

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования
«Гродненский государственный университет имени Янки Купалы»

Факультет математики и информатики
Кафедра современных технологий программирования

ДЕЛЕНДИК ЕКАТЕРИНА НИКОЛАЕВНА

Веб-сервис народного мониторинга цен на продукты питания

Курсовая работа
по дисциплине «Системы баз данных»
студентки 3 курса специальности
1-26 03 01 Управление информационными ресурсами
дневной формы получения образования

Научный руководитель
Изосимова Татьяна Николаевна,
доцент кафедры современных
технологий программирования

Гродно 2020

РЕЗЮМЕ

Курсовая работа: «Веб-сервис народного мониторинга цен на продукты питания», 37 страниц, 13 иллюстраций, 5 листингов, 7 использованных источников, 4 приложения.

БД, СУБД, SQL, JAVA, SPRING, HIBERNATE, КЛИЕНТ, СЕРВЕР

Объект исследования: веб-сервис для народного мониторинга цен на продукты питания.

Предмет исследования: языки программирования и средства реализации веб-приложений.

Цель исследования: проектирование и программная реализация веб-сервиса для народного мониторинга цен на продукты питания с дальнейшим его обновлением и использованием.

Автор работы подтверждает, что приведенный в курсовой работе материал правильно и объективно отражает состояние исследуемого процесса, а все заимствованные из литературных и других источников теоретические, методологические и методические положения и концепции сопровождаются ссылками на их авторов.

СОДЕРЖАНИЕ

| | |
|---|----|
| ВВЕДЕНИЕ..... | 4 |
| 1 ПРОЕКТИРОВАНИЕ ВЕБ-СЕРВИСА..... | 6 |
| 1.1 Требования к функциональным характеристикам | 6 |
| 1.1.1 База данных | 6 |
| 1.1.2 Серверная часть..... | 7 |
| 1.1.3 Клиентская часть..... | 8 |
| 1.2 Разработка интерфейса пользователя веб-сервиса..... | 9 |
| 1.3 Выводы по главе 1..... | 15 |
| 2 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ВЕБ-СЕРВИСА | 16 |
| 2.1 Обзор и анализ средств реализации..... | 16 |
| 2.2 Создание базы данных | 19 |
| 2.3 Разработка серверной части веб-сервиса..... | 20 |
| 2.4 Разработка клиентской части веб-сервиса | 24 |
| 2.5 Выводы по главе 2 | 25 |
| ЗАКЛЮЧЕНИЕ | 27 |
| СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ..... | 28 |
| ПРИЛОЖЕНИЕ 1 | 29 |
| ПРИЛОЖЕНИЕ 2..... | 32 |
| ПРИЛОЖЕНИЕ 3..... | 35 |
| ПРИЛОЖЕНИЕ 4..... | 37 |

ВВЕДЕНИЕ

На сегодняшний день наличие веб-сервиса для народного мониторинга цен на продукты питания и их характеристики в магазинах страны очень актуально.

Ввиду того что количество сетей продуктовых магазинов и просто магазинов в стране увеличивается каждый год, наличие подобного веб-сервиса – необходимое условие для успешного выбора магазина или конкретного товара для постоянных покупок людей.

Актуальность данной темы заключается в том, что до сих пор в Беларуси, а в частности в городе Гродно, нет подобного веб-сервиса, люди делают выбор неосознанно. Хороший веб-сервис для народного мониторинга цен на продукты питания – залог корректного и быстрого выбора сети продуктовых магазинов.

Данное приложение предназначено для мониторинга всей доступной информации о продуктах питания. Она решает следующие задачи:

- 1) демонстрация списка категорий товаров;
- 2) демонстрация списка подкатегорий товаров;
- 3) демонстрация списка продуктов из определенной подкатегории товаров;
- 4) добавление, удаление категорий/подкатегорий товаров;
- 5) добавление, удаление, обновление товаров;
- 6) демонстрация списка магазинов;
- 7) поиск товаров по определенным параметрам;
- 8) изменение цены товара;
- 9) демонстрация цен на товар за определенный промежуток времени.

Объектом курсовой работы является веб-сервис для народного мониторинга цен на продукты питания.

Предметом курсовой работы являются языки программирования и средства реализации веб-приложений.

Цель курсовой работы — спроектировать и программно реализовать веб-сервис для народного мониторинга цен на продукты питания с дальнейшим его обновлением и использованием. Для достижения поставленной цели предусмотрено решение следующих задач:

- 1) создать базу данных;
- 2) создать необходимые объекты (классы);
- 3) построить приложение по принципам ООП;
- 4) разработать методы и функции, необходимые для работы с объектами;
- 5) визуализировать архитектуру реализации приложения.

Используемые технологии в выполнении данной курсовой работы:

1. СУБД MySQL.
2. Язык программирования высокого уровня Java с использованием Spring MVC, Spring Security, Hibernate.
3. Язык программирования JavaScript.
4. HTML + CSS.

Используемое программное обеспечение в выполнении данной курсовой работы:

1. MySQL Workbench 8.0 CE.
2. IntelliJ IDEA 2018.3.5.
3. Postman.
4. Visual Studio Code.

В качестве теоретической и методологической базы при написании курсовой работы были использованы научные труды зарубежных ученых, периодические издания, интернет-источники.

Курсовая работа разрабатывается в паре с Елизаветой Кеврой. Курсовая работа состоит из введения, 2-ух глав, заключения, списка использованных источников и приложений.

1 ПРОЕКТИРОВАНИЕ ВЕБ-СЕРВИСА

1.1 Требования к функциональным характеристикам

Любой веб-сервис состоит из двух основных компонентов: клиентской и серверной частей, между которыми должно быть налажено взаимодействие. Именно серверная часть работает с базой данных.

1.1.1 База данных

В качестве этой системы управления базами данных был использован MySQL. Гибкость СУБД MySQL обеспечивается поддержкой большого количества типов таблиц. MySQL достаточно распространённая СУБД и, что самое важное, отлично подходит для обучения работы с базами данных в целом.

У СУБД MySQL есть ряд преимуществ:

- 1) богатый функционал: MySQL поддерживает большинство функционала SQL;
- 2) безопасность: большое количество функций, обеспечивающих безопасность, которые поддерживаются по умолчанию;
- 3) масштабируемость: MySQL легко работает с большими объемами данных и легко масштабируется.
- 4) скорость: упрощение некоторых стандартов позволяет MySQL значительно увеличить производительность.

Для визуального проектирования баз данных, работы с SQL-запросами, создания и эксплуатации БД был выбран такой инструмент, как MySQL Workbench версии 8.0 CE.

При создании базы данных важно избегать дублирования информации, придерживаться хороших практик при построении БД (корректное название таблиц и полей, корректная настройка типов данных каждого из полей и т.д.) и настроить каскадное удаление данных. Что касается последнего пункта, это важно, так как в базе данных для проектируемого веб-сервиса есть каскадные связи между объектами, и настройка на уровне базы данных даст возможность избежать большого количества строк в коде, который без такой настройки должен будет сам удалять записи, зависящие от другой записи, которую хотят удалить.

1.1.2 Серверная часть

Проектируемый веб-сервис должен быть многопользовательским. Так как почти все веб-сервисы предлагают какие-либо услуги пользователям вне зависимости от того, зарегистрированы ли пользователи в системе или нет. Необходимо реализовать разделение пользователей на две роли: пользователь и не авторизованный пользователь. Авторизованный пользователь имеет право доступа ко всем функциям. У не авторизованного пользователя права ограничены. При этом пользователи, которые будут храниться в базе данных, также должны быть разделены на два типа: обычный пользователь и администратор. Рассмотрим подробнее.

Функционал доступный неавторизованным пользователям:

- просмотр информации о данном веб-приложении;
- просмотр контактов;
- авторизация;
- регистрация;
- просмотр категорий товаров;
- просмотр подкатегорий товаров;
- просмотр продуктов из выбранной подкатегории товаров;
- просмотр магазинов, которые хранятся в базе данных веб-приложения.

Помимо вышеперечисленных функций зарегистрированный пользователь имеет больше возможностей, к ним относятся:

- создание товаров (т.е. добавление их в базу данных);
- изменение цены товара;
- поиск товаров по параметрам.

