Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Компьютерные системы и сети (КСиС)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту на тему

Клиент-серверное приложение с обменом информации по сети

БГУИР КП I–40 01 01 126 ПЗ

Выполнил

студент гр. 351001 Шиманчук А.О.

Проверил: Шамына. А.Ю.

Минск 2025

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ

Заведующий кафедрой ПОИТ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

Лапицкая Н.В. 2024г.

ЗАДАНИЕ

по курсовому проектированию

Студенту Шиманчуку Артему Олеговичу \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Тема работы клиент-серверное приложение с обменом информации по сети «Porsche Website»­ ­­­­­ ­­

2. Срок сдачи законченной работы *07.06.2024г.*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. Исходные данные к работе: Программа, реализующая функционал клиент-серверного приложения для создания полнофункционального веб-сайта Porsche. Включает разработку серверной части для обработки пользовательских запросов и хранения данных, а также клиентской части для предоставления интерактивного интерфейса. Использование современных веб-технологий (JavaScript, TypeScript, Node.js, Express, MySQL) для обеспечения высокой производительности и масштабируемости. Поддержка обмена данными по сети с использованием REST API. Внедрение методов безопасности для защиты данных пользователей.

4. Содержание расчетно-пояснительной записки (перечень вопросов, которые подлежат разработке)

*Введение*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*1 Описание используемых технологий\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*2*  Описание сервера \_\_\_\_\_\_*\_\_\_\_\_\_\_\_\_\_\_*

*3* Описание клиента *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*4* Интерфейс для взаимодействия*\_\_\_\_\_\_\_\_\_\_\_\_\_*

*5* [Тестирование](#_teqvt4kfphgq)*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*6 Схемы алгоритмов решения задач по ГОСТ 19.701-90*

*Заключение\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Список использованных источников\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Приложения* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5. Перечень графического материала (с точным обозначением обязательных чертежей и графиков)

*Схема алгоритма в формате А1*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6. Консультант по курсовой работе *Шамына А.Ю.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

7.Дата выдачи задания *05.03.2025г.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

8. Календарный график работы над проектом на весь период проектирования (с обозначением сроков выполнения и процентом от общего обьема работы):

*Раздел 1, Введение к 15.03.2025г. – 10 % готовности работы;\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Раздел 2 к 30.03.2025г. – 30% готовности работы\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Раздел 3 к 15.04.2025г. – 60% готовности работы\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Раздел 4, Заключение, Приложения к 10.05.2025г. – 90% готовности работы;*

*оформление пояснительной записки и графического материала к 15.05.2025г. – 100% готовности работы.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Защита курсового проекта с 15.05.2025г. по 07.06.2025г.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

РУКОВОДИТЕЛЬ *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Шамына А.Ю.*

*(подпись)*

Задание принял к исполнению *\_\_\_\_\_\_\_\_\_\_Шиманчук А.О. 05.03.2025г.*

*(дата и подпись студента)*

**СОДЕРЖАНИЕ**

Введение[………………………………………………………………………………5](#_30j0zll)

[1 Описание используемых технологий](#_soxq4vgwhbtw)……………………………………………..6

[1.1 Архитектурный стиль REST API……………………………………………7](#_lm3k2l3u6s1n)

[2 Описание сервера для приложения](#_ti7fnyi5n2mv)……………………………………………….8

[3 Описание клиента для приложения](#_jutpb9cwz41g) ……………………………………………..21

[3.1 Техническая часть клиента 22](#_xi5rkj9bac69)

[4 Интерфейс взаимодействия 27](#_kndrlc8y2nds)

[5 Тестирование 29](#_teqvt4kfphgq)

[5.1 Функиональное тестирование 31](#_k1h5xsikclin)

[5.2 Тестирование API 33](#_bgadom5razot)

[6 Схемы алгоритмов решения задач по ГОСТ 19.701-90 34](#_albngvmttuzm)

Заключение [37](#_mzwcwzom5w2q)

Список использованных источников [38](#_x6s423jtgyv3)

Приложение [А…………………………………………………………………..………39](#_1e6ee775c1p2)

## ВВЕДЕНИЕ

Современные веб-технологии развиваются с невероятной скоростью, изменяя подходы к созданию сайтов и взаимодействию пользователей с онлайн-ресурсами. В этом контексте клиент-серверные приложения играют ключевую роль, обеспечивая гибкость, скорость и масштабируемость современных веб-систем. Одной из самых популярных моделей является архитектура full stack, которая объединяет как серверную, так и клиентскую части приложения, создавая комплексные и функциональные решения.

Одним из примеров такого подхода является создание сайта для культового автомобильного бренда Porsche, сочетающего элегантность дизайна, высокую производительность и богатый функционал. Этот проект направлен на разработку полнофункционального клиент-серверного приложения, которое объединяет современные технологии, такие как JavaScript, Node.js, Express, TypeScript и MySQL.

Основной целью проекта является разработка интерактивного и динамического веб-приложения, способного обрабатывать большое количество запросов, поддерживать высокую скорость загрузки страниц и обеспечивать пользователей интуитивно понятным интерфейсом. Также важным аспектом является безопасность данных, что особенно важно для работы с персональной информацией клиентов.

Для разработки интерфейсов используется Figma, позволяющая создавать детализированные макеты и прототипы, а в качестве контентной основы используется официальный сайт Porsche, что позволяет сохранить аутентичность стиля и брендовые особенности.

## 1 ОПИСАНИЕ ИСПОЛЬЗУЕМЫХ ТЕХНОЛОГИЙ

В данном проекте для обмена данными между клиентской и серверной частями использован REST API. Этот подход был выбран благодаря своей простоте, надежности и масштабируемости, что делает его оптимальным для большинства веб-приложений, особенно при работе с базами данных и выполнении стандартных CRUD-операций.

Стоит отметить, что существует и другой стиль обмена данными — WebSockets, который обеспечивает постоянное двунаправленное соединение и лучше подходит для приложений с интерактивным обновлением данных в реальном времени, таких как чаты или онлайн-игры. Однако для целей данного проекта REST API обеспечивает необходимую функциональность и удобство реализации.

|  |  |  |
| --- | --- | --- |
| Характеристика | REST API | WebSockets |
| Принцип работы | Клиент отправляет запрос, сервер отвечает | Постоянное соединение между клиентом и сервером |
| Протокол | HTTP/HTTPS | WebSockets |
| Модель взаимодействия | Запрос-ответ (stateless) | Полный дуплекс (stateful) |
| Поддержка масштабирования | Легче масштабировать за счет статической природы | Сложнее масштабировать из-за постоянных соединений |
| Задержка данных | Высокая, так как каждое сообщение требует установления соединения | Низкая, соединение поддерживается постоянно |
| Безопасность | Традиционные методы HTTP, поддержка TLS | Поддержка TLS, но сложнее в реализации авторизации |
| Использование ресурсов | Менее требовательно к ресурсам сервера | Более требовательно из-за постоянных соединений |
| Применение | CRUD операции, RESTful сервисы, API для веб-приложений | Реалтайм-чаты, онлайн-игры, стриминг данных |

* 1. **Архитектурный стиль REST API**

REST API (Representational State Transfer Application Programming Interface) — это архитектурный стиль взаимодействия между клиентом и сервером, широко используемый для разработки веб-сервисов. REST основывается на стандартах и протоколах HTTP и предоставляет четкие принципы для создания распределенных систем с масштабируемой архитектурой.

