

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**  
**федеральное государственное бюджетное образовательное учреждение высшего образования**  
**«Российский экономический университет имени Г.В. Плеханова»**  
**Московский приборостроительный техникум**

**ОТЧЕТ**

по учебной практике

УП.04.01

Внедрение и поддержка программного обеспечения

Профессионального модуля ПМ.04

Сопровождение и обслуживание программного обеспечения компьютерных систем

Специальность 09.02.07

Информационные системы и программирование. Программист

Студент

\_\_\_\_\_

*подпись*

Казанин Р.П.

*фамилия, имя, отчество*

Группа

П50-2-18

Руководитель по практической подготовке от техникума

\_\_\_\_\_

*подпись*

Юрий Владимирович Севастьянов

*фамилия, имя, отчество*

«09».11.2021 года

## ОГЛАВЛЕНИЕ

ПРАКТИЧЕСКАЯ РАБОТА 1 .....	3
Тема: Знакомство с Java Spring и средой разработки IntelliJ IDEA. Методы Post и Get. Создание калькулятора.....	3
ПРАКТИЧЕСКАЯ РАБОТА 2 .....	11
Тема: Подключение зависимостей для работы с БД. Создание моделей. Создание и реализация контроллера, репозитория и представления. ....	11
ПРАКТИЧЕСКАЯ РАБОТА №3 .....	18
Тема: Работа со страницей подробнее и поиском .....	18
ПРАКТИЧЕСКАЯ РАБОТА 4 .....	24
Тема: Создание страницы, методов и ссылок для Редактирования. Создание методов удаления. ....	24
ПРАКТИЧЕСКАЯ РАБОТА 5 .....	29
Тема: Работа с валидацией .....	29
ПРАКТИЧЕСКАЯ РАБОТА №6 .....	32
Тема: Работа с связями .....	32
ИНДИВИДУАЛЬНЫЙ ПРОЕКТ.....	43
Тема: Закрепление материала.....	43

# ПРАКТИЧЕСКАЯ РАБОТА 1

Тема: Знакомство с Java Spring и средой разработки IntelliJ IDEA. Методы Post и Get. Создание калькулятора.

Цель работы: в данной практической работе необходимо ознакомиться с Java Spring и средой разработки IntelliJ IDEA, а также изучить такие аннотации как `@Controller`, `@RequestParam`, `@GetMapping` и `@PostMapping`, и попробовать их реализовать в приложении калькулятор.

1) `Controller` - обрабатывает запрос пользователя, создаёт соответствующую Модель и передаёт её для отображения в Вид

2) `RequestParam` - эта аннотация используется для того, чтобы методы обработчики могли получить параметры из http-запроса.

3) Аннотация `@GetMapping` — это просто аннотация, которая содержит `@RequestMapping (method = RequestMethod.GET)`. Она также позволяет более глубоко настроить метод-обработчик.

Ее параметры (они конвертируются в аналогичные параметры `@RequestMapping`):

`path` — URI

`headers` — заголовки

`name` — имя обработчика

`params` — параметры

`produces` — тип возвращаемых данных (JSON, XML, текст).

Используется в REST `consumes` — тип принимаемых данных. Используется в REST

По умолчанию аннотация принимает путь до метода. `@GetMapping ("managers") = @GetMapping (path = "managers")`

4) Аннотация `@PostMapping` - обрабатывает post-запросы. Все характеристики описаны в аннотации `@GetMapping`

5) `bootstrap` - это бесплатный фреймворк с открытым исходным кодом для создания веб-сайтов и веб-приложений. Это самый популярный

фреймворк HTML, CSS и JS для разработки адаптивных и мобильных проектов в Интернете.

Bootstrap довольно легок в подключении необходимо просто сначала скопировать нужный Html-код и вставить в проект, а затем зайти на сайт BootstrapCDN и скопировать оттуда ссылку на CSS стили и вставить вот в таком виде:

```
<head>
  <meta charset="UTF-8">
  <title>Home</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
```

Рисунок 1 - Ссылка на стили

Затем можно с Bootstrap скопировать любой дизайн, который нравится и подогнать его под себя. Вот один из примеров:

```
<body>
<div th:insert="fragments/header :: header"></div>
  <a href="/gun/add">Добавить оружие</a>
  <div th:each="el : ${Allguns}">
    <div class="bg-light p-5 rounded mt-3">
      <h1 align="center" th:text="${el.name}"></h1>

      <p align="center" class="lead" th:text="${el.getCalibr()}"></p>

      <p align="center" class="lead" th:text="${el.getCost()}"></p>
    </div>
  </div>
<div th:insert="fragments/footer :: footer"></div>
</body>
</html>
```

Рисунок 2 - Пример кода Bootstrap

И вот что я получил в итоге:

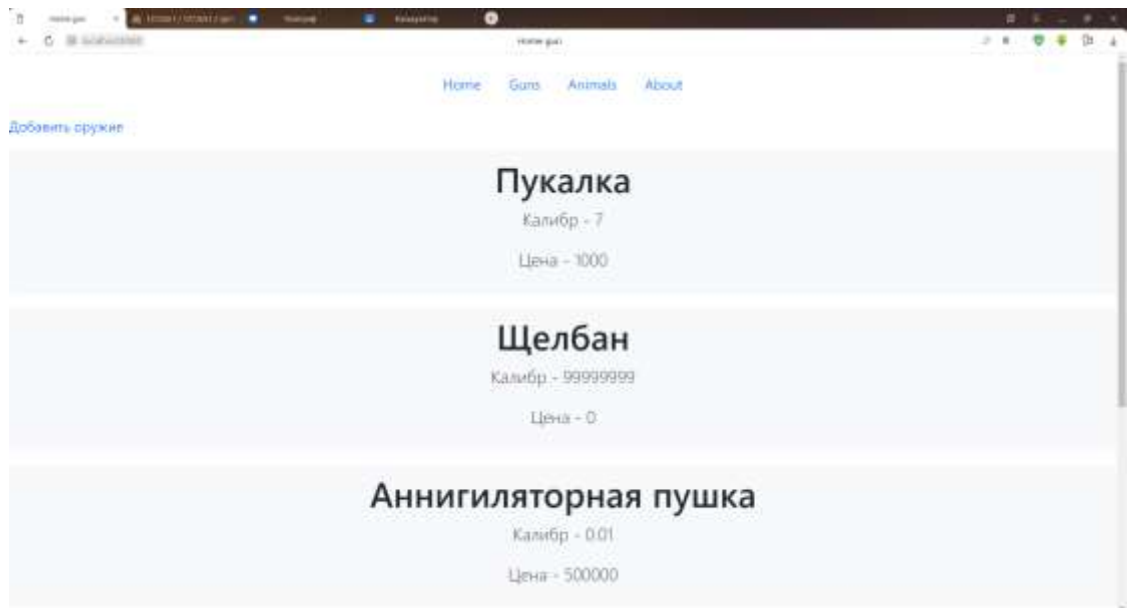


Рисунок 3 - Сайт с дизайном

б) `th:fragment` - необходимы для того чтобы не писать один и тот же код на каждой новой странице, а просто добавить ссылку и применить нужный контейнер.

Для того чтобы создать фрагменты нужно создать папку `fragments` в папке `templates`.

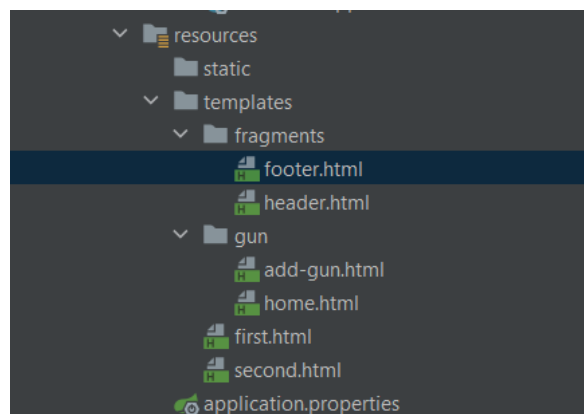


Рисунок 4 - Создание папки

Затем создаем `html`-файлы и добавляем туда нужный нам контейнер и называем его. Пример:

```

<div th:fragment="header">
  <div class="container">
    <header class="d-flex flex-wrap align-items-center justify-content-center justify-content-md-between py-3 mb-4 border-bottom">
      <a href="/" class="d-flex align-items-center col-md-1 mb-2 mb-md-0 text-dark text-decoration-none">
        <svg class="bi me-2" width="48" height="32" role="img" aria-label="Bootstrap"><use xlink:href="#bootstrap"></use></svg>
      </a>

      <ul class="nav col-12 col-md-auto mb-2 justify-content-center mb-md-0">
        <li><a href="/" class="nav-link px-2 link-secondary">Home</a></li>
        <li><a href="/home" class="nav-link px-2 link-dark">Features</a></li>
        <li><a href="/gun/" class="nav-link px-2 link-dark">Опытно</a></li>
        <li><a href="/s" class="nav-link px-2 link-dark">FAQs</a></li>
        <li><a href="/s" class="nav-link px-2 link-dark">About</a></li>
      </ul>

      <div class="col-md-3 text-end">
        <button type="button" class="btn btn-outline-primary me-2">Login</button>
        <button type="button" class="btn btn-primary">Sign up</button>
      </div>
    </header>
  </div>
</div>

```

Рисунок 5 - Контейнер «header»

И затем для использования данного контейнера просто вставляем ссылку в нужный html-файл вот так:

```

<body>
  <div th:insert="fragments/header :: header"></div>

```

Рисунок 6 - Использование фрагмента

Код программы:

На рисунке 1 изображены все зависимости используемы в данном проекте.

```

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;

```

Рисунок 7 - Все зависимости

На рисунке 2 изображен контроллер у которого есть атрибут name со значением «Hello world». Данный контроллер возвращает свои значения на http-страницу под название «first».

```

@GetMapping("/")
public String hello(Model model) {
    model.addAttribute( attributeName: "name", attributeValue: "Hello world");
    return "first";
}

```

Рисунок 8 - Контроллер «hello»

На рисунке 3 предоставлен контроллер, который при помощи метода get производит расчеты введенных пользователем значений и затем отправляет результат на другую страницу.

```

@GetMapping("/home")
public String home(@RequestParam(name = "num1",required = false) Double num1,
                  @RequestParam(name = "num2",required = false) Double num2,
                  @RequestParam(name = "act",required = false) String act,
                  Model model){
    model.addAttribute( attributeName: "digid1",num1);
    model.addAttribute( attributeName: "digid2",num2);
    model.addAttribute( attributeName: "action",act);

    Double res = 0.0;
    if(num1 != null && num2 != null) {
        if (act.equals("сложить")) res = num1 + num2;
        if (act.equals("вычесть")) res = num1 - num2;
        if (act.equals("умножить")) res = num1 * num2;
        if (act.equals("разделить")) res = num1 / num2;
    }
    model.addAttribute( attributeName: "res",res);

    return "second";
}

```

Рисунок 9 - Контроллер «home»

На рисунке 4 предоставлен контроллер, который при помощи метода post производит расчеты введенных пользователем значений и затем отправляет результат на другую страницу.

```

@PostMapping("/home")
public String ome(@RequestParam(name = "num1", required = false) Double num1,
                 @RequestParam(name = "num2", required = false) Double num2,
                 @RequestParam(name = "act", required = false) String act,
                 Model model){
    model.addAttribute("digid1", num1);
    model.addAttribute("digid2", num2);
    model.addAttribute("action", act);

    Double res = 0.0;
    if(num1 != null && num2 != null)
    {
        if(act.equals("сложить")) res = num1+num2;
        if(act.equals("вычесть")) res = num1-num2;
        if(act.equals("умножить")) res = num1*num2;
        if(act.equals("разделить")) res = num1/num2;
    }
    model.addAttribute("res", res);

    return "second";
}

```

Рисунок 10 - Контроллер «ome»

Далее на изображениях будет предоставлен результат работы программы:

hello world! оаоаоа ммм))

Сложить ▼

.....

Сложить ▼

Рисунок 11 - Главная страница



hello world! оаоаоа ммм))

<input type="text" value="4"/>	<input type="text" value="5"/>	Сложить ▾
<input type="button" value="отправить get"/>		

.....

<input type="text"/>	<input type="text"/>	Сложить ▾
<input type="button" value="отправить post"/>		

[на 2 стр](#)  
[на 2 стр ссылка 2](#)

Рисунок 12 - Ввод значений для get

Digid 1 - 4.0  
Digid 2 - 5.0  
Action - сложить  
Result - 9.0  
[на 1 стр](#)

Рисунок 13 - Ответ

hello world! оаоаоа ммм))

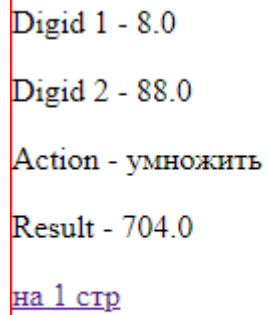
<input type="text"/>	<input type="text"/>	Сложить ▾
<input type="button" value="отправить get"/>		

.....

<input type="text" value="8"/>	<input type="text" value="88"/>	Умножить ▾
<input type="button" value="отправить post"/>		

[на 2 стр](#)  
[на 2 стр ссылка 2](#)

Рисунок 14 - Ввод значений для post

A screenshot of a web application's output, enclosed in a red rectangular border. The text is as follows:

Digid 1 - 8.0  
Digid 2 - 88.0  
Action - умножить  
Result - 704.0  
[на 1 стр](#)

The text is displayed in a monospaced font. The first four lines are in black, while the last line is a purple hyperlink.

Рисунок 15 - Ответ

Вывод: в данной практической работе я ознакомился с Java Spring и средой разработки IntelliJ IDEA, а также изучил такие аннотации как `@Controller`, `@RequestParam`, `@GetMapping` и `@PostMapping`, и разработал приложение калькулятор.

## ПРАКТИЧЕСКАЯ РАБОТА 2

Тема: Подключение зависимостей для работы с БД. Создание моделей.

Создание и реализация контроллера, репозитория и представления.

Цель работы: в данной практической работе необходимо описать подключение новых зависимостей в БД. Описать строку подключения. Создать новую Модель с 5 полям, где 3 из них будут типа String и 2 типа Int. Создать и описать Контроллер, Репозиторий и представления. Описать аннотации: @RequestMapping, @Autowired, @Entity, @Id, @GeneratedValue. Отобразить результат работы сайта.

1) Зависимость MySQL Connector - необходима для того чтобы можно было подключиться к базе данных MySQL.

