**3. ПРОЕКТНО-КОНСТРУКТОРСКАЯ ЧАСТЬ**

1. **Разработка структуры приложения**

Для представления принципа работы приложения ниже приведена ее

UML диаграмма вариантов использования (рис. 3.1).

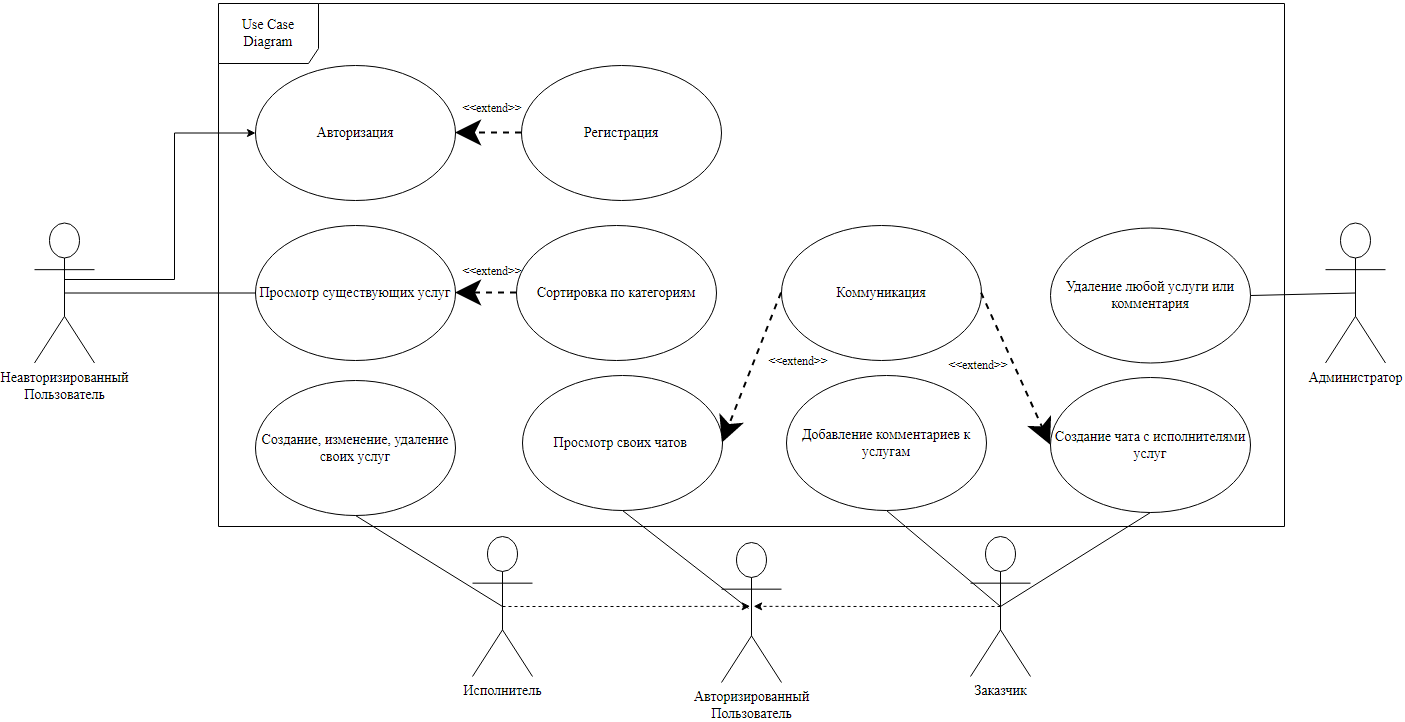


Рисунок 3.1 – UML диаграмма вариантов использования

При попадании на веб-приложение агрегатор услуг по ремонту пользователь сам решает, что ему необходимо сделать. Например, он может просматривать все добавленные услуги другими пользователями-исполнителями без необходимости регистрироваться. Однако, если пользователь-заказчик захочет связаться с пользователем-исполнителем услуги (с помощью чата) или оставить комментарий на услугу, то ему придётся пройти процесс регистрации, а затем авторизации. Если пользователь-исполнитель захочет добавить свою услугу, которая будет отображаться всем другим пользователям, то ему также необходимо пройти процессы регистрации и авторизации.

В любой момент работы с веб-приложением, пользователь может авторизоваться, зарегистрироваться или выйти из аккаунта.