Основной принцип REST — разделение клиента и сервера, при котором клиент отправляет запросы, а сервер отвечает с необходимыми данными. Взаимодействие происходит по модели «запрос-ответ», при этом сервер не хранит состояние клиента, что обеспечивает статeless архитектуру. Это значительно упрощает масштабирование и повышает отказоустойчивость системы.

REST API использует стандартные методы HTTP:

- GET — получение данных,

- POST — создание новых ресурсов,

- PUT — обновление существующих ресурсов,

- DELETE — удаление ресурсов.

Формат обмена данными чаще всего представлен в виде JSON, что обеспечивает простоту чтения и обработки как человеком, так и программой.

Преимущества REST API включают в себя:

Простоту и стандартизацию. Использование протокола HTTP и понятных методов делает REST легким для изучения и внедрения.

Масштабируемость. Отсутствие сохранения состояния на сервере облегчает горизонтальное масштабирование приложения при увеличении числа пользователей.

Кэширование. HTTP-протокол поддерживает кэширование, что уменьшает нагрузку на сервер и ускоряет обработку повторяющихся запросов.

Гибкость. REST API может использоваться в различных типах приложений — от мобильных до веб-сервисов и IoT-устройств.

Вместе с тем, REST API имеет ограничения — из-за модели запрос-ответ невозможна передача данных в реальном времени без постоянного опроса сервера (polling), что может создавать задержки и нагрузку. Для приложений с необходимостью мгновенного обмена данными применяют альтернативные решения, такие как WebSockets.

В рамках данного проекта REST API был выбран за его простоту, широкую поддержку и удобство интеграции с базой данных MySQL, а также за возможность легко организовать взаимодействие клиентской части на JavaScript с сервером на Node.js с использованием фреймворка Express.

**2 ОПИСАНИЕ СЕРВЕРА ПРИЛОЖЕНИЯ**

**2.1 Описание endpoin’ов сервера**

Серверная часть приложения разработана с использованием Node.js, TypeScript, фреймворка Express.js, что обеспечивает высокую производительность, гибкость и простоту настройки маршрутов. Для хранения данных используется MySQL, что позволяет эффективно работать с реляционными структурами данных и выполнять сложные SQL-запросы.

Основная задача серверной части — обработка запросов, поступающих от клиентской части приложения, и взаимодействие с базой данных для получения, добавления, изменения и удаления данных. Взаимодействие организовано через REST API с использованием следующих endpoint'ов:

|  |  |  |  |
| --- | --- | --- | --- |
| Endpoint | Запрос | Ответ | Описание |
| POST  http://localhost:3000  /API/register | Перадача пользовательских данных для регистрации | Сообщение об успешной регистрации или ошибке | Регистрирует пользователя в Базе Данных |
| POST  http://localhost:3000  /API/login | Перадача пользовательских данных для авторизации | При успешной авторизации - выдача JWT или сообщение об ошибке | Авторизирует пользователя в системе |
| GET  http://localhost:3000  /API/logout | Передача JWT токена | Сообщение об успешном выходе | Выход пользователя из системы |
| GET  http://localhost:3000  /API/refresh | Передача refresh  JWT | При успешном refresh возвращение нового access токена | Обновление токена |
| GET  http://localhost:3000  /API/verifyJwt | Перадача access JWT | Сообщение о валидности токена | Верификация токена |
| GET  http://localhost:3000  /API/carModels/  getAllModels | Передача access JWT | Возвращение всех моделей автомобилей | Получение моделей авто |
| GET  http://localhost:3000  /API/ carReview | Передача конкретной модели авто | Возвращение модели авто | Получение конкретной модели авто |
| POST  http://localhost:3000  /API  /usersBasket/addConfig | Передача конфигурации авто для добавления | При успешном добавлении возварщение сообщения об успешном добавлении в корзину | Добавление в БД  Конфигурации пользователя |
| GET  http://localhost:3000  /API  /usersBasket/ getUserConfig | Передача access JWT токена | Возвращение для конкретного пользователя всех его конфигураций | Поиск и возвращени из БД всех конфигураций |
| DELETE  http://localhost:3000  /API  /usersBasket/ deleteUserConfig | Передача id конфигурации и id пользователя для удаления конфигурации | Возвращение сообщения об успешном удалении из корзины | Удаление конфигурации из БД |

**2.2 Описание архитектуры и технической части сервера**

В данном курсовом проекте была выбрана архитектура VMC (View-Model-Controller), которая является одним из популярных подходов к построению веб-приложений. Эта архитектура помогает разделить приложение на логические модули, что упрощает разработку, тестирование и поддержку кода. Архитектура VMC похожа на классическую MVC, но обладает большей гибкостью и лучше подходит для современных JavaScript-фреймворков, таких как Express.js на стороне сервера.

Файл сервера:

import express from "express";

import { usersRouter } from "./routers/usersRoutes"

import { testRouter } from "./routers/testGelAllUsersRoute";

import { refreshTokenRouter } from "./routers/refreshJwtRoute"

import { carModelsRouter } from "./routers/carModelRoute"

import { carReviewRouter } from "./routers/carOverviewRoute"

import { verifyTokenRouter } from "./routers/verifyJwtRoute";

import { usersBasketRouter } from "./routers/usersBasketRoute"

//------------------------------------------------------------//

import cookieParser from "cookie-parser";

import { corsOptions } from "./config/corsOptions";

import cors from "cors"

//------------------------------------------------------------//

const server = express();

const PORT: number = +(process.env.PORT || 3000);

server.use(cors(corsOptions));

server.use(express.json())

server.use(express.urlencoded({ extended: true }))

server.use(cookieParser())

server.use('/API/auth', usersRouter)

// Маршруты для получения моделей и конкретной модели

server.use('/API/carModels', carModelsRouter)

server.use('/API/carReview', carReviewRouter)

// Маршруты для токенов

server.use('/API/refreshJwt', refreshTokenRouter)

server.use('/API/verifyJwt', verifyTokenRouter)

// Маршруты для корзины

server.use('/API/userBasket', usersBasketRouter)

server.listen(PORT, () => {

console.log(`server listening on port ${PORT}`);

});

Основные компоненты VMC:

View (Представление)

Представление отвечает за отображение данных пользователю и обработку взаимодействий с ним. В данном проекте используется чистый HTML, CSS и JavaScript для создания интерфейса. Именно на уровне View происходит получение данных от сервера через API и их отображение пользователю.

Пример использования представления в этом проекте — страницы с каталогом автомобилей Porsche, которые получают данные о доступных моделях через REST API и динамически отображают их в виде карточек.

Model (Модель)

Модель отвечает за бизнес-логику приложения и взаимодействие с базой данных. В этом проекте данные хранятся в базе данных MySQL, а модели реализованы на Node.js, что позволяет напрямую работать с SQL-запросами для более точного контроля над структурой данных.

Основные задачи модели:

Чтение данных из базы, Сохранение новых записей, Обновление существующих данных, Удаление записей.

Модель также отвечает за проверку данных перед сохранением в базе и обработку ошибок, связанных с доступом к базе данных. Например, модель для работы с таблицей автомобилей может включать методы для получения списка всех автомобилей, поиска автомобиля по ID, добавления нового автомобиля и удаления существующего.

Controller (Контроллер)

Контроллеры играют роль посредников между View и Model. Они получают запросы от клиента (обычно в формате JSON), передают их в модель для обработки, а затем возвращают результат обратно клиенту. Контроллеры отвечают за маршрутизацию запросов, проверку входных данных и обработку ошибок.