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.27</version>
</dependency>
```

Рисунок 16 - Зависимость «MySQL Connector»

Зависимость Spring Boot Starter Data JPA - предлагает набор очень мощных и сильно абстрагированных интерфейсов, которые используются для взаимодействия с любой базовой базой данных. Базы данных могут быть MySQL, MongoDB, Elasticsearch или любой другой поддерживаемой базой данных. Другие преимущества для Spring Data JPA включают в себя: Поддержка создания расширенных репозиториях на основе JPA Convention. Встроенная поддержка пагинации и динамическое выполнение запросов. Поддержка сопоставления сущностей на основе XML.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  <version>2.3.12.RELEASE</version>
</dependency>
```

Рисунок 17 - Зависимость «Spring Boot Starter Data JPA»

Чтобы подключить новую зависимость необходимо зайти на сайт MVNRepository ввести название нужной зависимости и скопировать код и вставить в файл pom.xml.

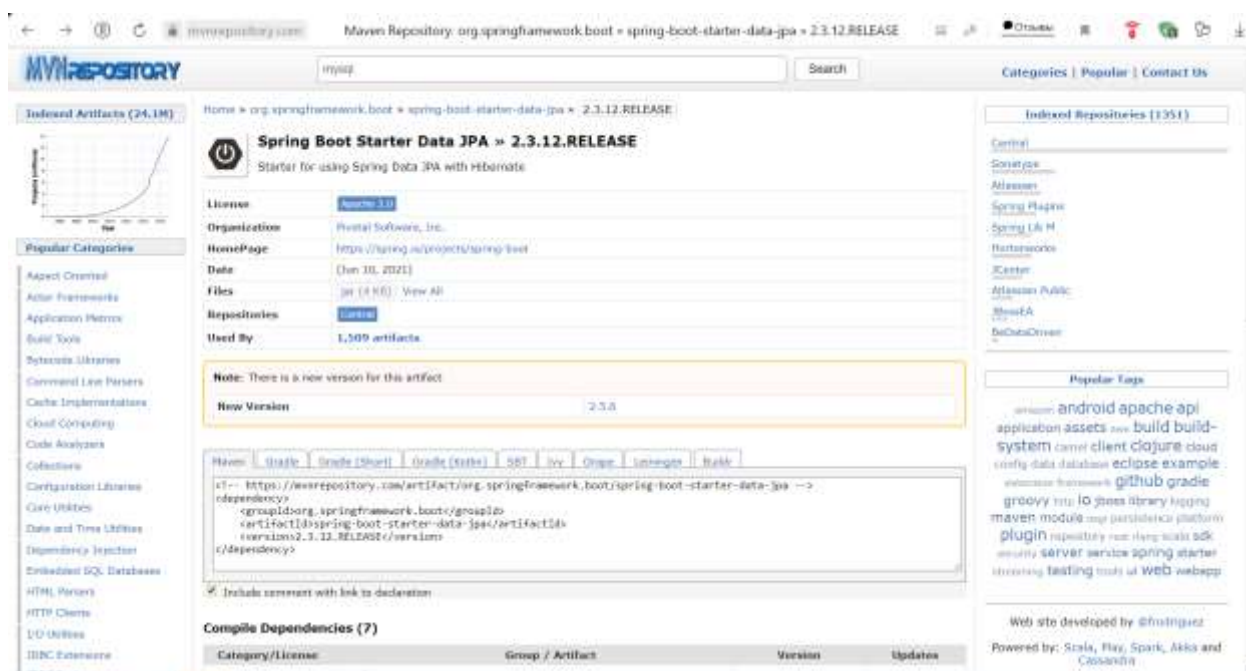


Рисунок 18 - Подключение новых зависимостей

2) Строку подключения необходимо прописать в файле application.properties.

Первая строка означает, что будет происходить с БД после добавления в нее новых данных. Например, update означает обновление БД после добавления в нее новых данных, create - означает создание новой базы данных, а create-drop - удаление старой и создание новой БД с обновленными данными.

Вторая строка означает название нашей БД (оно должно совпадать с названием БД в PhpMyAdmin), а также адрес ее подключения.

Третья строка означает имя пользователя для входа в БД.

Четвертая строка означает пароль пользователя для входа в БД.

```
server.port=8080  
  
spring.jpa.hibernate.ddl-auto=update  
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/springbd  
spring.datasource.username=root  
spring.datasource.password=|
```

Рисунок 19 - Строка подключения

### 3) Создание новой модели

Для того чтобы создать новую модель необходимо в папке models создать новый Java class.

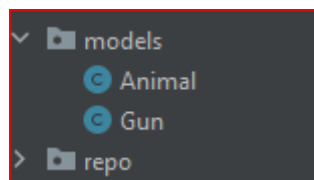


Рисунок 20 - Модель Animal

Затем в ней необходимо прописать необходимые поля для создания таблицы.

Аннотация `entity` - указывает на то, что данный класс является сущностью.

Аннотация `Id` - указывает, что данное поле является первичным ключом, т.е. это свойство будет использоваться для идентификации каждой уникальной записи.

Аннотация `GeneratedValue` - свойство будет генерироваться автоматически, в скобках можно указать каким образом. `Identity` - у каждой таблицы будет свой собственный подсчет `Id`. `Auto` - у всех таблиц будет один подсчет `Id`.

```

@Entity
public class Animal {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    public Long getId() {
        return id;
    }
    public Animal() {
    }
    public Animal(String color, String nazvanie, String name, Integer legs, Integer age) {
        this.color = color;
        this.nazvanie = nazvanie;
        this.name = name;
        this.legs = legs;
        this.age = age;
    }

    public String getColor() { return color; }

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

    public void setColor(String color) { this.color = color; }

    public void setNazvanie(String nazvanie) { this.nazvanie = nazvanie; }

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

    public void setLegs(Integer legs) { this.legs = legs; }

    public void setAge(Integer age) { this.age = age; }

    public String getNazvanie() { return nazvanie; }

    public String getName() { return name; }
}

```

Рисунок 21 - Класс

Далее создаем репозиторий для модели и называем его PassportRepository и подключаем расширение для нашей модели.

```

public interface GunRepository extends CrudRepository<Gun, Long> {
}

```

Рисунок 22 - Создание репозитория

Далее создаем контроллер для работы с моделью.

Аннотация RequestMapping - это квинтэссенция аннотации в Spring framework, которая позволяет нам сопоставлять HTTP-запросы с методами, которые мы хотели бы запустить.

```
@Controller
@RequestMapping("/animal")
public class AnimalController {

    @Autowired
    private AnimalRepository animalRepository;

    @GetMapping("/")
    public String homeanimal(Model model) {
        Iterable<Animal> animals = animalRepository.findAll();
        model.addAttribute("AllAnimals", animals);
        return "animal/home";
    }

    @GetMapping("/add")
    public String animalAddview(Model model) { return "animal/add-animal"; }

    @PostMapping("/add")
    public String animalAdd(
        @RequestParam("name")String name,
        @RequestParam("nazvanie")String nazvanie,
        @RequestParam("color")String color,

        @RequestParam("age")Integer age,
        @RequestParam("legz")Integer legz,

        Model model) {

        Animal animal = new Animal(color, nazvanie, name, legz, age);
        animalRepository.save(animal);

        return "redirect:/animal/";
    }
}
```

Рисунок 23 - Создаем контроллер

При создании метода PostMapping необходимо было добавить в модель конструктор для того чтобы можно было сохранять новых животных.

Затем в контроллере прописываем наш репозиторий чтобы автоматически заполнять все данные.

Аннотация Autowired - отвечает за авто заполнение зависимостей.



```
@Autowired
private AnimalRepository animalRepository;
```

Рисунок 24 - Подключение репозитория

Далее создаем страницы и прописываем в них данные для вывода таблицы и заполнения таблицы в БД.

Для вывода данных мы используем метод из контроллера homepassport.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Home animal</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
  <div th:insert="fragments/header :: header" ></div>
  <a href="/animal/add">Добавить животное</a>

  <div th:each="el : ${AllAnimals}">
    <div class="bg-light p-2 rounded mt-3">
      <h1 align="center" th:text="${el.getName()}"></h1>

      <p align="center" class="lead" th:text="Цвет : " + ${el.getColor()}"></p>
      <p align="center" class="lead" th:text="Название : " + ${el.getNazvanie()}"></p>
      <p align="center" class="lead" th:text="Возраст : " + ${el.getAge()}"></p>
      <p align="center" class="lead" th:text="Количество : " + ${el.getLegs()}"></p>
    </div>
  </div>

  <div th:insert="fragments/footer :: footer" ></div>
</body>
</html>
```

Рисунок 25 - Домашняя страница

Для добавления новых животных мы используем метод из контроллера addpassport.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>NewAnimal</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
  <div th:insert="fragments/header :: header" ></div>
  <form method="post">
    <input type="text" name="name" placeholder="Имя">
    <input type="text" name="nazvanie" placeholder="Название">
    <input type="text" name="color" placeholder="Цвет">

    <input type="text" name="age" placeholder="Возраст">
    <input type="text" name="legz" placeholder="Цена">

    <button type="submit" class="btn btn-warning">Добавить животное</button>
  </form>
  <div th:insert="fragments/footer :: footer" ></div>
</body>
</html>
```

Рисунок 26 - Добавление животного



Результат работы:

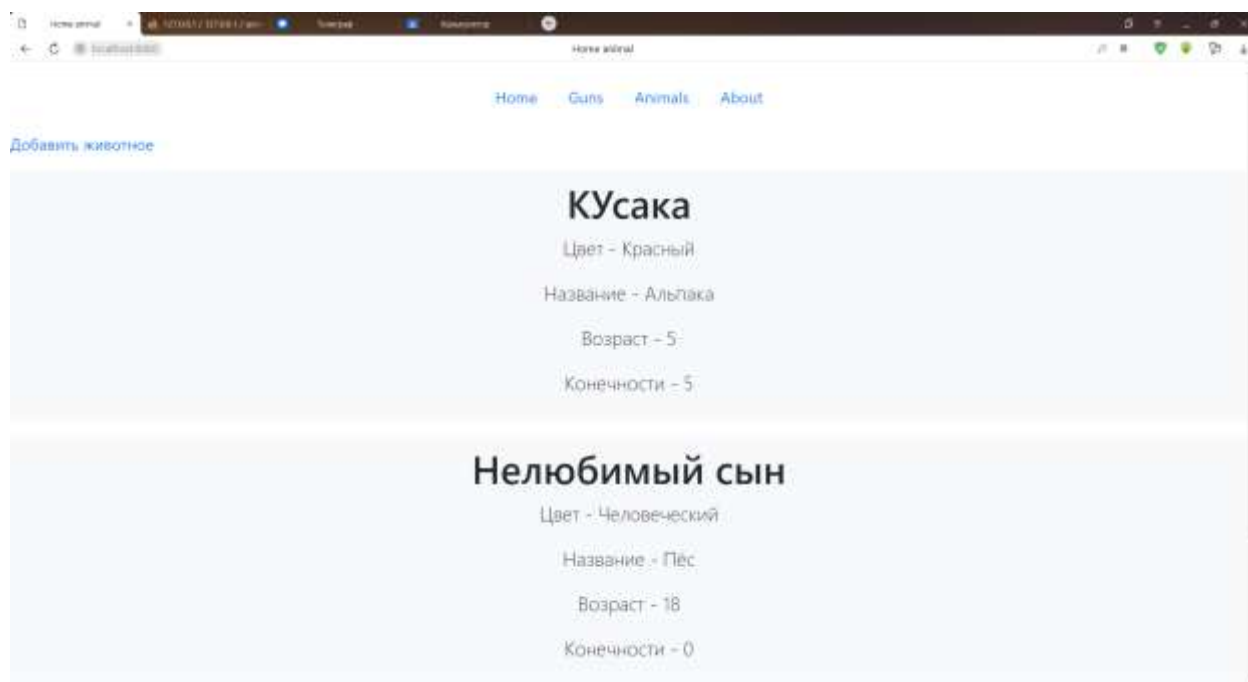


Рисунок 27 - Домашняя страница

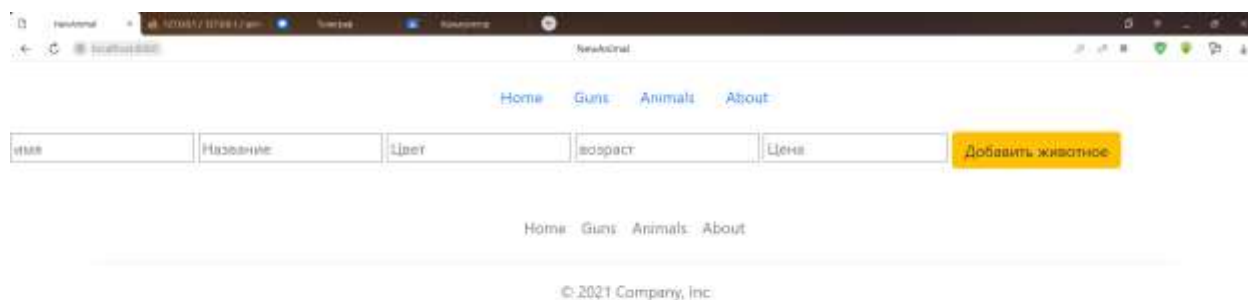


Рисунок 28 - Страница добавления животного

Вывод: в данной практической работе было описано подключение новых зависимостей в БД. Описана строка подключения. Создана новая Модель с 5 полям, где 3 из них будут типа String и 2 типа Int. Создан и описан Контроллер, Репозиторий и представления. Описаны аннотации: @RequestMapping, @Autowired, @Entity, @Id, @GeneratedValue. Отображен результат работы сайта.

## ПРАКТИЧЕСКАЯ РАБОТА №3

Тема: Работа со страницей подробнее и поиском

Цель работы: необходимо сделать страницу "Подробнее" и Поиск для модели из 2 практической.

Добавление ссылок на страницу поиска и страницу с подробным описанием выбранного элемента

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Home animal</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div th:insert="fragments/header :: header" ></div>
  <div th:align="center">
    <a href="/animal/add">Добавить животное</a> <br>
    <a href="/animal/filter">Фильтр</a>
  </div>
  <div th:each="el : ${AllAnimals}">
    <div align="center" class="bg-light p-2 rounded mt-3">
      <h1 align="center" th:text="${el.getName()}"></h1>
      <a align="center" th:href="/animal/animal-view/' + ${el.id}">Подробнее</a>
    </div>
  </div>
<div th:insert="Fragments/footer :: footer" ></div>
</body>
</html>
```

Рисунок 29 – Ссылки на страницы

Добавление в контроллере метода для отображения подробной информации по выбранному id

```
@RequestMapping("/{id}")
public String viewgun(@PathVariable(value = "id") Long id, Model model) {
    Optional<Animal> guns = animalRepository.findById(id);
    ArrayList<Animal> res = new ArrayList<>();
    guns.ifPresent(res::add);
    model.addAttribute( attributeName: "selectedAnimal",res);
    return "animal/view-animal";
}
```

Рисунок 30 – Отображение подробной информации

Аннотация @PathVariable используется для аргумента метода, чтобы привязать ее к значению переменной шаблона URI

#### Добавление метода поиска

```
public interface AnimalRepository extends CrudRepository<Animal, Long> {  
    List<Animal> findByName(String name);  
    List<Animal> findByNameContaining(String name);  
}
```

Рисунок 31 – Репозиторий

Реализация поиска. При вводе названия происходит поиск по точному значению, если ничего не найдено, будет работать поиск по схожему значению.