Модератор (администратор) поддерживает порядок на сайте и контролирует корректность информации, предоставляемой пользователями. Модератор может удалять любые услуги и комментарии к ним, если сочтет их содержание неприемлемым или вводящим в заблуждение.

Общее назначение программного средства – предоставление возможностей размещения и просмотра услуг по ремонту, а также коммуникации между заказчиком и исполнителем.

Реализуемая задача состоит в том, чтобы хранение всей информации, необходимой для существования веб-приложения осуществлялось в базе данных, а также производилось удобное взаимодействие с пользовательским интерфейсом.

1. **Разработка интерфейса взаимодействия пользователя с системой**

Проектирование макета веб-приложения происходило в Figma. Верстка сайта и проектирование веб-интерфейса осуществлялась с помощью React, Tailwind CSS, JavaScript.

React — JavaScript-библиотека с открытым исходным кодом для разработки пользовательских интерфейсов.  React разрабатывается и поддерживается Facebook, Instagram и сообществом отдельных разработчиков и корпораций.

Tailwind CSS — это CSS-фреймворк с открытым исходным кодом. Основная особенность этой библиотеки заключается в том, что, в отличие от других фреймворков CSS, таких как Bootstrap, она не предоставляет ряд предопределенных классов для таких элементов, как кнопки или таблицы.

На рисунке 3.2 представлен разработанный интерфейс взаимодействия пользователя с системой.

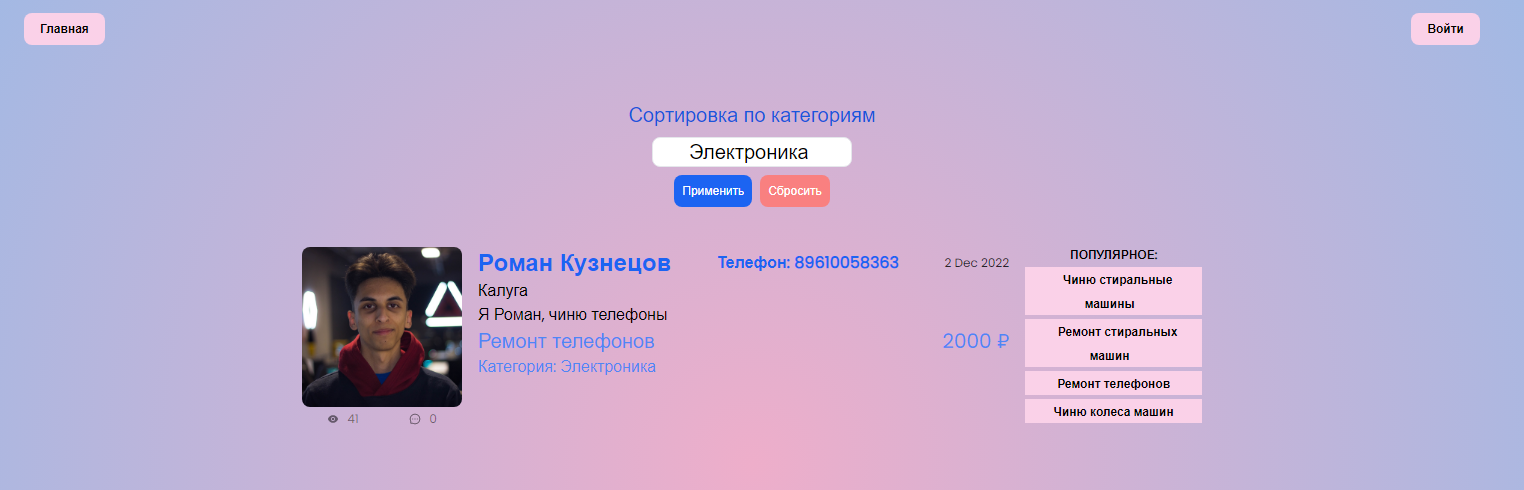


Рисунок 3.2 – Пользовательский интерфейс веб-приложения

1. **Физическая схема базы данных**

На рисунке 3.3 представлена физическая схема базы данных.

Документ «User» содержит информацию о пользователе: логин, пароль, имя, фамилию, город, номер телефона, является ли он администратором или нет, услуги этого пользователя.

Документ «Service» содержит информацию об услуге: логин владельца, заголовок, основной текст, категория, изображение пользователя, цена на услугу, просмотры, автор (владелец), комментарии к услуге.