Например, если пользователь отправляет запрос на получение списка всех автомобилей контроллер вызывает соответствующий метод модели для получения данных из базы и формирует ответ в формате JSON, который затем отправляется обратно клиенту.

Как взаимодействуют компоненты VMC: Пользователь открывает веб-страницу (View), которая отправляет HTTP-запрос к серверу для получения данных. Контроллер получает запрос, проверяет его корректность и передает управление модели. Модель взаимодействует с базой данных, выполняя необходимый SQL-запрос для получения или изменения данных. После обработки данных модель возвращает результат контроллеру. Контроллер отправляет сформированный ответ обратно клиенту, который затем отображает данные пользователю.

Классы моделей:

export class CarModelsModel {

static async getCarModels(model: any) {

const connection = await createConnToDataBase();

try{

const [rows] = await connection.execute(

`SELECT \* FROM models

WHERE model\_series = ?`, [`${model}`]

)

return rows;

}

catch(err){

console.log(err);

throw err;

}

finally {

await connection.end();

}

}

static async getAllCarModels() {

const connection = await createConnToDataBase();

try{

const [rows] = await connection.execute(

`SELECT \* FROM models`

)

return rows;

}

catch(err){

console.log(err);

throw err;

}

finally {

await connection.end();

}

}

}

export class CarReviewModel {

static async getCar(model\_name: any) {

const connection = await createConnToDataBase();

try{

const [rows] = await connection.execute(

`SELECT models.model\_name, models.model\_series, models\_overview.base\_price, models\_overview.preview\_images, models.id

FROM models\_overview

INNER JOIN porsche\_database.models on models\_overview.model\_id = models.id

WHERE models.model\_name = ?`,

[model\_name]

)

return rows;

}

catch(err){

console.log(err);

throw err;

}

finally {

await connection.end();

}

}

}

export class TokenModel {

static async updateRefreshToken(token: string, id: BigInt): Promise<void> {

const connection = await createConnToDataBase();

try{

await connection.execute(`

UPDATE users

SET refresh\_token = ?

WHERE id = ?;

`,[token, id]);

}

catch(err){

console.log(err);

throw err;

}

}

static async getUserByToken(token: string): Promise<RowDataPacket[]>{

const connection = await createConnToDataBase();

try{

const [user] = await connection.execute<RowDataPacket[]>(

`SELECT \* FROM users WHERE refresh\_token = ?`, [token]);

return user;

}

catch(err){

console.log(err);

throw err;

}

}

static async logOutByToken(id: string): Promise<void> {

const connection = await createConnToDataBase();

try{

await connection.execute(`

UPDATE users

SET refresh\_token = ?

WHERE id = ?

`, [null, id]);

}

catch(err){

console.log(err);

throw err;

}

}

}

static async getConfigByHash(configHash: any, userID: number){

const connection = await createConnToDataBase()

try{

const [rows] = await connection.execute<RowDataPacket[]>(`

SELECT \* FROM users\_basket

WHERE user\_id = ? AND config\_hash = ?`,

[userID, configHash]

)

return rows;

}

catch(err){

console.log(err);

throw err;

}

finally {

await connection.end();

}

}

static async insertConfig(config: any){

const connection = await createConnToDataBase()

try{

await connection.execute(`

INSERT INTO users\_basket (user\_id, model\_id, config\_hash,

exterior\_color, exterior\_color\_img,

interior\_color, interior\_color\_img,

wheels, wheels\_img,

package\_title, package\_description,

exhaust\_title, exhaust\_description,

model\_price)

VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)`,

[

config.user\_id, config.model\_id, config.hashConfig,

config.exterior\_color, config.exterior\_color\_img,

config.interior\_color, config.interior\_color\_img,

config.wheels, config.wheels\_img,

config.package\_title, config.package\_description,

config.exhaust\_title, config.exhaust\_description,

config.model\_price

]

);

}

catch(err){

console.log(err);

throw err;

}

finally {

await connection.end();

}

}

static async getConfig(user\_id: number){

const connection = await createConnToDataBase()

try{

const [rows] = await connection.execute<RowDataPacket[]>(`

SELECT users\_basket.\*,

models.fuel\_type, models.model\_name, models.model\_series,

models.model\_acceleration, models.power\_kw\_ps, models.top\_speed, models.overview\_image,

models\_overview.base\_price

FROM users\_basket

JOIN models ON users\_basket.model\_id = models.id

JOIN models\_overview ON models.id = models\_overview.model\_id

WHERE user\_id = ?`, [user\_id]

);

return rows;

}

catch(err){

console.log(err);

throw err;

}

finally {

await connection.end();

}

}

static async deleteConfigModel(config\_id: number, user\_id: number){

const connection = await createConnToDataBase()

try{

await connection.execute(`

DELETE FROM users\_basket

WHERE id = ? AND user\_id = ?`, [config\_id, user\_id]

);

}

catch(err){

console.log(err);

throw err;

}

finally {

await connection.end();

}

}

}

export class UsersModel {

static async getUserByEmail(email: string): Promise<RowDataPacket[]>{

let connection = await createConnToDataBase();

try{

const [rows] = await connection.execute<RowDataPacket[]>(`

SELECT id, user\_name, user\_surname, user\_email, user\_password FROM users WHERE user\_email = ?

`, [email])

return rows;

}

catch(err: any){

console.error(err);

throw err;

}

finally{

await connection.end()

}

}

static async createUserModel(user: userInterface): Promise<ResultSetHeader> {

const connection = await createConnToDataBase();

try{

const [rows] = await connection.execute<ResultSetHeader>(`

INSERT INTO users (user\_name, user\_surname, user\_email, user\_password)

VALUES (?, ?, ?, ?)

`, [user.userName, user.userSurname,user.userEmail,user.userPassword]);

return rows;

}

catch(err: any){

console.error(err);

return err;

}

finally{

await connection.end();

}

}

static async getAllUsersModel(): Promise<RowDataPacket[]>{

const connection = await createConnToDataBase();

try {

const [rows] = await connection.execute<RowDataPacket[]>(`SELECT \* FROM users`)

return rows;

}

catch(err: any){

throw err;

}

finally{

await connection.end();

}

}

}

Классы контроллеров:

export class CarModelsController {

static async getCarModels(req: Request, res: Response) {

try {

const model: any = req.query.model;

const carModels = await CarModelsModel.getCarModels(model)

res.status(200).json(carModels)

}

catch(err) {

res.status(500).json(err)

}

}

static async getAllModels(req: Request, res: Response) {

try {

const carModels = await CarModelsModel.getAllCarModels()

res.status(200).json(carModels)

}

catch(err) {

res.status(500).json(err)

}

}

}

export class CarReviewController {

static async getCar(req: Request, res: Response){

try {

const model\_name: any = req.query.model\_name;

const modelData = await CarReviewModel.getCar(model\_name);

res.status(200).json(modelData);

}

catch(err){

res.status(500).json(err)

}

}

}

xport class TokenController {

static async handleRefreshToken(req: Request, res: Response) {

// Проверка наличия refresh токена в куках

// Если пользователь не зарегистрирован, то этот код проверяет это

const cookies = req.cookies;

if (!cookies?.jwt) {

res.sendStatus(401);

return;

}

const refreshToken: string = cookies.jwt;

// Проверка соотв. полученного токена с токеном пользователя

// Если кто-то пытается подменить токен, то этот код проверяет его действительность

const users: RowDataPacket[] = await TokenModel.getUserByToken(refreshToken);

if (users.length === 0) {

res.sendStatus(403)

return;

