**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ……………………………………………………………………….. 3](#_gjdgxs)

[1 ОБЩИЕ СВЕДЕНИЯ ОБ ООО «Модсен»……………………………………. 4](#_30j0zll)

1.1 История организации……………………………………..…………… 4

[1.2 Охрана труда и техника безопасности ...……………………………...5](#_3znysh7)

[2](#_2et92p0) ОБЗОР ТЕХНОЛОГИИ *MERN* и *REST API*……………….…………………...[8](#_tyjcwt)

[2.1 Технология](#_3dy6vkm) *MERN* [………………….……………………………….](#_3dy6vkm).....[8](#_3dy6vkm)

[2.2](#_1t3h5sf) [Технология](#_3dy6vkm) *REST API*………………………………………………......9

[3 РАЗРАБОТКА *WEB API*……………………………………………………..](#_17dp8vu) ..13

[3.1 Постановка задачи…………………………………………………….](#_3rdcrjn) 13

[3.2](#_26in1rg) Модель данных………………………………………………………...[13](#_lnxbz9)

3.3 Уровень бизнес-логики……………………………………………….18

[3.4](#_1ksv4uv) Уровень представления………………………………………………. [24](#_44sinio)

[3.5 Тестирование *Web API*………………………………………………..](#_2jxsxqh)24

[ЗАКЛЮЧЕНИЕ……………………………………………………………….. 26](#_z337ya)

[Список использованных источников…………………………………………... 27](#_3j2qqm3)

[ПРИЛОЖЕНИЕ А](#_1y810tw) [Листинг программы……………………………………….. 28](#_4i7ojhp)

# ВВЕДЕНИЕ

Производственная практика является необходимым этапом формирования у обучающихся требуемых компетенций. Ее ценность заключается в приобретении новых и закреплении уже полученных знаний. В период прохождения производственной практики, обучающиеся должны закрепить теоретический материал, приобрести практические навыки.

Место практики: ООО «Модсен».

В ходе обучения важно придерживаться согласованности теории и практики. Как и любой другой этап образовательного процесса, пребывание студента на производстве преследует определенные цели и задачи.

Цели технологической практики: закрепление, расширение, углубление и систематизация теоретических знаний, а также приобретение навыков проектирования и конструирования информационных систем.

Задачи технологической практики:

* развитие и закрепление практических навыков выполнения анализа предметной области;
* приобретение практического опыта проектирования программных систем;
* развитие и закрепление практических навыков использования языков и инструментальных средств моделирования при проектировании системы;
* развитие и закрепление практических навыков создания программных систем с использованием современных сред разработки, поддерживающих возможность командной работы, контроля проекта и версий системы;
* развитие и закрепление практических навыков разработки документации к системе (технического задания, инструкций пользователя и программиста).

# 1 ОБЩИЕ СВЕДЕНИЯ ОБ ООО «ЛЕВЕРЕКС ИНТЕРНЕШНЛ»

* 1. История организации

*Modsen* – международная *IT*-компания. На данный момент в ней работают 150-200 сотрудников. Офисы компании размещены в Беларуси и Польше.

*Modsen* является молодой компанией, которая была основана в 2020 году. Но не смотря на возраст уже имеет большую историю разработки, большую и дружную команду и многожество партнёров, таких как *TopDevelopers*, *DESIGNRUSH*, *iTRate* и другие.

В следствие того, что компания молодая, она активно ищет новых партнёров и клиетов. И для этого компания предлагает следующее:

* Команда, которая подстроится под ваш часовой пояс;
* Краткосрочный период запуска продукта и быстрый выход на рынок;
* Первоклассные специалисты в сфере вашего бизнеса;
* Возможность собрать команду профессионалов высшего уровня.

Годы напряженной работы и сотни успешно реализованных программных проектов с постоянно растущей высококлассной командой инженеров-разработчиков позволяют компании создавать качественные цифровые решения для предприятий любого масштаба и отрасли, а также достигнуть следующего:

* Более 200 завершённых проектов;
* Создано программное обеспечение для 25 отраслей;
* 25 миллионов пользователей;
* 95% клиентов продолжают сотрудничество.

*Modsen* занимается:

* Фронтенд разработкой;
* Бэкенд-разработка;
* Услугами DevOps;
* Мобильной разработкой.



Рисунок 1.1 – Логотип компании ООО «Модсен»

* 1. Охрана труда и техника безопасности на рабочем месте

**программиста**

Техника безопасности на предприятии – это совокупность мероприятий организационного и технического характера, которые направлены на предотвращение на производстве несчастных случаев и на формирование безопасных условий труда.

С целью обеспечения охраны труда на всевозможных предприятиях, прикладываются все усилия для того, чтобы сделать труд работающих людей безопасным, а как итог, большие средства выделяются именно для осуществления этих целей.

Кроме того, специальная служба безопасности контролирует уровень безопасности технике на производстве, ее состояние, а также следит за тем, чтобы абсолютно все принимаемые на предприятие рабочие, обучались безопасным приемам работы.

Вводный инструктаж проводит инженер по технике безопасности или лицо, на которое приказом руководителя возложена работа по охране труда и технике безопасности. Инструктаж проводят со всеми принимаемыми на работу независимо от их образования, стажа работы по данной профессии или должности, а также с командированными, учащимися и студентами, прибывшими на практику.

Первичный инструктаж на рабочем месте проводит руководитель того подразделения, в котором предстоит работать данному работнику. Такой инструктаж проводят с каждым работником индивидуально с практическим показом безопасных приемов и методов труда.

Охрана труда – система обеспечения безопасности жизни и здоровья работающих в процессе трудовой деятельности, включающая правовые, социально-экономические, организационные, технические, психофизиологические, санитарно-гигиенические, лечебно-профилактические, реабилитационные и иные мероприятия и средства [2].

Руководством компании уделяется большое внимание улучшению эргономики рабочих мест, обеспечению гигиены и совершенствованию организации труда, регламентации режимов труда и отдыха.

Вся деятельность в области охраны труда на предприятии регламентирована действующим законодательством Республики Беларусь, санитарными нормами и правилами, гигиеническими нормативами, предписаниями надзорных органов.

Важным фактором создания безопасных условий труда является оптимизация организации рабочих мест. Рабочее место, хорошо приспособленное к трудовой деятельности инженера, правильно и целесообразно организованное в отношении пространства, формы, размера, обеспечивает ему удобное положение при работе и высокую производительность труда при наименьшем физическом и психическом напряжении.

Рабочее место для выполнения работ в положении сидя организуется в соответствии с ГОСТ 12.2.032-78 «Система стандартов безопасности труда. Рабочее место при выполнении работ сидя. Общие эргономические требования» [3].

Согласно ГОСТ 12.2.032-78, конструкция рабочего места и взаимное расположение всех его элементов должно соответствовать антропометрическим, физическим и психологическим требованиям. Большое значение имеет также характер работы. В частности, при организации рабочего места программиста должны быть соблюдены следующие основные условия:

* оптимальное размещение оборудования, входящего и состав рабочего места;
* достаточное рабочее пространство, позволяющее осуществлять все необходимые движения и перемещения;
* уровень акустического шума не должен превышать допустимого значения.

Рабочая поза сидя вызывает минимальное утомление программиста. Рациональная планировка рабочего места предусматривает четкий порядок и постоянство размещения предметов, средств труда и документации. То, что требуется для выполнения работ чаще, расположено в зоне легкой досягаемости рабочего пространства.

Помещения для работы программиста должны иметь естественное и искусственное освещение.

Искусственное освещение в помещениях эксплуатации ВДТ и ПЭВМ должно осуществляться системой общего равномерного освещения. В административно-общественных помещениях, в случаях преимущественной работы с документами, допускается применение системы комбинированного освещения (к общему освещению дополнительно устанавливаются светильники местного освещения, предназначенные для освещения зоны расположения документов).

Освещенность на поверхности стола в тоне размещения рабочего документа должна быть 300-500 лк. Местное освещение не должно создавать бликов на поверхности экрана и увеличивать освещенность экрана более 300 лк.

В качестве источников света при искусственном освещении должны применяться преимущественно люминесцентные лампы.

