МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО

Факультет автоматизированных и информационных систем

Кафедра «Информатика»

Отчёт по производственной (технологической) практике

на тему: «**WEB сервис по автоматизации службы доставки пиццы**»

|  |
| --- |
| Выполнил студент гр. ИП-32. |
| Суховенко Э. С. |
| Руководитель практики |
| от университета: Романькова Т. Л. |
| Руководитель практики |
| от предприятия: Семченко А. С. |

|  |  |
| --- | --- |
| Дата проверки:  Дата допуска к защите:  Дата защиты:  Оценка работы:  Подписи членов комиссии |  |
|  |
|  |
|  |
|  |
|  |

Гомель 2022

**СОДЕРЖАНИЕ**

|  |  |  |
| --- | --- | --- |
| Введение | | 3 |
| 1. Общие сведения об ООО «[ЛЕВЕРЕКС ИНТЕРНЕШНЛ](#_30j0zll)» | | 4 |
|  | 1.1 История организации | 4 |
|  | 1.2 Охрана труда и техника безопасности | 6 |
| 1. Обзор технологий | | 8 |
|  | 2.1 WEB разработка | 8 |
|  | 2.2 Технология REST API | 9 |
| 1. Разработка WEB API | | 12 |
|  | 3.1 Постановка задачи | 12 |
|  | 3.2 Используемые средства | 12 |
|  | 3.3 Модель данных | 14 |
|  | 3.4 Уровень доступа к данным | 16 |
|  | 3.5 Уровень представления | 19 |
| 1. Тестирование | | 21 |
|  | 4.1 Тестирование WEB API | 21 |
| Заключение | | 25 |
| Список используемых источников | | 26 |
| Приложение A | | 27 |

# ВВЕДЕНИЕ

Целью производственной практики является закрепление знаний по изучаемым дисциплинам и получение студентами практических навыков в период пребывания на предприятии (в организации).

Производственная практика – это обязательная составляющая образовательного процесса для каждого обучающегося технической профессии человека. Она необходима для подготовки квалифицированных работников, хорошо ориентирующихся не только в профильной теории, но и в реалиях трудовых будней. Этот этап обучения обычно осуществляется вне стен вуза – на базе учреждений, соответствующих будущей специальности студента.

Место практики: ООО «Леверекс Интернешнл».

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

*LeverX Group* – международная *IT*-компания, объединившая две команды: *LeverX* и *Emerline*. Сегодня здесь работают более 900 человек. Офисы компании размещены в Беларуси, США, Австрии, Польше, Украине, Латвии, Узбекистане, России [1].

*LeverX* является партнером немецкой компании *SAP*, разрабатывает и внедряет *SAP*-решения для таких крупных компаний, как *KIA Motors*, «KAMAЗ», *BMW*, *Home Credit Bank*, *PepsiCo* и других.

*Emerline* занимается разработкой *Web*- и *Mobile*-решений как для стартапов, так и для крупного бизнеса из Западной Европы и США. Кроме того, в портфеле компании есть собственные продукты: антистресс-раскраска, приложение для дрессировки собак и сканер документов.

Иностранное производственное унитарное предприятие «*LeverX Group*» г. Гомель создано в 2021 году. Учредитель – американское предприятие ООО «Леверекс Интернешнл», созданное в 2004 году. Основное направление деятельности: разработка программного обеспечения. *LeverX Group* уже более 17 лет является сертифицированным партнером немецкой компании *SAP*, а в 2020 году получила статус – *SAP Global Strategic Supplier for Technical Services*. Услуги компании оценили более 400 клиентов по всему миру. [1]. Логотип компании представлен на рисунке 1.1.



Рисунок 1.1 – Логотип компании ООО «Леверекс Интернешнл»

Также компания развивает отдельное направление разработки под брендом *Emerline*, создает собственные мобильные продукты и активно сотрудничает со стартапами. Особое внимание *Emerline* уделяет работе с инновационным стеком технологий, включая *IoT*, *AR*, *AI*, *Machine Learning* и *Data Science*.

*LeverX Group* объединяет две команды: *LeverX* и *Emerline*. Основная цель заключается в предоставлении полного цикла услуг по технологическому консалтингу, разработке и внедрению для предприятий для их успешной трансформации бизнеса. Глубокий опыт *LeverX* основан на успешном сотрудничестве с *SAP* в качестве Предпочтительного поставщика с 2004 года. Логотип команды *LeverX* показан на рисунке 1.2.



Рисунок 1.2 – Логотип команды *LeverX*

Темы привлечения талантов становятся ключевым направлением для того, чтобы быть ближе к существующим клиентам и помогать новым в решении их бизнес-задач и применении технологий. Компания *LeverX Group* проводит несколько инициатив по привлечению новых членов команды и сохранению нашей нынешней команды в:

1. налаживании сотрудничества с местными университетами
2. организацией мероприятий и встреч
3. оснащению своих офисов для организации комфортного и эффективного рабочего процесса.

Поскольку первой и главной целью компании *LeverX Group* является увеличение числа клиентов, необходима естественная миграция сотрудников между офисами. Тем не менее, эта тенденция продиктована потребностями бизнеса, такими как близость к клиентам, а не тенденциями переезда. Логотип команды *Emerline* показан на рисунке 1.3.



Рисунок 1.3 – Логотип команды *Emerline*

*LeverX Group* работает с компаниями и организациями для решения сложных задач современного делового мира. Каждый из наших партнеров является опытным, инновационным и зарекомендовавшим себя лидером в своих соответствующих областях.

* 1. Охрана труда и техника безопасности

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.

При правильной организации рабочего места производительность труда инженера возрастает с 8 до 20 процентов.

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.

# ОБЗОР ТЕХНОЛОГИЙ

* 1. WEB разработка

Прежде чем мы проведем обзор технологий, давайте разберемся, что такое веб-приложение и из чего оно состоит. *WEB*-приложение – это сайт с элементами интерактива. Они позволяют пользователям взаимодействовать с элементами на странице: нажимать кнопки, заполнять формы, запрашивать прайс, совершать покупки. Почтовые клиенты, соцсети, поисковики, интернет-магазины, программы для управления проектами – это всё примеры таких приложений.

C точки зрения архитектуры веб-приложения состоят из двух частей: клиентской и серверной. Клиентская часть также называется фронтэнд. По сути это то, что пользователи видят на экране устройства. Основные технологии:

* *HTML* – это стандартный язык разметки, который применяют для создания веб-проектов. Его элементы позволяют отображать стандартные блоки страниц, а также представляют форматированный текст, изображения, таблицы и формы ввода данных;
* каскадные таблицы стилей (*CSS*) – это язык разметки, который определяет оформление и макет элементов *HTML*. Таким образом, *HTML* задаёт структуру, а *CSS* – стиль. С помощью *CSS* задаются шрифты, цвета, стили, расположение отдельных элементов, а также отображение страниц на разных устройствах;
* *JavaScript* – это язык программирования, который помогает реализовывать сложное поведение веб-страницы;
* *ASP.NET* – технология создания веб-приложений и веб-сервисов от компании Microsoft. Она является составной частью платформы *Microsoft.NET* и развитием более старой технологии *Microsoft ASP*.

Популярные Фреймворки и библиотеки *JavaScript*:

* *React* – это библиотека с открытым кодом для создания пользовательских интерфейсов. С помощью *React* разработчики создают веб-приложения, которые изменяют отображение без перезагрузки страницы;
* *Angular* – это фреймворк от компании *Google*. Прежде всего он нацелен на разработку *SPA*-решений;
* *Vue* – это прогрессивный фреймворк для создания пользовательских интерфейсов. В отличие от фреймворков-монолитов, *Vue* подходит для постепенного внедрения. Он легко интегрируется с другими библиотеками и существующими проектами.

В *web*-приложениях также важна серверная часть. Под серверной частью понимают набор аппаратно-программных средств, с помощью которых реализована логика работы приложения. Это то, что происходит вне браузера и компьютера пользователя. К бэкенду относится панель администрирования, управление данными, логика их передачи по запросам фронтенда.

