, использующем Node.js Express.

Подключение библиотек:

подключаются библиотеки jsonwebtoken для работы с JSON Web Tokens и объект pool для взаимодействия с базой данных.

Middleware функция Auth:

Middleware функция принимает три параметра: **req** (объект запроса), **res** (объект ответа) и **next** (функция для передачи управления следующему в цепочке).

Извлечение токена из заголовков:

Извлекается токен из заголовков запроса. Заголовок "Authorization" должен содержать строку "Bearer" и сам токен.

Верификация токена:

Токен верифицируется с использованием секретного ключа JWT_SECRET. Если токен недействителен, генерируются соответствующие ошибки.

Получение пользователя из базы данных:

Пользователь извлекается из базы данных, объединяя результаты запросов к таблицам **users** и **admins** на основе userId, который был получен из декодированного токена.

Проверка наличия пользователя:

Если пользователь не найден, генерируется ошибка.

Проверка наличия администратора:

Дополнительно проверяется, является ли пользователь администратором, и устанавливается соответствующий флаг в объекте req.user.

Обработка ошибок:

Если происходит ошибка в процессе аутентификации, возвращается статус 401 с соответствующим сообщением об ошибке.

Экспорт middleware:

Middleware экспортируется для использования в других частях приложения.

API COMMUNITY (управления сообществом). Код на Node.js и Express включает функциональность добавления новых пользователей в сообщество, получения списка всех пользователей в сообществе и удаления пользователей из сообщества.

Добавление пользователя (AddUser):

const AddUser = async (req, res) => {

Функция принимает данные нового пользователя из тела запроса req.body. Затем извлекается идентификатор пользователя, создавшего нового пользователя createdByUserId из объекта запроса req.userId.

Затем выполняется SQL-запрос для получения информации о пользователе, создавшем нового пользователя. Если пользователь не найден, генерируется ошибка.

Далее создается SQL-запрос для вставки нового пользователя в базу данных. Вставленный пользователь возвращается в ответе, если операция выполнена успешно.

Получение всех пользователей (GetAllUsers):

const GetAllUsers = async (req, res) => {

Функция выполняет SQL-запрос для получения списка всех пользователей, включая информацию о пользователе, создавшем каждого из них.

Если запрос успешен и возвращает хотя бы одного пользователя, список пользователей возвращается в ответе. В противном случае возвращается ошибка "Пользователи не найдены".

Доступна для всех пользователей и администраторов.

Удаление пользователя (RemoveUser):

const RemoveUser = async (req, res) => {

Функция начинается с извлечения идентификатора пользователя из параметров запроса req.params.userId. Затем происходит проверка наличия идентификатора пользователя, и если его нет, возвращается ошибка.

Затем выполняется SQL-запрос для получения информации об удаляемом пользователе перед его удалением. После

этого выполняется SQL-запрос для удаления пользователя из базы данных.

Если удаление проходит успешно, возвращается успешный ответ вместе с информацией об удаленном пользователе. Доступна для **Администратор**.

Экспорт функций:

module.exports = { AddUser, GetAllUsers, RemoveUser };
Все три функции экспортируются для использования в других частях приложения. Экспортируются как объект, где ключи - это названия функций, а значения - сами функции.

Маршруты

/api/user

Этот маршрут предоставляет функционал, связанный с управлением пользователями. Все запросы отправляются на конечную точку /api/user и выполняют: вход юзера в систему, регистрация нового юзера и информация о текущем авторизованном юзере.

POST /api/user/login:

Описание: Этот запрос используется для аутентификации юзера в системе. Пользователь отправляет учетные данные и если данные верны, получает токен аутентификации.

Пример запроса:

POST /api/user/login

{

"username": "example_user",

"password": "example_password"

}

Пример ответа:

{

"token": "example_token"

}

POST /api/user/register:

Описание: Этот запрос используется для регистрации нового юзера. Пользователь отправляет данные для создания нового аккаунта.

Пример запроса:

POST /api/user/register

{

"username": "new_user",

"password": "new_password",

"email": "new_user@example.com"

}

Пример ответа:

{

"token": "example_token"

"message": "Пользователь успешно зарегистрирован"

}

GET /api/user/current:

Описание: Этот запрос используется для получения информации о текущем авторизованном юзере. Он требует предоставления токена аутентификации для получения данных о текущем пользователе.