```
@PostMapping("/filter")  
public String filtergun(@RequestParam("nameanimal")String name, Model model){  
    List<Animal> res = animalRepository.findByName(name);  
    if(res.size()>0){  
        model.addAttribute("attributeName: findainmals", res);  
    }else {  
        res = animalRepository.findByNameContaining(name);  
        model.addAttribute("attributeName: findainmals", res);  
    }  
    return "animal/filter-animal";  
}
```

Рисунок 32 – Реализация поиска

#### Создание страницы поиска

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css" rel="stylesheet">
  <title>Gun</title>
</head>
<body>
<div th:insert="fragments/header :: header" ></div>
<div align="center">
  <h1>Поиск</h1>
  <form method="post">
    <input type="text" name="nameanimal">
    <button type="submit" class="btn btn-info">Поиск</button>
  </form>
</div>

<div th:each="el : ${findainmals}">
  <div class="bg-light p-2 rounded mt-3">
    <h1 align="center" th:text="${el.getName()}"></h1>
    <a align="center" th:href="'/animal/animal-view/' + ${el.id}">Подробнее</a>
  </div>
</div>

<div th:insert="fragments/footer :: footer" ></div>
</body>
</html>

```

Рисунок 33 – Страница поиска

## Создание страницы подробного описания

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css" rel="stylesheet">
  <title>Animal</title>
</head>
<body>
<div th:insert="fragments/header :: header" ></div>
<div th:each="el : ${selectedAnimal}">
  <p align="center" class="lead" th:text="'Цвет - ' + ${el.getColor()}"></p>
  <p align="center" class="lead" th:text="'Название - ' + ${el.getNazvanie()}"></p>
  <p align="center" class="lead" th:text="'Возраст - ' + ${el.getAge()}"></p>
  <p align="center" class="lead" th:text="'Конечности - ' + ${el.getLegs()}"></p>
  <a align="center" th:href="/animal/">Назад</a>
</div>
<div th:insert="fragments/footer :: footer" ></div>
</body>
</html>

```

Рисунок 34 – Страница подробного описания

## Результат

[Home](#)

[Guns](#)

[Animals](#)

[About](#)

[Добавить животное](#)

[Фильтр](#)

## КУсака

[Подробнее](#)

## Нелюбимый сын

[Подробнее](#)

[Home](#)

[Guns](#)

[Animals](#)

[About](#)

© 2021 Company, Inc

[Home](#)

[Guns](#)

[Animals](#)

[About](#)

Цвет - Красный

Название - Альпака

Возраст - 5

Конечности - 5

[Home](#)

[Guns](#)

[Animals](#)

[About](#)

© 2021 Company, Inc

Рисунок 36 – Подробное описание

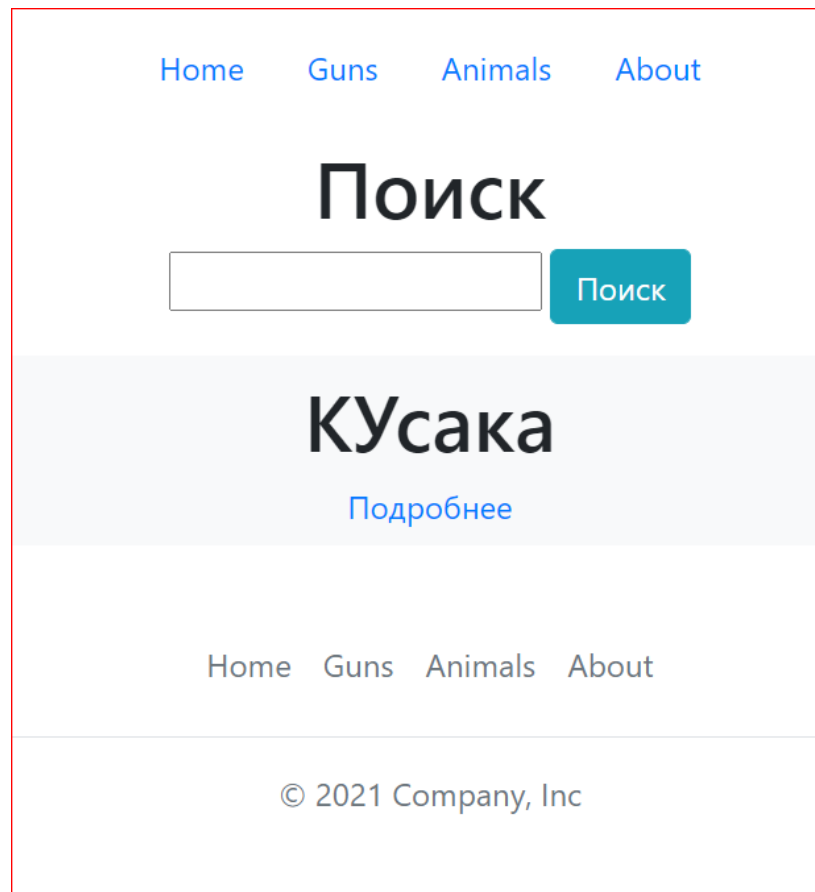


Рисунок 37 – Поиск по схожему значению

Вывод: В ходе выполнения данной практической работы была сделана страница "Подробнее" и реализован Поиск для модели из 2 практической.

## ПРАКТИЧЕСКАЯ РАБОТА 4

Тема: Создание страницы, методов и ссылок для Редактирования. Создание методов удаления.

Цель работы: необходимо сделать страницу "Изменить" и "Удалить" для модели из 2 практической.

Создадим страницу edit-animal.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
  <div th:insert="fragments/header :: header" ></div>

  <div th:each="el : ${editAnimal}">
    <form method="post">
      <input type="text" name="name" th:value="${el.getName()}" placeholder="Имя">
      <input type="text" name="nazvanie" th:value="${el.getNazvanie()}" placeholder="Название">
      <input type="text" name="color" th:value="${el.getColor()}" placeholder="Цвет">

      <input type="text" name="age" th:value="${el.getAge()}" placeholder="Возраст">
      <input type="text" name="legz" th:value="${el.getLegs()}" placeholder="Цена">

      <button type="submit" class="btn btn-warning">Изменить животное</button>
    </form>
  </div>
  <div th:insert="fragments/footer :: footer" ></div>
</body>
</html>
```

Рисунок 38 - edit-animal.html

Изменим страницу просмотра, добавим код из скриншота ниже в каждый элемент.

```
<form align="center" th:action="/animal/animal-view/'${el.getId()}'/del" method="post">
  <button type="submit" class="btn btn-warning">Удалить</button>
</form>
<a align="center" th:href="/animal/">Назад</a>
```

Рисунок 39 – изменение в animal-view



Далее перейдем к созданию контроллеров, на скриншоте ниже контроллер для открытия страницы изменения

```
@GetMapping("/animal-view/{id}/edit")
public String editanimalview(@PathVariable(value = "id")Long id, Model model){

    if(!animalRepository.existsById(id))
    {
        return "redirect:/animal/";
    }
    Optional<Animal> animals = animalRepository.findById(id);
    ArrayList<Animal> res = new ArrayList<>();
    animals.ifPresent(res::add);
    model.addAttribute("editAnimal",res);
    return "animal/edit-animal";
}
```

Рисунок 40 – контроллер editanimalview

Следующий контроллер применяет введенные нами изменения

```
@PostMapping("/animal-view/{id}/edit")
public String editanimal(
    @PathVariable(value = "id")Long id,

    @RequestParam("name")String name,
    @RequestParam("nazvanie")String nazvanie,
    @RequestParam("color")String color,

    @RequestParam("age")Integer age,
    @RequestParam("legz")Integer legz,
    Model model)
{
    Animal animal = animalRepository.findById(id).orElseThrow();
    animal.setName(name);
    animal.setNazvanie(nazvanie);
    animal.setColor(color);
    animal.setAge(age);
    animal.setLegs(legz);
    animalRepository.save(animal);
    return "redirect:/animal/";
}
```

Рисунок 41 – editanimal

Последний контроллер выполняет удаление из базы

```
@PostMapping("/animal-view/{id}/del")
public String deleteanimal(@PathVariable(value = "id")Long id,Model mode){
    Animal animal = animalRepository.findById(id).orElseThrow();
    animalRepository.delete(animal);
    return "redirect:/animal/";
}
```

Рисунок 42 – deleteanimal

Результат:

Имя - КУсака

Цвет - Красный

Название - Ебака

Возраст - 5

Конечности - 5

Редактировать

Удалить

Назад

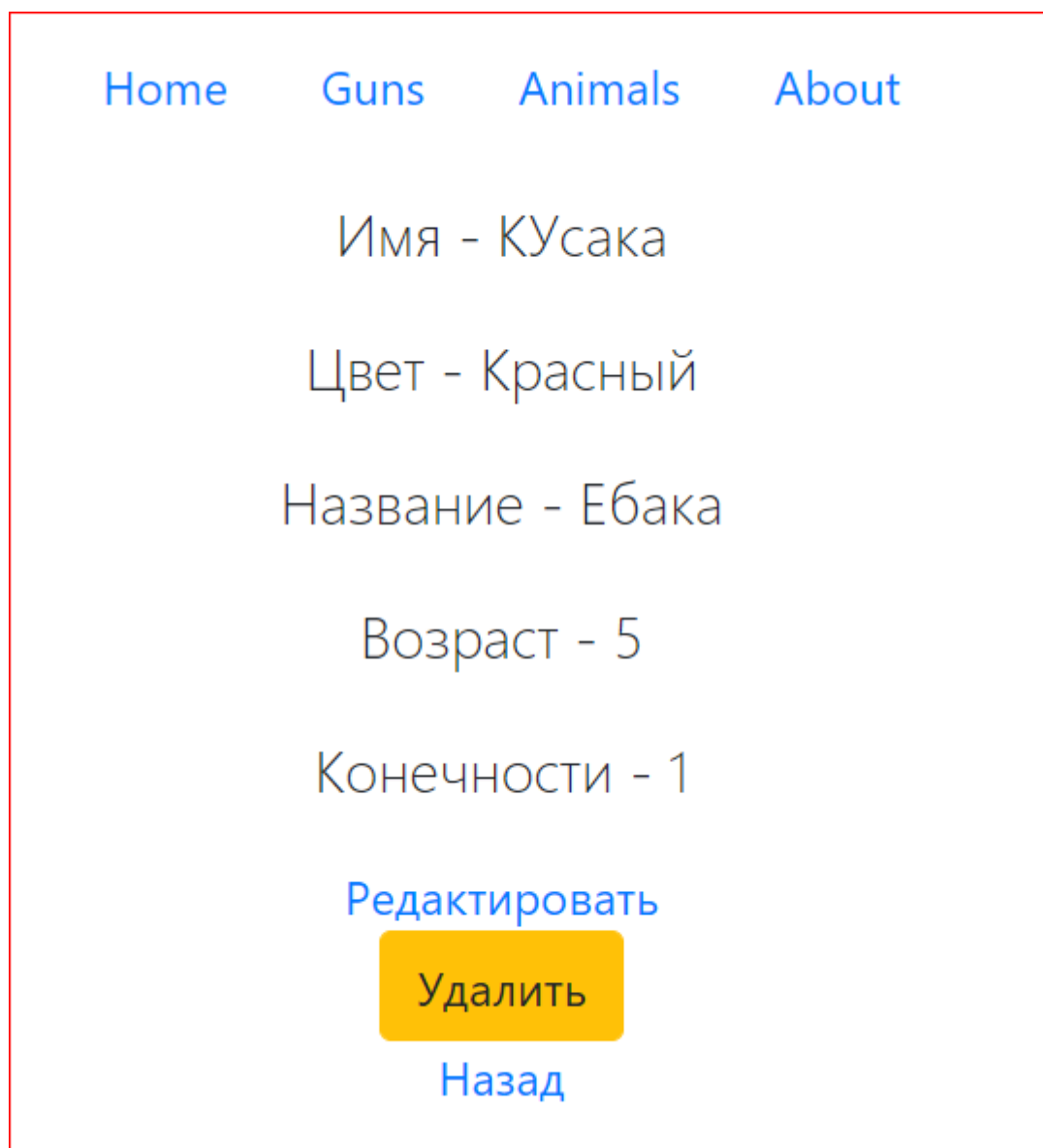
Рисунок 43 – окно просмотра одного животного

## Изменим кол-во конечностей



A screenshot of a web application interface. At the top, there is a navigation bar with links: Home, Guns, Animals, and About. Below the navigation bar, there is a form with five input fields. The first field contains 'КУсака', the second 'Ебака', the third 'Красный', the fourth '5', and the fifth '1'. To the right of these fields is a yellow button labeled 'Изменить животное'. Below the form, there is another navigation bar with the same links: Home, Guns, Animals, and About.

Рисунок 44 – изменение



A screenshot of a web application interface. At the top, there is a navigation bar with links: Home, Guns, Animals, and About. Below the navigation bar, the page displays the following information:  
Имя - КУсака  
Цвет - Красный  
Название - Ебака  
Возраст - 5  
Конечности - 1  
Below the information, there are two buttons: 'Редактировать' (in blue text) and 'Удалить' (in a yellow button). At the bottom, there is a link 'Назад' (in blue text).

Рисунок 45 – изменения вступили в силу

И в завершение нажмем кнопку удалить в окне просмотра

[Добавить животное](#)  
[Фильтр](#)

# Нелюбимый сын

[Подробнее](#)

Рисунок 46 – запись о кусаке удалена

Вывод: В данной работе я научился создавать станицу изменения и удаления данных, а также улучшил свои навыки работы с Java spring.

## ПРАКТИЧЕСКАЯ РАБОТА 5

### Тема: Работа с валидацией

Цель работы: Необходимо добавить валидацию для добавления и редактирования в таблицу той модели, которую создавали. Так же необходимо сделать пользовательские валидаторы.