Задача серверной разработки – сделать так, чтобы ответ от сервера доходил до клиента и спроектированные блоки функционировали нужным образом. А также создать для заказчика удобную и безопасную среду для наполнения и обновления контента на сайте. Основные технологии:

* *Node.js* – кроссплатформенная среда, которая выполняет код *JavaScript* вне браузера. *Node.js* позволяет разработчикам использовать *JavaScript*, чтобы получить инструменты командной строки. На стороне сервера с его помощью можно запускать сценарии для обработки динамического содержимого веб-страницы, перед тем как она будет доступна в веб-браузере пользователя.
* *Express* – фреймворк *Node.js*. *Express* сам использует модуль *http*, но вместе с тем предоставляет ряд готовых абстракций, которые упрощают создание сервера и серверной логики, в частности, обработка отправленных форм, работа с куками, *CORS* и т.д.
  1. Технология *REST API*

*REST* *API*, согласно [3] – механизм передачи состояния представления. *Web*-службы, которые пользуются системой *Representational State Transfer*, применяют термин *RESTful*. Отличие этого архитектурного стиля от других состоит в том, что у него нет единого стандарта, однако при этом допустимо использовать *XML, HTTP, JSON* и *URL*.

Термин состоит из двух аббревиатур, которые расшифровываются следующим образом. *API* (*Application Programming Interface*) (рисунок 2.1) – это код, который позволяет двум приложениям обмениваться данными с сервера. На русском языке его принято называть программным интерфейсом приложения. 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*.

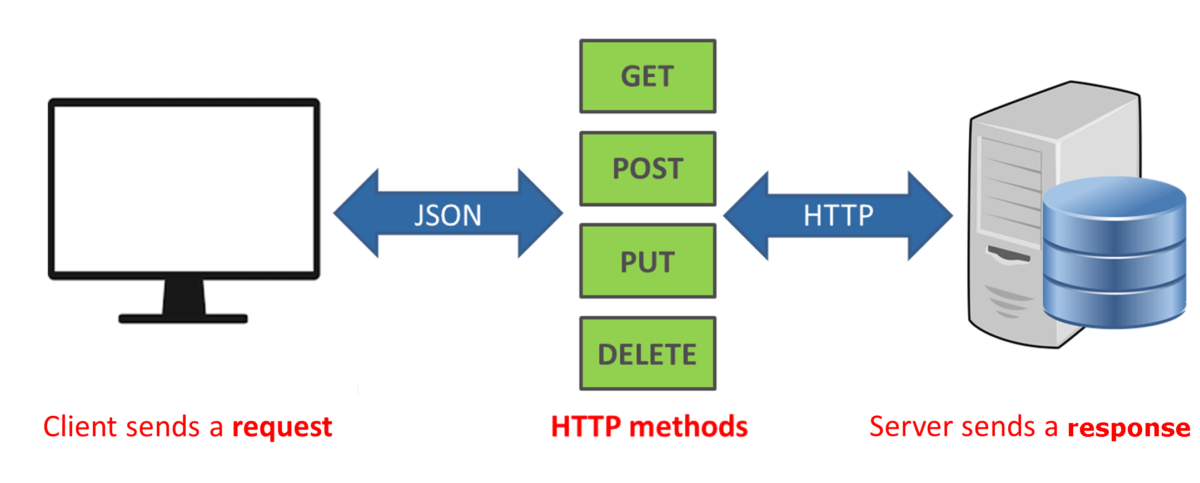


Рисунок 2.1 – Схема *Rest* *Api*

У *RESTful* есть 7 принципов написания кода интерфейсов:

* отделения клиента от сервера (*Client-Server*). Клиент – это пользовательский интерфейс сайта или приложения. В *REST API* код запросов остается на стороне клиента, а код для доступа к данным – на стороне сервера;
* 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. 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.

*API*, использующие протокол *HTTP* – это веб-сервисы. «Веб-сервис» – это веб-приложение, предоставляющее ресурсы в формате, используемом другими компьютерами. Веб-сервисы включают в себя различные типы API, в том числе *REST* и *SOAP API*. Веб-сервисы – это, в основном, запросы и ответы между клиентами и серверами (компьютер запрашивает ресурс, а веб-сервис отвечает на запрос). В случае веб-сервисов клиент, делающий запрос на ресурс, и сервер *API*, предоставляющий ответ, могут использовать любой язык программирования или платформу. Не имеет значения, какой язык программирования или платформа будут использоваться, потому что запрос сообщения и ответ сделаны через общий веб-протокол *HTTP*. Схема работы обычного сервиса погоды представлен на рисунке 2.2.

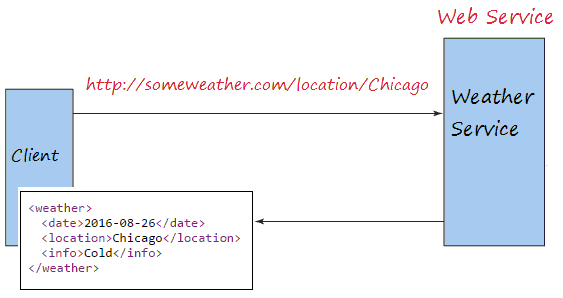


Рисунок 2.2 – Схема веб-сервиса погоды

Веб-протокол является частью веб-сервиса: они независимы от языка и поэтому совместимы между различными платформами и системами. При документировании *REST* *API* не имеет значения, строят ли инженеры *API* с помощью *Java*, *Ruby*, *Python* или какого-либо другого языка. Запросы выполняются через *HTTP*, и ответы возвращаются через *HTTP*.

Основным объектом манипуляции в HTTP является ресурс, на который указывает URI (Uniform Resource Identifier) в запросе клиента. Обычно такими ресурсами являются хранящиеся на сервере файлы, но ими могут быть логические объекты или что-то абстрактное. Особенностью протокола HTTP является возможность указать в запросе и ответе способ представления одного и того же ресурса по различным параметрам: формату, кодировке, языку и т. д. (в частности, для этого используется HTTP-заголовок). Именно благодаря возможности указания способа кодирования сообщения клиент и сервер могут обмениваться двоичными данными, хотя данный протокол является текстовым.

HTTP — протокол прикладного уровня; аналогичными ему являются FTP и SMTP. Обмен сообщениями идёт по обыкновенной схеме «запрос-ответ». Для идентификации ресурсов HTTP использует глобальные URI. В отличие от многих других протоколов, HTTP не сохраняет своего состояния. Это означает отсутствие сохранения промежуточного состояния между парами «запрос-ответ». Компоненты, использующие HTTP, могут самостоятельно осуществлять сохранение информации о состоянии, связанной с последними запросами и ответами (например, «куки» на стороне клиента, «сессии» на стороне сервера). Браузер, посылающий запросы, может отслеживать задержки ответов. Сервер может хранить IP-адреса и заголовки запросов последних клиентов. Однако сам протокол не осведомлён о предыдущих запросах и ответах, в нём не предусмотрена внутренняя поддержка состояния, к нему не предъявляются такие требования.