Администратор в свою очередь может удалять категории товаров и пользоваться теми же функциями, что и зарегистрированный пользователь.

Таким образом, на стороне сервера должны быть созданы REST контроллеры, которые позволят сделать REST-запросы со стороны клиента для получения, изменения, создания и удаления записей. На стороне сервера должны быть созданы все необходимые HTTP-методы для реализации всех требований к веб-сервису.

1.1.3 Клиентская часть

Взаимодействие между клиентской и серверной частями должно осуществляться посредством HTTP-запросов, благодаря REST архитектуре приложения.

Клиентская часть должна быть реализована отдельным модулем веб-приложения, запускаемого в браузере, предоставляющего удобный и

понятный интерфейс пользователю с использованием языка программирования JavaScript, вёрстка может быть реализована при помощи HTML и CSS.

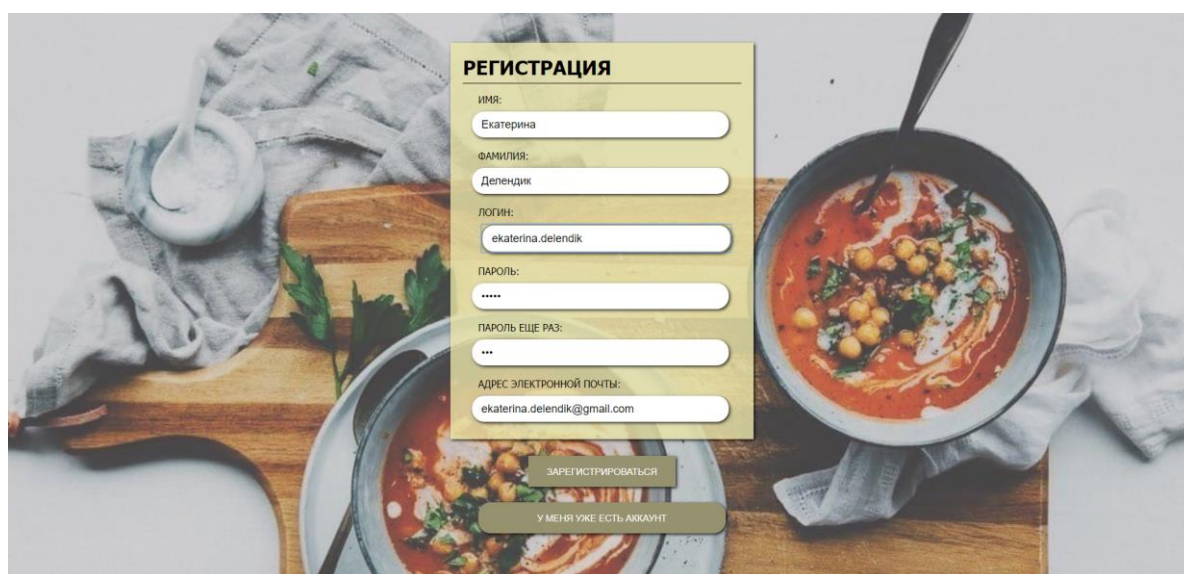
Требования, которые должны быть реализованы на стороне клиента:

- валидация данных при авторизации (незаполненные строки логина или пароля);
- вывод о предусмотренных ошибках со стороны сервера (неправильный логин или пароль);
- валидация данных при регистрации (проверка на доступность введенных символов, заполнение обязательных полей и т.д.);
- корректный переход между страницами;
- доступный и понятный пользователям интерфейс.

1.2 Разработка интерфейса пользователя веб-сервиса

Для данного веб-сервиса необходимо реализовать 3 вида интерфейса пользователя для разных ролей: пользователя, авторизованного пользователя, администратора.

Что касается анализа размещения информации на экране, во-первых, необходимые элементы, выделены более ярким цветом, для привлечения внимания. Во-вторых, все необходимые элементы большого размера и находятся в центре экрана. Пример страницы регистрации изображен на рисунке 1.1.



РЕГИСТРАЦИЯ

ИМЯ:
Екатерина

ФАМИЛИЯ:
Делендик

ЛОГИН:
ekaterina.delendik

ПАРОЛЬ:
.....

ПАРОЛЬ ЕЩЕ РАЗ:
...

АДРЕС ЭЛЕКТРОННОЙ ПОЧТЫ:
ekaterina.delendik@gmail.com

ЗАРЕГИСТРИРОВАТЬСЯ

У МЕНЯ УЖЕ ЕСТЬ АККАУНТ

Рисунок 1.1 – Форма регистрации

Примечание – Источник: собственная разработка.

Каждая страница соответствует общим требованиям, которым должен удовлетворять удобный интерфейс, а именно: принцип структуризации, простоты, видимости, обратной связи, повторного использования.

Как и описывалось в требованиях, при неправильно введенных данных на экране должно быть сообщение об ошибке (рисунок 1.2).

Важно отметить, что на всех полях ввода паролей настроен специальный тип “password”, который позволяет скрыть символы, вводимые пользователем. Это является хорошей практикой для интерфейса регистрации и авторизации.

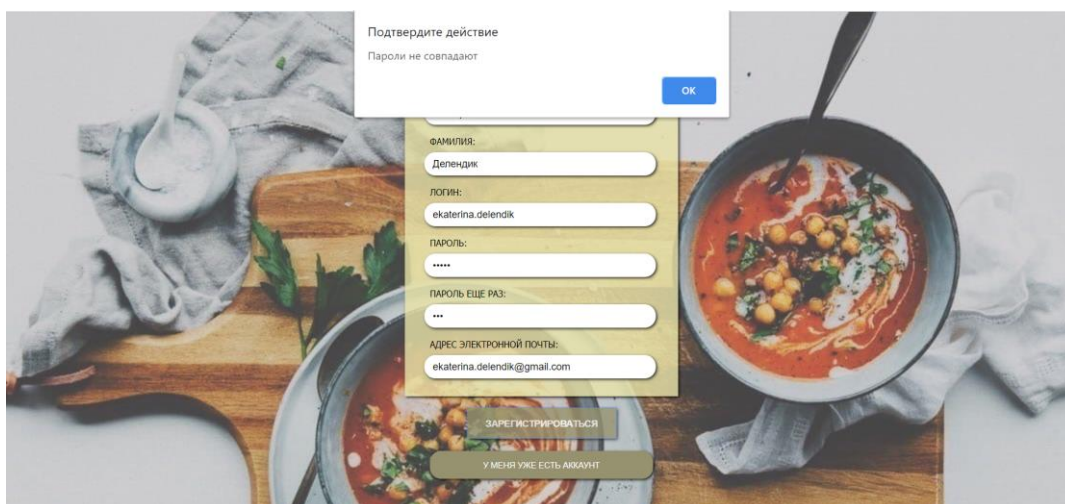


Рисунок 1.2 – Вывод ошибки при некорректном заполнении полей ввода пароля

Примечание – Источник: собственная разработка.

Если пользователь не заполнил некоторые из полей (рисунок 1.3), то выводится сообщение об ошибке.

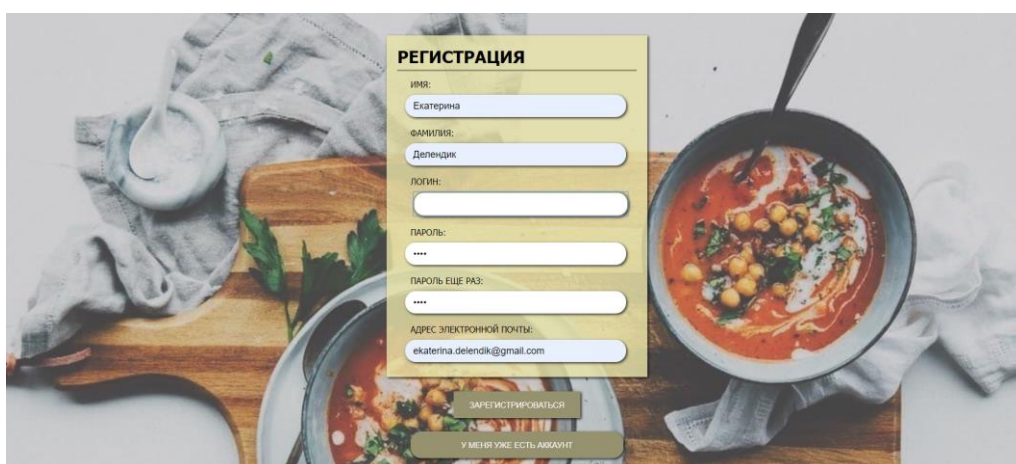


Рисунок 1.3 – Вывод ошибки из-за пустых полей для ввода

Примечание – Источник: собственная разработка.