Документ «Comment» содержит информацию о комментариях. Здесь происходит хранение текста комментария, имени, описание, история поставщика.

Документ «Chat» содержит информацию о чатах. Здесь хранится информация об обоих пользователях и сообщения этого чата.

Документ «ChatMessage» содержит информацию о сообщении чата. Здесь осуществляется хранение идентификаторов чата и отправителя, имени отправителя и само сообщение.

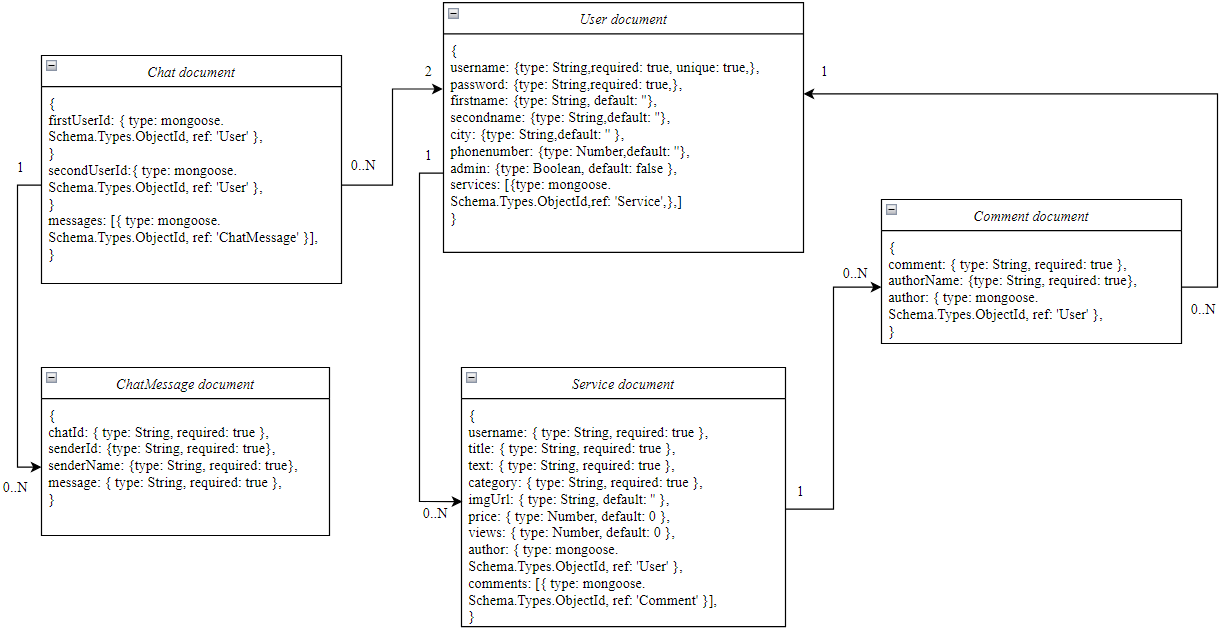


Рисунок 3.3 – Физическая модель данных

1. **Разработка архитектуры приложения**

Данная система представляет собой веб-приложение. Это особый тип программ, построенных по архитектуре "клиент-сервер". Особенность их заключается в том, что само веб-приложение находится и выполняется на сервере, а клиент при этом получает только результаты работы. Работа приложения основывается на получении запросов от пользователя (клиента), их обработке и выдачи результата. Передача запросов и результатов их обработки происходит через Интернет.

За счет наличия исполняемой части, веб-приложения способны выполнять практически те же операции, что и обычные Windows-приложения, с тем лишь ограничением, что код исполняется на сервере, в качестве интерфейса системы выступает браузер, а в качестве среды, посредством которой происходит обмен данными — Интернет.

К наиболее типичным операциям, выполняемым Web приложениями, относятся:

• прием данных от пользователя и сохранение их на сервере;

• выполнение различных действий по запросу пользователя: извлечение данных из базы данных (БД), добавление, удаление, изменение данных в БД, проведение сложных вычислений;

• аутентифицирование пользователя и отображение интерфейса системы, соответствующего данному пользователю;

• отображение постоянно изменяющейся оперативной информации и т.д.

При разработке данного веб-приложения в курсовой работе было принято решение использовать клиент-серверную архитектуру с использованием паттерна MVC.