#### Добавление зависимости

```
</dependency>
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.2.0.Final</version>
</dependency>
</dependencies>
```

Рисунок 47 – Зависимость

```
@NotEmpty(message = "Данное поле не должно быть пустым")
@Size(min = 2, max = 50, message = "Данное поле должно иметь размер от 2 до 50 символов")
private String name, type, form;

@NotNull
@Min(value = 200, message = "Минимальная стоимость - 200")
@Max(value = 200000, message = "Максимальная стоимость - 200000")
private int cost;

@NotNull
@Min(value = 1000, message = "минимальная длина артикула - 1111")
@Max(value = 999999, message = "максимальная длина артикула - 111111")
private int vendorcode;
```

Рисунок 48 – Ограничение

Аннотация @NotEmpty отвечает за то, чтобы строковой элемент таблицы не был пустым.

Аннотация @Size отвечает за допустимый размер строки.

Аннотация @NotNull отвечает за то, чтобы числовой элемент таблицы не был пустым.

Аннотация @Min отвечает за минимальное допустимое значение числа.

Аннотация @Max отвечает за максимальное допустимое значение числа.

## Изменение метода добавления нового элемента в БД

```
@GetMapping("/add")
public String addguitarview(Guitar guitar, Model model) { return "guitar/add-guitar"; }

@PostMapping("/add")
public String addguitar(@Valid Guitar guitar, BindingResult bindingResult, Model model) {
    if(bindingResult.hasErrors()) {
        if(guitar.getCost() == 123123) {
            ObjectError error = new ObjectError("Cost", defaultMessage: "Акционная цена");
            bindingResult.addError(error);
        }
        return "guitar/add-guitar";
    }
    guitarRepository.save(guitar);

    return "redirect:/guitar/";
}
```

Рисунок 49 – Контроллер

Аннотация @Valid отвечает за работу метода с валидатором.

Добавление валидаторов на форму добавления нового элемента в

БД

```
<form class="card p-2" method="post" th:object="${guitar}">
    <div>
        <div th:if="${fields.hasErrors('name')}" th:errors="${name}"></div>
        <input type="text" th:field="${name}" id="name" placeholder="Название" class="form-control">
        <div th:if="${fields.hasErrors('type')}" th:errors="${type}"></div>
        <input type="text" th:field="${type}" id="type" placeholder="Тип" class="form-control">
        <div th:if="${fields.hasErrors('form')}" th:errors="${form}"></div>
        <input type="text" th:field="${form}" id="form" placeholder="Форма" class="form-control">
        <div th:if="${fields.hasErrors('cost')}" th:errors="${cost}"></div>
        <div th:if="${fields.hasErrors('vendorcode')}">
            <div th:each="error in ${fields.errors('vendorcode')}" th:text="${error}"></div>
        </div>
        <input type="number" th:field="${cost}" id="cost" placeholder="Цена" class="form-control">
        <div th:if="${fields.hasErrors('vendorcode')}" th:errors="${vendorcode}"></div>
        <input type="number" th:field="${vendorcode}" id="vendorcode" placeholder="Артикул" class="form-control">
        <br>
        <a href="/guitar/" class="btn btn-warning">Отмена</a>
        <button type="submit" class="btn btn-success">Добавить новый guitar</button>
    </div>
</form>
```

Рисунок 50 – Страница добавления

Результат:

Данное поле не должно быть пустым  
Данное поле должно иметь размер от 2 до 50 символов  
Название

Данное поле должно иметь размер от 2 до 50 символов  
Данное поле не должно быть пустым  
Тип

Данное поле не должно быть пустым  
Данное поле должно иметь размер от 2 до 50 символов  
Форма

Минимальная стоимость - 200  
0

минимальная длина артикула - 1111  
0

Отмена Добавить новую гитару

Рисунок 51 – работа валидаторов

Данное поле должно иметь размер от 2 до 50 символов  
Данное поле не должно быть пустым  
Название

Данное поле не должно быть пустым  
Данное поле должно иметь размер от 2 до 50 символов  
Тип

Данное поле должно иметь размер от 2 до 50 символов  
Данное поле не должно быть пустым  
Форма

Акционные цены  
123123

минимальная длина артикула - 1111  
0

Отмена Добавить новую гитару

Рисунок 52 – пользовательский валидатор

Вывод: В процессе выполнения данной практической работы я научился добавлять валидацию для добавления в таблицу той модели, которую создавали. Так же был сделан пользовательский валидатор.

## ПРАКТИЧЕСКАЯ РАБОТА №6

### Тема: Работа с связями

Цель работы: Необходимо реализовать 3 типа связей между таблицами (Один к одному, один ко многим, Многие ко многим), описать их и продемонстрировать их работу.

Создадим модели Один к одному

```
@Entity
@Table(name = "passports")
public class OneOnePassport {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String series;
    private String number;

    @OneToOne(optional = true, mappedBy = "passport")
    private OneOnePerson owner;

    public Long getId() { return id; }

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

    public String getSeries() { return series; }

    public void setSeries(String series) { this.series = series; }

    public String getNumber() { return number; }

    public void setNumber(String number) { this.number = number; }

    public OneOnePerson getOwner() { return owner; }

    public void setOwner(OneOnePerson owner) { this.owner = owner; }
}
```

Рисунок 53 – модель паспорта



```

import javax.persistence.*;

@Entity
@Table(name = "users")
public class OneOnePerson {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;

    @OneToOne(optional = true, cascade = CascadeType.ALL)
    @JoinColumn(name = "passport")
    private OneOnePassport passport;

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public OneOnePassport getPassport() { return passport; }
    public void setPassport(OneOnePassport passport) { this.passport = passport; }
    public OneOnePerson(String name, OneOnePassport passport) {
        this.name = name;
        this.passport = passport;
    }
}

public OneOnePerson() {
}
}

```

Рисунок 54 – модель пользователей

@OneToOne указывает на то что поле связано с другой таблицей связью один к одному

@JoinColumn аннотация указывает на таблицу с которой осуществляется связь

Далее создадим контроллер с 2 методами для загрузки страницы и для добавления пользователя соответственно, а также html страницу для отображения данных.

```
@Controller
public class OneOneController {

    @Autowired
    private OneOnePersonRepo personRepo;
    @Autowired
    private OneOnePassportRepo passportRepo;

    @GetMapping("/personOneOne")
    public String Main(Model model){
        Iterable<OneOnePassport> passport = passportRepo.findAll();
        model.addAttribute("passport", passport);
        return "personOneOne";
    }

    @PostMapping("/personOneOne/add")
    public String blogPostAdd(@RequestParam String name, @RequestParam String number, Model model){
        OneOnePassport passport = passportRepo.findById(number);

        OneOnePerson person = new OneOnePerson(name, passport);

        personRepo.save(person);
        return "redirect:/personOneOne";
    }
}
```

Рисунок 55 – контроллер один к одному

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Добавление нового человека</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <div th:insert="fragments/header :: header"></div>

    <form th:action="${'/personOneOne/add'}" method="post">
        <input type="text" name="name" placeholder="Введите имя человека">
        <select name="number">
            <div th:each="element : ${passport}">
                <option th:value="${element.id}" th:text="${element.number}"></option>
            </div>
        </select>
        <br>
        <button type="submit">Добавить человека</button>
    </form>
</body>
</html>
```

Рисунок 56 – HTML страница

Результат

Павел

345687 ▾

Добавить человека

Рисунок 57 – заполнение данных

Олег

234789 ▾

Добавить человека

Рисунок 58 – заполнение данных

id	name	passport
1	Павел	1
2	Олег	2

Рисунок 59 – данные заполнились

Теперь создадим связь один ко многим для этого создадим 2 модели  
адрес и пользователь

```
@Table(name = "address")
public class OneManyAddress {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String city;
    private String street;
    private String building;
    @OneToMany(mappedBy = "address", fetch = FetchType.EAGER)
    private Collection<OneManyPerson> tenants;
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getStreet() {
        return street;
    }
    public void setStreet(String street) {
        this.street = street;
    }
    public String getBuilding() {
        return building;
    }
    public void setBuilding(String building) {
        this.building = building;
    }

    public Collection<OneManyPerson> getTenants() {
        return tenants;
    }
    public void setTenants(Collection<OneManyPerson> tenants) {
        this.tenants = tenants;
    }
}
```

Рисунок 60 – модель адрес

```

package com.example.TypeOfConnection.models;

import javax.persistence.*;

@Entity
@Table(name = "manyUsers")
public class OneManyPerson {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;

    @ManyToOne(optional = true, cascade = CascadeType.ALL)
    private OneManyAddress address;

    public Long getId() { return id; }

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

    public String getName() { return name; }

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

    public OneManyAddress getAddress() { return address; }

    public void setAddress(OneManyAddress address) { this.address = address; }

    public OneManyPerson(String name, OneManyAddress address) {
        this.name = name;
        this.address = address;
    }

    public OneManyPerson() {
    }
}

```

Рисунок 61 - модель пользователь

@ManyToOne()

@OneToMany()

Эти две аннотации указывают как связаны таблицы где ManyToOne –многие к одному и OneToMany – один ко многим соответственно.

Создадим контроллер с методами заполнения выпадающего списка и добавлением

```
@Controller
public class OneManyController {

    @Autowired
    public OneManyAddressRepo addressRepo;

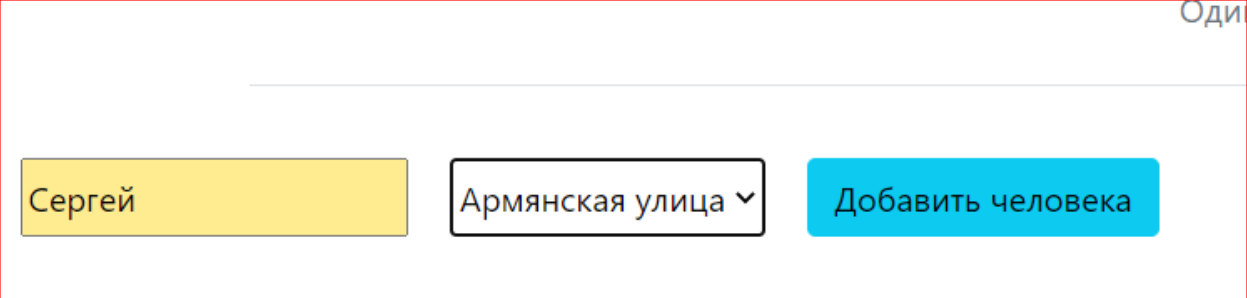
    @Autowired
    public OneManyPersonRepo personRepo;

    @GetMapping("/{personOneMany}")
    public String Main(Model model){
        Iterable<OneManyAddress> address = addressRepo.findAll();
        model.addAttribute("address", address);
        return "personOneMany";
    }

    @PostMapping("/{personOneMany}/add")
    public String blogPostAdd(@RequestParam String name, @RequestParam String street, Model model){
        OneManyAddress address = addressRepo.findByStreet(street);
        OneManyPerson person = new OneManyPerson(name, address);
        personRepo.save(person);
        return "redirect:/personOneMany";
    }
}
```

Рисунок 62 – контроллер

Результат



Один человек

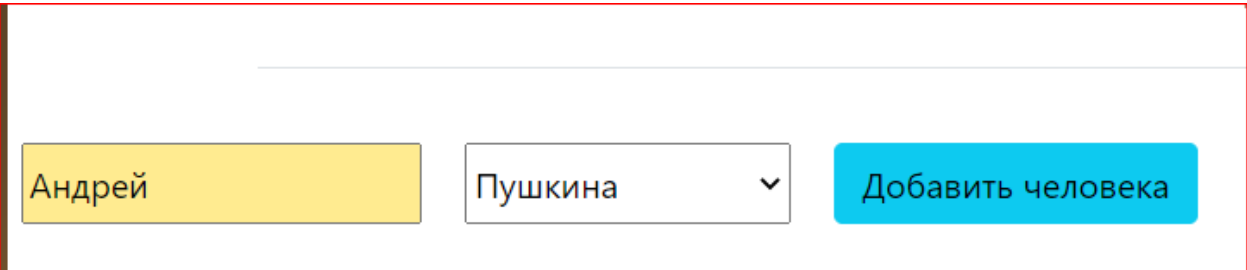
Добавить человека

Сергей

Армянская улица ▾

Добавить человека

Рисунок 63 – добавляем человека по адресу



Один человек

Добавить человека

Андрей

Пушкина ▾

Добавить человека

Рисунок 64 - добавляем человека по адресу

Все данные правильно внеслись в базу

id	name	address_id
4	Сергей	2
5	Андрей	1

Рисунок 65 — данные в базе

В завершение создадим последний вид связи

Многие ко многим для этого создадим снова 2 модели

```
package com.example.TypeOfConnection.models;

import ...

@Entity
@Table(name = "university")
public class ManyManyUniversity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column
    private String name;

    @ManyToMany
    @JoinTable(name = "student_university",
        joinColumns = @JoinColumn(name = "university_id"),
        inverseJoinColumns = @JoinColumn(name = "student_id"))
    private List<ManyManyStudent> students;

    public Long getId() { return id; }

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

    public String getName() { return name; }

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

    public List<ManyManyStudent> getStudents() { return students; }

    public void setStudents(List<ManyManyStudent> students) { this.students = students; }
}
```

Рисунок 66 — модель университета



```

package com.example.TypeOfConnection.models;

import ...