На данной странице имеются две кнопки: «Зарегистрироваться» и «У меня уже есть аккаунт», которая переходит на новую страницу для авторизации пользователя. Что касается самой формы для авторизации, валидация работает с такой же логикой, а интерфейс фактически такой же, как и в форме для регистрации (рисунок 1.4).

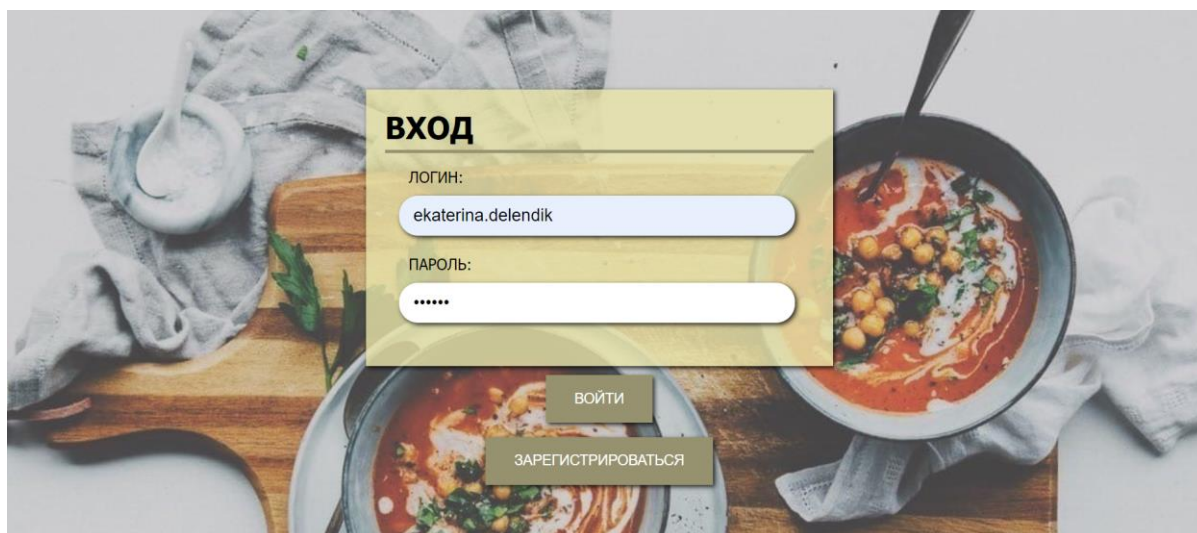


Рисунок 1.4 – Форма авторизации пользователей

Примечание – Источник: собственная разработка.

Далее продемонстрируем интерфейс на примере неавторизованного пользователя с возможным для него функционалом.

На главной странице размещён заголовок и информация о нашем веб-сервисе (рисунок 1.5.).

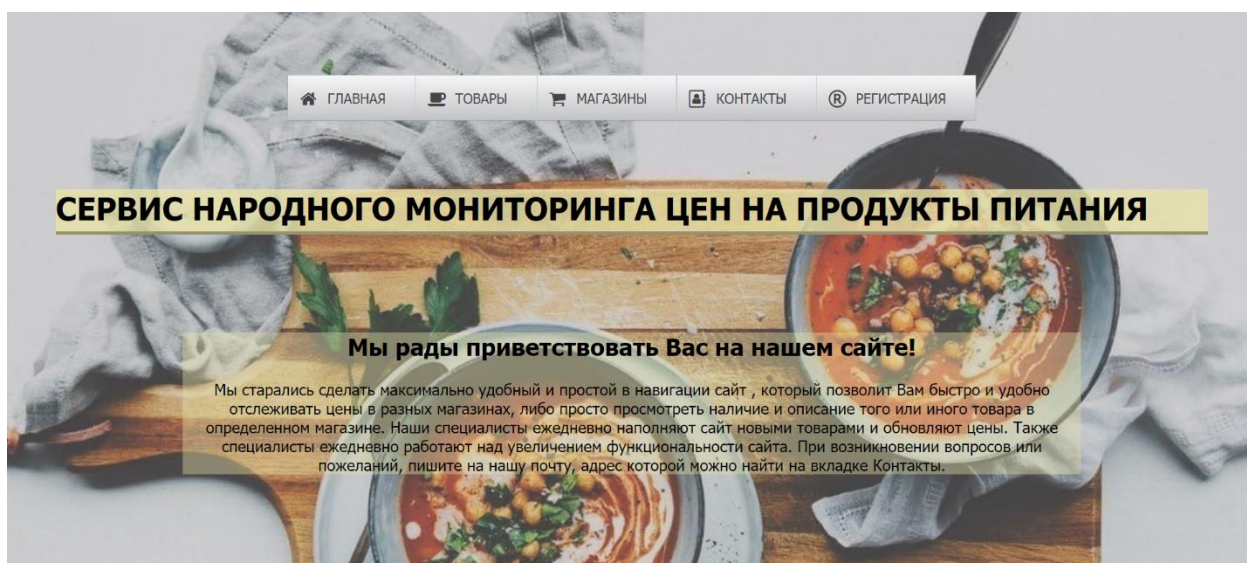


Рисунок 1.5 – Главная страница веб-сервиса

Примечание – Источник: собственная разработка.

На главной странице находится основная панель, которая содержит 5 кнопок: главная, товары, магазины, контакты, регистрация. При нажатии кнопки «Регистрация» пользователь перейдёт на страницу с формой для регистрации, которая была продемонстрирована выше. При нажатии кнопки «Контакты» пользователь окажется на странице, где указаны контакты разработчиков данного веб-сервиса (рисунок 1.6).

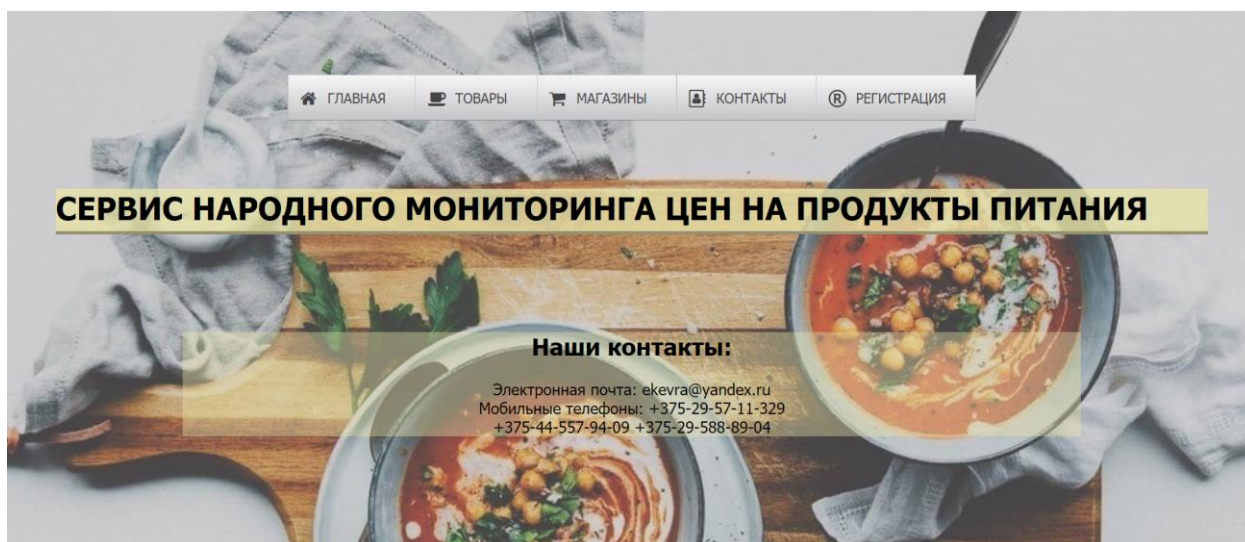


Рисунок 1.6 – Страница веб-сервиса, на которой указаны контакты разработчиков

Примечание – Источник: собственная разработка.