}

const foundUser: RowDataPacket = users[0];

jwt.verify(

refreshToken,

process.env.REFRESH\_TOKEN\_SECRET!,

(err, dec) => {

const decoded: JwtPayload = dec as JwtPayload;

if (err || foundUser.user\_name !== decoded.userName) {

res.sendStatus(403); // Forbidden

return;

}

const accessToken: string = jwt.sign(

{userName: decoded.userName, user\_id: decoded.user\_id},

process.env.ACCESS\_TOKEN\_SECRET!,

{ expiresIn: "60s" }

)

res.status(200).json({

accessToken,

"message": "Токен успешно обновлен"

})

}

)

}

static async handleLogOut(req: Request, res: Response){

// on client, also delete the accessToken

const cookies = req.cookies;

console.log(cookies);

if (!cookies?.jwt){

res.status(401).json({ message: 'Unauthorized: No token provided.' });

return;

}

const refreshToken = cookies.jwt;

const users: RowDataPacket[] = await TokenModel.getUserByToken(refreshToken);

if (users.length === 0) {

res.clearCookie('jwt', { httpOnly: true, secure: false });

res.status(404).json({ message: 'Token not found.' });

return;

}

const foundUser: RowDataPacket = users[0];

// Delete refreshToken from DataBase

await TokenModel.logOutByToken(foundUser.id);

res.clearCookie('jwt', { httpOnly: true, secure: false });

res.status(200).json({ message: 'Successfully logged out.'});

}

}

export class usersBasketController{

static async addConfig(req: Request, res: Response){

try {

// Получение id пользователя через JWT токен

const token: string = req.headers.authorization!.split(" ")[1];

const decoded: any = jwt.verify(token, process.env.ACCESS\_TOKEN\_SECRET!);

const user\_id: number = decoded.user\_id;

console.log("decoded: ", decoded);

// Получение тела запроса

const { model\_id, config } = req.body;

// Получение хеша конфигурации

const hashConfig: string = getConfigHash(config);

// Проверка на существование у пользователя такой конфигурации

const usersConfigs: RowDataPacket[] = await usersBasketModel.getConfigByHash(hashConfig, user\_id);

// Если существует такая конфигурация

if (usersConfigs.length !== 0){

// Конфликт с текущим состоянием ресурса

res.status(409).json({

message: `Configuration already exists in account`,

})

return;

}

// Добавление конфигурации в БД

const userConfigInfo: usersBasketRowInterface = {

user\_id,

model\_id,

hashConfig,

...config

}

await usersBasketModel.insertConfig(userConfigInfo);

// Если запись успешно добавлена

res.status(201).json({

message: `Configuration successfully saved in your collection`,

})

return;

}

catch (err: any){

res

.status(500)

.json({

"statusCode": 500,

"message" : `Server Error ${err.message}`,

})

return;

}

}

static async getUserConfig(req: Request, res: Response){

try{

const token: string = req.headers.authorization!.split(" ")[1];

const decoded: any = jwt.verify(token, process.env.ACCESS\_TOKEN\_SECRET!);

const user\_id: number = decoded.user\_id;

const userConfig: any = await usersBasketModel.getConfig(user\_id)

res.status(200).json({

message: `Basket successfully delivered`,

userConfig

})

}

catch(err: any){

res

.status(500)

.json({

"statusCode": 500,

"message" : `Server Error ${err.message}`,

})

return;

}

}

static async deleteConfig(req: Request, res: Response){

try{

// Получение id пользователя через JWT токен

const token: string = req.headers.authorization!.split(" ")[1];

const decoded: any = jwt.verify(token, process.env.ACCESS\_TOKEN\_SECRET!);

const user\_id: number = decoded.user\_id;

// Получение id конфигурации из query параметры

const config\_id: number = parseInt(<string>req.query.id);

// Удаление конфигурации пользователя

await usersBasketModel.deleteConfigModel(config\_id, user\_id);

res.status(200).json({

message: `Configuration successfully deleted`,

})

}

catch(err: any){

res

.status(500)

.json({

"statusCode": 500,

"message" : `Server Error ${err.message}`,

})

return;

}

}

}

export class AuthenticateUser {

static async createUser(req: Request, res: Response): Promise<void> {

try{

const errors = validationResult(req);

if (!errors.isEmpty()) {

res.status(400).json({

errors: errors.array()

})

}

const { userName, userSurname, userEmail, userPassword } = req.body;

const candidate: RowDataPacket[] = await UsersModel.getUserByEmail(userEmail);

if (candidate.length != 0){

res.status(409).json({

"statusCode": 409,

"message": "User with this email already exists",

})

}

const userData: userInterface = {

userName,

userSurname,

userEmail,

userPassword: await hashPassword(userPassword)

}

const rows: ResultSetHeader = await UsersModel.createUserModel(userData);

res.status(201).json({

"userID": rows.insertId,

"message": "User successfully created",

})

}

catch(err: any){

res.status(500).json({

"statusCode": 500,

"message" : `Server Error ${err.message}`,

})

}

}

static async loginUser(req: Request, res: Response): Promise<void> {

try {

// checking fields are filled in

const errors = validationResult(req);

if (!errors.isEmpty()) {

res.status(400).json({

errors: errors.array()

})

return;

}

// Getting data from the request body

const { userEmail, userPassword } = req.body;

// Getting user by email

const candidate: RowDataPacket[] = await UsersModel.getUserByEmail(userEmail);

// checking for user existence

if (candidate.length === 0){

res.status(401).json({"message": "Incorrect email or password"})

return;

}

if(!await bcrypt.compare(userPassword, candidate[0].user\_password)){

res.status(401).json({ "message": "Incorrect email or password"});

return;

}

const user: RowDataPacket = candidate[0];

console.log(user.id);

// return a JWT token and refresh JWT token

const accessToken: string = jwt.sign(

{userName: user.user\_name, user\_id: user.id},

process.env.ACCESS\_TOKEN\_SECRET!,

{expiresIn: "60s"}

)

const refreshToken: string = jwt.sign(

{userName: user.user\_name, user\_id: user.id},

process.env.REFRESH\_TOKEN\_SECRET!,

{expiresIn: "7d"}

)

console.log('saving jwt')

// saving in cookies JWT refresh

res.cookie('jwt', refreshToken, {

httpOnly: true,

secure: false,

maxAge: 7 \* 24 \* 60 \* 60 \* 1000

});

console.log('jwt saved in cookies')

await TokenModel.updateRefreshToken(refreshToken, user.id);

res.status(200).json({

userID: user.id,

accessToken,

message: `Authorization successful\n Welcome ${user.user\_name}`,

})

return;

}

catch(err:any){

res

.status(500)

.json({

"statusCode": 500,

"message" : `Server Error ${err.message}`,

})

return;

}

}

static async getAllUsers(req: Request, res: Response): Promise<void> {

const users: RowDataPacket[] = await UsersModel.getAllUsersModel()

res.json(users)

}

}

**3 ОПИСАНИЕ КЛИЕНТА ДЛЯ ПРИЛОЖЕНИЯ**

**3.1** **Техническая часть клиента**

Клиентская часть веб-приложения, разработанного в рамках курсового проекта, представляет собой набор HTML-страниц, стилизованных с помощью CSS и использующих JavaScript для динамического взаимодействия с сервером через REST API. Основная цель клиентской части — обеспечить удобный, быстрый и безопасный доступ к функционалу приложения, включая просмотр моделей автомобилей, регистрацию, авторизацию, управление корзиной и конфигурирование автомобилей.