# РАЗРАБОТКА *WEB API*

* 1. Постановка задачи

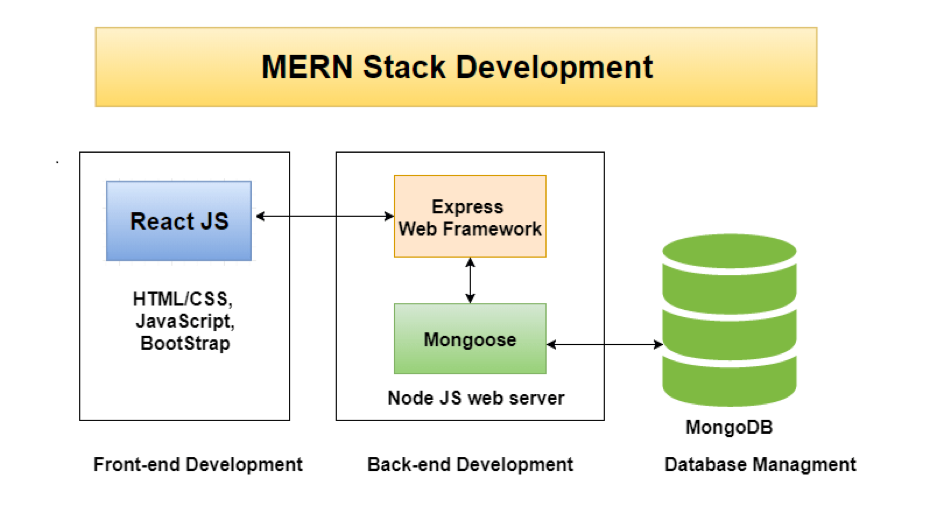
Необходимо разработать *WEB-*сервис, который позволяет отдавать данные в удобном для пользователя виде, благодаря чему им удобно пользоваться при написании собственных приложений. *WEB*-сервис должен быть разработан с использованием серверного фреймворка *Express*. База данных должна описываться с использованием *ORM*-библиотеки *Sequalize*. Валидация должна производится с помощью специальной библиотеки *Express-Validator*.

* 1. **Используемые средства**

В качестве стека технологий был выбран *SERN*. *SERN* – это аббревиатура из следующих технологий:

* *SQLite;*
* *Express.js;*
* *React.js;*
* *Node.js.*

Согласно [4], данный стек предлагает следующую архитектуру (рисунок 3.1):



**SERN Stack Development**

**Sequalize**

**SQLite**

**Database Management**

Рисунок 3.1 – Архитектура приложения с использованием стека *SERN*

На рисунке 3.1 можно выделить следующие уровни:

* уровень данных. Данный уровень хранит все данные приложения. Он представлен сервером;
* бизнес-уровень работает как посредник между уровнем данных и уровнем представления. Все данные проходят через бизнес-уровень перед тем, как их будет использовать уровень представления;
* уровень представления. Это уровень, на котором пользователи взаимодействуют с приложением. В данной курсовой работе он реализован с помощью библиотеки *React*. Уровень представления содержит отделенный код (код, который отвечает только за разметку, но не поведение).

*SQLite* используется в качестве базы данных, *Node.js* и *Express.js* для серверной части, а *React.js* для создания клиентской части. Все эти технологии объединяет то, что в их основе используется язык *JavaScript*, что упрощает процесс разработки.

*Express* – это фреймворк, созданный на базе платформы *Node.js*. Его преимущества значительно упростят и укоротят внутренний код. В стеке Express играет немаловажную роль, так как с его помощью настраивается маршрутизация и создается *API*-сервер для взаимодействия между клиентской и серверной частью приложения. Данный фреймворк предоставляет большое *API* для работы с данными.

*Node.js* – среда выполнения, позволяющая запускать код *JavaScript* на сервере, благодаря чему фронтенд-разработчик может создавать легко интегрируемые приложения.

*React* –это декларативная, эффективная и гибкая *JavaScript*-библиотека для создания пользовательских интерфейсов. Она позволяет вам собирать сложный *UI* из маленьких изолированных кусочков кода, называемых «компонентами»*.* Согласно [5], она имеет следующие преимущества:

* эффективность. React хранит в памяти две версии виртуального *DOM* –  обновленный виртуальный *DOM* и его резервную копию, созданную до обновления. После обновления React сравнивает обе версии между собой, чтобы найти измененные элементы, а затем –  обновляет исключительно изменившуюся часть реального *DOM*. Подобный процесс на первый взгляд кажется переусложненным и трудоемким, однако он занимает гораздо меньше времени, чем обновление реальной объектной модели документа целиком, следовательно, он оптимизирует работу с *DOM*;
* высокая производительность. Одна из жизненно важных целей любого стартапа – написать веб-приложение быстрым и отзывчивым, обеспечить наилучшее обслуживание клиентов. Виртуальная *DOM* (рисунок 3.2), в отличие от реального *DOM*, занимает мало места и быстро обновляется, тем самым повышая производительность приложения. Виртуальная *DOM* позволяет странице немедленно получать ответы от сервера и отображать обновления. Например, *Facebook* применяет технологию виртуального *DOM* для обновления чатов и лент пользователей без перезагрузки страницы;

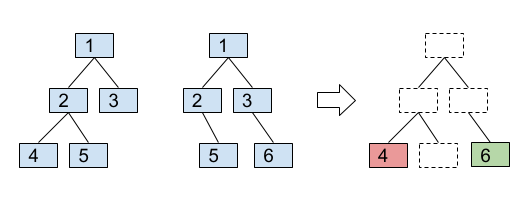


Рисунок 3.2 – Диаграмма виртуальной *DOM*

* повторное применение компонентов. При работе с *ReactJS* создаются многоразовые компоненты: чаще всего, компонент пользовательского интерфейса можно использовать в других частях кода или даже в разных проектах практически без изменений. Более того, разработчикам *React*-приложений доступны библиотеки готовых компонентов с открытым исходным кодом. Программирование на *React* сводится к тому, что мы разрабатываем различные компоненты, а затем встраиваем их в *DOM* (рисунок 3.3).

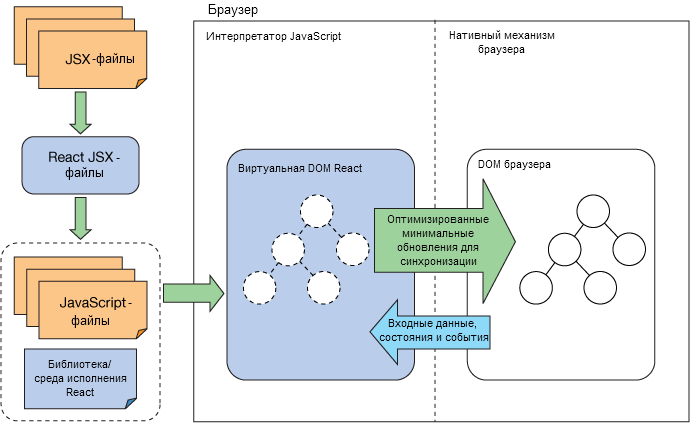


Рисунок 3.3 – Диаграмма приложения на *React*

*SERN* – это эффективный стек для разработки веб-приложений. Он удобный в использовании и лёгок в изучении.

* 1. Модель данных

В качестве хранилища данных была использована база данных *SQLite*. Данная база данных является реляционной. Реляционная БД – это СУБД, специально предназначенная для хранения не иерархических структур данных (таблиц) и обычно реализуемая с помощью подхода *SQL*.

На рисунке 3.4 представлена схемы используемой базы данных.

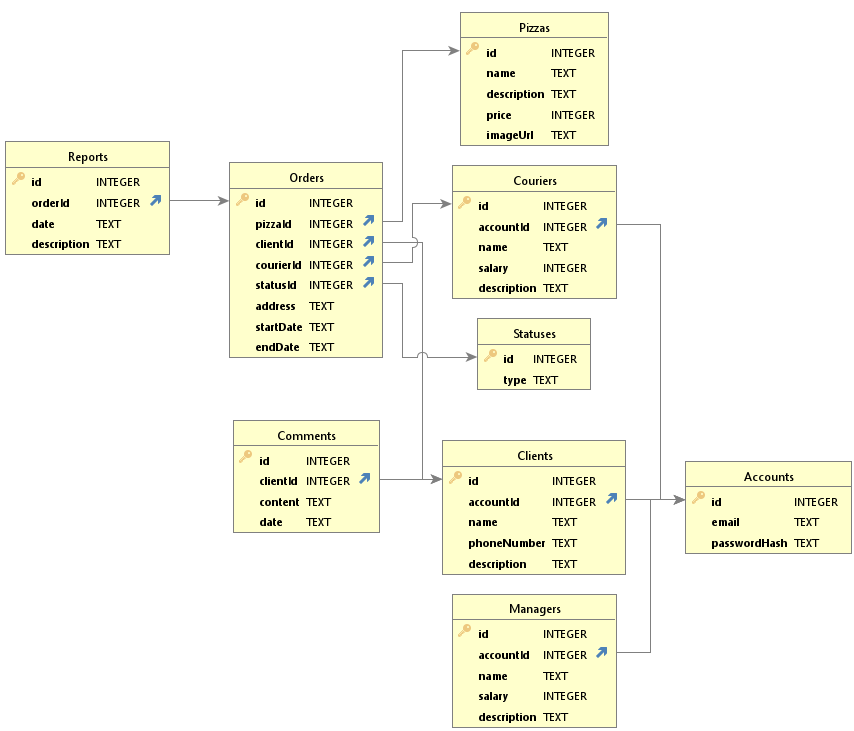
****

Рисунок 3.4 – Схема базы данных

Данная схема отражает модель базы данных, на которой мы можем заметить, что у нас имеется девять таблиц: *Pizzas, Couriers, Statuses, Orders, Reports, Comments, Clients, Managers, Accounts*.