Model-View-Controller (MVC) – схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо. Общий вид паттерна представлен на рисунке 3.4.

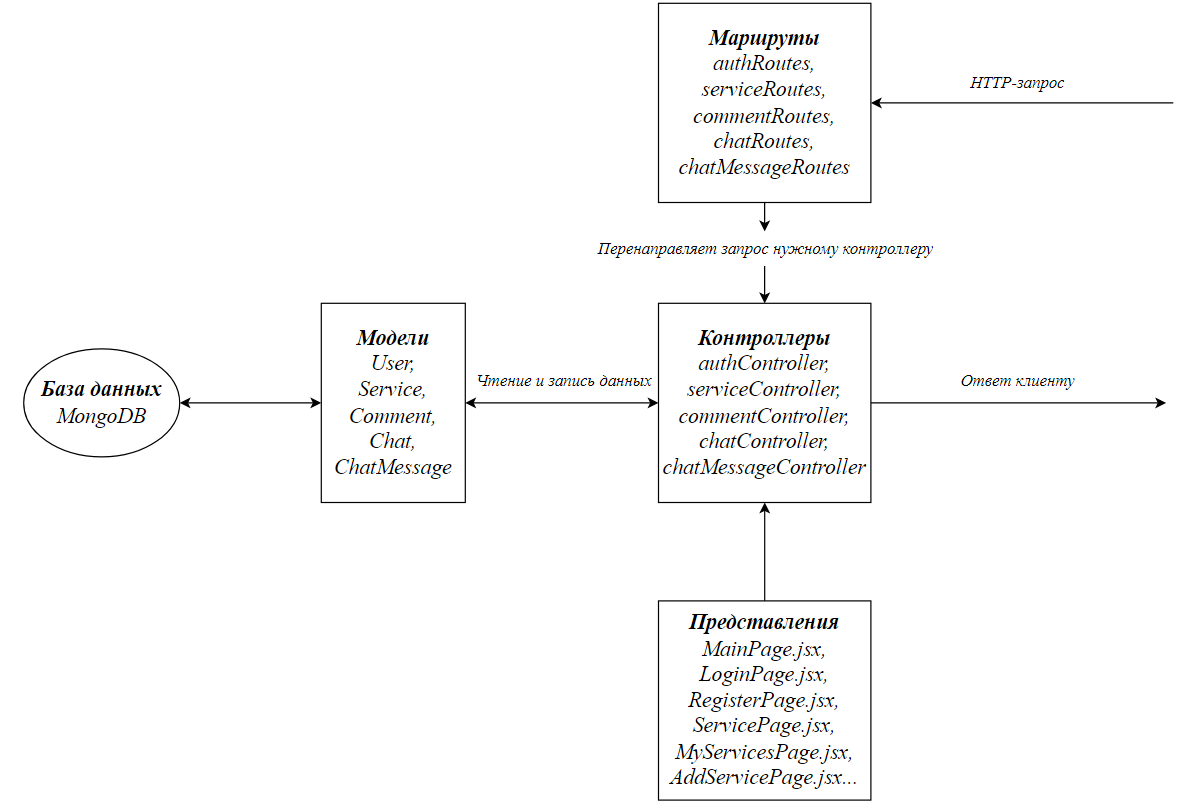


Рисунок 3.4 – Архитектура приложения

В этой схеме модель является независимым компонентом - любые изменения контроллера или представления никак не влияют на модель. Контроллер и представление являются относительно независимыми компонентами. Так, из представления можно обращаться к определённому контроллеру, а из контроллера генерировать представления, но при этом нередко их можно изменять независимо друг от друга.

Такое разграничение компонентов приложения позволяет реализовать концепцию разделение ответственности, при которой каждый компонент отвечает за свою строго очерченную сферу. В связи с чем легче построить работу над отдельными компонентами. И благодаря этому приложение легче разрабатывать, поддерживать и тестировать отдельные компоненты.

Концепция MVC разделяет данные, представление и обработку действий пользователя на компоненты:

• ***Модель / Model*** — предоставляет собой объектную модель некой предметной области, включает в себя данные и методы работы с этими данными, реагирует на запросы из контроллера, возвращая данные и/или изменяя своё состояние. При этом модель не содержит в себе информации о способах визуализации данных или форматах их представления, а также не взаимодействует с пользователем напрямую.