Основные компоненты клиентской части:

1. HTML и структура страниц:
   * auth.html — страница аутентификации.
   * reg.html — страница регистрации.
   * landing.html — главная страница с общим описанием моделей.
   * modelOverview.html — страница с каталогом всех моделей.
   * modelReview.html — страница с детальной информацией о конкретной модели.
   * basket.html — страница с корзиной пользователя, где можно управлять конфигурациями.
2. CSS для стилизации:
   * Используются адаптивные стили для корректного отображения приложения на различных устройствах.
   * Применены flexbox и grid для построения сложных интерфейсов.
   * Акцент на минималистичном и современном дизайне, соответствующем стилю бренда Porsche.
3. JavaScript для динамики:
   * Используется чистый JavaScript для обработки событий, валидации форм и взаимодействия с сервером.
   * Основные функции включают отправку запросов к REST API, обработку ответов и динамическое обновление интерфейса.

Основные модули и функции JavaScript:

1. Аутентификация и регистрация:
   * auth.js — управление авторизацией пользователя.
   * Отправка данных формы на /API/auth/login методом POST.
   * Получение и сохранение JWT токена в cookies для последующих запросов.
   * Обработка ошибок (например, неверный пароль или отсутствие пользователя в базе).
2. Работа с токенами:
   * Сохранение и использование access и refresh токенов для обеспечения безопасности.
   * Обновление токена при его истечении с помощью /API/refresh.
   * Автоматическое перенаправление на страницу авторизации при истечении токена.
3. Загрузка и отображение данных:
   * Асинхронные запросы для получения списка моделей с сервера.
   * Использование методов GET для получения данных с эндпоинта /API/carModels/getAllModels.
   * Динамическое создание элементов интерфейса на основе полученных данных.
4. Управление корзиной:
   * Добавление и удаление конфигураций автомобилей.
   * Хранение конфигураций на сервере и их отображение в корзине пользователя.
   * Защита данных пользователя с использованием JWT.
5. Обработка ошибок и уведомления:
   * Реализованы методы для обработки ошибок сервера и уведомления пользователя о статусе операций.
   * Используются модальные окна и всплывающие уведомления для более комфортного взаимодействия с пользователем.
6. Проверка авторизации:
   * Проверка валидности токена при каждом запросе.
   * Автоматическое перенаправление на страницу авторизации при обнаружении просроченного токена.
7. Модульность и переиспользование кода:
   * Код разделён на отдельные модули для улучшения читаемости и поддерживаемости.
8. Асинхронное программирование:
   * Активное использование async/await для обработки запросов и повышения отзывчивости приложения.

const authModule = () => {

const formElement = document.querySelector('.login\_\_form');

const popUpMessage = new PopUpMessage();

formElement.addEventListener('submit', async (e) => {

e.preventDefault();

const userEmail = formElement.querySelector('.input\_\_email').value;

const userPassword = formElement.querySelector('.input\_\_password').value;

if (userEmail === "" || userPassword === "") {

popUpMessage.show(

"Authorization failed",

"All fields are required!"

)

return;

}

if(userPassword.length < 8) {

popUpMessage.show(

"Authorization failed",

"Password must be at least 8 characters long!"

)

return;

}

const emailRegex = /^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;

if (!emailRegex.test(userEmail)) {

popUpMessage.show(

"Authorization failed",

"Please, enter a valid email address (example@domain.com)"

)

return;

}

try{

const response = await fetch('http://localhost:3000/API/auth/login', {

method: 'POST',

credentials: 'include',

headers: {

"Content-Type": "application/json"

},

body: JSON.stringify({

userEmail,

userPassword

})

})

if (response.ok){

const userData = await response.json();

console.log(userData);

popUpMessage.show(

"Authorization successful",

`${userData.message}`

)

localStorage.setItem('token', userData.accessToken);

localStorage.setItem('userID', userData.userID);

setTimeout(() => {

window.location.href='landing.html';

}, 3000);

}

else{

const errorData = await response.json();

popUpMessage.show(

"Authorization failed",

`${errorData.message}`

)

}

}

catch(err){

popUpMessage.show(

"Authorization failed",

"An error occurred during login"

)

console.error(err);

}

})

}

Обращение к API корзины:

const deleteFetch = async () => {

try{

const url = `http://localhost:3000/API/userBasket/deleteUserConfig?id=${configId}`

const response = await fetch(url,{

method: 'DELETE',

headers: {

Authorization: `Bearer ${localStorage.getItem('token')}`

},

});

if (response.status === 200) {

// Если удаление прошло успешно, то удаляем li из списка и li и summary

const deleteLiItem = deleteBtn.closest('.basket\_\_list-item');

const id = deleteLiItem.id;

deleteLiItem.remove();

const deleteLiItemSummary = document.getElementById(id);

deleteLiItemSummary.remove()

popUpMessage.show(

'Deletion Message',

'Configuration successfully deleted'

)

// Если список конфигураций пуст

const basketList = document.querySelector('.basket\_\_summary-list');

if (basketList.children.length === 0) {

const basketContainer = document.querySelector('.basket\_\_container');

basketContainer.remove();

const basketSection = document.querySelector('.basket');

const div = this.createEmptyBasketContainer()

basketSection.appendChild(div);

window.scroll(0,0)

}

}

}

catch(err){

console.log(err)

}

}

const fetchBasket = async () => {

try{

const url = 'http://localhost:3000/API/userBasket/getUserConfig';

const response = await fetch(url, {

method: 'GET',

headers: {

'Accept': 'application/json',

Authorization: 'Bearer ' + localStorage.getItem('token')

},

});

const basket = await response.json();

return basket.userConfig;

}

catch(err){

console.log(err);

}

}

const requestToAddConfigAPI = async () => {

// Запрос к API для сохранения в корзину

try{

const apiURL = "http://localhost:3000/API/userBasket/addConfig";

const bodyJSON = {

model\_id,

config,

}

const response = await fetch(apiURL, {

method: 'POST',

headers: {

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

'Authorization': `Bearer ${localStorage.getItem('token')}`,

},

body: JSON.stringify(bodyJSON),

});

const data = await response.json();

console.log(data);

// Если конфигурация успешно добавлена

if(response.ok){

popUpMessage.show(

'Success',

'Configuration successfully saved, thanks'

)

}

// Если такая конфигурация уже существует

else if (response.status === 409){

popUpMessage.show(

'Save conflict',

'This configuration already exists in your account'

)

}

}

catch(error){

console.log(error)

}

}

**4 ИНТЕРФЕЙС ВЗАИМОДЕЙСТВИЯ**

Клиентская часть приложения отвечает за взаимодействие пользователя с системой, предоставляя удобный и интуитивно понятный интерфейс для работы с данными, хранящимися на сервере. В данном проекте клиент реализован с использованием HTML, CSS и JavaScript, что позволяет создать современный и адаптивный интерфейс для сайта Porsche.

Основные задачи клиентской части включают отправку HTTP-запросов к серверу для получения, добавления, обновления и удаления данных, обработку пользовательских событий и динамическое обновление интерфейса без необходимости полной перезагрузки страницы. Это улучшает отзывчивость приложения и делает его более интерактивным для пользователя.

Архитектура клиентской части:

Клиентская часть состоит из нескольких основных компонентов:

HTML — структура страниц, включающая элементы интерфейса, такие как кнопки, формы и таблицы.

CSS — стилизация элементов для создания современного и привлекательного дизайна. Используются flexbox и grid для адаптивных макетов.

JavaScript — логика обработки данных, отправка запросов и динамическое обновление страницы.