Конструкция рабочего стола должна обеспечивать оптимальное размещение на рабочей поверхности используемого оборудования с учетом его количества и конструктивных особенностей (размер ВДТ и ПЭВМ, клавиатуры и др.), характера выполняемой работы. При этом допускается использование рабочих столов различных конструкций, отвечающих современным требованиям эргономики.

Конструкция рабочего стула (кресла) должна обеспечивать поддержание рациональной рабочей позы при работе, позволять изменять позу с целью снижения статического напряжения мышц шейно-плечевой области и спины для предупреждения развития утомления.

Тип рабочего стула (кресла) должен выбираться в зависимости от характера и продолжительности работы.

Рабочий стул (кресло) должен быть подъемно-поворотным и регулируемым по высоте и углам наклона сиденья и спинки, а также расстоянию спинки от переднего края сиденья, при этом регулировка каждого параметра должна быть независимой, легко осуществляемой и иметь надежную фиксацию.

Экран видеомонитора должен находиться от глаз пользователя на оптимальном расстоянии 600-700 мм, но не ближе 500 мм с учетом размеров алфавитно-цифровых знаков и символов.

Высота рабочей поверхности стола должна регулироваться в пределах 680-800 мм, при отсутствии такой возможности высота рабочей поверхности стола должна составлять 725 мм.

Рабочий стол должен иметь пространство для ног высотой не менее 600 мм, шириной – не менее 500 мм, глубиной на уровне колен – не менее 450 мм и на уровне вытянутых ног – не менее 650 мм.

Клавиатуру следует располагать на поверхности стола на расстоянии не менее чем 300 мм от края, обращенного к пользователю или на специальной, регулируемой по высоте рабочей поверхности, отделенной от основной столешницы.

Помимо требований к организации рабочего места СанПиН 9-131 РБ 2000 устанавливает требования к микроклимату рабочей зоны: влажности, температуре, скорости потока воздуха и пр.

Техника безопасности – это система организационных мероприятий и технических средств, предотвращающих воздействие на работающих опасных производственных факторов.

# ОБЗОР ТЕХНОЛОГИИ *MERN* И *REST API*

* 1. Технология *MERN*

Стек – это сочетание технологий, используемых для создания веб-приложения. Любое веб-приложение будет создано с использованием нескольких технологий (фреймворки, библиотеки, базы данных и т. Д.).

Стек *MERN* – это *JavaScript*-стек, разработанный для упрощения процесса разработки. MERN включает в себя четыре компонента с открытым исходным кодом: *MongoDB*, *Express*, *React* и *Node.js.* Эти компоненты обеспечивают комплексную среду для работы разработчиков.

Подробно рассмотрим каждый из них:

* *MongoDB*: кроссплатформенная база данных документов. *MongoDBNoSQL* (нереляционная) документно-ориентированная база данных. В то время как традиционные реляционные базы данных имеют типичный дизайн схемы, основанный на столбцах и таблицах, *MongoDB* не требует схем. Данные хранятся в гибких документах с помощью языка запросов на основе *JSON* (*JavaScript Object Notation*). Содержание, размер и количество полей в документах могут отличаться от одного к другому. Это означает, что структура данных будет меняться со временем. *MongoDB* известен своей гибкостью и простотой масштабирования;
* *Express*: базовая платформа веб-приложений. *Express* является структурой веб-приложения для *Node.js*, другого компонента *MERN*. Вместо того, чтобы писать полный код веб-сервера вручную на *Node.js*, разработчики используют *Express*, чтобы упростить задачу написания кода сервера. Нет необходимости повторять один и тот же код снова и снова, как это было бы с *HTTP*-модулем *Node.js*. Платформа *Express* предназначена для создания надежных веб-приложений и *API*. Он известен своей быстрой скоростью и минималистской структурой, а многие функции доступны в виде плагинов;
* *React*: библиотека *JavaScript* для создания пользовательских интерфейсов. Первоначально был создан инженером-программистом в *Facebook*, а позже был открыт. Он поддерживается *Facebook*, а также сообществом разработчиков и отдельных разработчиков. Библиотека *React* может использоваться для создания представлений, отображаемых в *HTML*. Реактивные взгляды носят декларативный характер. Это означает, что разработчикам не нужно беспокоиться об управлении последствиями изменений состояния представления (объекта, определяющего поведение компонентов) или изменений в данных. Вместо того чтобы полагаться на шаблоны для автоматизации создания повторяющихся элементов *HTML* или *DOM* (объектная модель документа), *React* использует полнофункциональный язык программирования (*JavaScript*) для создания повторяющихся или условных элементов *DOM*. С *React* один и тот же код может выполняться как на сервере, так и в браузере. *React* якоря стека *MERN*. В каком-то смысле это определяющая особенность стека. Это один из компонентов, который отличает *MERN* от *MEAN*, другого популярного стека *JavaScript*, в котором вместо библиотеки *React* используется *AngularJS* (среда интерфейсного веб-приложения).

Также для тестирование используется *cypress*. *Cypress* – это *open-source* фреймфорк для *E2E* тестирования. Это также как и *Puppeteer* относительно молодой инструмент, однако он вносит новые концепции и решения в способы осуществления автоматизации и тестирования. Ключевой особенностью, *Cypress* является то, что он исполняется внутри самого браузера. Это в том числе означает, что *Cypress* всегда отслеживает моменты вызова всякого рода событий в браузере и никогда не упустит любые манипуляции с элементами страницы, что намного уменьшает вероятность появления *floating* тестов.

Достоинства:

* Встроенный набор инструментов для тестирования построенный на форке *mocha, chai, sinon;*
* Встроенный механизм автоматического ожидания;
* Это собственно означает, что при написании сценарием нет необходимости писать *async*/*await* функции как это делается в *Puppeteer* и *Selenium*. *Cypress* сам подождет когда появится нужный элемент, подождет когда закончится анимация, и подождет когда очередной сетевой запрос завершится;
* *Time machine* фича, которая позволяет в *Сypress test runner* откатываться на определенные шаги в последовательности выполнения теста;
* Исчерпывающая документация с большим набором примеров;
* Возможность написания в том числе и *unit* тестов.
  1. Технология *REST API*

*REST API* — это способ взаимодействия сайтов и веб-приложений с сервером. Его также называют *RESTful*.

Термин состоит из двух аббревиатур, которые расшифровываются следующим образом. *API* (*Application Programming Interface*) — это код, который позволяет двум приложениям обмениваться данными с сервера. На русском языке его принято называть программным интерфейсом приложения. *REST* (*Representational State Transfer*) — это способ создания *API* с помощью протокола *HTTP*. На русском его называют «передачей состояния представления».

Технологию *REST API* применяют везде, где пользователю сайта или веб-приложения нужно предоставить данные с сервера. Например, при нажатии иконки с видео на видеохостинге *REST API* проводит операции и запускает ролик с сервера в браузере. В настоящее время это самый распространенный способ организации API. Он вытеснил ранее популярные способы *SOAP* и *WSDL*.

У *RESTful* нет единого стандарта работы: его называют «архитектурным стилем» для операций по работе с серверов. Такой подход в 2000 году в своей диссертации ввел программист и исследователь Рой Филдинг, один из создателей протокола *HTTP*.

*2.2.1* Принципы RESTAPI. У *RESTful* есть 7 принципов написания кода интерфейсов.

Отделения клиента от сервера (*Client-Server*). Клиент — это пользовательский интерфейс сайта или приложения, например, поисковая строка видеохостинга. В *REST API* код запросов остается на стороне клиента, а код для доступа к данным — на стороне сервера. Это упрощает организацию *API*, позволяет легко переносить пользовательский интерфейс на другую платформу и дает возможность лучше масштабировать серверное хранение данных.

Отсутствие записи состояния клиента (*Stateless*). Сервер не должен хранить информацию о состоянии (проведенных операций) клиента. Каждый запрос от клиента должен содержать только ту информацию, которая нужна для получения данных от сервера.

Кэшируемость (*Casheable*). В данных запроса должно быть указано, нужно ли кэшировать данные (сохранять в специальном буфере для частых запросов). Если такое указание есть, клиент получит право обращаться к этому буферу при необходимости.