• ***Представление / View*** — отвечает за отображение информации (визуализацию). Одни и те же данные могут представляться различными способами и в различных форматах. Например, коллекцию объектов при помощи разных представлений можно представить на уровне пользовательского интерфейса как в табличном виде, так и списком; на уровне API можно экспортировать данные как в JSON, так в XML или XSLX.

• ***Контроллер / Controller*** — обеспечивает связь между пользователем и системой, использует модель и представление для реализации необходимой реакции на действия пользователя. Как правило, на уровне контроллера осуществляется фильтрация полученных данных и авторизация — проверяются права пользователя на выполнение действий или получение информации.

• ***Система маршрутизации*** как дополнительный компонент сопоставляет запросы с маршрутами и выбирает для обработки запросов определенный контроллер.

В общем случае, когда к приложению приходит запрос, система маршрутизации выбирает нужный контроллер для обработки запроса. Контроллер обрабатывает запрос. В процессе обработки он может обращаться к данным через модели и для рендеринга ответа использовать представления. Результат обработки контроллера отправляется в ответ клиенту. Ответ представляет html-страницу, которую пользователь видит в своем браузере.

В приложении определены роутеры, которые сопоставляются с адресами, например, "localhost:3002/". В файле index.js:

// http://localhost:3002

app.use('/api/auth', authRoutes)

app.use('/api/services, serviceRoutes)

app.use('/api/comments', commentRoutes)

app.use('/api/chat', chatRoutes)

app.use('/api/messages', messagesRoutes)

Контроллеры подключаются как стандартные модули, и затем функции контроллеров используются для обработки маршрутов. Кроме того, для упрощения логической организации маршруты, которые обрабатываются конкретными контроллерами authController, serviceController и другие организованы в отдельных роутерах – authRoutes, serviceRoutes … .

Однако в данном случае опять же стоит отметить, что внутри каждого роутера может быть определено множество маршрутов. В этом случае для упрощения управления маршрутами каждый роутер организуется и подключается в виде отдельного модуля. Для этого в проекте организована папка routes.

В папке routes соответственно хранятся данные роутеры. Для обработки запросов в нашем приложении используется Express. В нём определено ряд встроенных функций, и одной из таких является функция app.get(). Она обрабатывает GET-запросы протокола HTTP и позволяет связать маршруты с определенными обработчиками. Для этого первым параметром передается маршрут, а вторым - обработчик, который будет вызываться, если запрос к серверу соответствует данному маршруту.

Вся логика обработки запросов лежит в контроллерах. Для этого в каталоге приложения определена папка, которая называется controllers. В данной папке хранятся следующие контроллеры нашего приложения: authController, serviceController, commentController, chatController, chatMessageController. Каждый файл представляет собой отдельный контроллер, который содержит набор функций-обработчиков маршрутов.

Так в контроллере serviceController содержатся следующие функции-обработчики: createService(), getAll(), sortedServices(), getById(), getMyServices(), removeService(), updateService() и др. Чтобы получить список всех услуг, по адресу "localhost:3002/" определен маршрут и обработчик, представленный функцией контроллера getAll():

router.get('/', getAll)

Также нужно подключить конкретную функцию-обработчик, например, import { getAll} from '../controllers/serviceController.js'

Когда фреймворк Express получает запрос, этот запрос передается в конвейер обработки. Конвейер состоит из набора компонентов или middleware, которые получают данные запроса и решают, как его обрабатывать.

Здесь конвейер обработки состоял из вызовов router.get(), которые сравнивали запрошенный адрес с маршрутом, и если между адресом и маршрутом было соответствие, то данный запрос обрабатывался методом router.get().

При переходе по адресу / будет срабатывать метод router.get, который отправит пользователю файл MainPage.jsx.

При обработке запросов фреймворк Express опирается на систему маршрутизации. В приложении определяются маршруты, а также обработчики этих маршрутов. Если запрос соответствует определенному маршруту, то вызывается для обработки запроса соответствующий обработчик.

Для обработки данных по определенному маршруту можно использовать ряд функций, в частности: use, get, post, put, delete. Так, данные удаляются с помощью метода DELETE, поэтому для обработки определяем функцию:

router.delete('/:id', checkAuth, removeService)

Первый параметр функции - адрес, на который идет отправка - /services/:id, второй параметр – небольшой middleware, который проверят авторизирован ли пользователь, третий параметр – функция-обработчик.