@Entity
@Table(name = "student")
public class ManyManyStudent {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;

    @ManyToMany
    @JoinTable(name = "student_university",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "university_id"))
    private List<ManyManyUniversity> universities;

    public Long getId() { return id; }

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

    public String getName() { return name; }

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

    public List<ManyManyUniversity> getUniversities() { return universities; }

    public void setUniversities(List<ManyManyUniversity> universities) { this.universities = universities; }
}

```

Рисунок 67 – модель студента

Анотация `@ManyToMany` в сочетании с `@JoinTable` создает промежуточную таблицу с именем `name`

`@JoinColumn` создает связь в промежуточной таблице



```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Добавление нового студента</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div th:insert="fragments/header :: header"></div>

<form th:action="${'/personManyMany/add'}" method="post">
  <select name="student">
    <div th:each="element : ${students}">
      <option th:text="${element.name}"></option>
    </div>
  </select>
  <select name="universiti">
    <div th:each="element : ${universities}">
      <option th:text="${element.name}"></option>
    </div>
  </select>
  <br>
  <button type="submit">Добавить человека</button>
</form>
</body>
</html>

```

Рисунок 68 – страница связи многие ко многим

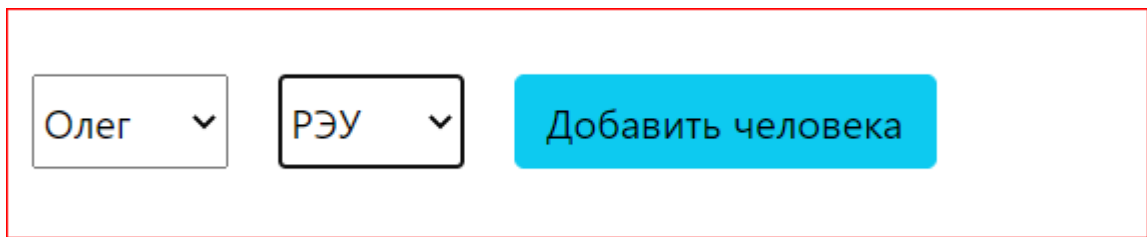
## Результат

The screenshot shows a web form with two dropdown menus and a button. The first dropdown menu has 'Олег' selected, and the second dropdown menu has 'МИФИ' selected. To the right of the dropdowns is a blue button with the text 'Добавить человека'.

Рисунок 69 – заполняем данные

The screenshot shows the same web form as Figure 69, but with different data. The first dropdown menu now has 'Наташа' selected, while the second dropdown menu still has 'МИФИ' selected. The blue button 'Добавить человека' remains the same.

Рисунок 70 - заполняем данные



Олег ▼ РЭУ ▼ Добавить человека

Рисунок 71 - заполняем данные

university_id	student_id
1	1
2	1
1	2

Рисунок 72 данные в базе данных

Вывод: В процессе выполнения работы я научился создавать таблицы со связями 3х типов, а также заполнять их данными пользовательского ввода, получил практические навыки работы с ними.

# ИНДИВИДУАЛЬНЫЙ ПРОЕКТ

Тема: Закрепление материала

Цели работы: Целью данного индивидуального проекта является разработка и реализация модели информационной системы управления интернет-магазином. Для решения данной задачи будут использованы возможности языка программирования Java, среда программирования IntelliJ IDEA 2021.2.3 и OpenServer версии 5.4.0

Описание предметной области: Выбрав необходимые товары пользователь обычно имеет возможность просмотреть и оформить покупку. Информация о покупателе храниться в базе данных магазина сайт специализируется на продаже товаров для животных, возможности: Просмотр информации о животных, просмотр товаров и добавление их в корзину, просмотр новостей, новости могут содержать ссылки на товары подходящей тематики (например, статья о расчесывании может иметь ссылки на расчёски и шампуни) на сайте есть возможность регистрации и авторизации, без сайт подразумевает разделение ролей на пользователей и администраторов администратор имеет возможность добавлять статьи, виды животных, статьи, новости и привязывать товары к статьям.

## Логическая модель базы данных:

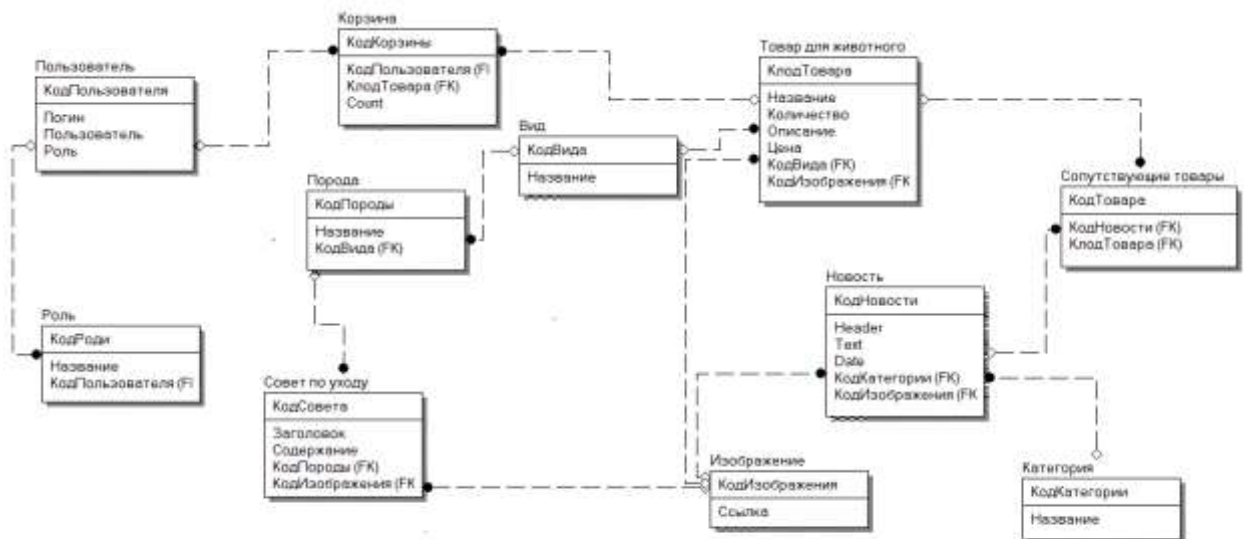


Рисунок 73 – логическая модель

## Физическая модель:

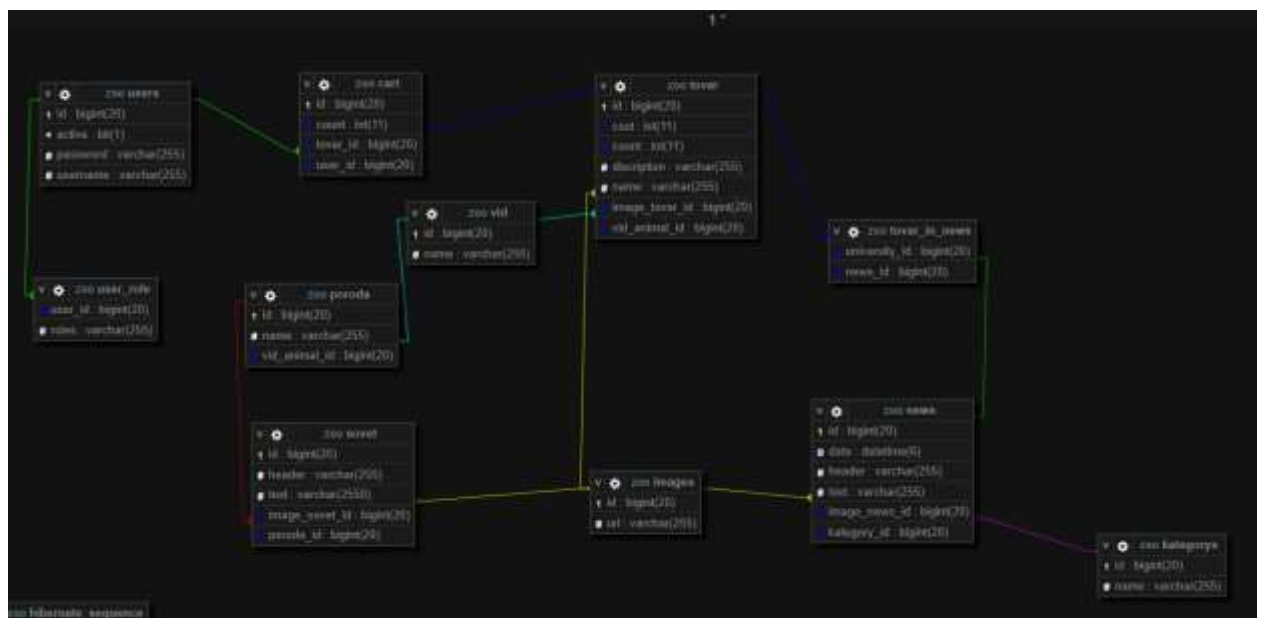


Рисунок 74 – Физическая модель

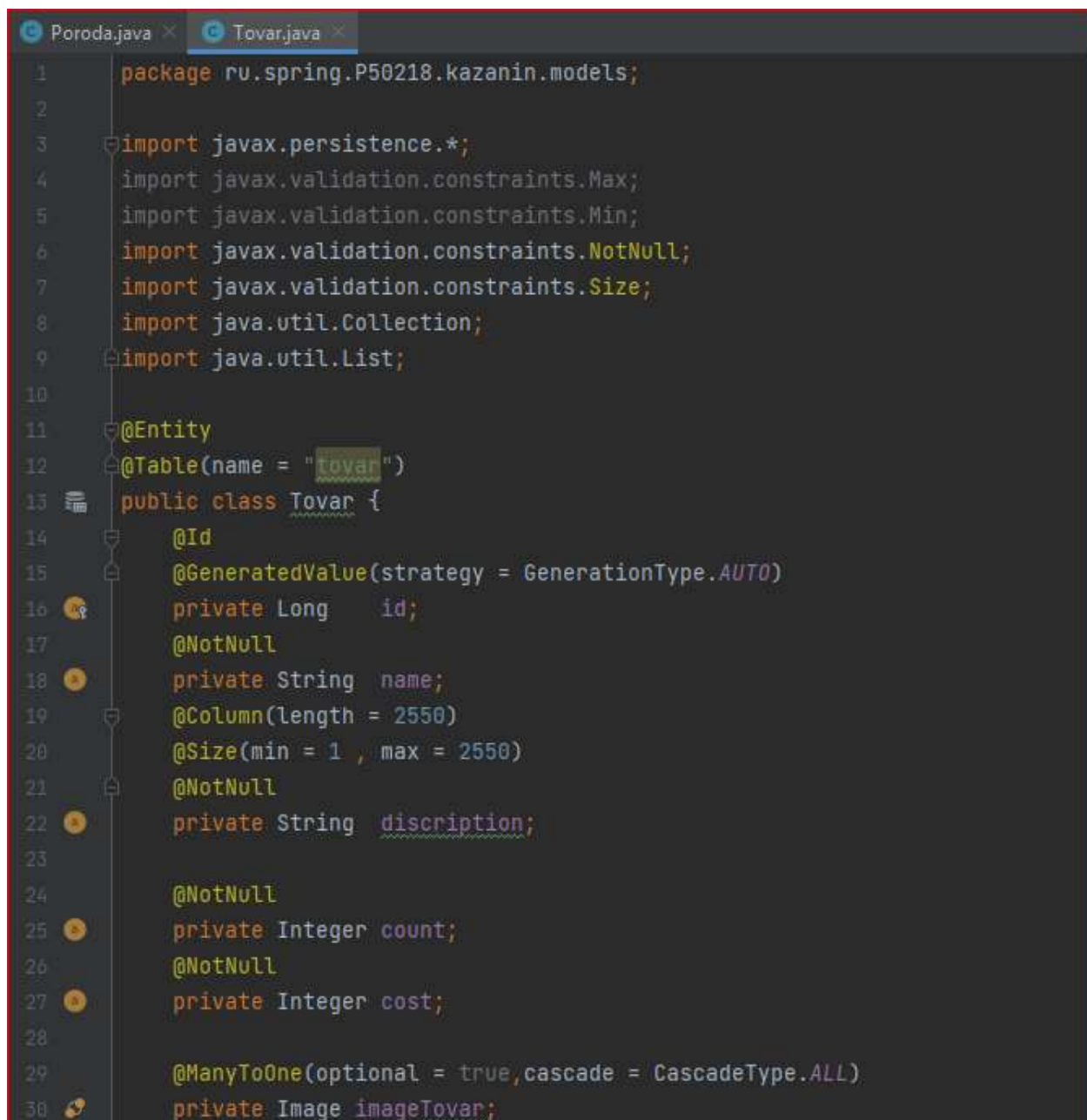
## Словарь данных

Таблица 1 – Словарь данных

№	Логическое название	Физическое название	Тип данных	Ограничения
1	2	3	4	5
user_role				
1	Код	id	Long	PK, NOT NULL
2	Название	name	Varchar(255)	NOT NULL
users				
1	Код	id	Long	PK, NOT NULL
2	Логин	Login	Varchar(255)	NOT NULL
3	Пароль	Password	Varchar(255)	NOT NULL
images				
1	Код	id	Long	PK, NOT NULL
2	Ссылка	Url	Varchar(2550)	NOT NULL
vid				
1	Код	id	Long	PK, NOT NULL
2	Название	name	string	NOT NULL
poroda				
1	Код	id	Long	PK, NOT NULL
2	Название	name	Varchar(255)	NOT NULL
3	КодВида	vid_animal_id	Long	FK NOT NULL
sovet				
1	Код	id	Long	PK, NOT NULL
2	Заголовок	header	Varchar(255)	NOT NULL
3	Содержание	text	Varchar(2550)	NOT NULL
4	КодИзображения	image_sovet_id	Long	FK NOT NULL
5	КодПороды	Poroda_id	Long	FK NOT NULL
tovar				
1	Код	id	Long	PK, NOT NULL
2	Цена	cost	int	NOT NULL
3	Количество	count	int	NOT NULL
4	Описание	description	Varchar(2550)	NOT NULL
5	Название	Name	Varchar(255)	NOT NULL
6	КодИзображения	image_tovar_id	Long	FK NOT NULL
7	КодВида	image_vid_id	Long	FK NOT NULL
Kategoryys				
1	Код	Id	Long	PK, NOT NULL
2	Название	name	Varchar(255)	NOT NULL
news				
1	Код	id	Long	PK, NOT NULL
2	Заголовок	header	Varchar(255)	NOT NULL
3	Содержание	text	Varchar(2550)	NOT NULL
4	КодИзображения	image_sovet_id	Long	FK NOT NULL
5	КодКатегории	kategory_id	Long	FK NOT NULL
6	Дата	Date	Date	NOT NULL

№	Логическое название	Физическое название	Тип данных	Ограничения
1	2	3	4	5
tovar_in_news				
1	КодТовара	tovar_id	Long	FK NOT NULL
2	КодНовости	news_id	Long	FK NOT NULL
chart				
1	Код	id	Long	PK, NOT NULL
2	Количество	count	int	NOT NULL
3	КодТовара	tovar_id	Long	FK NOT NULL
4	КодПользователя	user_id	Long	FK NOT NULL

Реализация программы:



```
1 package ru.spring.P50218.kazanin.models;
2
3 import javax.persistence.*;
4 import javax.validation.constraints.Max;
5 import javax.validation.constraints.Min;
6 import javax.validation.constraints.NotNull;
7 import javax.validation.constraints.Size;
8 import java.util.Collection;
9 import java.util.List;
10
11 @Entity
12 @Table(name = "товар")
13 public class Tovar {
14     @Id
15     @GeneratedValue(strategy = GenerationType.AUTO)
16     private Long id;
17     @NotNull
18     private String name;
19     @Column(length = 2550)
20     @Size(min = 1, max = 2550)
21     @NotNull
22     private String discription;
23
24     @NotNull
25     private Integer count;
26     @NotNull
27     private Integer cost;
28
29     @ManyToOne(optional = true, cascade = CascadeType.ALL)
30     private Image imageTovar;
```

Рисунок 75 – модель товара с ограничениями

## Html страницы для Регистрации и авторизации:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"
      xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
<head>
  <title>Spring Security Example </title>
  <link rel="stylesheet" href="https://bootstrap5.ru/css/docs.css">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha1"
</head>
<body class="text-center">
<div th:if="${param.error}">
  Не верный логин или пароль
</div>
<div align="center" th:if="${param.logout}">
  You have been logged out.
</div>
<form align="center" class="nav-item" th:action="@{/login}" method="post">
  <div align="center"><label> User Name : <input type="text" name="username"/> </label></div>
  <div align="center"><label> Password: <input type="password" name="password"/> </label></div>
  <div align="center"> <button type="submit" class="btn btn-primary">Войти</button></div>
</form>
<a href="/registration">Зарегистрироваться</a>
</body>
</html>
```

Рисунок 76 – Авторизация

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"
      xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
<head>
  <meta charset="UTF-8">
  <title>Register</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
  <form th:align="center" method="post">
    <label align="center">Логин : <input type="text" name="username"/></label><br>
    <label align="center">Пароль : <input type="password" name="password"/></label><br>
    <button align="center" type="submit">Зарегистрироваться</button>
  </form>
</body>
</html>
```

Рисунок 77 – Регистрация



```

package ru.spring.P50218.kazanin.config;

import ...