Оставшиеся две кнопки «Товары» и «Магазины» уже связаны непосредственно с серверной стороной веб-сервиса. При наведении курсора мышки на кнопку «Товары» выводится список с категориями товаров, которые хранятся в базе данных (рисунок 1.7).

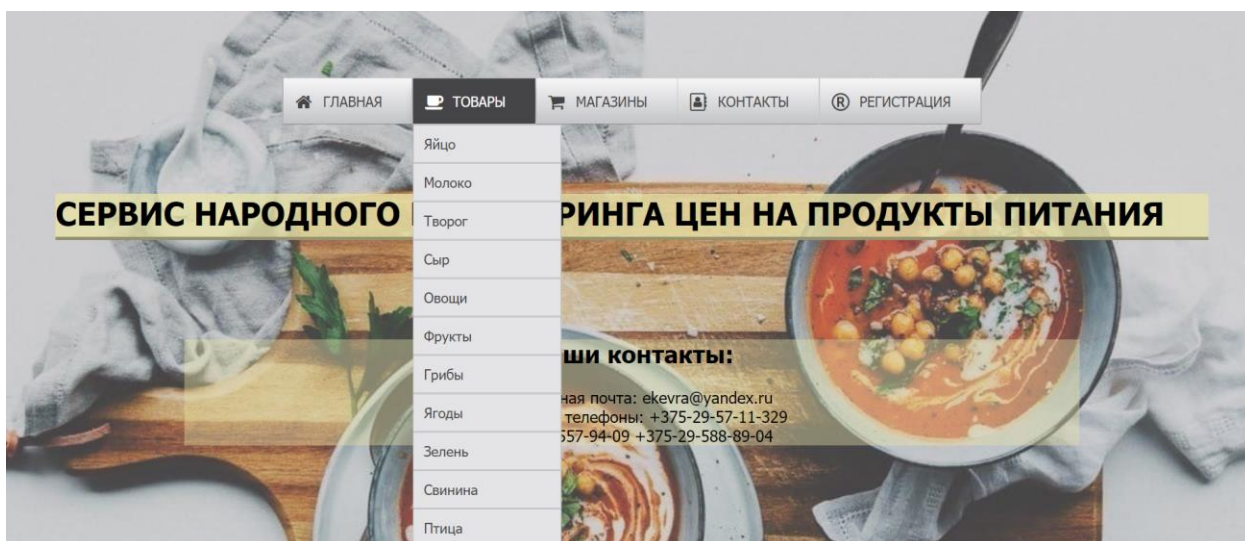


Рисунок 1.7 – Список категорий товаров из базы данных

Примечание – Источник: собственная разработка.

При загрузке главной страницы делается запрос на сервер, который берёт необходимую информацию для данной страницы. В данном случае речь идёт о динамической разметке страницы, которая зависит от данных, полученных из БД. Это значит, что при отсутствии всех категорий кнопка «Товары» в целом не будет актуальна. То же касается и кнопки любой из категорий при отсутствии подкатегорий, связанных с ней. Здесь наблюдаются те самые связи, которые в базе данных определяются как каскадные, где категория без товаров может существовать, хотя и не имеет смысла, а товар без категории не может существовать, т.к. исходя из данного интерфейса получить данные о товарах нельзя, не выбрав категории и подкатеорию.

При наведении курсора мыши на любую из категорий на экране выводится список подкатегорий, которые хранятся в базе данных и которые принадлежат именно той категории, которую выбирает пользователь. (рисунок 1.8).

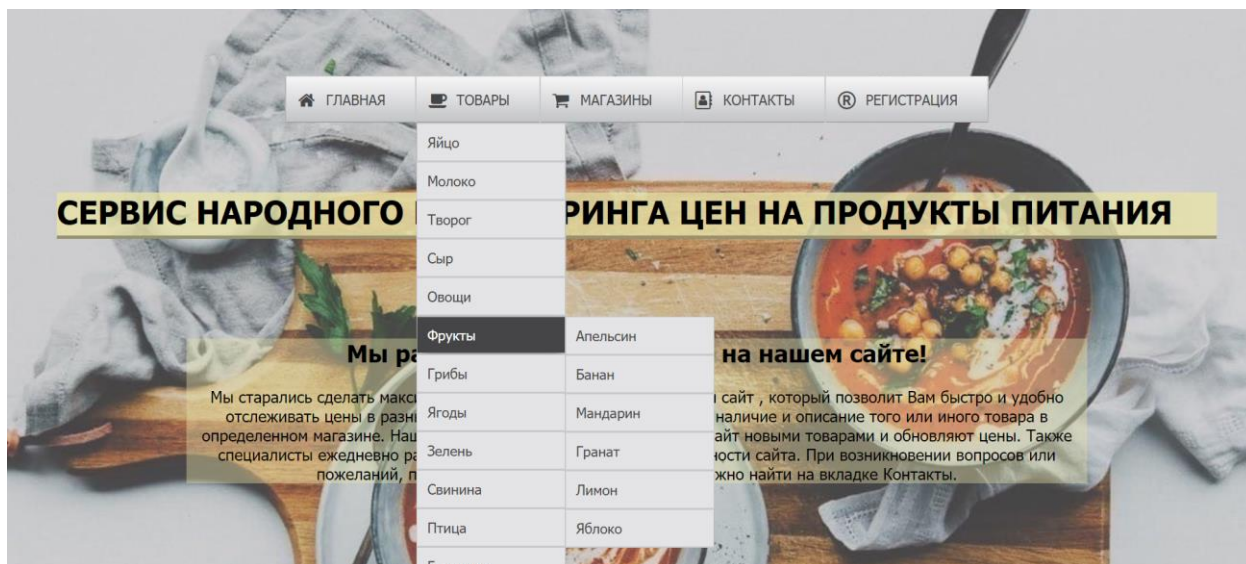


Рисунок 1.8 – Список категорий и соответствующих подкатегорий из базы данных

Примечание – Источник: собственная разработка.

При нажатии любой кнопки из подкатегорий пользователь перейдёт на страницу с таблицей всех товаров из базы данных, соответствующих выбранной подкатегории (рисунок 1.9). Следует отметить, что магазинов одной сети в базе данных много, и для того, чтобы не перегружать информацией пользователя и выводить на экран только уникальные названия магазинов без обозначения их адресов, на стороне клиента реализован метод, который фильтрует названия всех магазинов и выводит на экран в доступном виде только уникальные значения.

| Наименование продукта | Производитель | Вес | Цена | Магазины |
|--|------------------|------|------|-------------------|
| Молоко ультрапастеризованное "Простоквашино" 3.2%, 930 мл. | Простоквашино | 0.93 | 2.17 | Санта, Рублевский |
| Молоко ультрапастеризованное 2.5%. | Бабушкина крынка | 0.9 | 1.45 | Евроопт |

Рисунок 1.9 – Список товаров из определённой подкатегории

Примечание – Источник: собственная разработка.

Такое же поведение программы можно наблюдать при нажатии на кнопку «Магазины». Список всех магазинов из базы данных будет продемонстрирован пользователю (рисунок 1.10).

| Магазины |
|--|
| Евроопт, Гродно, ул. Победы 47 |
| Евроопт, Гродно, улица А. Дубко 17 |
| Евроопт, Гродно, ул. Ольги Соломовой 104 |
| Родны кут, Гродно, ул. Кремко, 10 |
| Родны кут, Гродно, ул. Гожская 3 |
| Санта, Гродно, бул. Ленинского Комсомола, 44 |
| Рублевский, Минск, пр-т. Победителей 1 |

Рисунок 1.10 – Список магазинов из базы данных

Примечание – Источник: собственная разработка.

Таким образом, можно сделать вывод о вполне интуитивном и простом интерфейсе веб-сервиса, который позволяет пользователю наблюдать, а также мониторить цены и характеристику продуктов питания.

1.3 Выводы по главе 1.