Единство интерфейса (*Uniform Interface*). Все данные должны запрашиваться через один *URL*-адрес стандартными протоколами, например, *HTTP*. Это упрощает архитектуру сайта или приложения и делает взаимодействие с сервером понятнее.

Многоуровневость системы (*Layered System*). В *RESTful* сервера могут располагаться на разных уровнях, при этом каждый сервер взаимодействует только с ближайшими уровнями и не связан запросами с другими.

Предоставление кода по запросу (*Code on Demand*). Серверы могут отправлять клиенту код (например, скрипт для запуска видео). Так общий код приложения или сайта становится сложнее только при необходимости.

Начало от нуля (*Starting with the Null Style*). Клиент знает только одну точку входа на сервер. Дальнейшие возможности по взаимодействию обеспечиваются сервером.

*2.2.2* Стандарты. Сам по себе *RESTful* не является стандартом или протоколом. Разработчики руководствуются принципами *REST* *API* для создания эффективной работы с серверов для своих сайтов и приложений. Принципы позволяют выстраивать серверную архитектуру с помощью других протоколов: *HTTP*, *URL*, *JSON* и *XML*.

Это отличает *REST* *API* от метода простого протокола доступа к объектам *SOAP* (*Simple Object Access Protocol*), созданного *Microsoft* в 1998 году. В *SOAP* взаимодействие по каждому протоколу нужно прописывать отдельно только в формате *XML*. Также в *SOAP* нет кэшируемости запросов, более объемная документация и реализация словаря, отдельного от *HTTP*. Это делает стиль *REST* *API* более легким в реализации, чем стандарт *SOAP*.

Несмотря на отсутствие стандартов, при создании *REST* *API* есть общепринятые лучшие практики, например:

* использование защищенного протокола *HTTPS*;
* использование инструментов для разработки *API* *Blueprint* и *Swagger*;
* применение приложения для тестирования *Get Postman*;
* применение как можно большего количества *HTTP*-кодов (список) ;
* архивирование больших блоков данных.

*2.2.3* Архитектура. *REST API* основывается на протоколе передачи гипертекста *HTTP* (*Hypertext Transfer Protocol*). Это стандартный протокол в интернете, созданный для передачи гипертекста. Сейчас с помощью *HTTP* отправляют любые другие типы данных.

Каждый объект на сервере в *HTTP* имеет свой уникальный *URL*-адрес в строгом последовательном формате. Например, второй модуль обучающего видео про *JS* будет храниться на сервере по адресу http://lesson/JS/2.

В *REST API* есть 4 метода *HTTP*, которые используют для действий с объектами на серверах:

* *GET* (получение информации о данных или списка объектов)
* *DELETE* (удаление данных)
* *POST* (добавление или замена данных)
* *PUT* (регулярное обновление данных)

Такие запросы еще называют идентификаторами *CRUD*: *create* (создать), *read* (прочесть), *update* (обновить) *delete* (удалить). Это стандартный набор действий для работы с данными. Например, чтобы обновить видео про *JS* по адресу http://lesson/JS/2 *REST API* будет использовать метод *PUT*, а для его удаления — *DELETE*.

В каждом *HTTP*-запросе есть заголовок, за которым следует описание объекта на сервере — это и есть его состояние.

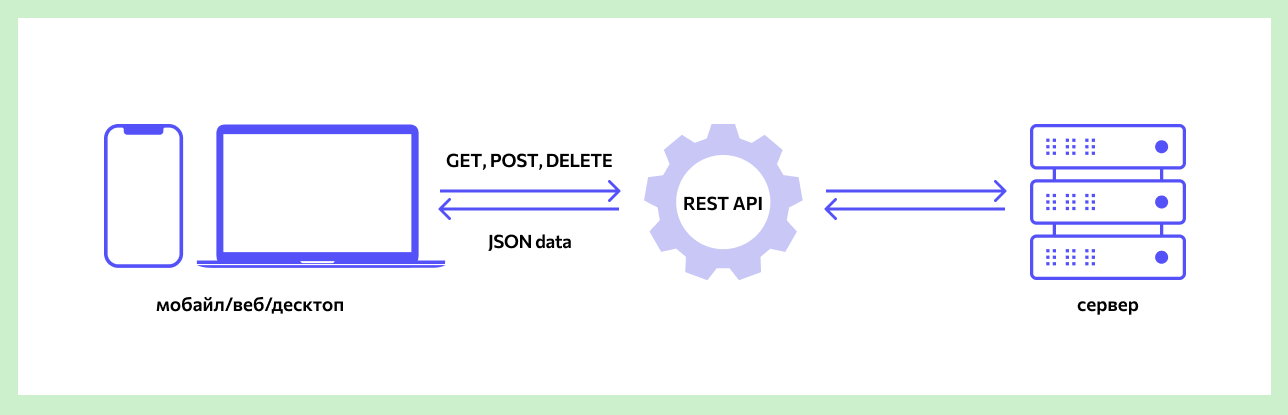


Рисунок 2.1 – Схема работы *REST API*

*2.2.4* Как работает *RESTful*. Например, на сайте такси есть система выбора типа машины: эконом, стандарт и люкс. В *REST API* каждому типу будет присвоен свой *URL* на странице бронирования:

* *http://taxi.com/order/econom*
* *http:// taxi.com / order /standart*
* *http:// taxi.com / order /lux*

Такие *URL* однозначно определяют ресурс на сервисе — данные о доступных такси каждого класса. Чтобы взаимодействовать с этими ресурсам *REST API* применяет *CRUD*-команды протокола *HTTP*. Например, *GET* *econom* для передачи клиенту информации о такси класса эконом. В *RESTful* такие запросы будут кэшироваться — клиенту не нужно обращаться к серверу снова при повторном запросе.

Также такая архитектура позволяет расставить приоритеты в обслуживании. Например, использование более производительных серверов для запросов на такси класса люкс. Такую архитектуру легко масштабировать: при появлении новых классов такси, система будет обращаться напрямуя к ресурсам по новым *URL*.

*2.2.5* Гдеприменяют. REST API рекомендуют использовать в следующих случаях:

* ограниченная пропускная способность соединения с сервром
* есть необходимость кэшировать запросы
* приложение или сайт будет значительно масштабироваться
* приложение или сайт использует *AJAX* (метод фонового обмена данными с сервером)

*REST API* используют чаще альтернативных методов, например *SOAP*. Помимо сайтов и веб-приложений *RESTful* используют для облачных вычислений.

# РАЗРАБОТКА *WEB API*

* 1. Постановка задачи

Задачей на данную работу являлось написание *web*-сервиса с открытым *API*, который позволяет отдавать данные в отличном от *HTML* формате, благодаря чему им удобно пользоваться при написании других приложений.

Для реализации необходимо использовать *Express* – это фреймворк *Node.js*, а в качестве базы данных необходимо использовать *MongoDb* – это документная база данных.

Тестировать приложение можно с помощью *Postman*, это открытая *API*, созданная *Swashbuckle* для того, чтобы программистам бэк-энда не требовалось изучать фронт-энд, чтобы продемонстрировать работу своего кода.

Уровень представления реак компонента

* 1. Модель данных

В результате проектирования необходимо получить базу данных, которая сможет обеспечить хранение всех необходимых данных для корректной работы приложения.

Как было оговорено выше база данных является документной.

Документная база данных – это тип нереляционных баз данных, предназначенный для хранения и запроса данных в виде документов в формате, подобном *JSON*. Документные базы данных позволяют разработчикам хранить и запрашивать данные в БД с помощью той же документной модели, которую они используют в коде приложения. Гибкий, полуструктурированный, иерархический характер документов и документных баз данных позволяет им развиваться в соответствии с потребностями приложений. Документные базы данных обеспечивают гибкость индексации, производительность выполнения стандартных запросов и аналитику наборов документов.