Реляционная база данных – это набор данных с предопределенными связями между ними. Эти данные организованны в виде набора таблиц, состоящих из столбцов и строк. В таблицах хранится информация об объектах, представленных в базе данных. В каждом столбце таблицы хранится определенный тип данных, в каждой ячейке – значение атрибута. Каждая стока таблицы представляет собой набор связанных значений, относящихся к одному объекту или сущности. Каждая строка в таблице может быть помечена уникальным идентификатором, называемым первичным ключом, а строки из нескольких таблиц могут быть связаны с помощью внешних ключей. К этим данным можно получить доступ многими способами, и при этом реорганизовывать таблицы БД не требуется.

* 1. Уровень доступа к данным

Для описания моделей базы данных в приложении использовалась специальная *ORM*-библиотека (*Object Relational Mapping*) *Sequalize*. *Sequalize* – библиотека для работы с *SQLite*, которая позволяет сопоставлять объекты классов и строки таблиц из базы данных [6].

Данные, которые используются в *Sequalize*, описываются определенной схемой (рисунок 3.5).

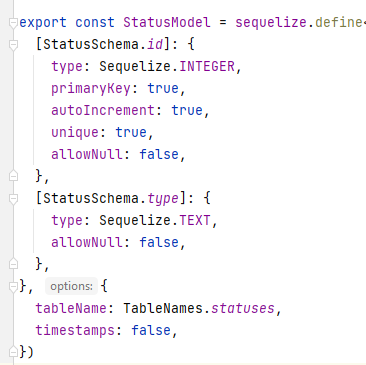


Рисунок 3.5 – Схема модели *Status*

Для того, чтобы использовать *Sequalize*, мы должны сначала импортировать (затем можно деструктуризировать) два объекта – *Shema* и *model*. Данная схема описывает небольшой документ из базы данных. Флаг *required* означает, что данное поле обязательное. Флаг *unique* говорит о том, что поле должно быть уникальное.

На рисунке 3.6 представлена более сложная схема со связями в других таблицах.

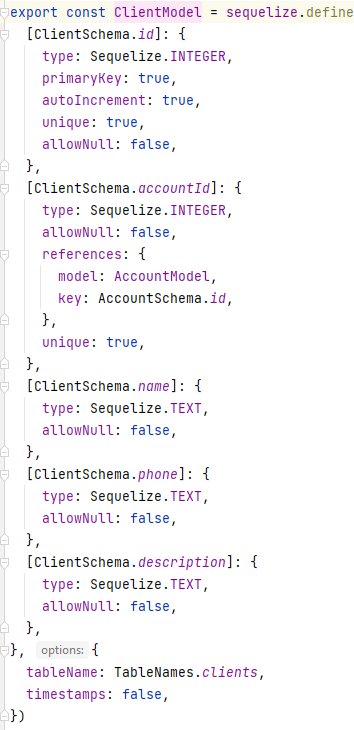


Рисунок 3.6 – *Sequalize* схема

Каждый ключ в нашей схеме определяет свойство в наших документах, которое будет приведено к связанному с ним *SchemaType*. Например, мы определили *description* свойства, который будет приведен к *String SchemaType*. Ключи можно делать внешними, задавая ссылку на поле в другой таблице.

Данная схема нужна для создания объектов, которые соответствуют объекту из базы данных. Для управления данными используется *Express*. Для доступа к методам библиотеки нужно экспортировать *express Router*. Это позволит нам регистрировать маршруты и использовать их в нашем приложении.

Стандартный *Get* запрос выглядит так (рисунок 3.7):



Рисунок 3.7 – *Get* запрос

На рисунке 3.7 можно увидеть, что мы сначала экспортируем *Router* и модели *ClientModel, CommentModel*. Данный запрос регистрирует маршрут «*/comments/getAll*». Данный маршрут используется в слое представления в *fetch* запросе, предоставляемым *api* браузера. В случае успешного запроса в качестве результата пользователь получит все объекты из таблицы *Comments*. Если же запрос завершился неудачно (код 400), то пользователь получит соответствующее сообщение.

На рисунке 3.8 представлен более сложный запрос (*Post* запрос).



Рисунок 3.8 – *Post* запрос

На рисунке 3.9 представлен *Delete* запрос.

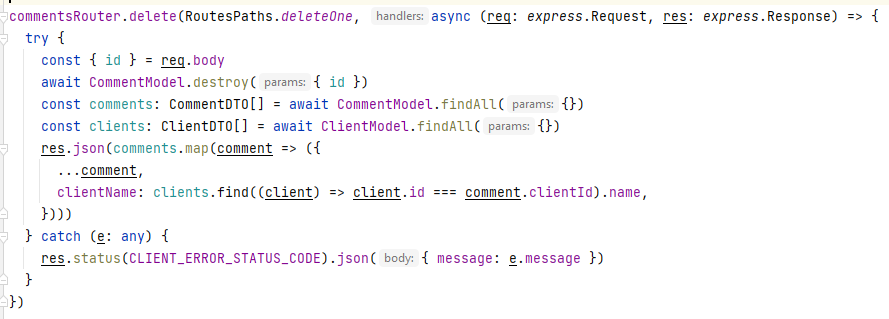


Рисунок 3.9 – *Delete* запрос

На рисунках 3.8-3.9 можно заметить, что перед основным телом функции, мы проверяем входные данные с помощью *Express-Validator*. Данная библиотека предоставляет большое *API*, с помощью которого мы можем проверять входные данные на любые условия, в том числе мы также можем создать свои собственные условия. Это полезно, когда формат входных данных зависит от определенных символов в строке или определенного свойства в объекте. Для этого используется метод *custom* из объекта *check*, который принимает сгенерированное нами условие.

* 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.

Для взаимодействия пользователя и приложения существую такие элементы управления, как кнопки (*button*), поля ввода (*input*), выпадающий список (*select*) и тп. С помощью эти единиц *web*-приложения пользователь активирует запросы, изменяет и добавляет данные в базу данных.

Библиотека *React* предоставляет компонентный подход, что облегчает настройку и стилизацию приложения. Компонента представляет собой *html* и *javascript* код (рисунок 3.10).