В процессе проектирования были составлены требования к функциональным характеристикам. Из них стало известно, что пользоваться веб-сервисом полностью могут только авторизованные пользователи. А для контроля веб-сервисом в целом используются роли администраторов. На основе перечисленных требований были построены макеты интерфейсов. Данные макеты просты и понятны каждому пользователю, так как при их построении использовались основные правила создания интерфейсов.

2 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ВЕБ-СЕРВИСА

2.1 Обзор и анализ средств реализации

Серверная часть приложения написана на Java с использованием SpringMVC, Hibernate. Java – это объектно-ориентированный язык программирования. Java отвечает трем основным принципам объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм. Крайне важно учитывать каждый из них при написании программы.

Как уже было сказано выше, для реализации веб-сервиса был использован Hibernate. Для взаимодействия с базой данных необходим фреймворк для работы с ней – Hibernate. Целью Hibernate является освобождение разработчика от значительного объема сравнительно низкоуровневого программирования при работе в объектно-ориентированных средствах в реляционной базе данных. Разработчик может использовать Hibernate как в процессе проектирования системы классов и таблиц «с нуля», так и для работы с уже существующей базой данных. Библиотека не только решает задачу связи классов Java с таблицами базы данных (и типов данных Java с типами данных SQL), но и также предоставляет средства для автоматической генерации и обновления набора таблиц, построения запросов и обработки полученных данных и может значительно уменьшить время разработки, которое обычно тратится на ручное написание SQL- и JDBC-кода. Hibernate автоматизирует генерацию SQL-запросов и освобождает разработчика от ручной обработки результирующего набора данных и преобразования объектов, максимально облегчая перенос приложения на любые базы данных SQL.

Что касается используемого SpringMVC Framework, его целью является поддержка в Spring архитектуры модель-представление-контроллер (model-view-controller). Spring обеспечивает готовые компоненты, которые могут быть использованы (и используются) для разработки веб-приложений. Главной целью MVC является разделение объектов, бизнес-логики и внешнего вида приложения. Все эти компоненты слабо связаны между собой и при желании мы можем изменить, например, внешний вид приложения, не внося существенные изменения в остальные два компонента.

В качестве среды разработки использована программа IntelliJ IDEA. IntelliJ IDEA- интегрированная среда разработки программного обеспечения для многих языков программирования, в частности Java, JavaScript, Python и многих других. Важно отметить, что при настройке взаимосвязи между

проектом Java и СУБД MySQL, IntelliJ IDEA поддерживает работу с консолью MySQL и таблицами БД в целом прямо внутри проекта. Это, конечно, очень удобно для разработчика, который разрабатывает на стороне сервера.

Клиентская и серверная часть взаимодействуют между собой через REST-архитектуру.

REST (Representational state transfer) – это стиль архитектуры программного обеспечения для распределенных систем, таких как World Wide Web, который, как правило, используется для построения веб-служб. Термин REST был введен в 2000 году Роем Филдингом, одним из авторов HTTP-протокола. Системы, поддерживающие REST, называются RESTful-системами.

В общем случае REST является очень простым интерфейсом управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат.

Таким образом, на стороне сервера были созданы REST контроллеры, которые позволяют сделать REST-запросы со стороны клиента для получения, изменения, создания и удаления записей. На стороне сервера были созданы все необходимые HTTP-методы для реализации всех требований к веб-сервису. Данные методы тестировались через программное обеспечение Postman (рисунок 2.1 – 2.2).

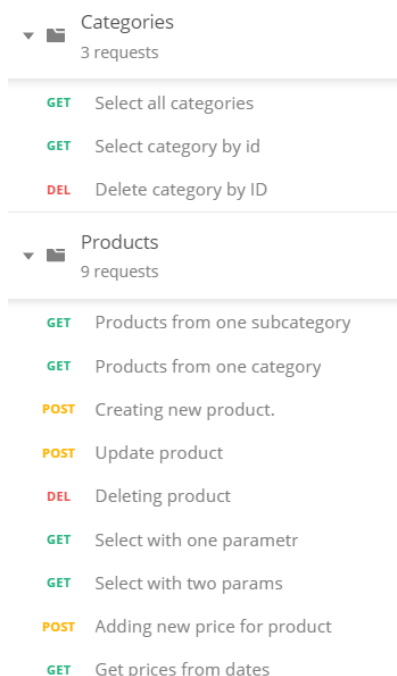


Рисунок 2.1 – Список реализованных методов для REST-запросов

Примечание – Источник: собственная разработка.

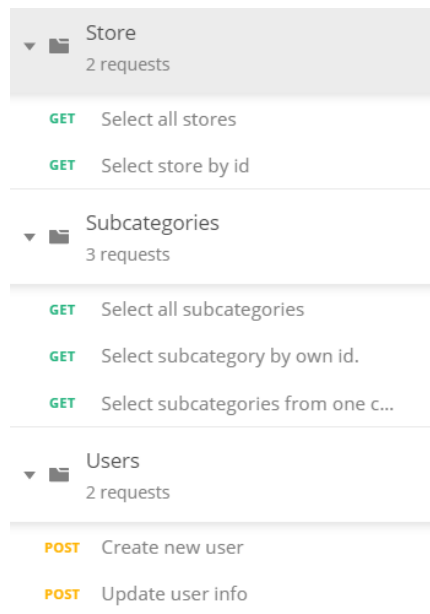


Рисунок 2.2 – Список реализованных методов для REST-запросов

Примечание – Источник: собственная разработка.

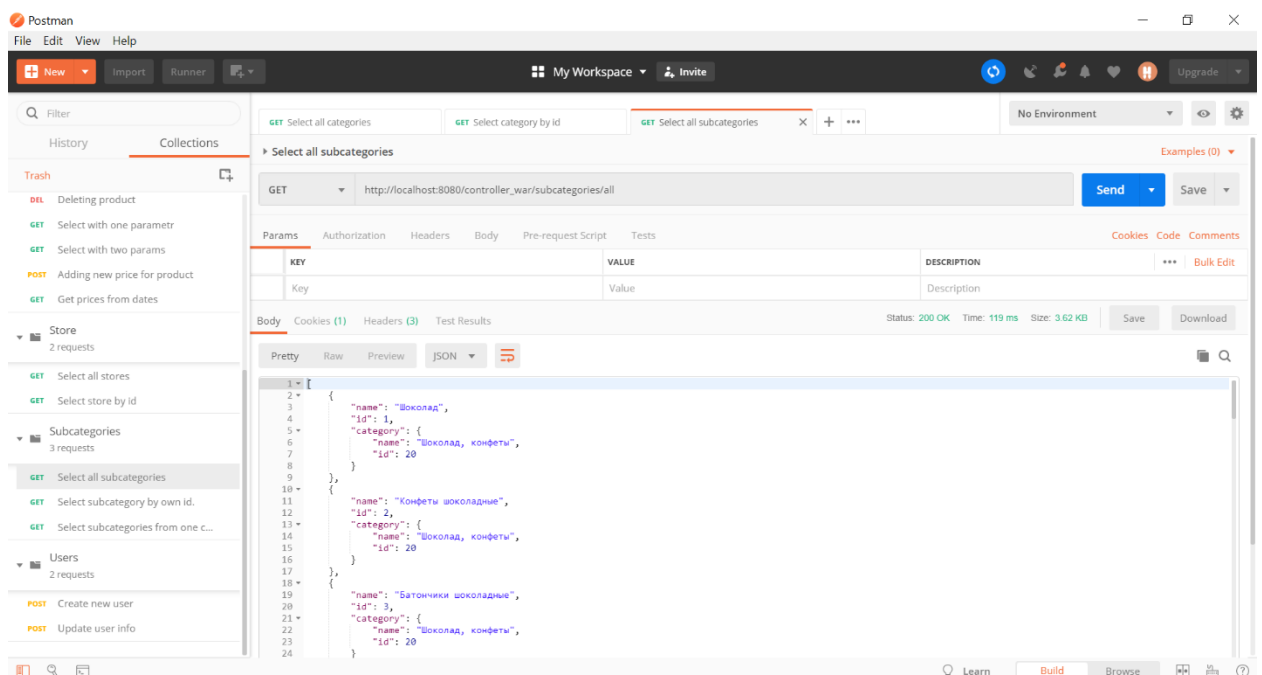


Рисунок 2.3 – Тестирование REST-запросов

Примечание – Источник: собственная разработка.