Для получения отправленного id услуги используем выражение типа request.params.id, где request.params инкапсулирует передаваемые параметры, а id - ключ данных, который соответствует значению id соответствующей услуги для удаления. Функция-обработчик removeService контроллера serviceController:

export const removeService = async (req, res) => {

    try {

        const service = await Service.findByIdAndDelete(req.params.id)

        if (!service) return res.json({ message: 'Такой услуги не существует' })

        await User.findByIdAndUpdate(req.userId, {

            $pull: { services: req.params.id },

        })

        res.json({ message: 'Услуга была удалена.' })

    } catch (error) {

        res.json({ message: 'Что-то пошло не так.' })

    }

}

Таким образом, за счет выноса логики обработки маршрутов и организации маршрутов в роутеры в отдельные модули общий код приложения стал проще и яснее.

В данном приложении для управления между состояниями данных и представлениями пользовательского интерфейса используется React и Redux toolkit. Redux Toolkit - это пакет, облегчающий работу с Redux. Он был разработан для решения трех главных проблем:

* слишком сложная настройка хранилища (store);
* для того чтобы заставить Redux делать что-то полезное, приходится использовать дополнительные пакеты;
* слишком много шаблонного кода (boilerplate).

Redux Toolkit предоставляет инструменты для настройки хранилища и выполнения наиболее распространенных операций, а также содержит полезные утилиты, позволяющие упростить код.

Разработка любого Redux-приложения предполагает создание и настройку хранилища. configureStore() позволит упростить процесс настройки:

import { configureStore } from '@reduxjs/toolkit'

import authSlice from './features/auth/authSlice'

import serviceSlice from './features/service/serviceSlice'

import commentSlice from './features/comment/commentSlice'

export const store = configureStore({

    reducer: {

        auth: authSlice,

        service: serviceSlice,

        comment: commentSlice,

    },

})

Редукторы (reducers) - это самая важная часть Redux. Как правило, редуктор отвечает за:

* определение характера ответа на основе поля type объекта операции;
* обновление состояния посредством копирования части состояния и модификации этой копии.

createSlice() автоматически генерирует типы и создателей операции на основе переданного названия редуктора. Создателя операции определяют action creators. Параметры, необходимые для выполнения операции, обычно, помещаются в поле payload. createSlice() анализирует функции, определенные в поле reducers (в нашем случае createComment), создает редуктор для каждого случая и генерирует создателя, использующего название редуктора в качестве типа операции. Таким образом, редуктор createComment становится типом операции comment/createComment, а создатель createComment() возвращает операцию с этим типом.

Хранилище Redux ничего не знает об асинхронной логике. Оно знает только о том, как синхронно отправлять операции, обновлять состояние посредством вызова корневого редуктора и уведомлять UI об изменениях. Любые асинхронные операции должны выполняться за пределами хранилища.

Существует несколько посредников для реализации асинхронности в Redux. Рекомендуемым является Redux Thunk. Он прекрасно подходит для большинства случаев, а использование синтаксиса async/await делает его еще лучше. createAsyncThunk() упрощает процесс выполнения асинхронных запросов - мы передаем ему строку для префикса типа операции и колбек создателя полезной нагрузки (payload), выполняющего реальную асинхронную логику и возвращающего промис с результатом. createAsyncThunk() возвращает преобразователя, который заботится об отправке правильных операций на основе возвращенного промиса, и типы операции, которые можно обработать в редукторах.

Таким образом, создание комментария createSlice вместе с action creator и createAsyncThunk будет выглядеть следующим образом:

const initialState = {

    comments: [],

    loading: false,

}

// Создание преобразователя

export const createComment = createAsyncThunk(

    'comment/createComment',

    async ({ serviceId, comment, author }) => {

        try {

            console.log('author', author)

            const { data } = await axios.post(`/comments/${serviceId}`, {

                serviceId,

                comment,

                author,

            })

            return data

        } catch (error) {

            console.log(error)

        }

    },

)

// Определяем редуктор и комментарии с помощью `createSlice()` и Обрабатываем операции в редукторе

export const commentSlice = createSlice({

    name: 'comment',

    initialState,

    reducers: {},

    extraReducers: {

        // Создание комментария

        [createComment.pending]: (state) => {

            state.loading = true

        },

        [createComment.fulfilled]: (state, action) => {

            state.loading = false

// Добавляем комментарий в массив состояния

            state.comments.push(action.payload)