@Configuration
public class MvcConfig implements WebMvcConfigurer {

    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController( urlPathOrPattern: "/login").setViewName("login");
    }

}

```

Рисунок 78 – Mvc Контроллер

Так же создадим класс WebSecurityConfig для авторизации и регистрации получим интерфейс Password Encoder для шифрования пароля.

```

@Autowired
private DataSource dataSource;

@Bean
public PasswordEncoder getPasswordEncoder() { return new BCryptPasswordEncoder( strength: 8); }

@Autowired
private PasswordEncoder passwordEncoder;

```

Рисунок 79 – Часть класса WebSecurityConfig

Так же создадим 2 переопределяемых метода для регистрации и авторизации в первом методе укажем страницы доступные для посещения без авторизации и ссылку на страницу регистрации. Во втором методе для авторизации передадим объект базы данных и шифратор пароля, после выполним SELECT запросы для получения пользователя и роли.

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests() .anyRequest().authenticated().and().authorizeRequests()
        .antMatchers( .antMatchers( "/", "/registration").permitAll()
        .anyRequest().authenticated()
        .and() .httpSecurity()
        .formLogin() .loginPage("/login")
        .permitAll()
        .and() .httpSecurity()
        .logout() .logoutUrl("/logout")
        .permitAll()
        .and() .httpSecurity()
        .csrf().disable();
}

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception{
    auth.jdbcAuthentication()
        .dataSource(dataSource)
        .passwordEncoder(passwordEncoder)
        .usersByUsernameQuery("select username,password,active from users where username=?")
        .authoritiesByUsernameQuery("select u.username, ur.role from users u inner join user_role ur on u.id = ur.user_id where u.username=?");
}

```

Рисунок 80 - Часть класса WebSecurityConfig

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-+8n0xVWZ3E5R50omZ0m"
<title th:text="${tovar.name}"></title>
<div th:insert="fragments/bg"></div>
</head>
<body>
<div th:insert="fragments/header :: header"></div>
<div style="display: flex; justify-content: space-around; align-items: center; padding: 10px;">
<div style="text-align: center; width: 40%;">

<div style="margin-top: 10px; display: flex; justify-content: space-between; width: 100%;">
<div th:text="${tovar.name}"></div>
<div th:text="${tovar.count}"></div>
</div>
</div>
<div style="width: 50%; text-align: left;">
<h2 th:text="${tovar.name}"></h2>
<p class="lead" th:text="${tovar.description}"></p>
<a class="btn btn-lg btn-primary" th:href="${tovari}" role="button">Назад</a>
<a class="btn btn-lg btn-primary" style="margin-left: 10px;" th:href="${tovari}/${tovar.id}/addToChart" role="button">Добавить в корзину</a>
</div>
</div>
<div th:insert="fragments/footer"></div>
</body>
</html>

```

Рисунок 81 – страница просмотра товара

```

@GetMapping("/{id}/view/{id}")
public String openview(
    @PathVariable(name = "id") Long id_tovara,
    Model model) {
    Tovar tovar = tovarRepository.findFirstById(id_tovara);
    model.addAttribute("tovar", tovar);
    return "tovari/tovar-view";
}