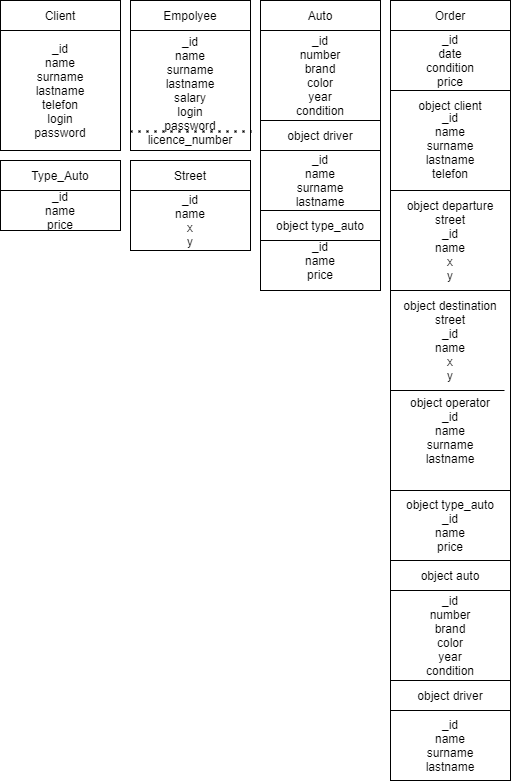


Рисунок 3.1 – Общая схема базы данных *MongoDb*

Данная схема отражает модель базы данных, включающую в себя шесть коллекций: *Clients, Employee, Auto, Orders, Type\_Auto, Street.* В коллекциях содержатся документы, которые в свою очередь содержат вложенные документы.

Коллекция *Orders* предназначена для хранения заказов. Информация о полях представлена в таблице 3.1.

Таблица 3.1 – Информация о полях коллекции *Orders*

|  |  |  |
| --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение |
| *\_id* | *ObjectId* | Хранение уникального номера. |
| *date* | *string* | Хранение даты заказа |
| *condition* | *int32* | Хранение состояния заказа |
| *price* | *double* | Хранение цены заказа |
| *\_id* | *ObjectId* | Хранение уникального номера клиента. |
| *surname* | *string* | Хранение фамилии клиента. |
| *name* | *string* | Хранение имени клиента. |
| *lastname* | *string* | Хранение отчества клиента. |
| *telefon* | *string* | Хранение номера телефона клиента. |
| *name* | *string* | Хранение названия улицы отправления. |
| *x* | *int32* | Хранение расположения улицы отправления по координате x. |
| *y* | *int32* | Хранение расположения улицы отправления по координате y. |
| *name* | *string* | Хранение названия улицы назначения. |
| *x* | *int32* | Хранение расположения улицы назначения по координате x. |
| *y* | *int32* | Хранение расположения улицы назначения по координате y. |
| *\_id* | *ObjectId* | Хранение уникального номера оператора. |
| *surname* | *string* | Хранение фамилии оператора. |
| *name* | *string* | Хранение имени оператора. |
| *lastname* | *string* | Хранение отчества оператора. |
| *name* | *string* | Хранение названия типа. |
| *price* | *double* | Хранение цены за км. |
| *\_id* | *ObjectId* | Хранение уникального номера. |
| *number* | *string* | Хранение номера машины. |
| *brand* | *string* | Хранение марки машины. |

Продолжение таблицы 3.1

|  |  |  |
| --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение |
| *color* | *string* | Хранение цвета машины. |
| *year* | *int32* | Хранение года выпуска машины. |
| *condition* | *int32* | Хранение состояния машины. |
| *\_id* | *ObjectId* | Хранение уникального номера водителя. |
| *surname* | *string* | Хранение фамилии |
| *name* | *string* | Хранение имени |
| *lastname* | *string* | Хранение отчества |

Коллекция «*Employee*» предназначена для хранения данных о сотрудниках. Информация о полях представлена в таблице 3.2.

Таблица 3.2 – Информация о полях коллекции *Employee*

|  |  |  |
| --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение |
| *\_id* | *ObjectId* | Хранение уникального номера. |
| *surname* | *string* | Хранение фамилии. |
| *name* | *string* | Хранение имени. |
| *lastname* | *string* | Хранение отчества. |
| *salary* | *double* | Хранение зарплаты. |
| *login* | *string* | Хранение логина. |
| *password* | *string* | Хранение пароля. |
| *type* | *string* | Хранение должности. |
| *license\_number* | *int32* | Хранение номера водительских прав для водителя. |

Коллекция «*Client*» предназначена для хранения данных о клиентах сервиса. Информация о полях представлена в таблице 3.3.

Таблица 3.3– Информация о полях коллекции *Client*

|  |  |  |
| --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение |
| *\_id* | *ObjectId* | Хранение уникального номера. |
| *surname* | *string* | Хранение фамилии клиента |
| *name* | *string* | Хранение имени клиента |
| *lastname* | *string* | Хранение отчества клиента |
| *telefon* | *string* | Хранение номера телефона |
| *password* | *string* | Хранение пароля клиента |
| *login* | *string* | Хранение логина клиента |

Коллекция «*Type\_Auto*» предназначена для хранения типов автомобилей. Информация о полях представлена в таблице 3.4.

Таблица 3.4 – Информация о полях коллекции *Type\_Auto*

|  |  |  |
| --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение |
| *\_id* | *ObjectId* | Хранение уникального номера. |
| *name* | *string* | Хранение названия типа. |
| *price* | *double* | Хранение цены за км. |

Коллекция «*Street*» предназначена для хранения информации о улицах на которых работает такси. Информация о полях представлена в таблице 3.5.

Таблица 3.5 – Информация о полях коллекции *Street*

|  |  |  |
| --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение |
| *\_id* | *ObjectId* | Хранение уникального номера. |
| *name* | *string* | Хранение названия улицы. |
| *x* | *int32* | Хранение координаты x. |
| *y* | *int32* | Хранение координаты y. |

Коллекция «*Auto*» предназначена для хранения информации об автомобилях. Информация о полях представлена в таблице 3.6.

Таблица 3.6 – Информация о полях коллекции *Auto*

|  |  |  |
| --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение |
| *\_id* | *ObjectId* | Хранение уникального номера. |
| *number* | *string* | Хранение номера машины. |
| *brand* | *string* | Хранение марки машины. |
| *color* | *string* | Хранение цвета машины. |
| *year* | *int32* | Хранение года выпуска машины. |
| *condition* | *int32* | Хранение состояния машины. |
| *\_id* | *ObjectId* | Хранение уникального номера типа. |
| *name* | *string* | Хранение названия типа. |
| *price* | *double* | Хранение цены за км. |
| *\_id* | *ObjectId* | Хранение уникального номера водителя. |
| *surname* | *string* | Хранение фамилии |
| *name* | *string* | Хранение имени |
| *lastname* | *string* | Хранение отчества |

* 1. Уровень доступа к данным

Уровень доступа к данным (или же *Data Access Layer*) хранит модели, описывающие используемые сущности, также здесь размещаются специфичные классы для работы с разными технологиями доступа к данным.

Путем анализа предметной области можно выделить следующие модели, соответствующие сущностям базы данных, описание которых представлено в таблице 3.7.

Таблица 3.7 – Описание доменных классов

|  |  |
| --- | --- |
| Исходное название класса | Назначение класса |
| *Street* | Хранение названия и расположение улицы. |
| *Employee* | Хранение данных о сотрудниках. |
| *Auto* | Хранение информации об автомобилях в автопарке. |
| *Client* | Хранение данных о клиентах. |
| *Auto\_Type* | Хранение название и расценку видов автомобилей. |
| *Order* | Хранение информации сделанных клиентами заказах. |

Каждой таблице из базы данных соответствует один класс модели из библиотеки *Mongoose,* который описывает операции *CRUD* для доступа к базе данных, из этого следует что для каждой вышеописанной сущности должен существовать класс модели, который будет реализовывать *CRUD* операции.

Реализация классов доступа к базе данных находится на *DataAccess*, содержащий классы типа *Entity*, *Model*.

Классы *Entitys* полностью повторяют таблицы из базы данных и служат для хранения информации из таблиц на слое *DataAccess*.

Классы *Models* предназначены для обращения к базе данных от соответствующих классов *DTO*, реализуют интерфейс *IModel<T>*. Описание интерфейса представлено в таблице 3.8.