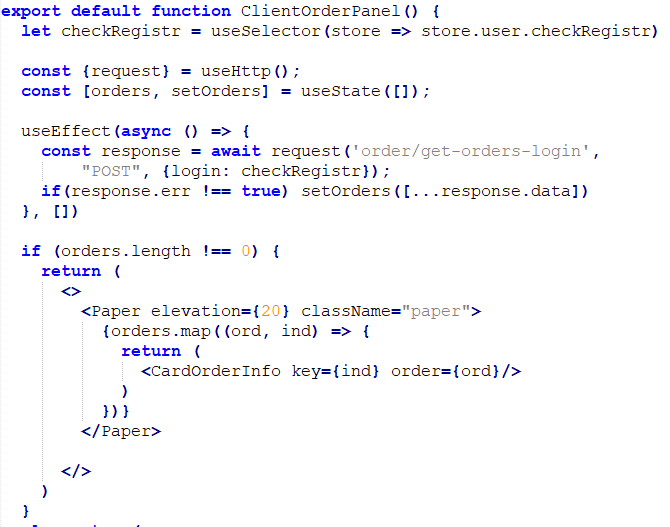


Рисунок 3.10 – Компонента в *React*

Ключевым понятием в *React* является *JSX*. *JSX* обладает всей мощью *JavaScript*. В *JSX* можно использовать любые *JavaScript*-выражения внутри фигурных скобок. Каждый *React*-элемент является *JavaScript*-объектом, который можно сохранить в переменную или использовать внутри программы.

Компонента, представленная на рисунке 3.10, оборачивает в *JSX* метод массива map, что позволило нам произвести рендер всего массива.

1. **ТЕСТИРОВАНИЕ**

4.1 Тестирование *WEB API*

Работа с базами данных является основой задачей ведения бизнеса, ведь продавцы должны иметь информацию о том, кто и что сделал с каким-то товаром и так далее.

Для тестирования был выбран исполнитель тестов на *JavaScript* *Jest*. *Jest* – это известная *opensource*-библиотека для модульного тестирования кода *JavaScript*. Она была создана и развивается благодаря *Facebook*. Написана на *Node.js*. Обычной тест представлен на рисунке 4.1.

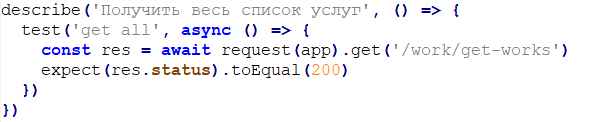


Рисунок 4.1 – Тест на получение всех услуг

Для запуска тестирования необходимо ввести в терминале команду *npm run test*.

На рисунке 4.2 представлена функция тестирования *Post* запроса.

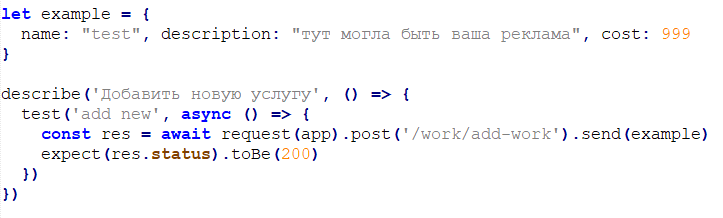


Рисунок 4.2 – Тест на добавление данных

На рисунке 4.3 представлена функция тестирования *Put* запроса.

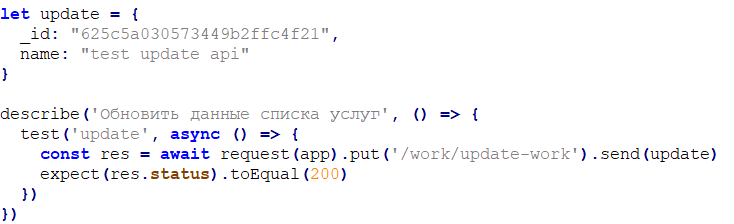


Рисунок 4.3 – Тест на обновление данных

На рисунке 4.4 представлена функция тестирования *Delete* запроса.

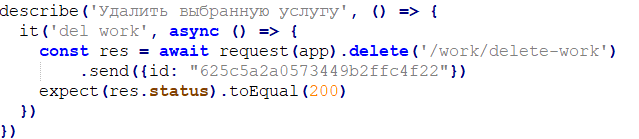


Рисунок 4.4 – Тест на удаление данных

На рисунке 4.5 представлен результат тестирования *API.*

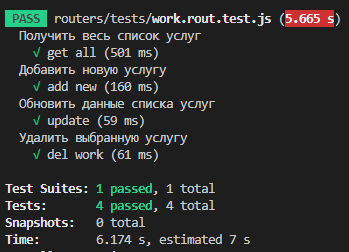


Рисунок 4.5 – Результат тестирования сервиса

Данная библиотека обладает следующими преимуществами:

* очень быстрая;
* тестирование снимками;
* потрясающий интерактивный режим отслеживания, который запускает тесты только тех компонентов, которые подверглись изменениям;
* полезные сообщения об ошибках;
* простая конфигурация;
* *mock* и *spy*-объекты;
* активная разработка.

Благодаря своему богатому *API*, *JEST* широко используется для тестирования *JS*, *React* приложений.

Также для тестирования использовался *Postman*. *Postman* – это *HTTP*-клиент для тестирования *API*. *HTTP*-клиенты тестируют отправку запросов с клиента на сервер и получение ответа от сервера. С помощью *Postman* программист может:

* составлять и отправлять *HTTP*-запросы к *API*;
* создавать коллекции (набор последовательных запросов) и папки запросов для сокращения времени тестирования;
* менять параметры запросов (например, ключи авторизации и *URL*);
* менять окружения для запросов (например, на тестовом стенде, локально или на сервере) [7].

Для начала работы необходимо установить локальный клиент и указать хост и порт сервера. Далее указываем путь к соответствующему *rout* (рисунок 4.6) (в данном случае */api/pizzas*).

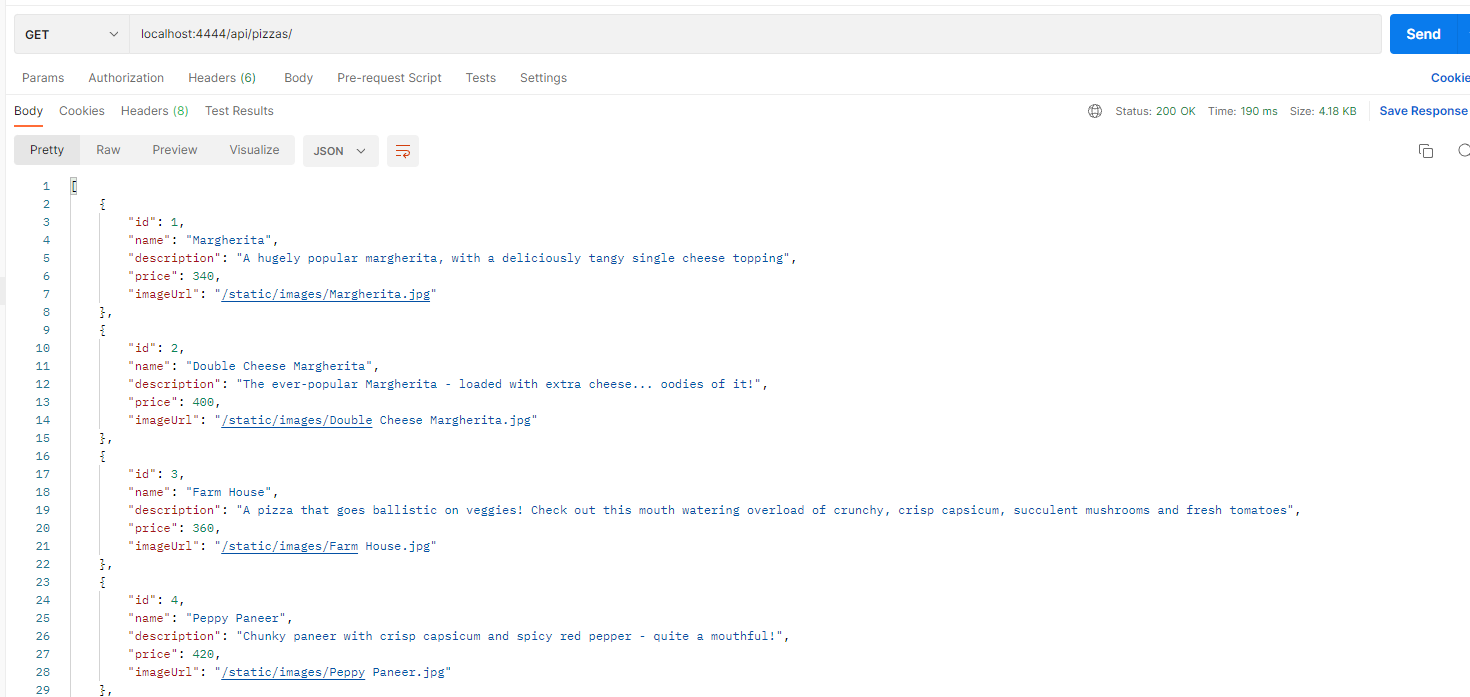


Рисунок 4.6 – Запрос на получение данных в Postman

Для более сложного запроса сначала нужно настроить заголовки (рисунок 4.7) и указать *body* (рисунок 4.8) – тело запроса, который несет в себе полезную информацию.