@GetMapping("/{id}/addToChart")
public String addToChart(
    @PathVariable(name = "id") Long id_tovara,
    Model model) {
    Tovar tovar = tovarRepository.findFirstById(id_tovara);
    if (tovar.getCount() > 0) {
        User user = userRepository.findByUsername(getUsername());
        Chart chart = new Chart();
        chart.setUser(user);
        chart.setTovar(tovar);
        chart.setCount(1);
        chartRepository.save(chart);
    }
    Iterable<Tovar> alltovari = tovarRepository.findAll();
    Iterable<VidAnimal> allvids = vidAnimalRepository.findAll();
    User user = userRepository.findByUsername(getUsername());
    for (var i : user.getRoles()) {
        if (i.equals(Role.ADMIN)) model.addAttribute("isAdmin", true);
    }
    model.addAttribute("alltovari", alltovari);
    model.addAttribute("allvids", allvids);
    return "tovari/tovari-list-view";
}

```

Рисунок 82 – часть контроллера товаров

Работа программы:

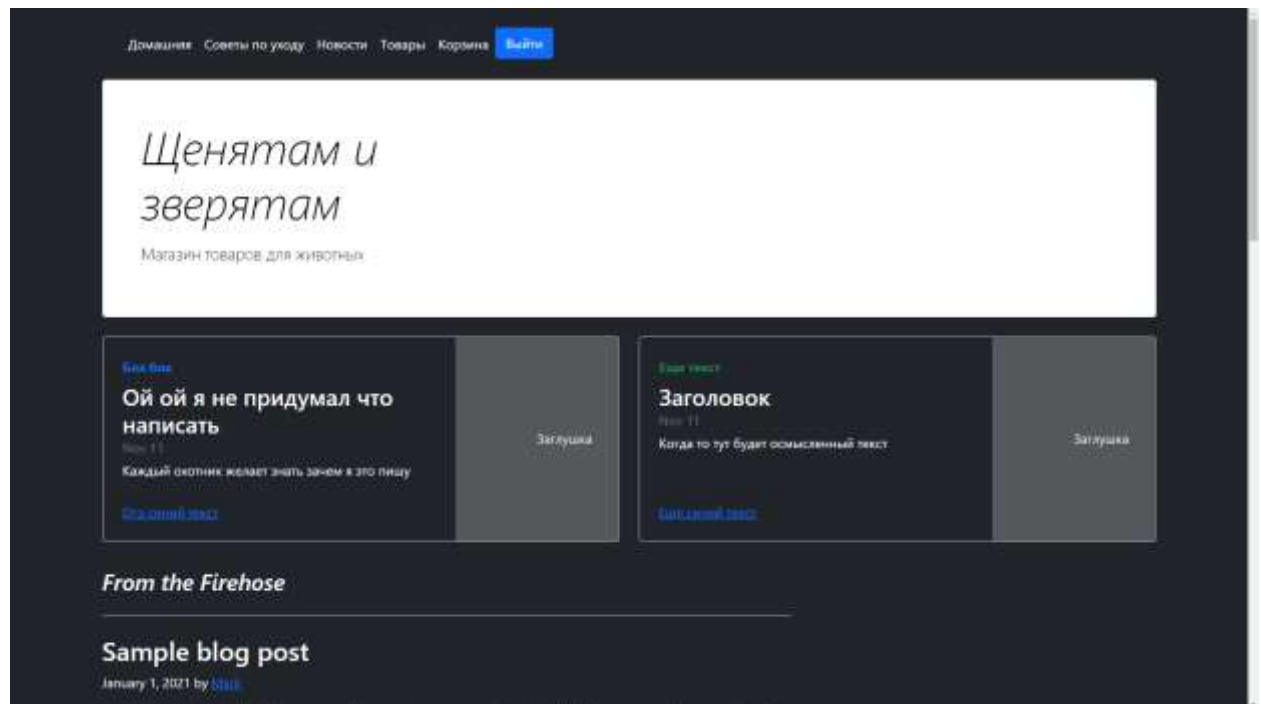


Рисунок 83 – Главная страница

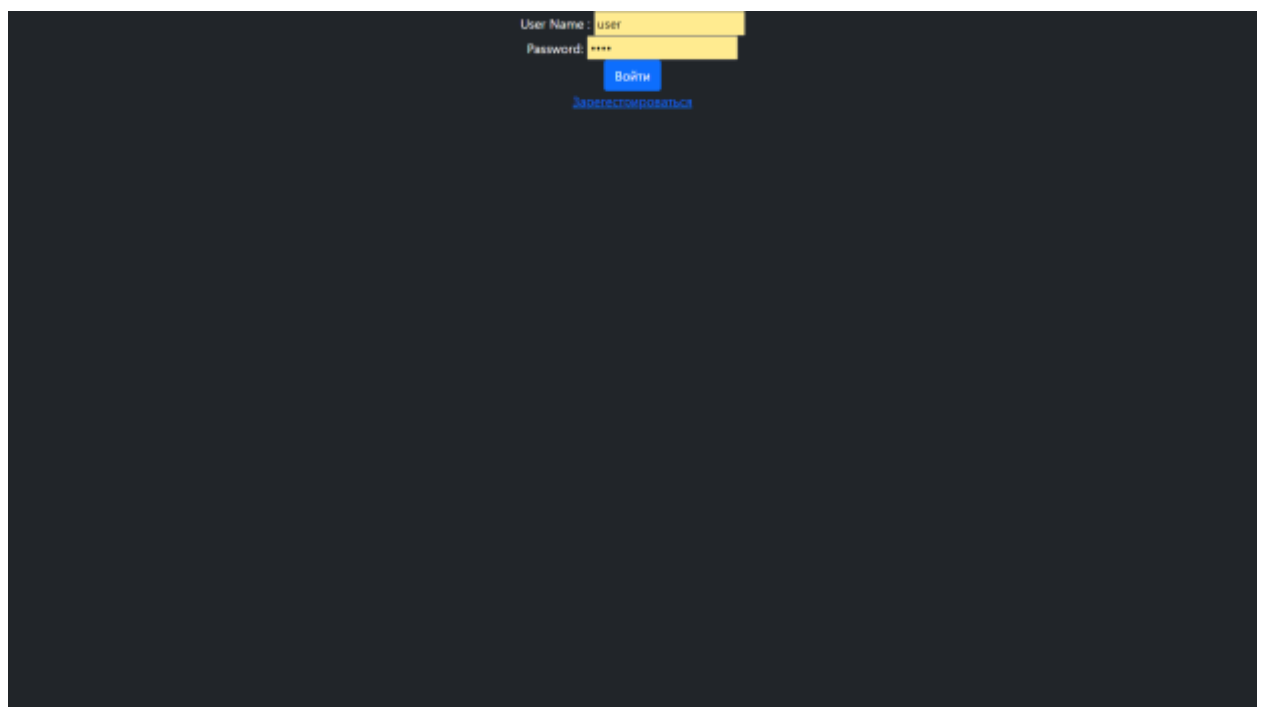


Рисунок 84 – Авторизация

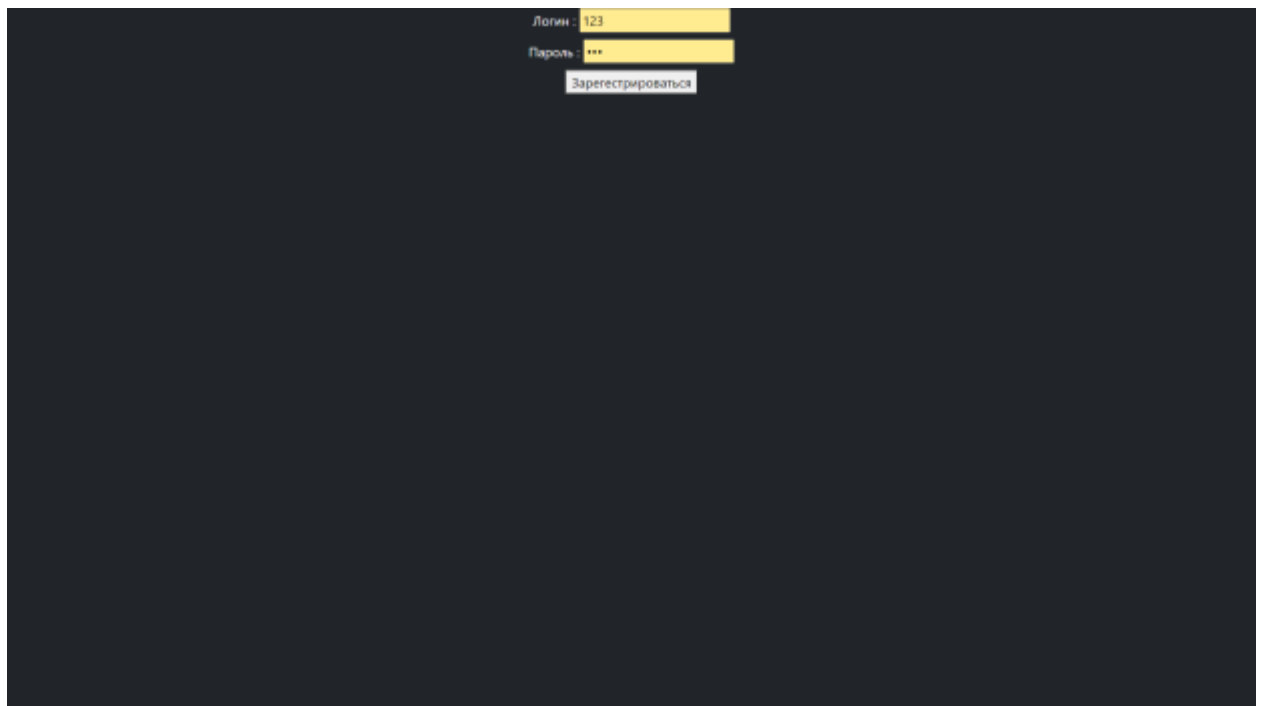


Рисунок 85 – Регистрация

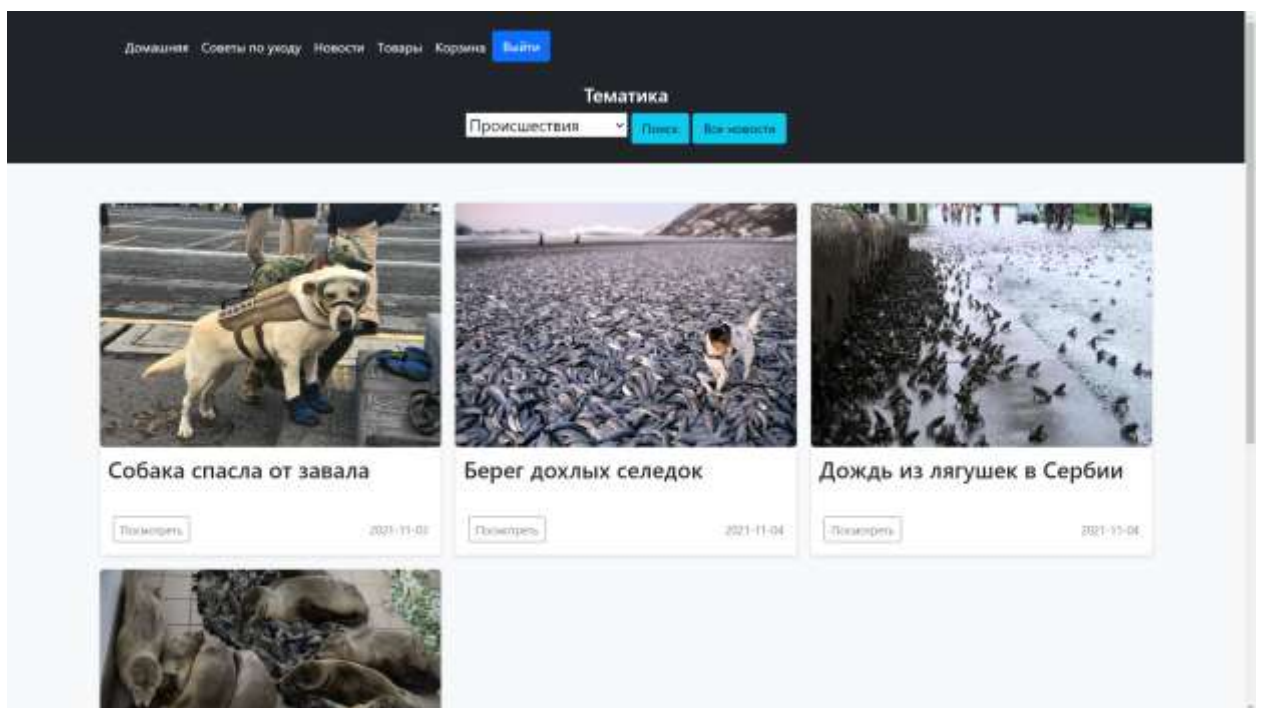


Рисунок 86 – авторизация пользователя