Основные страницы приложения включают:

Основные страницы приложения:

landing.html (Лендинг-страница)

Главная страница, которая служит первым впечатлением о сайте. Обычно включает в себя крупные баннеры, анимации, промо-материалы и ключевые ссылки для навигации. Эта страница содержит: Приветственный блок с лозунгами бренда Porsche, Слайдер с новыми моделями и акциями,

Блоки с преимуществами бренда (мощность, технологии, дизайн),Форма подписки на рассылку новостей, Кнопки для быстрой регистрации и авторизации.

auth.html (Страница аутентификации)

Страница, на которой пользователи вводят свои учетные данные для входа в систему. Обычно включает форму с полями для ввода email и пароля, а также кнопкой входа. Дополнительные элементы: Ссылка на страницу регистрации, если пользователь еще не зарегистрирован.

reg.html (Страница регистрации)

Страница для создания нового аккаунта. Включает форму для ввода необходимых данных: Имя пользователя, Email, Пароль, Подтверждение пароля, Поле для согласия с политикой конфиденциальности. После успешной регистрации пользователь перенаправляется на страницу аутентификации или сразу попадает в личный кабинет.

modelOverview.html (Страница рядов моделей)

Страница, которая отображает все доступные ряды моделей Porsche, такие как 911, 718. Для каждой модели отображается краткая информация, такая как: Название модели, Краткое описание, Изображение, Кнопка для перехода на детальную страницу модели (modelReview.html).

modelReview.html (Страница с конкретной моделью)

Страница, отображающая полную информацию о конкретной модели автомобиля. Обычно включает: Крупное изображение автомобиля, Детальное описание, Галерея изображений, Кнопка добавления в корзину, Список комплектаций и доступных опций.

basket.html (Страница с корзиной)

Страница, на которой пользователи могут увидеть добавленные автомобили перед оформлением заказа. Включает: Список выбранных моделей, Подсчет общей стоимости, Функцию удаления моделей из корзины, Кнопку для перехода к оформлению заказа, Поддержку динамического обновления списка товаров.

## 5 ТЕСТИРОВАНИЕ

Тестирование веб-приложения — это важный этап разработки, который позволяет убедиться в правильной работе всех компонентов, стабильности системы и отсутствии критических ошибок. Для вашего проекта, построенного на Node.js, Express.js и MySQL, тестирование включает несколько основных направлений, описанных в подпунктах.

**5.1 Функциональное тестирование**

Функциональное тестирование — это процесс проверки, соответствует ли разработанное приложение его функциональным требованиям и техническому заданию. Оно направлено на проверку корректности работы всех основных функций приложения, взаимодействия компонентов и обработки данных. Для вашего веб-приложения Porsche, функциональное тестирование включает проверку основных пользовательских сценариев, таких как регистрация, авторизация, просмотр моделей автомобилей и управление корзиной.

Основные цели функционального тестирования:

* Проверка правильности выполнения всех бизнес-логик и пользовательских сценариев.
* Обеспечение корректной обработки данных и сообщений.
* Подтверждение того, что приложение работает в соответствии с ожидаемыми требованиями.

Ключевые сценарии функционального тестирования:

1. Тестирование регистрации и авторизации:
   * Пользователь может создать новый аккаунт, заполнив поля формы регистрации (имя, email, пароль).
   * Валидация введенных данных: проверка формата email, длины пароля и совпадения полей пароля.
   * При успешной регистрации сервер должен вернуть подтверждающее сообщение, а данные пользователя должны быть сохранены в базе данных.
   * После регистрации пользователь может войти в систему, введя email и пароль. При этом сервер должен выдать JWT токен для аутентификации.
   * Проверка обработки ошибок: неправильный пароль, несуществующий аккаунт, повторная регистрация с тем же email.
2. Тестирование сессий и токенов:
   * Проверка корректности генерации JWT токенов при авторизации.
   * Обновление токенов при истечении срока действия (реализация маршрута /API/refresh).
   * Защита маршрутов, требующих аутентификации, от неавторизованных запросов.
   * Тестирование выхода из системы (инвалидизация токенов).
3. Тестирование каталога автомобилей:
   * Отправка GET-запросов для получения списка всех моделей автомобилей.
   * Проверка корректности фильтрации моделей по сериям (например, 911).
   * Открытие детальной страницы автомобиля с полной информацией о модели (название, цена, характеристики).
   * Проверка корректного отображения изображений и текста.
4. Тестирование корзины:
   * Добавление автомобиля в корзину с сохранением конфигурации пользователя.
   * Получение списка всех добавленных в корзину моделей.
   * Удаление конкретной модели из корзины.
   * Проверка корректности расчета общей стоимости товаров в корзине.
5. Тестирование форм и сообщений об ошибках:
   * Валидация всех полей форм перед отправкой данных на сервер.
   * Обработка ошибок на клиенте (например, пустые поля, неправильный формат email).
   * Корректное отображение сообщений об успешных действиях (например, "Пользователь успешно зарегистрирован").

**5.2 Тестирование API**

Тестирование API является важным этапом в разработке клиент-серверных приложений, так как оно позволяет проверить корректность и стабильность взаимодействия между фронтендом и серверной частью. В данном проекте с использованием REST API тестирование сосредоточено на проверке правильности работы эндпоинтов, корректности принимаемых и возвращаемых данных, а также устойчивости системы к ошибкам.

Основные задачи тестирования API:

* Проверка корректного выполнения всех HTTP-запросов (GET, POST, DELETE и т.д.) к серверу.
* Валидация структуры и содержания запросов и ответов в формате JSON.
* Тестирование обработки ошибок — например, проверка поведения сервера при неверных или неполных данных.
* Проверка авторизации и аутентификации — правильность работы JWT-токенов, доступ к защищённым маршрутам.
* Тестирование производительности API — как быстро сервер отвечает на запросы, устойчивость при высокой нагрузке.

Примеры тестируемых эндпоинтов:

* POST /API/register — отправка данных для регистрации пользователя. Проверяется корректность сохранения данных и выдача статуса успешной регистрации или ошибки при дублировании пользователя.
* POST /API/login — проверка аутентификации пользователя, получение JWT-токена. Тестируется обработка неправильных данных и успешный вход.
* GET /API/carModels/getAllModels — получение списка всех моделей автомобилей. Проверяется корректность возвращаемых данных и структура ответа.
* POST /API/usersBasket/addConfig — добавление конфигурации автомобиля в корзину. Тестируется правильность обработки данных и возврат сообщения об успехе.
* DELETE /API/usersBasket/deleteUserConfig — удаление конфигурации из корзины по ID. Проверяется корректность удаления и возврат статуса.

Методы тестирования API:

Ручное тестирование с использованием инструментов, таких как Postman, для отправки различных запросов и анализа ответов.