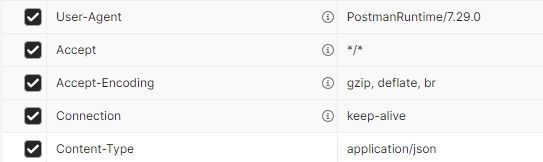


Рисунок 4.7 – Заголовки запроса

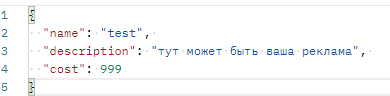


Рисунок 4.8 – Тело запроса

На рисунке 4.9 представлен *Post* запрос по маршруту «*api/work/add-work*».

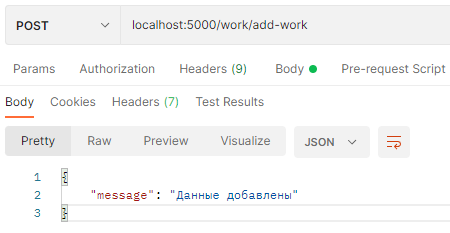


Рисунок 4.9 – *Post* запрос в *Postman*

Все, что производится человеком, может содержать ошибки. Именно поэтому любой продукт нуждается в проверке – тестировании, прежде чем его можно будет эффективно и безопасно использовать. То же самое справедливо и для программного обеспечения.

# 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. Компания Leverx [Электронный ресурс] – Режим доступа: <https://www.leverx-software.com> – Дата доступа : 20.06.2022.
2. Система стандартов безопасности труда [Электронный ресурс] – Режим доступа: <https://ohranatruda.of.by/gost-12-0-230-2-2015-sistema-standartov-bezopasnosti-truda-sistemy-upravleniya-okhranoj-truda-otsenka-sootvetstviya-trebovaniya.html> – Дата доступа: 20.06.2022.
3. Введение в REST API – RESTful веб-сервисы [Электронный ресурс] – Режим доступа: <https://habr.com/ru/post/483202/> – Дата доступа : 20.06.2022.
4. Плюсы и минусы React: виртуальная DOM, синтаксис JSX и другие аргументы для спора [Электронный ресурс] – Режим доступа: <https://nuancesprog.ru/p/14500/> – Дата доступа : 22.06.2022.
5. Использование базы данных (с помощью Sequalize) [Электронный ресурс] – Режим доступа: <https://developer.mozilla.org/ru/docs/Learn/Server-side/Express_Nodejs/sequalize> – Дата доступа : 23.06.2022.
6. Postman [Электронный ресурс] – Режим доступа: <https://blog.skillfactory.ru/glossary/postman/> – Дата доступа : 28.06.2022.

**ПРИЛОЖЕНИЕ А**

**(обязательное)**

# Листинг программного кода

httpHook.js

import React, { useState, useCallback } from 'react';  
  
export default function useHttp() {  
  
 const [loading, setLoading] = useState(false)  
  
 const request = useCallback(async (url, method = "GET", body = null, headers = {}) => {  
 setLoading(true);  
 try {  
 if (body) {  
 body = JSON.stringify(body);  
 headers['Content-Type'] = 'application/json'  
 }  
  
 const response = await fetch(url, { method, body, headers })  
 const data = await response.json();  
  
 if (!response.ok) {  
 setLoading(false);  
 if(data.errors !== undefined)  
 {  
 return {message: (data.errors[0].msg || "Что-то пошло не так (http-hooks)"), err: true};  
 }  
 return {message: (data.message || "Что-то пошло не так (http-hooks)"), err: true};  
 }  
 setLoading(false);  
 return data;  
  
 } catch (e) {  
 setLoading(false);  
 throw e;  
 }  
 }, [])  
  
 return { loading, request }  
};

auth-rout.js

const { Router } = require('express');  
const User = require('../models/Client');  
const { check, validationResult } = require('express-validator')  
  
const router = Router();  
  
router.post('/login',  
 [check('login', 'Неверный логин').isLength({ min: 5, max: 15 }),  
 check('password', 'Неверный пароль').isLength({ min: 5, max: 15 }),],  
 async (req, res) => {  
 try {  
 let errors = validationResult(req);  
 if (!errors.isEmpty()) {  
 return res.status(400).json({  
 errors: errors.array(),  
 message: "Некоректные данные при входе"  
 })  
 }  
  
 const { login, password } = req.body;  
  
 console.log('auth: ' + login + ' ' + password)  
  
 const user = await User.findOne({ login: login });  
 if (!user) {  
 return res.status(400).json({ message: 'Такого логина не существует' })  
 }  
  
 if (password !== user.password)  
 return res.status(400).json({ message: 'Пользователя с таким паролем не существует' })  
  
 return res.status(200).json({ login: login, password: password, message: 'Вход выполнен успешно' })  
  
 }  
 catch (err) {  
 return res.status(500).json({ message: 'Что-то пошло не так' })  
 }  
 });  
  
router.post('/registr',  
 [check('fio', 'Неверное имя').isLength({ min: 1, max: 15 }),  
 check('login', 'Неверный логин').isLength({ min: 5, max: 15 }),  
 check('phone', 'Неверный телефон').isLength({ min: 5, max: 15 }),  
 check('password', 'Неверный пароль').isLength({ min: 5, max: 15 }),  
 check('email', 'Неверный email').isEmail()],  
 async (req, res) => {  
 try {  
 const errors = validationResult(req);  
 if (!errors.isEmpty()) {  
 return res.status(400).json({  
 errors: errors.array(),  
 message: "Некоректные данные при регистрации"  
 })  
 }  
  
 const { fio, login, phone, password, email } = req.body;  
  
 const candidate = await User.findOne({ login: login })  
 if (candidate) {  
 return res.status(400).json({ message: 'Такой пользователь уже существует' })  
 }  
  
 const user = new User({  
 fio: fio,  
 login: login,  
 phone: phone,  
 password: password,  
 email: email  
 })  
  
 await user.save(function (err) {  
 if (err)  
 return res.status(400).json({ message: 'Не удалось зарегистровать нового пользователя (err save)' })  
 })  
  
 return res.status(200).json({ message: 'Регистрация прошла успешно' })  
  