        },

    },

})

export default commentSlice.reducer

// Позже в PostPage.jsx, отправляем `thunk`

dispatch(createComment({ serviceId, comment, author }))

// На сервере в commentController принимаем параметры из request.body

export const createComment = async (req, res) => {

        const { serviceId, comment, author } = req.body

…

        const newComment = new Comment({ comment, authorName, author })

        await newComment.save()

        await Service.findByIdAndUpdate(serviceId, {$push: { comments: newComment.\_id },})

…

Для интеграции MongoDB с проектом используется Object Document Mapper (ODM) Mongoose для создания схем и моделей данных приложения. В приложении определены следующие модели: User, Service, Comment, Chat, ChatMessage. Mongoose выступает в качестве внешнего интерфейса для MongoDB, базы данных NoSQL, использующей документно-ориентированную модель данных.

Модели определяются с помощью Schema. Схема позволяет определять поля, хранящиеся в каждом документе, вместе с их требованиями к проверке и значениями по умолчанию. Также можно определить статические вспомогательные методы, для того, чтобы упростить работу с типами данных.

Например, модель комментария в нашем приложении будет:

import mongoose from 'mongoose'

const CommentSchema = new mongoose.Schema(

{

comment: { type: String, required: true },

authorName: {type: String, required: true},

author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },

},

    { timestamps: true },

)

export default mongoose.model('Comment', CommentSchema)

Затем схемы «компилируются» в модели с помощью mongoose.model() метода. Метод save() сохраняет текущий обьект в базу данных. Когда у нас есть модель, мы можем использовать ее для поиска, создания, обновления и удаления объектов данного типа.

Организация взаимодействия приложения с сервером посредством протокола HTTP осуществляется за счет принципа REST (Representational state transfer).

Другой отличительной чертой REST является то, что API-интерфейсы REST фокусируются на ресурсах (то есть вещах, а не действиях) и способах доступа к ресурсам. Ресурсы, как правило, являются разными типами информации. Мы получаем доступ к ресурсам через URL (Uniform Resource Locators), так же как переход к URL-адресу в вашем браузере позволяет подключиться к информационному ресурсу. URL-адреса сопровождаются методом, который указывает, как вы хотите взаимодействовать с ресурсом

REST работает на уровне HTTP и использует нижеприведённые методы для работы с ресурсами на сервере:

POST — для создания ресурса;

GET — для его получения;

PUT — для его обновления;

DELETE — для его удаления.

Разумеется, этот перечень сопровождается такими опциями, как обработка ошибок в запросе, разграничение доступа к данным и валидация входящих данных, в общем, всеми возможными проверками, которые сервер выполняет перед тем, как выполнить желание клиента.

В данном случае, например, необходимо иметь возможность добавлять новые услуги, просматривать все существующие услуги и конкретную услугу пользователя.

Эти операции соответствуют следующим маршрутам:

• POST /services— создание новой услуги;

• GET /services/:id — получение одной услуги;

• GET /services — получение всех услуг.

Что происходит, когда клиент хочет получить подробную информацию о услуге с идентификатором 6389fabd6ab54b47891c88db из списка всех услуг с помощью REST API веб-приложения агрегатора? Клиент должен обмениваться данными с сервером, на котором размещен API, используя протокол HTTP, как показано на рисунке 3.5