**6 Схемы алгоритмов решения задач по ГОСТ 19.701-90**

**6.1 Схема алгоритма авторизации**

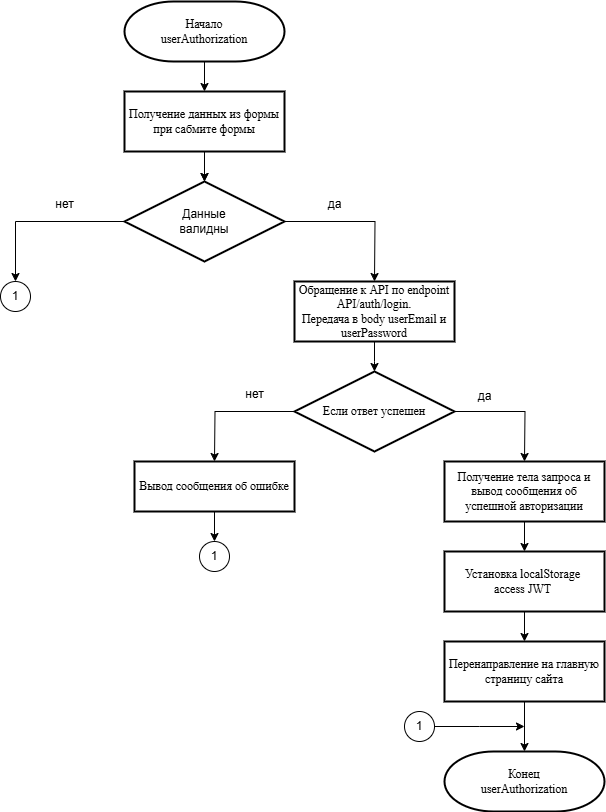


Рисунок 6.1 – авторизация пользователя

**6.2 Схема алгоритма добавления в корзину конфигурации**

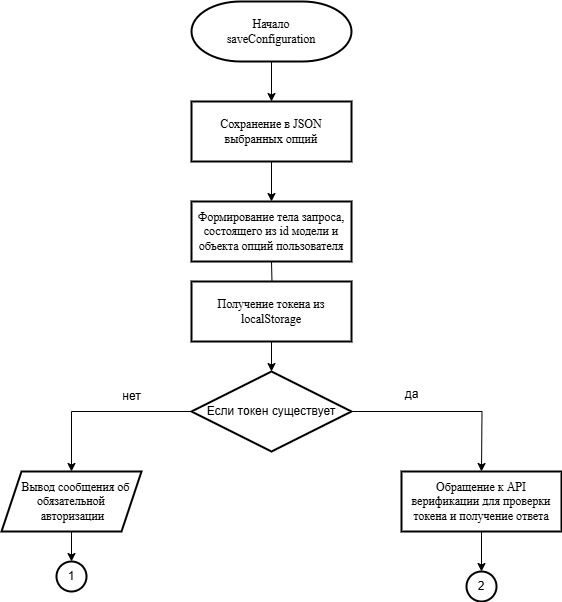


Рисунок 6.2 – сохранения конфигурации пользователя(часть1)

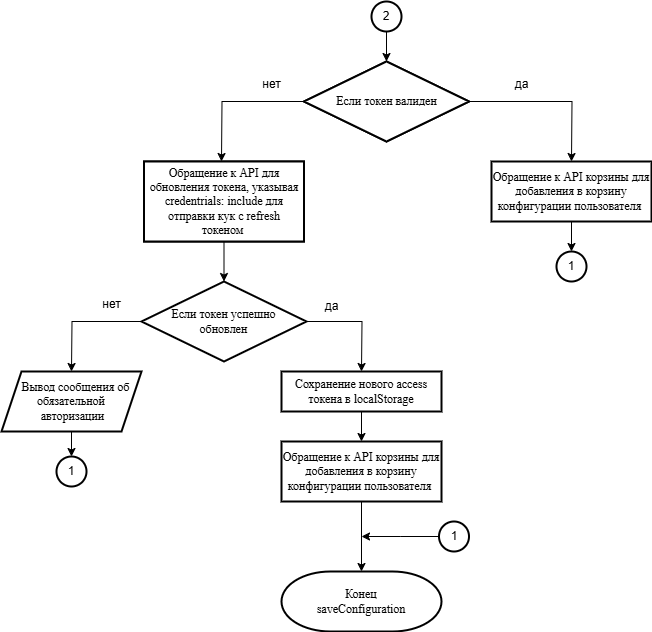


Рисунок 6.2 – сохранения конфигурации пользователя(часть2)

## ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта был разработан полнофункциональный клиент-серверный веб-сайт, посвящённый моделям автомобилей Porsche. Основной задачей проекта было создание удобного, быстрого и надёжного приложения для обмена данными между клиентом и сервером, а также предоставление пользователю полного набора возможностей для взаимодействия с каталогом автомобилей.

Использование архитектуры VMC позволило чётко разделить логику представления, обработки данных и взаимодействия с базой, что значительно упростило разработку, тестирование и дальнейшее сопровождение проекта. Серверная часть, реализованная на Node.js с использованием Express и TypeScript, обеспечивает надёжное и безопасное взаимодействие с базой данных MySQL через REST API. Это решение позволило построить гибкую и расширяемую архитектуру, которая легко адаптируется к новым требованиям и дополнительным функциям.

Клиентская часть, построенная на чистом JavaScript, предоставляет удобный и интуитивно понятный интерфейс для пользователей, позволяя выполнять операции регистрации, авторизации, просмотра моделей, добавления автомобилей в корзину и их последующего заказа. Использование простых, но эффективных методов взаимодействия с сервером (через REST API) обеспечило высокую скорость отклика приложения и удобство его использования на различных устройствах.

Тестирование, проведённое в процессе разработки, подтвердило корректность работы основных функций, надёжность API и устойчивость системы к ошибкам. Реализованное приложение отвечает поставленным требованиям и может служить основой для дальнейшего расширения функционала, включая интеграцию более сложных механизмов поиска, фильтрации, персонализации и анализа пользовательских данных.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Node.js - JavaScript runtime built on Chrome's V8 JavaScript engine [Электронный ресурс]. – Режим доступа: https://nodejs.org/. – Дата доступа: 16.05.2025.

[2] Express.js - Fast, unopinionated, minimalist web framework for Node.js [Электронный ресурс]. – Режим доступа: https://expressjs.com/. – Дата доступа: 16.05.2025.

[3] TypeScript - JavaScript with syntax for types [Электронный ресурс]. – Режим доступа: https://www.typescriptlang.org/. – Дата доступа: 16.05.2025.

[4] MySQL - The world's most popular open source database [Электронный ресурс]. – Режим доступа: https://www.mysql.com/. – Дата доступа: 16.05.2025.

[5] ECMAScript - Language Specification [Электронный ресурс]. – Режим доступа: https://tc39.es/ecma262/. – Дата доступа: 16.05.2025.

[6] Porsche Official Website [Электронный ресурс]. – Режим доступа: https://www.porsche.com/. – Дата доступа: 16.05.2025.

[7] Figma - Online design tool for teams [Электронный ресурс]. – Режим доступа: https://www.figma.com/. – Дата доступа: 16.05.2025.

[8] JSON Web Tokens (JWT) [Электронный ресурс]. – Режим доступа: https://jwt.io/. – Дата доступа: 16.05.2025.

[9] REST API Design - Best Practices [Электронный ресурс]. – Режим доступа: https://restfulapi.net/. – Дата доступа: 16.05.2025.

[10] CORS - Cross-Origin Resource Sharing [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. – Дата доступа: 16.05.2025.