Таблица 3.8 – Описание интерфейса *IModel*

|  |  |
| --- | --- |
| Название метода | Назначение |
| *findAll* | Обращение к указанной таблице и возврат всех существующих данных |
| *findOne* | Обращение к элементу по его фильтру и возврат данных |
| *Create* | Создание новой записи в таблице |
| *Update* | Обновление информации о записи в таблице из базы данных |
| *Delete* | Удаление записи из таблицы базы данных |

Интерфейсы и классы реализации пользовательских функций располагаются в слое *BusinessLogic*, они необходимы для обработки данных полученных с пользовательского интерфейса. Так, нагрузка на приложение будет разделена на серверную и пользовательскую часть, где на пользовательской части будут создаваться запросы, которые будут отправляться на сервер, где они и будут обрабатываться.

Классы *DTOs* повторяют классы *Enities* на слое *DataAccess*.

Классы *Routers* реализуют методы для обработки пользовательских запросов, в данном случае это обработчики *clientRouter, streetRouter, driverRouter, operatorRouter, commentRouter, signInRouter, signUpRouter*, реализующие методы необходимые для выполнения поставленных задач.

Рассмотрим *clientRouter* из *Routers*, он отвечает за методы, необходимые для работы с заказами, в том числе оформление, отмена и просмотр. Реализуемые классом методы представлены в таблице 3.9.

Таблица 3.9 – Описание методов, реализуемых *clientRouter*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название | Передаваемые значения | Возвращаемые значения | Описание метода | |
| *getOrdersByClientId* | *int, boolean* | *Order[]* | Обращение к модели заказов для получения всех либо текущих заказов, оформленных клиентом | |
| *createOrder* | *Int, string, syring, string* | *void* | | Обращение к модели заказов для создания заказа |
| *payOrder* | *int* | *Void* | | Обращение к модели заказов для оплаты заказа |

Обработчик *driverRouter* работает с запросами роли водитель, описание которых представлено в таблице 3.10.

Таблица 3.10 – Описание методов, реализуемых *driverRouter*

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Передаваемые значения | Возвращаемые значения | Описание метода |
| *getOrdersByDriverId* | *String, int* | *Order[]* | Обращение к модели заказов для получения заказов c нужным состоянием для курьера по *id* |
| *answerOrder* | *String,int* | *void* | Обращение к модели заказов для отказа или выполнения заказа |

Обработчик *operatorRouter* работает с запросами роли оператора, описание которых представлено в таблице 3.11.

Таблица 3.11 – Описание методов, реализуемых *operatorRouter*

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Передаваемые значения | Возвращаемые значения | Описание метода |
| *getOrdersByState* | int | *Order[]* | Обращение к модели заказов для получения заказов c нужным состоянием |
| *editDriver* | *String, string* | *void* | Обращение к модели заказов для назначения водителя на выполнения |

Обработчик *streetRouter* работает с запросами, связанными с каталогом улиц, описание которых представлено в таблице 3.12.

Таблица 3.12 – Описание методов, реализуемых *streetRouter*

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Передаваемые значения | Возвращаемые значения | Описание метода |
| *addStreet* | *String, int, int* | *Void* | Обращение к модели улиц для создания новой улицы |
| *updateStreet* | *String, int, int* | *Void* | Обращение к модели улиц для обновления данных |
| *getStreet* | *-* | *Street[]* | Обращение к модели улиц для возращения всех улиц |
| *deleteStreet* | *string* | *Void* | Ображение к модели улиц для удаления выбранной улицы |

Обработчик *commentsRouter* работает с запросами, связанными с комментариями, описание которых представлено в таблице 3.13.

Таблица 3.13 – Описание методов, реализуемых *commentsRouter*

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Передаваемые значения | Возвращаемые значения | Описание метода |
| *getAll* | – | *Comment[]* | Обращение к модели комментариев для получения всех комментариев |

Продолжение таблицы 3.13

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Передаваемые значения | Возвращаемые значения | Описание метода |
| *addOne* | *String, string* | *Comment* | Обращение к модели комментариев для добавления комментария |

Обработчик *signInRouter* работает с запросами, связанными со входом, описание запросов представлено в таблице 3.14.

Таблица 3.14 – Описание методов, реализуемых *signInRouter*

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Передаваемые значения | Возвращаемые значения | Описание метода |
| *trySignIn* | *string, string* | *Account || null* | Обращение к моделям клиентов, менеджеров и курьеров для получения профиля по заданным логину и паролю |

Обработчик *signUpRouter* работает с запросами, связанными со входом, описание запросов представлено в таблице 3.15.

Таблица 3.15 – Описание методов, реализуемых *signUpRouter*

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Передаваемые значения | Возвращаемые значения | Описание метода |
| *trySignUp* | *Client*  *& Account* | *(Account & Client) || null* | Создания новой учётной записи |

* 1. Уровень представления

*Presentation layer* (уровень представления) – это тот уровень, с которым непосредственно взаимодействует пользователь. Этот уровень включает компоненты пользовательского интерфейса, механизм получения ввода от пользователя. Применительно к *React*, который входит в стек *MERN*,на данном уровне расположены представления и все те компоненты, который составляют пользовательский интерфейс.

В данной работе для уровня представления используется *React*, который является библиотекой *JavaScript*. *React* использует компонентный подход вместе с композицией, который вместе дают простой инструмент для постройки простых и понятных интерфейсов.

После компиляции *React*, с помощью *babel*, он представляет собой простой *JavaScript* файл. Для такой компиляции *babel* необходима информация о проекте и всех зависимостях. Эта информация хранится в файле *package.json*.

Уровень представления связывается с другими уровнями посредством вызовов интерфейса прикладных программ (*API*).

* 1. Тестирование *Web API*

Правильно работающий сервис является основной задачей данного проекта, и для успешной работы нельзя обойтись без тестирования.

Для тестирования был выбран фреймворк *JavaScript* – *SuperTest*.

*Supertest* – это тестовый фреймворк *JavaScript*, который делает асинхронное тестирование простым и интересным. *Supertest* поставляется с множеством отличных функций, которые можно найти на веб-сайте. *Supertest* обеспечивает абстракцию высокого уровня для тестирования *HTTP*. Это необходимо, так как вы будете тестировать *API*.

Для тестирования была выбрана сущность улицы.

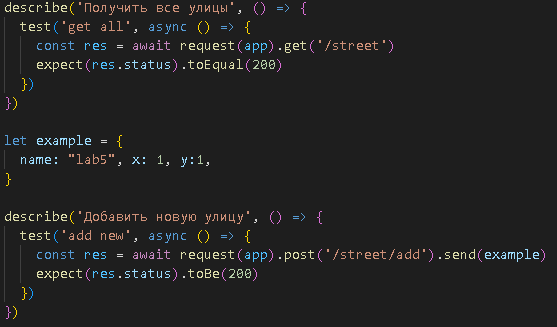


Рисунок 3.2 – Пример тестов

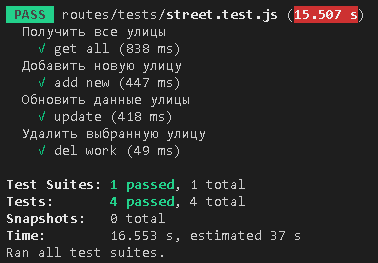


Рисунок 3.3 – Результат тестов

# ЗАКЛЮЧЕНИЕ

В процессе прохождения производственной практики, были приобретены необходимые практические умения и навыки работы, путём непосредственного участия в разработке. Во время её прохождения будущий программист применяет полученные в процессе обучения знания, умения и навыки на практике.

Основными задачами производственной практики являются:

* получения практического опыта работы в качестве программиста;
* улучшения качества профессиональной подготовки;
* закрепление полученных знаний по общим и специальным дисциплинам;
* проверка умения студентов пользоваться персональным компьютером.

Во время прохождения практики, мною были выполнены все задачи, которые были поставлены. Достигнута цель производственной практики, а именно, я овладел необходимыми компетенциями, систематизацией, обобщением и углубление теоретических знаний.