2.2 Создание базы данных.

Для хранения информации была выбрана СУБД MySQL. При помощи такого ПО, как MySQL Workbench, были созданы таблицы, поля и настройки базы данных.

Используемая база данных для разрабатываемого веб-сервиса содержит в себе 9 таблиц (рисунок 2.3):

- 1) **user** (**id**, name, surname, email);
- 2) **credentials** (**login**, password, role, user_id);
- 3) **store** (**id**, name, city, address);
- 4) **product_store** (**store_id**, **product_id**);
- 5) **product** (**id**, name, manufacture, weight, subcategory_id, price_id);
- 6) **subcategory** (**id**, name, id_category);
- 7) **category** (**id**, name);
- 8) **price** (**id**, price, start_date, isActive);
- 9) **old_prices** (**price_id**, **product_id**).

Персональными ключами в каждой из таблиц являются выделенные поля. Можно обратить внимание, что большинство из них id. В таблицах product_store и old_prices персональный ключ состоит из двух полей, то есть он формируется из уникальности связки двух id. И в таблице credentials персональным ключом является login.

В базе данных реализованы такие связи, как:

- 1) один-к-одному (credentials - user);
- 2) один-ко-многим (category – subcategory, subcategory - product);
- 3) многие-ко-многим (product - store).

В хорошей практике создания база данных следует избегать реализации связи многие-ко-многим, которая необходима веб-сервису для связи между продуктами и магазинами. В таком случае была создана связывающая таблица product_store, которая хранит в себе Id записи из таблицы store и Id записи из таблицы product. Связь между таблицами product – product_store и store – product_store один-ко-многим.

Персональными ключами в каждой из таблиц являются id. В таблицах product_store и old_prices персональный ключ состоит из двух полей, то есть он формируется из уникальности связки двух id. И в таблице credentials персональным ключом является login.

Также в таблице настроены такие параметры, как обязательные и необязательные поля для заполнения таблиц, настроены внешние ключи для реализации связей между таблицами, каскадные удаления записей. Последний пункт обозначает, что при удалении категории удалятся и подкатегории, соответствующие ей. Модель созданной базы данных продемонстрирована на рисунке 2.4.

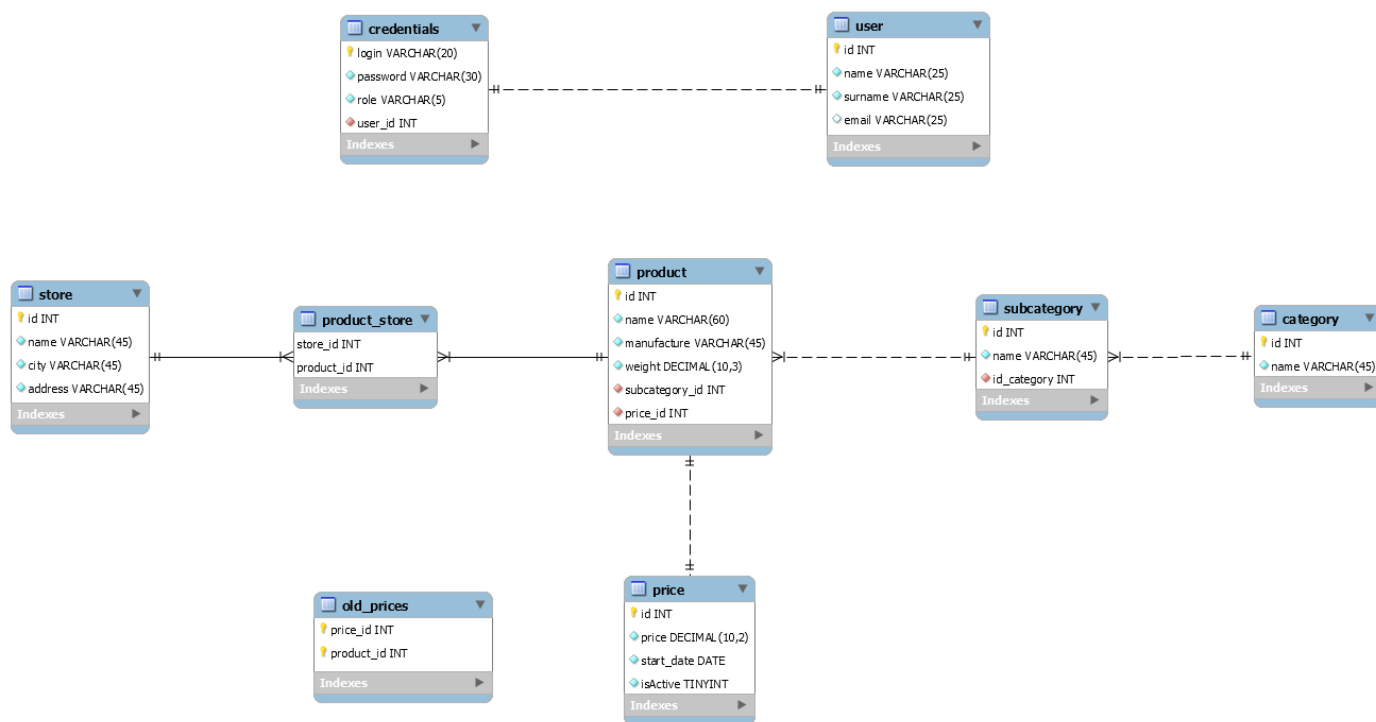


Рисунок 2.4 – Модель базы данных

Примечание – Источник: собственная разработка.

2.3 Разработка серверной части веб-сервиса.

Для того чтобы строго структурировать серверную часть приложения был использован Maven и весь проект разделён на 5 модулей:

1. Model – модуль, который содержит классы, объекты которых соответствуют каждой из таблицы из базы данных. Именно в этом модуле используются аннотации Hibernate, которые во многом упрощают работу разработчика. Пример класса Product продемонстрирован в приложении 1.

2. API (Application Programming Interface) – модуль, который содержит в себе все интерфейсы, которые используются в приложении. Интерфейсы используются для реализации слабой связанности, что является важной частью построения качественного веб-приложения. В этом модуле есть интерфейс GenericDao, который описывает все CRUD методы, код данного интерфейса описан в листинге 2.1.

```

public interface GenericDao<T> {

    void create(T t);

    void delete(T t);

    T getById(Integer id);

    List<T> getAll();

    void update(T t);
}

```

Листинг 2.1 – Интерфейс GenericDao из модуля API

Примечание – Источник: собственная разработка.

3. DAO (Data Access Object) – это модуль, который работает непосредственно с базой данных. Именно в этом модуле делаются все запросы в базу данных и реализуются все crud-методы. В этом модуле создан абстрактный класс AbstractDao (листинг 2.2), который реализует все crud-методы для каждого из отдельных классов dao, а в каждом из них уже реализована частная логика для каждого из класса-моделей, пример такого класса продемонстрирован на листинге 2.3.

```

public abstract class AbstractDao<T> implements GenericDao<T> {
    private Class clazz;

    @PersistenceContext
    protected EntityManager entityManager;

    public AbstractDao(Class<T> clazz) {
        this.clazz = clazz;
    }

    public void create(T object) {
        entityManager.persist(object);
    }

    public void delete(T object) {
        entityManager.remove(object);
    }
}

```

```

    }
    public T getById(Integer id) {
        T object = (T) entityManager.find(clazz, id);
        return object;
    }

    public List<T> getAll() {
        CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
        CriteriaQuery<T> criteriaQuery = criteriaBuilder.createQuery(clazz);
        Root<T> rootQuery = criteriaQuery.from(clazz);
        criteriaQuery.select(rootQuery);
        TypedQuery<T> query = entityManager.createQuery(criteriaQuery);
        List<T> allitems = query.getResultList();
        return allitems;
    }

    public void update(T object) {
        entityManager.merge(object);
    }
}

```

Листинг 2.2 – Абстрактный класс AbstractDao из модуля DAO

Примечание – Источник: собственная разработка.

```

@Repository
public class CredDao extends AbstractDao<Cred> implements ICredDao {

    private static final Logger logger = LogManager.getLogger(CredDao.class);
    private CredDao() {
        super(Cred.class);
    }

    public Cred getByLogin(String login){
        Cred object = (Cred) entityManager.find(Cred.class, login);
        return object;
    }
}

```

Листинг 2.3 – Класс ProductDao из модуля DAO

Примечание – Источник: собственная разработка.