Пример запроса:

GET /api/user/current

(С заголовком "Authorization: Bearer {token}")

Пример ответа:

{

"username": "current_user",

"email": "current_user@example.com"

"token": "example_token"

/api/posts

Этот маршрут предоставляет возможность управления постами в системе с учетом информации о пользователях, которые создали соответствующие посты.

GET /api/posts:

Описание: Этот запрос используется для получения списка всех постов в системе с добавленной информацией о пользователях, которые создали эти посты.

Пример запроса:

GET /api/posts

Пример ответа:

[

{

"id": 1,

"title": "Заголовок поста 1",

"content": "Содержание поста 1",

"user_name": "Имя пользователя",

"user_email": "example@example.com",

},

{

"id": 2,

"title": "Заголовок поста 2",

"content": "Содержание поста 2",

"user_name": "Имя другого пользователя",

"user_email": "another@example.com"

},

]

GET /api/posts/:id:

Описание: Этот запрос используется для получения информации о конкретном посте по его идентификатору с добавленной информацией о пользователе, создавшем этот пост.

Пример запроса:

GET /api/posts/1

Пример ответа:

{

"id": 1,

"title": "Заголовок поста 1",

"content": "Содержание поста 1",

"user_name": "Имя пользователя",

"user_email": "example@example.com",

}

POST /api/posts:

Описание: Этот запрос используется для создания нового поста в системе. Он также автоматически добавляет информацию о пользователе, создавшем пост.

Пример запроса:

POST /api/posts

{

"title": "Новый пост",

"text": "Содержание нового поста",

"tags": ["tag1", "tag2"]

}

Пример ответа:

{

"message": "Пост успешно создан",

"post": {

"id": 3,

"title": "Новый пост",

"content": "Содержание нового поста",

"user_name": "Имя пользователя",

"user_email": "example@example.com",

}

}

DELETE /api/posts/:id:

Описание: Этот запрос используется для удаления существующего поста по его идентификатору.

Пример запроса:

DELETE /api/posts/1

Пример ответа:

{

"message": "Пост успешно удален",

"deletedPost": {

"id": 1,

"title": "Заголовок поста 1",

"content": "Содержание поста 1",

"user_name": "Имя пользователя",

"user_email": "example@example.com"

}

}

PATCH /api/posts/:id:

Описание: Этот запрос используется для обновления информации о существующем посте по его идентификатору.

Пример запроса:

PATCH /api/posts/1

{

"title": "Новый заголовок для поста",

"text": "Новое содержание для поста",

"tags": ["new_tag1", "new_tag2"]

}

Пример ответа:

{

"message": "Пост успешно обновлен",

"updatedPost": {

"id": 1,

"title": "Новый заголовок для поста",

"content": "Новое содержание для поста",

"user_name": "Имя пользователя",

"user_email": "example@example.com"

}

}

/api/cart

Этот маршрут предоставляет возможность управления корзиной пользователя. Запросы отправляются на конечную точку /api/cart и выполняют операции, связанные с созданием заказа, включая добавление выбранных товаров в корзину.

POST /api/cart/makeOrder:

Описание: Этот запрос используется для создания заказа, включая выбранные пользователем товары.

Этот маршрут предназначен для создания заказа с учетом выбранных пользователем товаров и сохранения информации о заказе в базе данных. Пример запроса и ответа показывает формат данных, отправляемых при создании заказа и информацию, возвращаемую после успешного создания заказа.

Пример запроса:

POST /api/cart/makeOrder

{

"selectedItems": [

{

"itemId": 1,

"itemName": "Название товара",

"quantity": 2

},

{

"itemId": 2,

"itemName": "Другой товар",

"quantity": 1

}

]

}

Пример ответа:

{

"message": "Заказ успешно создан",

"order": {

"order_id": 1,

"user_id": 123,

"items": [

{

"itemId": 1,

"itemName": "Название товара",

"quantity": 2

},

{

"itemId": 2,

"itemName": "Другой товар",

"quantity": 1
}
]
}
}