## ПРИЛОЖЕНИЕ А

**(обязательное)**

**Исходный код клиента и сервера для добавления в корзину конфигурации**

// --------------------------------------------------------------------------------------------------//

// Получение активных слайдов и информации выбранной пользователем //

// --------------------------------------------------------------------------------------------------//

// Работа с активным слайдом ЭКСТЕРЬЕРА //

// Получение выбранного слайда

const exteriorChosenSlide = exteriorSwiper.slides[exteriorSwiper.activeIndex]

// Получение выбранного изображения и описания

const exteriorChosenImg = exteriorChosenSlide.querySelector('.color\_\_swiper-slide-picture img');

const exteriorChosenCaption = exteriorChosenSlide.querySelector('.color\_\_swiper-slide-picture figcaption');

// Сохранение в JSON изображения и описания

const exterior = {

"exterior\_color" : exteriorChosenCaption.textContent.trim(),

"exterior\_color\_img": exteriorChosenImg.getAttribute('src'),

}

// Работа с активным слайдом ИНТЕРЬЕРА //

// Получение выбранного слайда

const interiorChosenSlide = interiorSwiper.slides[interiorSwiper.activeIndex]

// Получение выбранного изображения и описания

const interiorChosenImg = interiorChosenSlide.querySelector('figure img');

const interiorChosenCaption = interiorChosenSlide.querySelector('figure figcaption');

// Сохранение в JSON изображения и описания

const interior = {

"interior\_color" : interiorChosenCaption.textContent.trim(),

"interior\_color\_img": interiorChosenImg.getAttribute('src'),

}

// Работа с активным слайдом ДИСКОВ //

// Получение выбранного слайда

const wheelsChosenSlide = wheelsSwiper.slides[wheelsSwiper.activeIndex]

// Получение выбранного изображения и описания

const wheelsChosenImg = wheelsChosenSlide.querySelector('.wheels\_\_swiper-slide-picture img');

const wheelsChosenCaption = wheelsChosenSlide.querySelector('.wheels\_\_swiper-slide-picture figcaption');

// Сохранение в JSON изображения и описания

const wheels = {

"wheels" : wheelsChosenCaption.textContent.trim(),

"wheels\_img": wheelsChosenImg.getAttribute('src'),

}

// Работа с активным слайдом ПАКЕТОВ //

// Получение выбранного слайда

const packageChosenSlide = packageSwiper.slides[packageSwiper.activeIndex]

// Получение выбранного изображения и описания

const packageChosenTitle = packageChosenSlide.querySelector('.package\_\_swiper-slide-title');

const packageChosenCaption = packageChosenSlide.querySelector('.package\_\_swiper-slide-description');

// Сохранение в JSON изображения и описания

const carPackage = {

"package\_title" : packageChosenTitle.textContent.trim(),

"package\_description": packageChosenCaption.textContent.trim().replace(/\s+/g, ' '),

}

// Работа с активным слайдом ВЫХЛОПА //

// Получение выбранного слайда

const exhaustChosenSlide = exhaustSwiper.slides[exhaustSwiper.activeIndex]

// Получение выбранного изображения и описания

const exhaustChosenTitle = exhaustChosenSlide.querySelector('.exhaust\_\_swiper-slide-title');

const exhaustChosenCaption = exhaustChosenSlide.querySelector('.exhaust\_\_swiper-slide-description');

// Сохранение в JSON изображения и описания

const exhaust = {

"exhaust\_title" : exhaustChosenTitle.textContent.trim(),

"exhaust\_description": exhaustChosenCaption.textContent.trim().replace(/\s+/g, ' '),

}

// Получение итоговой цены конфигурации //

const modelPriceEl = document.querySelector('.total-price-for-save')

const model\_price = parseInt(modelPriceEl.textContent)

// Итоговая конфигурация

const config = {

exterior\_color: exterior.exterior\_color,

exterior\_color\_img: exterior.exterior\_color\_img,

interior\_color: interior.interior\_color,

interior\_color\_img: interior.interior\_color\_img,

wheels: wheels.wheels,

wheels\_img: wheels.wheels\_img,

package\_title: carPackage.package\_title,

package\_description: carPackage.package\_description,

exhaust\_title: exhaust.exhaust\_title,

exhaust\_description: exhaust.exhaust\_description,

model\_price: model\_price,

}

const model\_id = localStorage.getItem('modelID');

const popUpMessage = new PopUpMessage();

// Функция для запроса к API для сохранения в корзину

const requestToAddConfigAPI = async () => {

// Запрос к API для сохранения в корзину

try{

const apiURL = "http://localhost:3000/API/userBasket/addConfig";

const bodyJSON = {

model\_id,

config,

}

const response = await fetch(apiURL, {

method: 'POST',

headers: {

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

'Authorization': `Bearer ${localStorage.getItem('token')}`,

},

body: JSON.stringify(bodyJSON),

});

const data = await response.json();

console.log(data);

// Если конфигурация успешно добавлена

if(response.ok){

popUpMessage.show(

'Success',

'Configuration successfully saved, thanks'

)

}

// Если такая конфигурация уже существует

else if (response.status === 409){

popUpMessage.show(

'Save conflict',

'This configuration already exists in your account'

)

}

}

catch(error){

console.log(error)

}

}

// Верификация токена

const token = localStorage.getItem('token');

if(!token){

popUpMessage.show(

"Authorization required",

"To continue, you must log in."

)

return

}

try{

const response = await TokenService.verifyToken(token);

// Если токен валиден, то сохранение в корзину

if (response.ok){

await requestToAddConfigAPI();

}

// Если не валиден, то попытка обновления

else{

const response = await TokenService.refreshToken();

// Если успешно обновлен

if(response.ok){

const data = await response.json();

localStorage.setItem('token', data.accessToken);

await requestToAddConfigAPI();

}

else{

popUpMessage.show(

"Session expired\n",

"Your session has expired. Please log in again."

)

setTimeout(AuthMenu.logOut, 2000)

}

}

}

catch(error){

console.log(error)

}

})

static async addConfig(req: Request, res: Response){

try {

// Получение id пользователя через JWT токен

const token: string = req.headers.authorization!.split(" ")[1];

const decoded: any = jwt.verify(token, process.env.ACCESS\_TOKEN\_SECRET!);

const user\_id: number = decoded.user\_id;

console.log("decoded: ", decoded);

// Получение тела запроса

const { model\_id, config } = req.body;

// Получение хеша конфигурации

const hashConfig: string = getConfigHash(config);

// Проверка на существование у пользователя такой конфигурации

const usersConfigs: RowDataPacket[] = await usersBasketModel.getConfigByHash(hashConfig, user\_id);

// Если существует такая конфигурация

if (usersConfigs.length !== 0){

// Конфликт с текущим состоянием ресурса

res.status(409).json({

message: `Configuration already exists in account`,

})

return;

}

// Добавление конфигурации в БД

const userConfigInfo: usersBasketRowInterface = {

user\_id,

model\_id,

hashConfig,

...config

}

await usersBasketModel.insertConfig(userConfigInfo);

// Если запись успешно добавлена

res.status(201).json({

message: `Configuration successfully saved in your collection`,

})

return;

}

catch (err: any){

res

.status(500)

.json({

"statusCode": 500,

"message" : `Server Error ${err.message}`,

})

return;

}

}

static async insertConfig(config: any){

const connection = await createConnToDataBase()

try{

await connection.execute(`

INSERT INTO users\_basket (user\_id, model\_id, config\_hash,

exterior\_color, exterior\_color\_img,

interior\_color, interior\_color\_img,

wheels, wheels\_img,

package\_title, package\_description,

exhaust\_title, exhaust\_description,

model\_price)

VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)`,

[

config.user\_id, config.model\_id, config.hashConfig,

config.exterior\_color, config.exterior\_color\_img,

config.interior\_color, config.interior\_color\_img,

config.wheels, config.wheels\_img,

config.package\_title, config.package\_description,

config.exhaust\_title, config.exhaust\_description,

config.model\_price

]

);

}

catch(err){

console.log(err);

throw err;

}

finally {

await connection.end();

}

}