4. Service – модуль, который содержит в себе обрабатывающие классы, т.е. классы, в которых методы реализуют определённую логику, эти методы наследуются из интерфейсов из модуля `api`. На данном уровне серверного приложения уже подключен и в целом имеет смысл быть логгер, который позволяет отслеживать необрабатываемые исключения (см. приложение 2). Также в этом модуле созданы все классы DTO(Data Transfer Object). Это является хорошей практикой, передавать на сторону клиента объекты именно классов DTO. Они выглядят фактически так же, как и классы из модуля `model`. Просто они не содержат аннотаций. Пример такого класса изображен в приложении 3.

5. Controller – модуль, который теснее всех связан с внешней средой, а точнее с клиентской стороной приложения. Именно в этом модуле используется SpringMVC. Здесь используется многочисленные аннотации с параметрами и без. Хорошей практикой при написании контроллеров является то, что в них находится минимально логики, таким образом такие классы являются просто связующими между сервисным модулем и клиентской части приложения. Здесь также прописываются HTTP-методы и URL, по которым будут осуществляться запросы(приложение 4).

Таким образом видно, что вся серверная часть строго структурирована. Использованы лучшие практики ООП. Код написан понятно и аккуратно. В целом в данной части приложения легко искать нужные части кода, контролировать необрабатываемые ситуации, что очень важно для дальнейшего развития приложения.

2.4 Разработка клиентской части веб-сервиса.

Клиентскую часть приложения можно разделить на две части: разработка на JavaScript и верстка HTML + CSS. Ни в первом, ни во втором случаях не используются сторонние библиотеки. Пример разметки страницы входа в веб-сервис изображён на листинге 2.4. Также для удобной работы все файлы находятся в определённой структуре, где отдельно лежат картинки для фонов CSS, отдельно сама разметка и отдельно файлы CSS. Как уже и оговаривалось, клиент и сервер общаются через REST-запросы, пример такого запроса, где клиентская сторона получает записи категорий, изображён на листинге 2.5.

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Вход</title>
```

```

        <link rel="stylesheet" href="css/style2.css">
</head>
<body>
    <div class="margin100">
        <div class="signinForm">
            <h1 title="Форма входа на сайт">Вход</h1>
                <div class="group">
                    <label for="">Логин:</label>
                    <input type="text" id="inputLogin">
                </div>
                <div class="group">
                    <label for="">Пароль:</label>
                    <input type="password" id="inputPassword">
                </div>
                <div class="button1">
                    <center><button
onclick="checkingLoginAndPassword();">Войти</button></center>
                </div>
                <div class="button2">
                    <center><button
onclick='location.href="loginPage.html"'>Зарегистрироваться</button></cente
r>
                </div>
            </div>
        </div>
        <script src="javascript/signin.js"></script>
    </body>
</html>

```

Листинг 2.4 – HTML-разметка страницы входа в систему

Примечание – Источник: собственная разработка.

```

const url = 'http://localhost:8080/controller_war';

const categoryRequest = new XMLHttpRequest();
categoryRequest.open('GET', url + '/categories/all', false);
var categories = [];

```

```
categoryRequest.onload = function() {  
    categories = JSON.parse(this.response);  
}  
categoryRequest.send();
```

Листинг 2.5 – REST-запрос

Примечание – Источник: собственная разработка.

Таким образом, реализована связь между клиентской частью веб-сервиса и серверной. То есть браузер, клиентская часть получает обработанную информацию из базы данных, что и является полной готовой структурой веб-приложения.

2.5 Выводы по главе 2.

При помощи СУБД MySQL, языка программирования Java, JavaScript и фреймворков SpringMVC и Hibernate реализован веб-сервис для народного мониторинга цен на продукты питания. Созданы условия для успешного дальнейшего его обновления и использования. Также описаны алгоритмы взаимодействия пользователя и программы. В ходе данной разработки были улучшены знания в работе с базами данных и такими фреймворками, как Hibernate и Spring MVC. Был изучен язык программирования JavaScript и улучшены навыки в разметке данных.

ЗАКЛЮЧЕНИЕ

В процессе выполнения курсовой работы получены знания в работе с базами данных и такими фреймворками, как Hibernate и SpringMVC, изучен язык программирования JavaScript, улучшены навыки в разметке данных и в работе с REST-архитектурой, что очень важно для современного разработчика. Кроме того разработано полноценное веб-приложение: сервис для народного мониторинга цен на продукты питания.

Для достижения поставленной цели решены следующие задачи. Во-первых, создана база данных. Во-вторых, созданы необходимые объекты (классы), построено приложение по принципам ООП и разработаны методы и функции, необходимые для работы с объектами. В-третьих, визуализирована архитектура реализации приложения. Визуализация в приложении простая и удобная, что, конечно, будет очень важно при дальнейшем использовании.

Таким образом, программа представляет собой сложную систему взаимодействующих друг с другом объектов. В процессе создания приложения реализованы алгоритмы, которые в дальнейшем можно усовершенствовать. Все поставленные задачи выполнены. Созданное приложение представляет собой законченный программный продукт.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Учебник по JavaFX 8 — Часть 5: Работа с базами данных
/http://code.makery.ch/library/javafx-8-tutorial/ru/part5/
2. <https://metanit.com/java/tutorial/> — руководство по Java [Электронный ресурс] – Режим доступа: <https://metanit.com/java/tutorial/> – Дата доступа: 14.04.2019
3. www.programmer-lib.ru/java – обучение программированию [Электронный ресурс] – Режим доступа: <http://programmer-lib.ru/csharp> – Дата доступа: 21.04.2020
4. www.professorweb.ru – JavaFX [Электронный ресурс] – Режим доступа: <http://professorweb.ru> – Дата доступа: 02.05.2020
5. www.code-live.ru – уроки программирования [Электронный ресурс] – Режим доступа: <http://code-live.ru> – Дата доступа: 23.04.2020
6. <http://hibernate.org/> – официальный сайт Hibernate [Электронный ресурс] – Режим доступа: <http://hibernate.org/> - Дата доступа: 14.04.2020
7. <https://www.mysql.com/>-официальный сайт MySQL[Электронный ресурс]-Режим доступа: <https://www.mysql.com/>-Дата доступа: 12.04.2020

ПРИЛОЖЕНИЕ 1

Класс Product

```
@Entity
@Table(name = "product")
public class Product {

    @Id
    private Integer id;

    @Column
    private String name;

    @Column
    private String manufacture;

    @Column
    private Double weight;

    @ManyToOne
    @JoinColumn(name = "subcategory_id")
    private Subcategory subcategory;

    @OneToOne
    private Price price;

    @ManyToMany(mappedBy = "products", fetch = FetchType.EAGER)
    private List<Store> stores;

    public Product() {
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
}
```

```

public void setName(String name) {
    this.name = name;
}

public String getManufacture() {
    return manufacture;
}

public void setManufacture(String manufacture) {
    this.manufacture = manufacture;
}

public Double getWeight() {
    return weight;
}

public void setWeight(Double weight) {
    this.weight = weight;
}

public Subcategory getSubcategory() {
    return subcategory;
}

public void setSubcategory(Subcategory subcategory) {
    this.subcategory = subcategory;
}

public Price getPrice() {
    return price;
}

public void setPrice(Price price) {
    this.price = price;
}

public List<Store> getStores() {
    return stores;
}

public void setStores(List<Store> stores) {
    this.stores = stores;
}

```

@Override

```
public boolean equals(Object object) {  
    if (this == object) return true;  
    if (object == null || getClass() != object.getClass()) return false;  
    Product product = (Product) object;  
    return name.equals(product.getName()) &&  
        id.equals(product.getId()) &&  
        manufacture.equals(product.getManufacture())&&  
        price.equals(product.getPrice())&&  
        stores.equals(product.getStores()) &&  
        subcategory.equals(product.getSubcategory())&&  
        weight.equals(product.getWeight());  
}
```

@Override