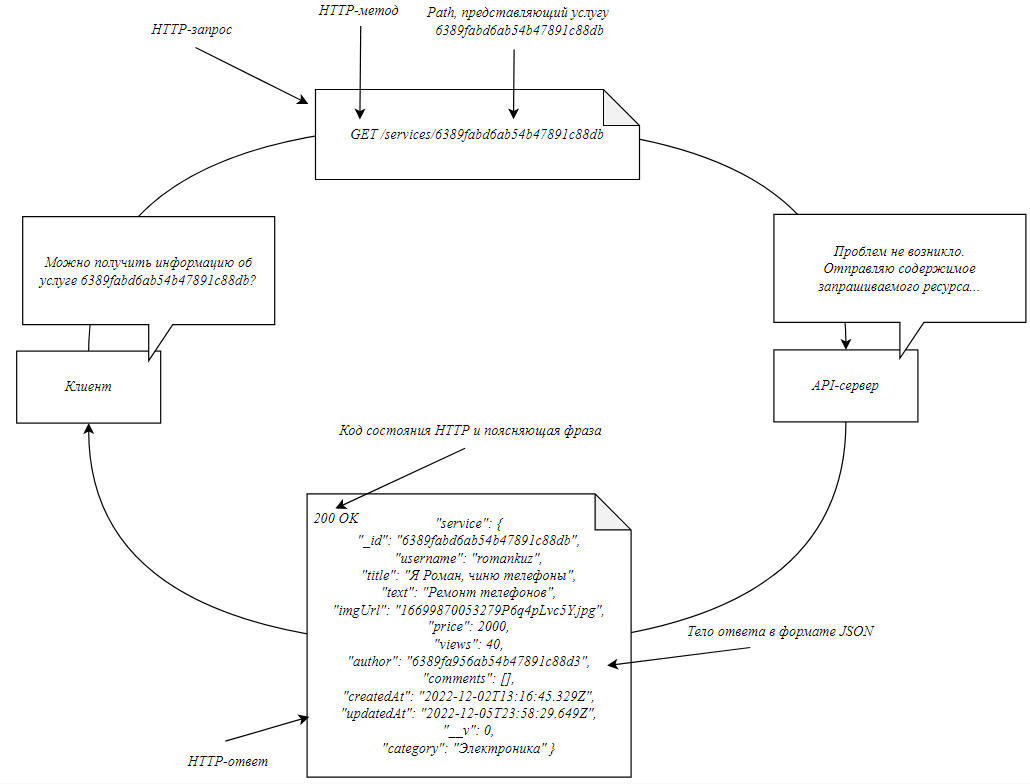


Рисунок 3.5 - Вызов REST API с использованием протокола HTTP

Поскольку получение информации об услуге представлено GET /services/:id, клиент отправляет HTTP-запрос GET /services/6389fabd6ab54b47891c88db на сервер API веб-приложения. В ответ сервер возвращает ответ, 200 OK, за которым следует информация о запрашиваемой услуге. (Обратите внимание, что мы упростили этот обмен данными по протоколу HTTP, чтобы сосредоточиться только на тех элементах, которые важны для нас).

Запрос состоит из HTTP-метода GET и пути /services/6389fabd6ab54b47891c88db. Путь – это адрес, идентифицирующий ресурс на сервере; в данном случае услуга 6389fabd6ab54b47891c88db находится в services. HTTP-метод указывает на то, что потребитель хочет сделать с этим ресурсом: GET означает, что он хочет получить ресурс. С функциональной точки зрения такой запрос означает что-то вроде: «Привет, могу ли я получить информацию об услуге с идентификатором 6389fabd6ab54b47891c88db»? Но с точки зрения протокола HTTP это означает: «Привет, могу ли я определить ресурс по пути /services/6389fabd6ab54b47891c88db»?

Первая часть ответа состоит из кода состояния HTTP 200 и поясняющей фразы OK. Код состояния сообщает нам, как прошла обработка запроса. Благодаря поясняющей фразе мы можем догадаться, что код состояния HTTP 200 означает, что все прошло нормально. Вторая часть ответа называется телом ответа. Она содержит содержимое ресурса, идентифицированного путем в запросе, который в данном случае представляет собой информацию об услуге 6389fabd6ab54b47891c88db в виде данных в формате JSON. С функциональной точки зрения ответ, возвращаемый сервером API, в основном означает: «Конечно, вот информация о запрашиваемой услуге». С точки зрения HTTP имеется в виду: «Проблем не возникло. Отправляю содержимое запрашиваемого ресурса...».

Преимущества использования данного подхода заключаются в следующем:

• Независимость сервера от клиента — серверы и клиенты могут быть мгновенно заменены другими независимо друг от друга, так как интерфейс между ними не меняется. Сервер не хранит состояний клиента.

• Уникальность адресов ресурсов — каждая единица данных (любой степени вложенности) имеет свой собственный уникальный URL, который, по сути, целиком является однозначным идентификатором ресурса.

• Независимость формата хранения данных от формата их передачи — сервер может поддерживать несколько различных форматов для передачи одних и тех же данных (JSON, XML и т. д.), но хранит данные в своем внутреннем формате, независимо от поддерживаемых.

• Присутствие в ответе всех необходимых метаданных — помимо самих данных сервер должен возвращать детали обработки запроса, например, сообщения об ошибках, различные свойства ресурса, необходимые для дальнейшей работы с ним, например, общее число записей в коллекции для правильного отображения постраничной навигации