 }  
 catch (err) {  
 return res.status(500).json({ message: 'Что-то пошло не так' })  
 }  
 });  
  
module.exports = router;

client-rout.js

const { Router } = require('express');  
const Client = require('../models/Client');  
const { check, validationResult } = require('express-validator')  
  
const router = Router();  
  
router.get('/get-client-login', async (req, res) => {  
 try {  
 const { login } = req.body;  
  
 const user = await Client.findOne({ login: login });  
  
 if (!user) {  
 return res.status(400).json({ message: 'Такого пользователя не существует' })  
 }  
  
 res.status(200).json({ data: user });  
 } catch (err) {  
 res.status(500).json('Что то пошло нет так: ' + err);  
 }  
});  
  
router.get('/get-clients', async (req, res) => {  
 try {  
 const clients = await Client.find();  
  
 if (clients.length === 0) {  
 return res.status(400).json({ message: 'Пользователей в бд нет' })  
 }  
  
 res.status(200).json({ data: clients });  
 } catch (err) {  
 res.status(500).json('Что то пошло нет так: ' + err);  
 }  
});  
  
router.get('/get-feedbacks', async (req, res) => {  
 try {  
 const clients = await Client.find({});  
  
 if (clients.length === 0) {  
 return res.status(400).json({ message: 'Пользователей в бд нет' })  
 }  
  
 const feedbacks = clients.filter(c => {  
 if(c.feedback) return { id: c.\_id, fio: c.fio, feedback: c.feedback }  
 })  
  
 res.status(200).json({ data: feedbacks });  
 } catch (err) {  
 res.status(500).json('Что то пошло нет так: ' + err);  
 }  
});  
  
router.put('/update-feedback', async (req, res) => {  
 try {  
 const { login, newFeedback } = req.body;  
 const client = await Client.findOne({ 'login': login })  
  
 if (client == null)  
 return res.status(400).json({ message: 'Клиент не найден' })  
  
 client.feedback = newFeedback;  
  
 await client.save(function (err) {  
 if (err) {  
 return res.status(400).json({ message: 'Не удалось обновить или добавить отзыв (err save)' })  
 }  
 else  
 return res.status(200).json({ message: 'Отзыв успешно добавлен' })  
 })  
 } catch (err) {  
 res.status(500).json('Что то пошло нет так: ' + err);  
 }  
});  
  
router.put('/delete-feedback', [  
 check("id", "Неверный индентификатор").isLength({min: 1, max: 50})  
], async (req, res) => {  
 try {  
 let errors = validationResult(req);  
 if (!errors.isEmpty()) {  
 return res.status(400).json({  
 errors: errors.array(),  
 message: "Индентификатор имел не верный формат"  
 })  
 }  
 let {id} = req.body;  
 const client = await Client.findOne({ \_id: id })  
 client.feedback = null;  
 await client.save(function (err) {  
 if (err) {  
 return res.status(400).json({ message: 'Не удалось удалить (err save)' })  
 }  
 else  
 return res.status(200).json({ message: 'Отзыв успешно удален' })  
 })  
 return res.status(200).json({message: "Отзыв успешно удален"})  
 } catch (err) {  
 return res.status(400).json({message: "Отзыв не была удален"})  
 }  
})  
  
module.exports = router;

employee-rout.js

const { Router } = require('express');  
const Employee = require('../models/Employee');  
const { check, validationResult } = require('express-validator')  
  
const router = Router();  
  
router.get('/get-employee-login', async (req, res) => {  
 try {  
 const { login } = req.body;  
  
 const empl = await Employee.findOne({ login: login });  
  
 if (!empl) {  
 return res.status(400).json({ message: 'Такого сотрудника не существует' })  
 }  
  
 res.status(200).json({ data: empl });  
 } catch (err) {  
 res.status(500).json('Что то пошло нет так: ' + err);  
 }  
});  
  
  
router.post('/login-empl',  
 [check('login', 'Неверный логин').isLength({ min: 5, max: 15 }),  
 check('password', 'Неверный пароль').isLength({ min: 5, max: 15 }),],  
 async (req, res) => {  
 try {  
 let errors = validationResult(req);  
 if (!errors.isEmpty()) {  
 return res.status(400).json({  
 errors: errors.array(),  
 message: "Некоректные данные при входе"  
 })  
 }  
  
 const { login, password } = req.body;  
  
 const empl = await Employee.findOne({ login: login });  
 if (!empl) {  
 return res.status(400).json({ message: 'Такого логина не существует' })  
 }  
  
 if (password !== empl.password)  
 return res.status(400).json({ message: 'Сотрудника с таким паролем не существует' })  
  
 return res.status(200).json({ login: empl.login, password: empl.password, position: empl.position,  
 message: 'Вход выполнен успешно' })  
  
 }  
 catch (err) {  
 return res.status(500).json({ message: 'Что-то пошло не так' })  
 }  
 });  
  
router.get('/get-employee', async (req, res) => {  
 try {  
 const empls = await Employee.find();  
  
 if (empls.length === 0) {  
 return res.status(400).json({ message: 'Сотрудников в бд нет' })  
 }  
  
 res.status(200).json({ data: empls });  
 } catch (err) {  
 res.status(500).json('Что то пошло нет так: ' + err);  
 }  
});  
  
router.get('/get-employee-master', async (req, res) => {  
 try {  
 const empls = await Employee.find({position: "master"});  
  
 if (empls.length === 0) {  
 return res.status(400).json({ message: 'Сотрудников в бд нет' })  
 }  
  
 res.status(200).json({ data: empls });  
 } catch (err) {  
 res.status(500).json('Что то пошло нет так: ' + err);  
 }  
});  
  
  
  
  
module.exports = router;

mark-rout.js

const { Router } = require('express');  
const Mark = require('../models/Mark');  
  
const router = Router();  
  
router.get('/get-marks', async (req, res) => {  
 try {  
 const marks = await Mark.find({})  
  
 if (marks.length === 0)  
 return res.status(400).json({ message: 'Данные отсуствуют' })  
  
 res.status(200).json({ data: marks });  
 } catch (err) {  
 res.status(500).json('Что то пошло нет так: ' + err);  
 }  
});  
  
router.post('/add', async (req, res) => {  
 try {  
 const { name, models } = req.body;  
  
 let condidat = new Mark({  
 name: name,  
 models: models  
 })  
  
 await condidat.save(function (err) {  
 if (err) {  
 console.log(err);  
 return  
 }  
 console.log("save");  
 })  
 res.status(201).json({ message: 'Данные добавлены' })  
 } catch (err) {  
 res.status(500).json('Что то пошло нет так', err);  
 }  
  
});  
  
module.exports = router;

order-rout.js

const {Router} = require('express');  
const {check, validationResult} = require('express-validator');  
const Order = require('../models/Order');  
const Client = require('../models/Client');  
const mongoose = require("mongoose");  
  
const router = Router();  
  
router.post('/create', async (req, res) => {  
 try {  
 const {login, mark, yearRelese, typeEngine, gosnumber, problemText, dateRegistr} = req.body;  
  
 const auto = {  
 yearRelese: yearRelese, mark: mark, typeEngine: typeEngine, gosnumber: gosnumber  
 }  
  
 const clientFind = await Client.findOne({login: login});  
  
 const client = new Client({  
 fio: clientFind.fio, phone: clientFind.phone, login: login, password: clientFind.password  
 })  
  
 const order = new Order({  
 \_id: new mongoose.Types.ObjectId(),  
 auto: auto,  
 client: client,  
 dateRegistr: new Date().toLocaleString(),  
 payment: '0',  
 status: 'Ожидание',  
 problemText: problemText,  
 })  
  
 await order.save(function (err) {  
 if (err) {  
 console.log(err)  
 return res.status(400).json({message: 'Не удалось зарегистровать заказ (err save)'})  
 } else return res.status(200).json({message: 'Регистрация заказа прошла успешно'})  
 })  
  