```
public int hashCode() {  
  
    final int prime = 31;  
    int result = 1;  
  
    result = prime * result + ((price == null) ? 0 : price.hashCode());  
    result = prime * result + ((name == null) ? 0 : name.hashCode());  
    result = prime * result + ((manufacture == null) ? 0 :  
manufacture.hashCode());  
    result = prime * result + ((weight == null) ? 0 : weight.hashCode());  
    result = prime * result + ((subcategory == null) ? 0 :  
subcategory.hashCode());  
    result = prime * result + ((stores == null) ? 0 : stores.hashCode());  
    result = prime * result + id;  
  
    return result;  
}
```

ПРИЛОЖЕНИЕ 2

Класс CategoryService

```
@Service
@Transactional
public class CategoryService implements ICategoryService<CategoryDTO> {

    private static final Logger logger =
LogManager.getLogger(CategoryService.class);
    private ICategoryDao categoryDao;

    public CategoryService() {
    }

    @Autowired
    public void setCategoryDao(ICategoryDao categoryDao) {
        this.categoryDao = categoryDao;
    }

    public List<Category> getAllCategories() {
        List<Category> categories = new ArrayList<Category>();
        try {
            categories = categoryDao.getAll();
        } catch (HibernateException e) {
            logger.log(Level.ERROR, "The import failed." + e.getMessage());
            throw new ServiceException("Some problems with getting information.",
e.getCause());
        }
        return categories;
    }

    public Category getCategoryById(Integer id){
        Category category = new Category();
        try {
            category = categoryDao.getById(id);
        } catch (HibernateException e) {
            logger.log(Level.ERROR, "The import failed." + e.getMessage());
            throw new ServiceException("Some problems with getting information.",
e.getCause());
        }
        return category;
    }

    public List<CategoryDTO> categoriesDto() {
```

```

List<Category> categories = getAllCategories();
ModelMapper modelMapper = new ModelMapper();
List<CategoryDTO> categoryDTOs = new ArrayList<CategoryDTO>();
for (Category category : categories) {
    categoryDTOs.add(modelMapper.map(category, CategoryDTO.class));
}
return categoryDTOs;
}

public CategoryDTO categoryDTO(Integer id){
    Category category = getCategoryById(id);
    ModelMapper modelMapper = new ModelMapper();
    CategoryDTO categoryDTO = modelMapper.map(category,
CategoryDTO.class);
    return categoryDTO;
}

@Transactional
public void createCategory(CategoryDTO categoryDTO) {
    try{
        categoryDao.create(getCategory(categoryDTO));
    }
    catch (HibernateException e){
        logger.log(Level.ERROR, "Creating failed." + e.getMessage());
        throw new ServiceException("Creating failed.", e.getCause());
    }
}

@Transactional
public void updateCategory(CategoryDTO categoryDTO) {

    Category newCategory = getCategory(categoryDTO);
    Category oldCategory = categoryDao.getById(newCategory.getId());
    if(oldCategory != null){
        oldCategory = newCategory;
        try{
            categoryDao.update(oldCategory);
        }
        catch (HibernateException e){
            logger.log(Level.ERROR, "Updating failed." + e.getMessage());
            throw new ServiceException("Updating failed.", e.getCause());
        }
    }
}
}

```

```

@Transactional
public void deleteCategory(Integer id) {
    Category category = categoryDao.getById(id);
    if(category != null){
        try{
            categoryDao.delete(category);
        }
        catch (HibernateException e){
            logger.log(Level.ERROR, "Deleting failed." + e.getMessage());
            throw new ServiceException("Deleting failed.", e.getCause());
        }
    }
}

private Category getCategory(CategoryDTO categoryDTO){
    ModelMapper modelMapper = new ModelMapper();
    return modelMapper.map(categoryDTO, Category.class);
}
}

```


ПРИЛОЖЕНИЕ 3

Класс ProductDTO

```
public class ProductDTO {  
  
    private Integer id;  
  
    private String name;  
  
    private String manufacture;  
  
    private Double weight;  
  
    private SubcategoryDTO subcategory;  
  
    private PriceDTO price;  
  
    private List<StoreDTO> stores;  
  
    public Integer getId() {  
        return id;  
    }  
  
    public void setId(Integer id) {  
        this.id = id;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getManufacture() {  
        return manufacture;  
    }  
  
    public void setManufacture(String manufacture) {  
        this.manufacture = manufacture;  
    }  
  
    public Double getWeight() {
```

```
        return weight;
    }

    public void setWeight(Double weight) {
        this.weight = weight;
    }

    public SubcategoryDTO getSubcategory() {
        return subcategory;
    }

    public void setSubcategory(SubcategoryDTO subcategory) {
        this.subcategory = subcategory;
    }

    public PriceDTO getPrice() {
        return price;
    }

    public void setPrice(PriceDTO price) {
        this.price = price;
    }

    public List<StoreDTO> getStores() {
        return stores;
    }

    public void setStores(List<StoreDTO> stores) {
        this.stores = stores;
    }
}
```

ПРИЛОЖЕНИЕ 4

Класс ProductController

```
@RestController
@RequestMapping("/products")
public class ProductController {

    private static final Logger logger =
LogManager.getLogger(ProductController.class);
    private IProductService productService;
    private IPriceService priceService;

    @Autowired
    public void setPriceService(IPriceService priceService) {
        this.priceService = priceService;
    }
    @Autowired
    public void setProductService(IProductService productService) {
        this.productService = productService;
    }

    @GetMapping("/subcategory/{id}")
    public List<ProductDTO> getProductFromSubcategory(@PathVariable
Integer id) {
        return productService.getProductsDTOFromOneSubcategory(id);
    }

    @GetMapping("/category/{id}")
    public List<ProductDTO> getProductFromCategory(@PathVariable Integer
id) {
        return productService.getProductsDTOFromOneCategory(id);
    }

    @PostMapping("/add")
    public ResponseEntity<ProductDTO> createProduct(@RequestBody
ProductDTO product) {
        productService.createProduct(product);
        return ResponseEntity.status(HttpStatus.OK).body(product);
    }

    @PostMapping("/update")
    public ResponseEntity<ProductDTO> updateProduct(@RequestBody
ProductDTO product) {
        productService.updateProduct(product);
    }
}
```

```

        return ResponseEntity.status(HttpStatus.OK).body(product);
    }

    @PostMapping("/newPrice/{id}")
    public void addPrice(@PathVariable Integer id, @RequestBody
    PriceWithoutDate priceWithoutDate) {

        Date date = new Date();
        try{
            String stringDate = priceWithoutDate.getStart_date();
            date = new SimpleDateFormat("dd-MM-yyyy").parse(stringDate);
        }
        catch(ParseException e){
            logger.log(Level.ERROR, "There is an error with date parsing." +
            e.getMessage());
            throw new ServiceException("There is an error with date parsing.",
            e.getCause());
        }
        PriceDTO priceDTO = new PriceDTO(priceWithoutDate.getId(),
            priceWithoutDate.getPrice(),
            date,
            priceWithoutDate.getIsActive());
        productService.updatePriceOfProduct(id, priceDTO);
    }

    @DeleteMapping("/delete/{id}")
    public void deleteProduct(@PathVariable Integer id){
        productService.deleteProduct(id);
    }

    @GetMapping
    public List<ProductDTO> findProducts(@RequestParam(value = "name",
    required = false) String name,
        @RequestParam(value = "manufacture", required =
    false) String manufacture){
        return productService.getByNameOrManufacture(name, manufacture);
    }

    @GetMapping("/prices")
    public List<PriceDTO> getPricesOfProduct(@RequestParam(value = "id",
    required = true) Integer id,
        @RequestParam(value = "firstDate", required = false)
    String firstDate,
        @RequestParam(value = "secondDate", required =
    false) String secondDate){

```

```
    if(firstDate == null && secondDate == null){  
        return priceService.getPricesOfProduct(id);  
    }  
    else{  
        return priceService.getPricesOfProduct(id, firstDate, secondDate);  
    }  
}  
}
```