Данная практика является хорошим практическим опытом для дальнейшей самостоятельной деятельности. За время пройденной практики я познакомился с новыми интересными фактами. Закрепил свои теоретические знания, лучше ознакомился со своей профессией, а также данный опыт послужит хорошей ступенькой в моей дальнейшей карьерной лестнице.

# Список использованных источников

1. Официальный сайт ООО «Модсен» [Электронный ресурс]. Режим доступа: <https://modsen.com> – Дата доступа: 02.07.2022.
2. Юридический словарь [Электронный ресурс]. Режим доступа: http://multilang.pravo.by/ru/Term/Index/525?langName=ru&ch=Все&size=25&page=4&type=3 – Дата доступа: 02.07.2022.
3. Система стандартов безопасности труда. Рабочее место при выполнении работ сидя. Общие эргономические требования : ГОСТ 12.2.032-78. – Введ. 01.01.1979 – Москва : ИПК Издательство стандартов, 2001. – 27с.
4. Основы *REST*: теория и практика [Электронный ресурс]. Режим доступа: https://tproger.ru/articles/osnovy-rest-teorija-i-praktika/ – Дата доступа: 02.07.2021.

# ПРИЛОЖЕНИЕ А

(обязательное)

Листинг программы

**server.js**

const express = require('express');

const mongoose = require('mongoose')

const app = express();

const PORT = process.env.PORT || 5000

app.use(express.json({extended: true}))

app.use('/client', require('./routes/auth.route'))

app.use('/order', require('./routes/order.route'))

app.use('/type\_auto', require('./routes/type\_auto.route'))

app.use('/street', require('./routes/street.route'))

app.use('/autos', require('./routes/auto.route'))

app.use('/comment', require('./routes/client.route'))

async function start(){

try{

await mongoose.connect('mongodb://localhost:27017/Taxi', {

useUnifiedTopology: true,

useNewUrlParser: true

});

app.listen(PORT, ()=>{

console.log(`Server run in ${PORT}`)

})

}catch(err){console.log(err)}

}

start();

module.exports = app

**Auto.js**

const { Schema, model, Types } = require('mongoose');

const schema = new Schema({

\_id: { type: Types.ObjectId, required: false },

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

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

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

year: { type: Number, required: true },

condition: { type: Number, required: true },

type\_auto: {

\_id: { type: Types.ObjectId, required: false },

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

price: { type: Number, required: true }

},

driver: {

\_id: { type: Types.ObjectId, required: false },

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

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

lastname: { type: String, required: true }

}

}, { versionKey: false })

module.exports = model('Auto', schema)

**Client.js**

const {Schema, model, Types} = require('mongoose');

const schema = new Schema({

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

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

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

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

login: {type: String, required: true, unique: true},

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

comment: {type: String}

}, {versionKey: false})

module.exports = model('Client', schema)

**Employee.js**

const { Schema, model, Types } = require('mongoose');

const schema = new Schema({

\_id:{type: Types.ObjectId, required: false},

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

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

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

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

login: { type: String, required: true, unique: true },

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

licence\_number: { type: String }

}, { versionKey: false })

module.exports = model('Employee', schema)

**Order.js**

const { Schema, model, Types } = require('mongoose');

const schema = new Schema({

\_id: { type: Types.ObjectId, required: false },

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

condition: { type: Number, required: true },

price: { type: Number, required: false },

client: {

\_id: { type: Types.ObjectId, required: false },

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

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

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

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

},

departure\_street: {

\_id: { type: Types.ObjectId, required: false },

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

x: { type: Number, required: true },

y: { type: Number, required: true }

},

destination\_street: {

\_id: { type: Types.ObjectId, required: false },

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

x: { type: Number, required: true },

y: { type: Number, required: true }

},

operator: {type: Object,

\_id: { type: Types.ObjectId, required: false },

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

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

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

required: false

},

type\_auto: {

\_id: { type: Types.ObjectId, required: false },

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

price: { type: Number, required: true }

},

auto: {type: Object,

\_id: { type: Types.ObjectId, required: false },

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

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

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

year: { type: Number, required: true },

condition: { type: Number, required: true },

driver: {type: Object,

\_id: { type: Types.ObjectId, required: false },

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

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

lastname: { type: String, required: true }

},

required:false

}

})

module.exports = model('Order', schema)

**Street.js**

const { Schema, model, Types } = require('mongoose');

const schema = new Schema({

\_id:{type: Types.ObjectId, required: false},

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

x: { type: Number, required: true },

y: { type: Number, required: true }

}, { versionKey: false })

module.exports = model('Street', schema)

**Type\_Auto.js**

const { Schema, model, Types } = require('mongoose');

const schema = new Schema({

\_id:{type: Types.ObjectId, required: false},

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

price: { type: Number, required: true }

}, { versionKey: false })

module.exports = model('Type\_auto', schema)

**auth.route.js**

const { Router } = require('express');

const router = Router();

const Client = require('../models/Client')

const { check, validationResult } = require('express-validator')

const bcrypt = require('bcryptjs')

const jwt = require('jsonwebtoken')

const Employee = require('../models/Employee')

router.post('/registretion',

[

check('name', 'Некорректное имя').isLength({ min: 2 }),

check('surname', 'Некорректное surname').isLength({ min: 5 }),

check('lastname', 'Некорректное lastname').isLength({ min: 5 }),

check('login', 'Некорректное login').isLength({ min: 5 }),

check('password', 'Некорректное password').isLength({ min: 5 })

],

async (req, res) => {

try {

const errors = validationResult(req)

if (!errors.isEmpty()) {

return res.status(400).json({

errors: errors.array(),

message: 'Некорректные данные при регистрации'

})

}

const { name, surname, lastname, phone, login, password } = req.body;

const isUsed = await Client.findOne({ login })

if (isUsed) {

return res.status(300).json({ message: 'Данный логин занят' })

}

const hashPassword = await bcrypt.hash(password, 12)

const user = new Client({

name, surname, lastname, phone, login, password: hashPassword, comment: ''

})

await user.save();

res.status(201).json({ message: 'Учётная запись созданна' })

} catch (err) { console.log(err) }

})

router.post('/login',

[

check('login', 'Некорректное login').exists(),

check('password', 'Некорректное password').exists()

],

async (req, res) => {

try {

const errors = validationResult(req)

if (!errors.isEmpty()) {

return res.status(400).json({

errors: errors.array(),

message: 'Некорректные данные при регистрации'

})

}

const { login, password } = req.body;

const client = await Client.findOne({ login })

const employee = await Employee.findOne({ login })

if (!client && !employee) {

return res.status(400).json({ message: "Такого логина нет" })

} else {

if (!employee) {

const isMatch = bcrypt.compare(password, client.password)

if (!isMatch) {

return res.status(400).json({ message: 'Пароль не верный' })

}

const jwtSecret = 'fdsfds6f78dsfj96dsf54fas8a98778989ada55ccdsd5'

const token = jwt.sign(

{ clientId: client.id },

jwtSecret,

{ expiresIn: '1h' }

)

res.json({ token, clientId: client.id, operator: false, driver: false })

}

else {

const isMatch = (password === employee.password)

if (!isMatch) {

return res.status(400).json({ message: 'Пароль не верный' })

}

const jwtSecret = 'fdsfds6f78dsfj96dsf54fas8a98778989ada55ccdsd5'

const token = jwt.sign(

{ clientId: employee.id },

jwtSecret,

{ expiresIn: '1h' }

)

res.json({ token, clientId: employee.id, operator: !employee.licence\_number, driver: !!employee.licence\_number })

}

}

} catch (err) { console.log(err) }

})

module.exports = router;

**auto.route.js**

const { Router } = require('express')

const router = Router()

const Auto = require('../models/Auto')

const Order = require('../models/Order')

const Client = require('../models/Client')

const Street = require('../models/Street')

const Type\_auto = require('../models/Type\_auto')

const mongoose = require('mongoose');

router.get('/', async (req, res) => {

try {

let {type} = req.query

let autos = await Auto.find({'condition': 0, 'type\_auto.name': type})

res.json(autos)

} catch (error) {

console.log(error)