 } catch (err) {  
 return res.status(500).json({message: 'Что-то пошло не так'})  
 }  
});  
  
router.get('/get-orders', async (req, res) => {  
 try {  
 const orders = await Order.find({})  
  
 if (orders.length === 0) return res.status(400).json({message: 'Данные отсуствуют'})  
  
 res.status(200).json({data: orders});  
 } catch (err) {  
 res.status(500).json('Что то пошло нет так: ' + err);  
 }  
});  
  
  
router.post('/get-orders-login', async (req, res) => {  
 try {  
 const {login} = req.body;  
 const orders = await Order.find({'client.login': login})  
  
 if (orders.length === 0) return res.status(400).json({message: 'Заказов пока что нет'})  
  
 res.status(200).json({data: orders});  
 } catch (err) {  
 res.status(500).json({message: 'Что то пошло нет так: ' + err});  
 }  
});  
  
router.post('/get-orders-master-login', async (req, res) => {  
 try {  
 const {login} = req.body;  
 const orders = await Order.find({'master.login': login})  
  
 if (orders.length === 0) return res.status(400).json({message: 'Заказов пока что нет'})  
  
 res.status(200).json({data: orders});  
 } catch (err) {  
 res.status(500).json({message: 'Что то пошло нет так: ' + err});  
 }  
});  
  
router.put("/update-master-cost",[  
 check('id', 'Неверный id').isLength({ min: 1, max: 50 }),  
 check("cost", "Неверно введена стоимость").isNumeric().isLength({min: 1, max: 10000000}),  
 check("status", "Неверный статус").isLength({min: 1, max: 50}),  
 check("master", "Ошибка ввода данных мастера").notEmpty()  
], async (req, res) => {  
 try {  
 let errors = validationResult(req);  
 if (!errors.isEmpty()) {  
 return res.status(400).json({  
 errors: errors.array(),  
 message: "Введены некоректные данные"  
 })  
 }  
  
 const {master, id, cost, status} = req.body;  
 const order = await Order.findById(id)  
  
 if (order === null) {  
 return res.status(400).json({message: "Заказа с id" + id + " не найден"});  
 }  
  
 order.master = {  
 fio: master.fio,  
 email: master.email,  
 login: master.login,  
 }  
  
 order.cost = cost;  
 order.status = status;  
  
 await order.save(function (err) {  
 if (err)  
 return res.status(400).json({message: "Не удалось обновить данные заказа"})  
 else  
 return res.status(200).json({message: "Заказ успешно обновлен"});  
 })  
  
 } catch (err) {  
 res.status(500).json({message: "Что-то пошло не так: " + err})  
 }  
})  
  
router.put("/update-payment", async (req, res) => {  
 try {  
 const {id} = req.body;  
 console.log(req.body)  
 const order = await Order.findById(id);  
  
 if (!order) {  
 return res.status(400).json({message: "Заказа с id" + id + " не найден"});  
 }  
  
 order.payment = '1'  
  
 await order.save(function (err) {  
 if (err)  
 return res.status(400).json({message: "Не удалось обновить данные заказа"})  
 else  
 return res.status(200).json({message: "Заказ успешно обновлен"});  
 })  
  
 } catch (err) {  
 res.status(500).json({message: "Что-то пошло не так: " + err})  
 }  
})  
  
  
module.exports = router;

work-rout.js

const {Router} = require('express')  
const Work = require('../models/Work')  
const {check, validationResult} = require('express-validator')  
  
const router = Router()  
  
router.get('/get-works', async (req, res) => {  
 const works = await Work.find({});  
 if (works.length === 0)  
 return res.status(400).json({message: 'Данные в базе данных отсуствуют'})  
 return res.status(200).json({works: works})  
})  
  
router.post("/add-work", [  
 check("name", "Неверно введено название").isLength({min: 3, max: 20}),  
 check("description", "Неверно введено описание").isLength({min: 5, max: 100}),  
 check("cost", "Неверно введена стоимость").isNumeric().isLength({min: 1, max: 10000000})  
], async (req, res) => {  
 try {  
 let errors = validationResult(req);  
 if (!errors.isEmpty()) {  
 return res.status(400).json({  
 errors: errors.array(),  
 message: "Введены некоректные данные"  
 })  
 }  
 const {name, description, cost} = req.body;  
  
 let condidat = new Work({name: name, description: description, cost: parseFloat(cost)})  
  
 await condidat.save(function (err) {  
 // if (err) return res.status(400).json({message: "Не удалось сохранить данные"})  
 if (err) return res.status(400)  
 })  
  
 return res.status(200).json({message: 'Данные добавлены'})  
  
 } catch (err) {  
 return res.status(500).json({message: 'Что то пошло нет так'});  
 }  
})  
  
router.put("/update-work", async (req, res) => {  
 try {  
 const {cost, description, name, \_id} = req.body;  
  
 const work = await Work.findById(\_id);  
 if (work === null)  
 return res.status(400).json({message: `Услуга с id ${\_id} не найдена`})  
  
 work.name = name ?? work.name;  
 work.description = description ?? work.description;  
 work.cost = cost ?? work.cost;  
  
 await work.save(function (err) {  
 if (err) {  
 return res.status(400).json({message: 'Не удалось обновить услугу'})  
 } else  
 return res.status(200).json({message: 'Услуга успешно обновлена'})  
 })  
  
 } catch (err) {  
 return res.status(500).json({message: "Что-то пошло не так"})  
 }  
})  
  
router.delete('/delete-work', [  
 check("id", "Неверный индентификатор").isLength({min: 1, max: 50})  
], async (req, res) => {  
 try {  
 let errors = validationResult(req);  
 if (!errors.isEmpty()) {  
 return res.status(400).json({  
 errors: errors.array(),  
 message: "Индентификатор имел не верный формат"  
 })  
 }  
 let {id} = req.body;  
 await Work.deleteOne({\_id: id});  
 return res.status(200).json({message: "Услуга успешно удалена"})  
 } catch (err) {  
 return res.status(400).json({message: "Услуга не была удалена"})  
 }  
})  
  
module.exports = router;

server.js

const express = require('express');   
const mongoose = require('mongoose');  
const app = express();   
const PORT = process.env.PORT || 5000;   
  
const bp = require('body-parser');  
app.use(bp.json())  
app.use(bp.urlencoded({ extended: true }))  
  
app.use('/auth', require('./routers/auth-rout'))  
app.use('/work', require('./routers/work-rout'))  
app.use('/mark', require('./routers/mark-rout'))  
app.use('/order', require('./routers/order-rout'))  
app.use('/client', require('./routers/client-rout'))  
app.use('/employee', require('./routers/employee-rout'))  
  
async function start() {  
 try {  
 await mongoose.connect('mongodb://localhost:27017/WebAppSto');  
 app.listen(PORT, () => console.log(`Server has been started on port ${PORT}!`));  
 }  
 catch (e) {  
 console.log(`Server Error: ${e.message}`);  
 process.exit(1);  
 }  
}  
  
start();  
module.exports = app

rout.test.js

const request = require('supertest')

const app = require('../../server')

describe('Получить весь список услуг', () => {

  test('get all', async () => {

    const res = await request(app).get('/work/get-works')

    expect(res.status).toEqual(200)

  })

})

let example = {

  name: "test", description: "тут может быть ваша реклама", cost: 999

}

describe('Добавить новую услугу', () => {

  test('add new', async () => {

    const res = await request(app).post('/work/add-work').send(example)

    expect(res.status).toBe(200)

  })

})

let update = {

  \_id: "625c5a030573449b2ffc4f21",

  name: "test update api"

}

describe('Обновить данные списка услуг', () => {

  test('update', async () => {

    const res = await request(app).put('/work/update-work').send(update)

    expect(res.status).toEqual(400)

  })

})

describe('Удалить выбранную услугу', () => {

  it('del work', async () => {

    const res = await request(app).delete('/work/delete-work').send({id: "625c5a2a0573449b2ffc4f22"})

    expect(res.status).toEqual(200)

  })

})

Order.js

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

const mongoose = require("mongoose");

const schema = new Schema({

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

    auto: {

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

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

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

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

    },

    client: {

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

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

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

    },

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

    master: {

        fio: {type: String},

        login: {type: String},

        email: {type: String},

    },

    payment: {type: String},

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

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

    cost: {type: String},

}, {versionKey: false})

module.exports = model('orders', schema);

Work.js

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

const mongoose = require("mongoose");

const schema = new Schema({

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

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

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

    name: {type: String, required: true}

}, { versionKey: false })

module.exports = model('works', schema);

Employee.js

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

const schema = new Schema({

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

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

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

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

    email: {type: String},

    position: {type: String},

}, { versionKey: false });

module.exports = model('employee', schema);

Client.js

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

const schema = new Schema({

    feedback: {type: String},

    fio: {type: String},

    login: {type: String},

    phone: {type: String},

    password: {type: String},

    email: {type: String}

}, { versionKey: false });

module.exports = model('clients', schema);

Mark.js

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

const schema = new Schema({

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

    models: {type: Array, required: true },

}, { versionKey: false });

module.exports = model('marks', schema);