}

})

module.exports = router

**client.route.js**

const { Router } = require('express')

const router = Router()

const Client = require('../models/Client')

router.post('/update',

async (req, res) => {

try {

const { clientId, com } = req.body;

console.log('comment ', clientId, com)

const client = await Client.findOne({ '\_id': clientId })

if (!client) {

console.log('No')

return res.status(300).json({ message: 'Такого заказа нет' })

}

client.comment = com

await client.save()

res.status(200).json({ message: 'ЗАпись обновлена' })

} catch (err) { console.log(err) }

})

router.get('/', async (req, res) => {

try {

let client = await Client.find()

res.json(client)

} catch (error) {

console.log(error)

}

})

module.exports = router

**order.route.js**

const { Router } = require('express')

const router = Router()

const Order = require('../models/Order')

const Client = require('../models/Client')

const Street = require('../models/Street')

const Type\_auto = require('../models/Type\_auto')

const Auto = require('../models/Auto')

const mongoose = require('mongoose');

router.get('/', async (req, res) => {

try {

const { clientId, cond, isDriver, isOperator } = req.query

console.log('Просит по id ', clientId, ' state ', cond, ' driver ', isDriver, ' operator ', isOperator)

let order

if (isDriver === 'true') {

order = await Order.find({ 'auto.driver.\_id': mongoose.mongo.ObjectId(clientId), 'condition': cond })

console.log('Просит driver ')

}

else {

if (isOperator === 'true') {

order = await Order.find({ 'condition': cond })

console.log('Просит operator ')

}

else {

order = await Order.find({ 'client.\_id': clientId, 'condition': cond })

console.log('Просит client ')

}

}

console.log(order)

res.json(order)

} catch (error) {

console.log(error)

}

})

router.post('/add',

async (req, res) => {

try {

const { clientId, sendStreet1, sendStreet2, sendTypeAuto } = req.body;

console.log(clientId)

console.log(sendStreet1)

console.log(sendStreet2)

console.log(sendTypeAuto)

console.log('!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!')

const isUsed = await Order.findOne({ 'client.\_id': clientId, 'condition': 0 })

if (isUsed) {

return res.status(300).json({ message: 'У данного пользователя уже есть активный заказ' })

}

const client = await Client.findById(clientId)

console.log(client)

var date = new Date();

const order = new Order({

\_id: new mongoose.mongo.ObjectId(),

data: date.getFullYear.toString(),

condition: 0,

price: Math.sqrt((sendStreet1.x - sendStreet2.x) \*\* 2 + (sendStreet1.y - sendStreet2.y) \*\* 2) \* sendTypeAuto.price,

client: {

\_id: client.\_id,

name: client.name,

surname: client.surname,

lastname: client.lastname,

phone: client.phone,

},

departure\_street: {

\_id: sendStreet1.\_id,

name: sendStreet1.name,

x: sendStreet1.x,

y: sendStreet1.y

},

destination\_street: {

\_id: sendStreet2.\_id,

name: sendStreet2.name,

x: sendStreet2.x,

y: sendStreet2.y

},

type\_auto: {

\_id: sendTypeAuto.\_id,

name: sendTypeAuto.name,

price: sendTypeAuto.price

},

auto: null,

operator: null

})

await order.save();

res.status(201).json({ message: 'Учётная запись созданна' })

} catch (err) { console.log(err) }

})

router.post('/update/driver',

async (req, res) => {

try {

const { orderId, con } = req.body;

console.log(orderId)

const isOrder = await Order.findOne({ '\_id': orderId })

console.log(isOrder)

if (!isOrder) {

console.log('No')

return res.status(300).json({ message: 'Такого заказа нет' })

}

isOrder.condition = con

console.log('-------------------')

console.log(isOrder)

await isOrder.save()

res.status(201).json({ message: 'ЗАпись обновлена' })

} catch (err) { console.log(err) }

})

router.post('/update/operator',

async (req, res) => {

try {

const { orderId, number, con } = req.body;

console.log('!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!')

console.log(orderId, number)

if (con = 1) {

const isOrder = await Order.findOne({ '\_id': orderId })

const selectAuto = await Auto.findOne({ 'number': number })

console.log(isOrder)

if (!isOrder) {

console.log('No')

return res.status(300).json({ message: 'Такого заказа нет' })

}

isOrder.condition = 1

isOrder.auto = {

\_id: selectAuto.\_id,

number: selectAuto.number,

brand: selectAuto.brand,

color: selectAuto.color,

year: selectAuto.year,

condition: selectAuto.condition,

driver: {

\_id: selectAuto.driver.\_id,

name: selectAuto.driver.name,

surname: selectAuto.driver.surname,

lastname: selectAuto.driver.lastname

}

}

console.log('-------------------')

console.log(isOrder)

await isOrder.save()

}

else{

const isOrder = await Order.findOne({ '\_id': orderId })

console.log(isOrder)

if (!isOrder) {

console.log('No')

return res.status(300).json({ message: 'Такого заказа нет' })

}

isOrder.condition = -1

await isOrder.save()

}

res.status(201).json({ message: 'ЗАпись обновлена' })

} catch (err) { console.log(err) }

})

module.exports = router

**street.route.js**

const { Router } = require('express')

const router = Router()

const Street = require('../models/Street')

const mongoose = require('mongoose')

router.get('/', async (req, res) => {

try {

const streets = await Street.find()

res.status(200).json(streets)

} catch (error) {

console.log(error)

}

})

router.post('/add',

async (req, res) => {

try {

const { name, x, y } = req.body;

const street = new Street({

\_id: new mongoose.mongo.ObjectId(),

name,

x,

y,

})

await street.save();

res.status(200).json({ message: 'Учётная запись созданна' })

} catch (err) { console.log(err) }

})

router.post('/update',

async (req, res) => {

try {

const { streetId, name, x, y } = req.body;

console.log('update',streetId, name, x, y)

const street = await Street.findOne({ '\_id': streetId })

if (!street) {

console.log('No')

return res.status(300).json({ message: 'Такого заказа нет' })

}

street.name = name

street.x = x

street.y = y

await street.save()

res.status(200).json({ message: 'ЗАпись обновлена' })

} catch (err) { console.log(err) }

})

router.post('/delete',

async (req, res) => {

try {

const { streetId} = req.body;

console.log(streetId)

const street = await Street.deleteOne({'\_id': streetId})

res.status(200).json({ message: 'ЗАпись удалена' })

} catch (err) { console.log(err) }

})

module.exports = router

**type\_auto.route.js**

const {Router} = require('express')

const router = Router()

const Type\_auto = require('../models/Type\_auto')

router.get('/', async (req, res) => {

try {

const type\_auto = await Type\_auto.find()

res.json(type\_auto)

} catch (error) {

console.log(error)

}

})

module.exports = router

**App.js**

import './App.css';

import useRoutes from './Routes';

import { AuthContext } from './context/AuthContext';

import { useAuth } from './hooks/auth.hook';

function App() {

const { login, logout, token, clientId, isReady, isOperator, isDriver } = useAuth();

const isLogin = !!token;

const routes = useRoutes(isLogin, isOperator, isDriver);

return (

<AuthContext.Provider value={{login, logout, token, clientId, isReady, isLogin, isOperator, isDriver}}>

<div>

{routes}

</div>

</AuthContext.Provider>

);

}

export default App;

**Routes.js**

import { BrowserRouter, Routes, Route, Link } from 'react-router-dom'

import Main from './pages/Main/Main';

import Login from './pages/Login/Login'

import Registretion from './pages/Registretion/Registretion';

import ClientArea from './pages/ClientArea/ClientArea';

import Order from './pages/Order/Order';

import DriverArea from './pages/DriverArea/DriverArea';

import OperatorArea from './pages/OperatorArea/OperatorArea'

import Comments from './pages/Comments/Comments'

function useRoutes(isLogin, isOperator, isDriver) {

if (isLogin) {

if(isOperator){

}

if(isDriver){

return (

<Routes>

<Route path='/' element={<Main />}></Route>

<Route path='/driverroom' element={<DriverArea />}></Route>

</Routes>

)

}

if(isOperator){

return(

<Routes>

<Route path='/' element={<Main />}></Route>

<Route path='/operatorroom' element={<OperatorArea />}></Route>

</Routes>

)

}

return (

<Routes>

<Route path='/' element={<Main />}></Route>

<Route path='/clientroom' element={<ClientArea />}></Route>

<Route path='/order' element={<Order />}></Route>

<Route path='/comment' element={<Comments/>}></Route>

</Routes>

)

}

return (

<Routes>

<Route path='/' element={<Main />}></Route>

<Route path='/login' element={<Login />}> </Route>

<Route path='/registretion' element={<Registretion />}></Route>

</Routes>

)

}

export default useRoutes;

**Registretion.jsx**

import './Registretion.css'

import Header from '../../components/Header/Header'

import Footer from '../../components/Footer/Footer'

import { Link } from 'react-router-dom'

import { useState } from 'react'

import axios from "axios"

import Box from '@mui/material/Box';

import TextField from '@mui/material/TextField';

import Button from '@mui/material/Button';

function Registretion() {

const [form, setForm] = useState({

name: '',

surname: '',

lastname: '',

phone: '',

login: '',

password: ''

})

const changeHandler = (event) => {

setForm({ ...form, [event.target.name]: event.target.value });

console.log(form);

}

const registerHandler = async () => {

try {

await axios.post('/client/registretion', { ...form }, {

headers: {

'Content-Type': 'application/json'

}

}).then(res => console.log(res))

} catch (err) { console.log(err) }

}

return (

<div>

<Header />

<div className='main-registretion'>

<div className='img-registretion'>

<img />

</div>

<div className='form-registreiteon'>

<Box

component="form"

sx={{

'& > :not(style)': { m: 1, width: '25ch' },

}}

noValidate

autoComplete="off"

>

<TextField onChange={changeHandler} id="standard-basic" label="Введите имя" variant="standard" name='name' />

<TextField onChange={changeHandler} id="standard-basic" label="Введите фамилию" variant="standard" name='surname' />

<TextField onChange={changeHandler} id="standard-basic" label="Введите отчество" variant="standard" name='lastname' />

<TextField onChange={changeHandler} id="standard-basic" label="Введите телефон" variant="standard" name='phone' />

<TextField onChange={changeHandler} id="standard-basic" label="Введите логин" variant="standard" name='login' />

<TextField onChange={changeHandler} id="standard-basic" label="Введите пароль" variant="standard" name='password' />

<TextField onChange={changeHandler} id="standard-basic" label="Повторите пароль" variant="standard" name='password2' />

<Button onClick={registerHandler} variant="outlined">Зарегистрироваться</Button>

<Link to='/login'>Уже есть аккаунт</Link>

</Box>

</div>

</div>

<Footer />

</div>

)

}

export default Registretion;

**Login.jsx**

import './Login.css'

import Header from '../../components/Header/Header'

import Footer from '../../components/Footer/Footer'

import { Link } from 'react-router-dom'

import Box from '@mui/material/Box';

import TextField from '@mui/material/TextField';

import Button from '@mui/material/Button';

import axios from "axios"

import { useState, useContext } from 'react';

import { AuthContext } from '../../context/AuthContext';

function Login() {

const [form, setForm] = useState({

login: '',

password: ''

})

const {login} = useContext(AuthContext)

const changeHandler = (event) => {

setForm({ ...form, [event.target.name]: event.target.value });

console.log(form);

}

const loginHandler = async () => {

try {

await axios.post('/client/login', { ...form }, {

headers: {

'Content-Type': 'application/json'

}

}).then(res => {

login(res.data.token, res.data.clientId, res.data.operator, res.data.driver)

console.log(res.data)

})

} catch (err) { console.log(err) }

}

return (

<div>

<Header />

<div className='main-registretion'>

<div className='img-registretion'>

<img />

</div>

<div className='form-registreiteon'>

<Box

component="form"

sx={{

'& > :not(style)': { m: 1, width: '25ch' },

}}

noValidate

autoComplete="off"

>

<TextField onChange={changeHandler} id="standard-basic" label="Введите логин" variant="standard" name='login' />

<TextField onChange={changeHandler} id="standard-basic" label="Введите пароль" variant="standard" name='password' />

<Button onClick={loginHandler} variant="outlined">Войти</Button>

<Link to='/registretion'>Нет аккаунта</Link>

</Box>

</div>

</div>

<Footer />

</div>

)

}

export default Login;

**Order.jsx**

import React, { useCallback, useContext, useState, useEffect } from 'react';

import Header from '../../components/Header/Header'

import Footer from '../../components/Footer/Footer'

import './Order.css'

import Box from '@mui/material/Box';

import InputLabel from '@mui/material/InputLabel';

import MenuItem from '@mui/material/MenuItem';

import FormControl from '@mui/material/FormControl';

import Select from '@mui/material/Select';

import Button from '@mui/material/Button';

import axios from 'axios';

import { AuthContext } from '../../context/AuthContext'

function Order() {

const [fromWhere, setFromWhere] = React.useState('');

const [where, setWhere] = React.useState('');

const [type, setType] = React.useState('');

console.log(fromWhere)

console.log(where)

console.log(type)

const FromWhereHandleChange = (event) => {

setFromWhere(event.target.value);

};

const WhereHandleChange = (event) => {

setWhere(event.target.value);

};

const TypeHandleChange = (event) => {

setType(event.target.value);

};

const { clientId } = useContext(AuthContext)

const [type\_auto, setType\_auto] = useState([])

const [street, setStreet] = useState([])

const getStreet = useEffect(() => {

axios.get('/street', {

headers: {

'Content-Type': 'application/json'

}

})

.then((res) => setStreet(res.data))

.catch((error) => console.log(error))

}, [])

const getType\_auto = useEffect(() => {

axios.get('/type\_auto', {

headers: {

'Content-Type': 'application/json'

}

})

.then((res) => setType\_auto(res.data))

.catch((error) => console.log(error))

}, [])

const AddOrderHandler = async () => {

try {

const sendStreet1 = street.filter((item) => item.name === fromWhere)[0]

const sendStreet2 = street.filter((item) => item.name === where)[0]

const sendTypeAuto = type\_auto.filter((item) => item.name === type)[0]

await axios.post('/order/add', { clientId, sendStreet1, sendStreet2, sendTypeAuto }, {

headers: {

'Content-Type': 'application/json'

}

}).then(res => console.log(res))

} catch (err) { console.log(err) }

}

return (

<div>

<Header />

<FormControl variant="standard" sx={{ m: 1, minWidth: 120 }}>

<InputLabel id="demo-simple-select-standard-label">От куда</InputLabel>

<Select

labelId="demo-simple-select-standard-label"

id="demo-simple-select-standard"

value={fromWhere}

onChange={FromWhereHandleChange}

label="Age"

>

{street.filter((item) => item.name !== where)

.map((item, ind) =>

<MenuItem key={ind} value = {item.name}>{item.name}</MenuItem>)}

</Select>

</FormControl>

<FormControl variant="standard" sx={{ m: 1, minWidth: 120 }}>

<InputLabel id="demo-simple-select-standard-label">Куда</InputLabel>

<Select

labelId="demo-simple-select-standard-label"

id="demo-simple-select-standard"

value={where}

onChange={WhereHandleChange}

label="Age"

>

{street.filter((item) => item.name !== fromWhere)

.map((item, ind) =>

<MenuItem key={ind} value = {item.name}>{item.name}</MenuItem>)}

</Select>

</FormControl>

<FormControl variant="standard" sx={{ m: 1, minWidth: 120 }}>

<InputLabel id="demo-simple-select-standard-label">Тип авто</InputLabel>

<Select

labelId="demo-simple-select-standard-label"

id="demo-simple-select-standard"

value={type}

onChange={TypeHandleChange}

label="Age"

>

{type\_auto.map((item, ind) =>

<MenuItem key={ind} value = {item.name}>{item.name}</MenuItem>)}

</Select>

</FormControl>

<Button onClick={AddOrderHandler} variant="outlined">Заказать</Button>

<Footer />

</div>

)

}

export default Order;