В случае если авторизован админ у нас появятся дополнительные ссылки

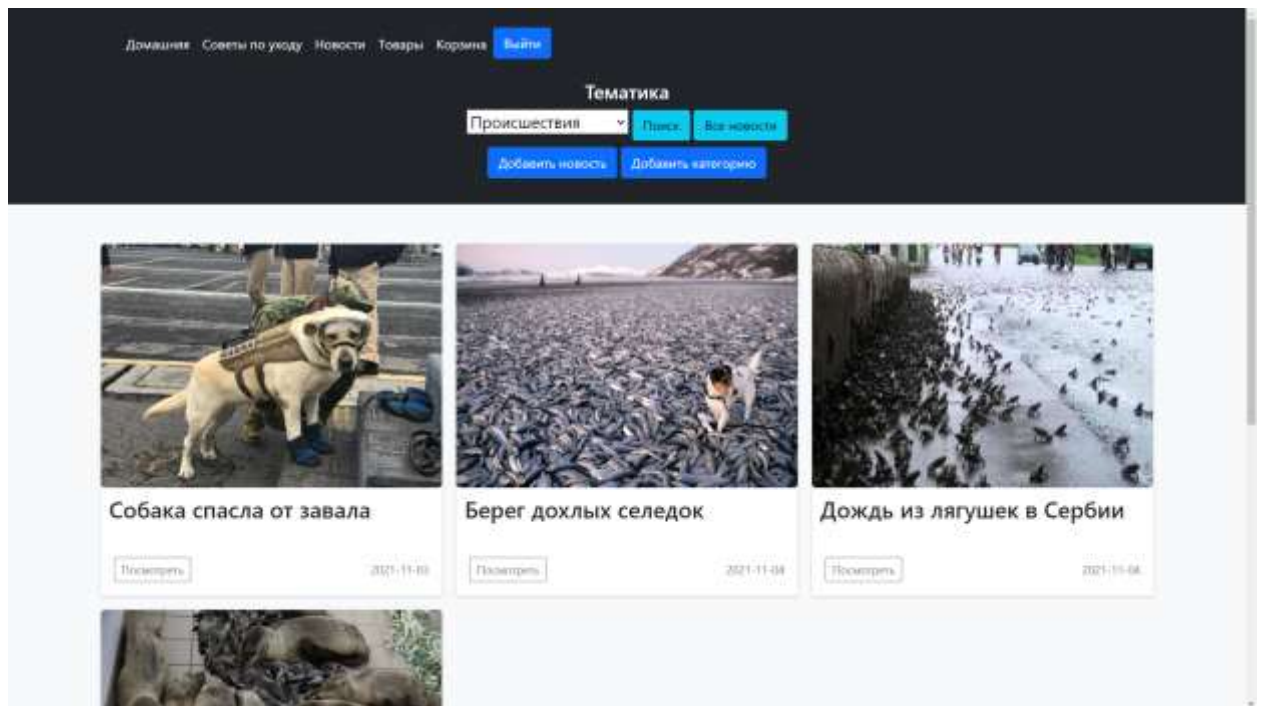


Рисунок 87 – Появились кнопки добавления новостей

Добавим новость о котятках, для этого создадим категорию котят

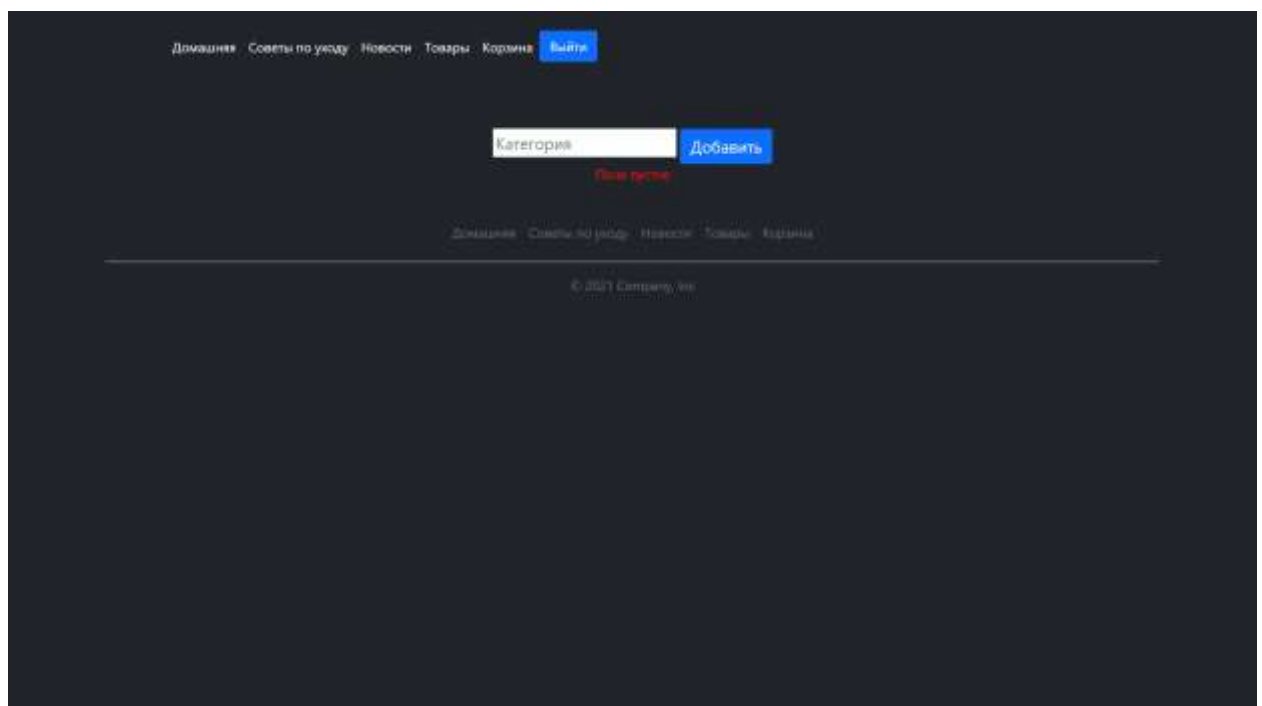


Рисунок 88 – Поле не может быть пустым

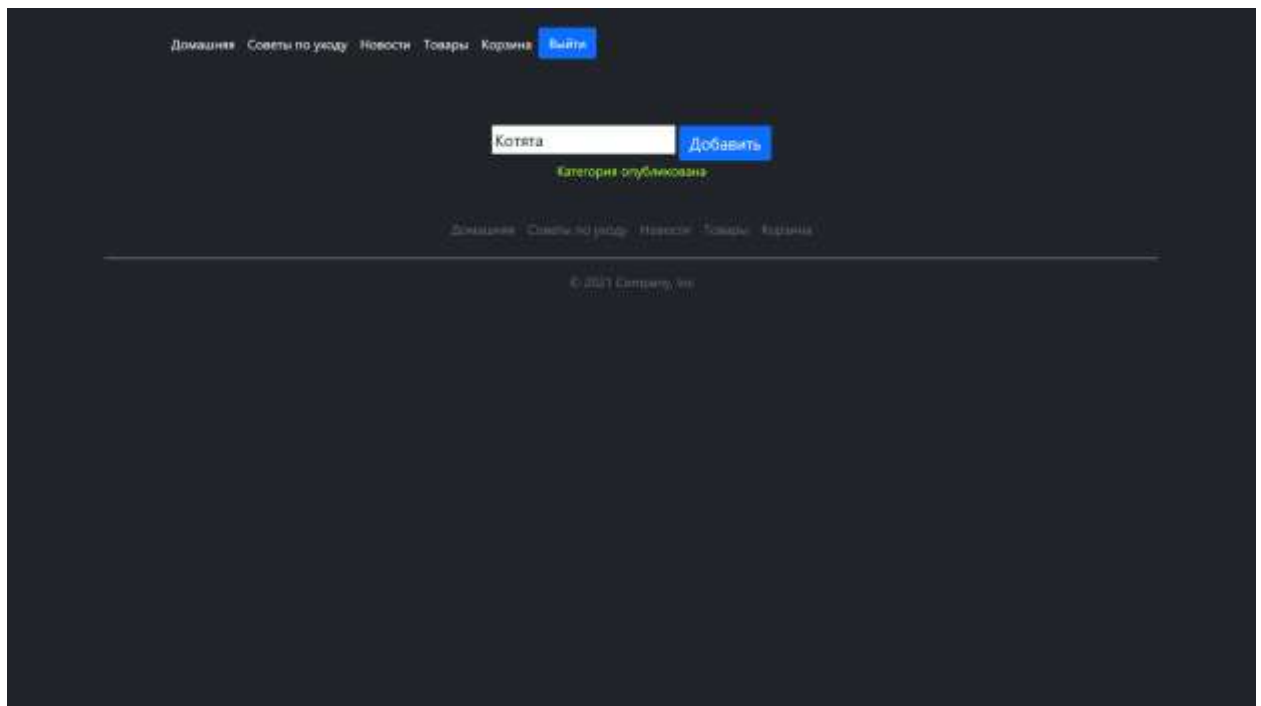


Рисунок 89 – Категория добавлена

Далее перейдем в окно добавления новости и добавим новость и котят, после нажатия на кнопку опубликовать страница обновится и появится оповещение о статусе публикации.

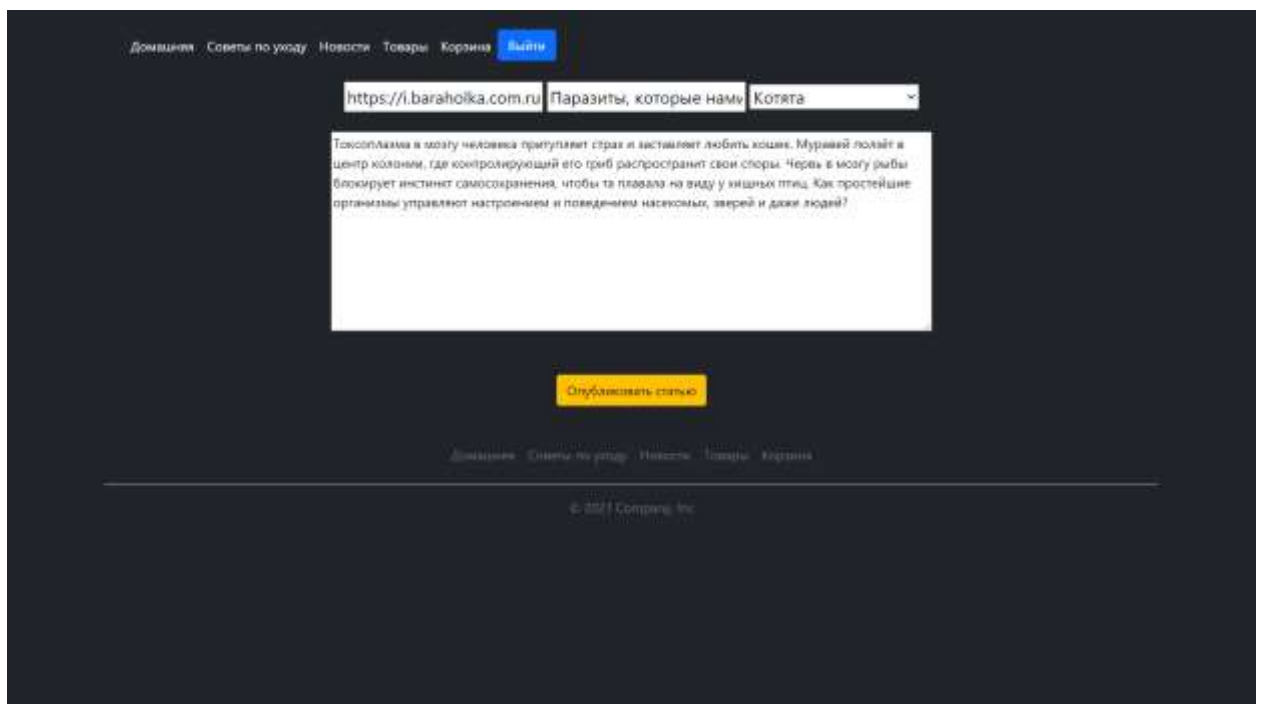


Рисунок 90 – Ввод данных

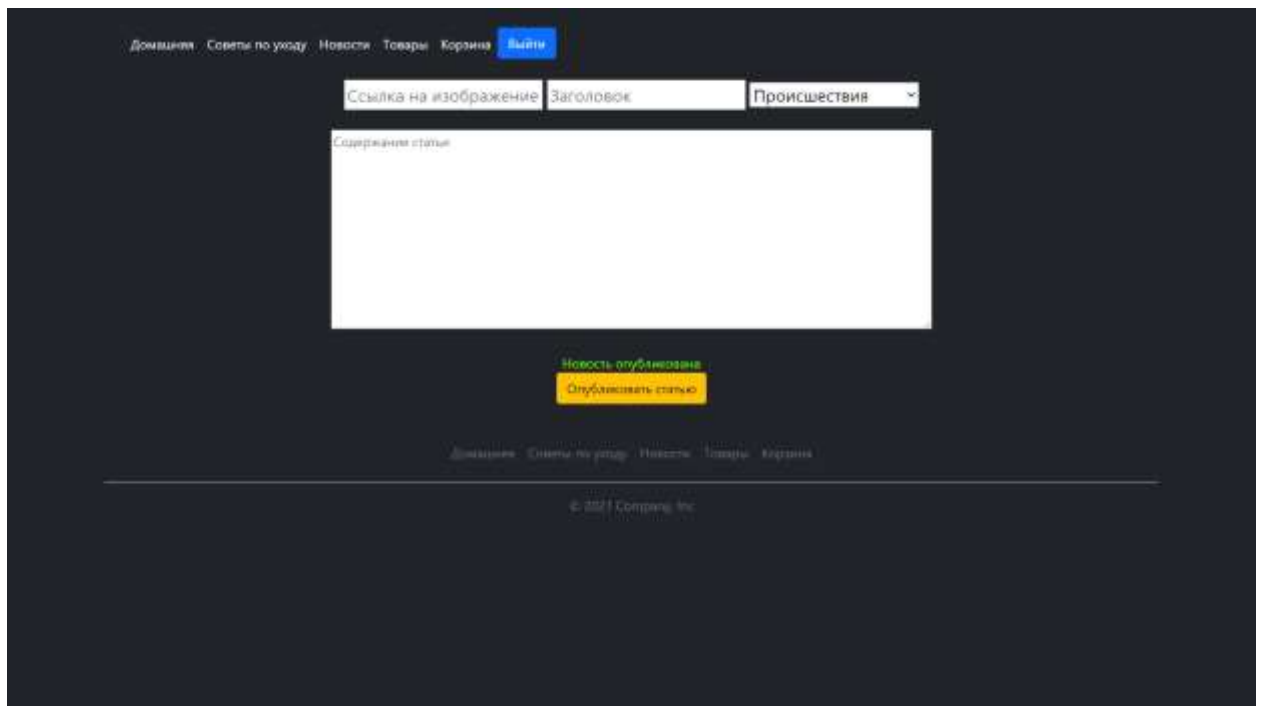
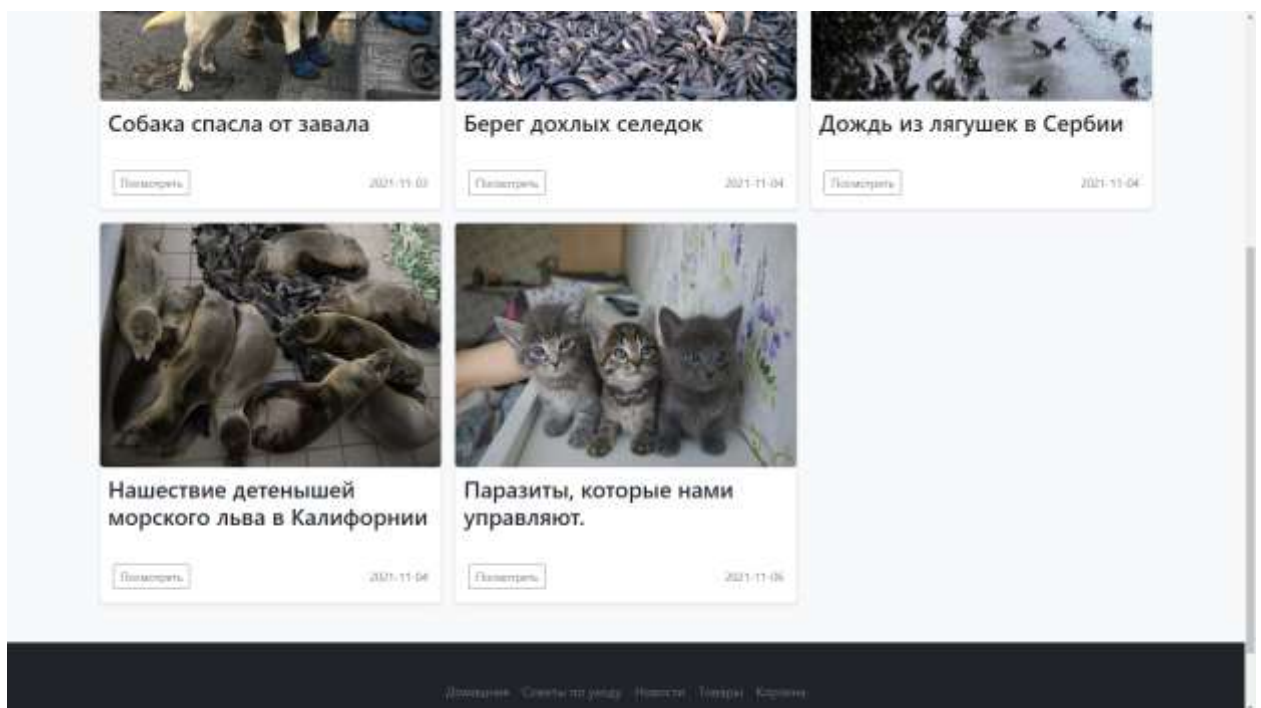


Рисунок 91 – Новость опубликована

Теперь она видна в списке новостей



При нажатии кнопки просмотреть откроется страница подробного просмотра, в случае авторизации за админа будет видна ссылка для перехода на страницу добавления



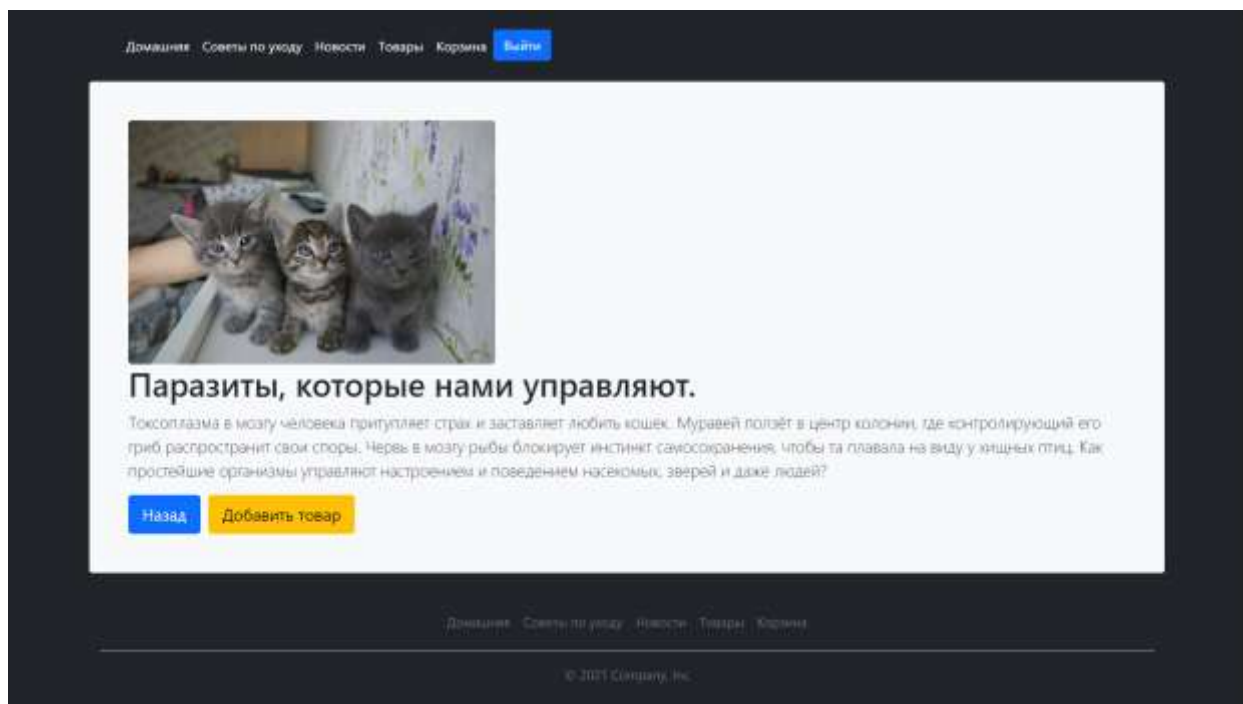


Рисунок 92 – страница просмотра

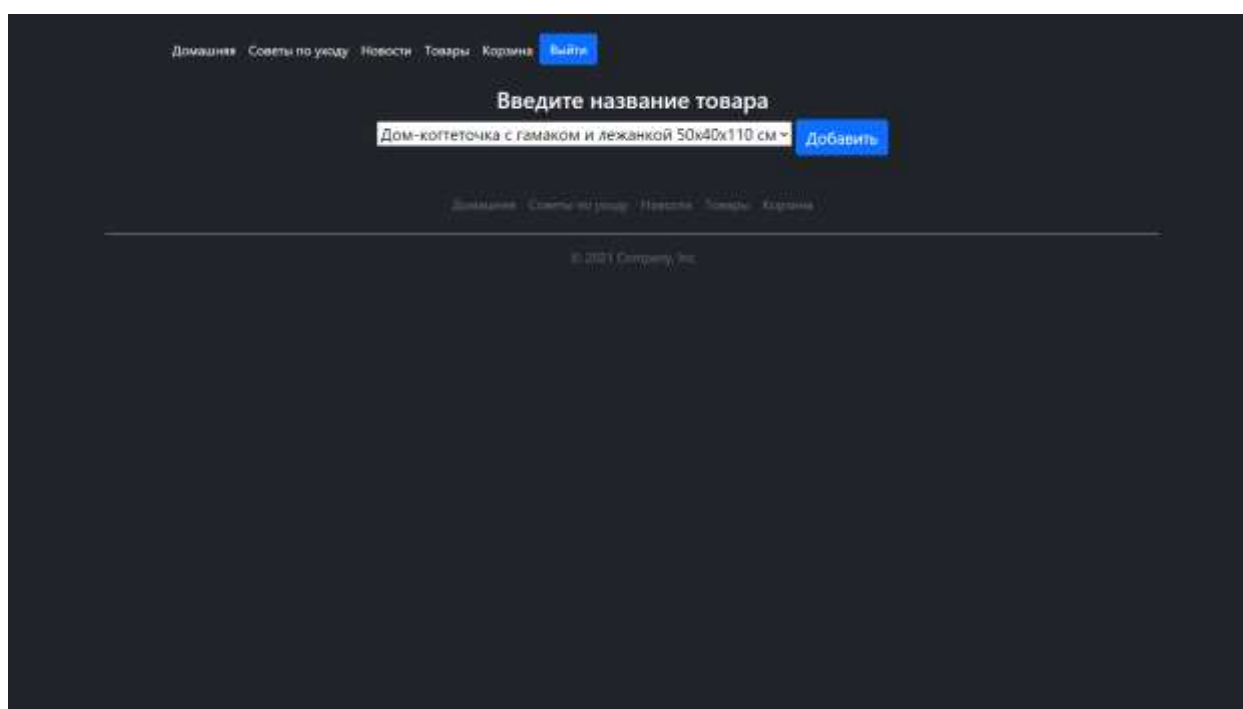


Рисунок 93 – Добавление товара к новости



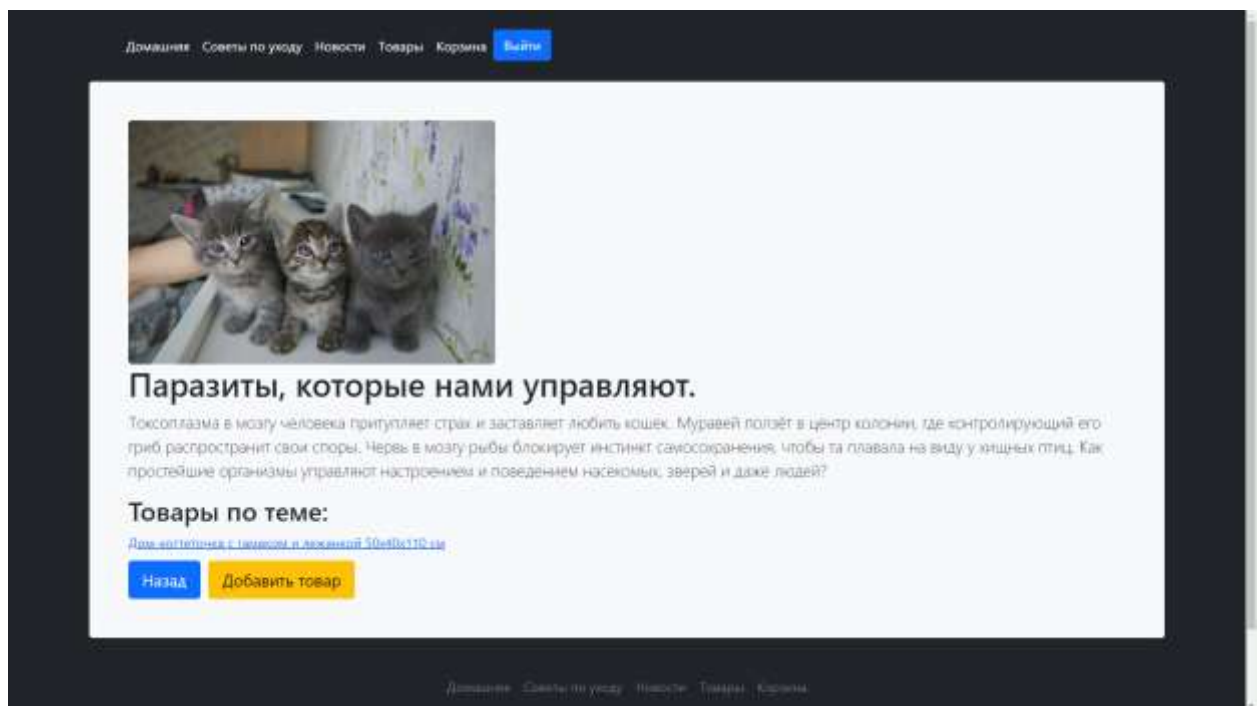


Рисунок 94 – после добавления появится ссылка на товар

Перейдем по ней

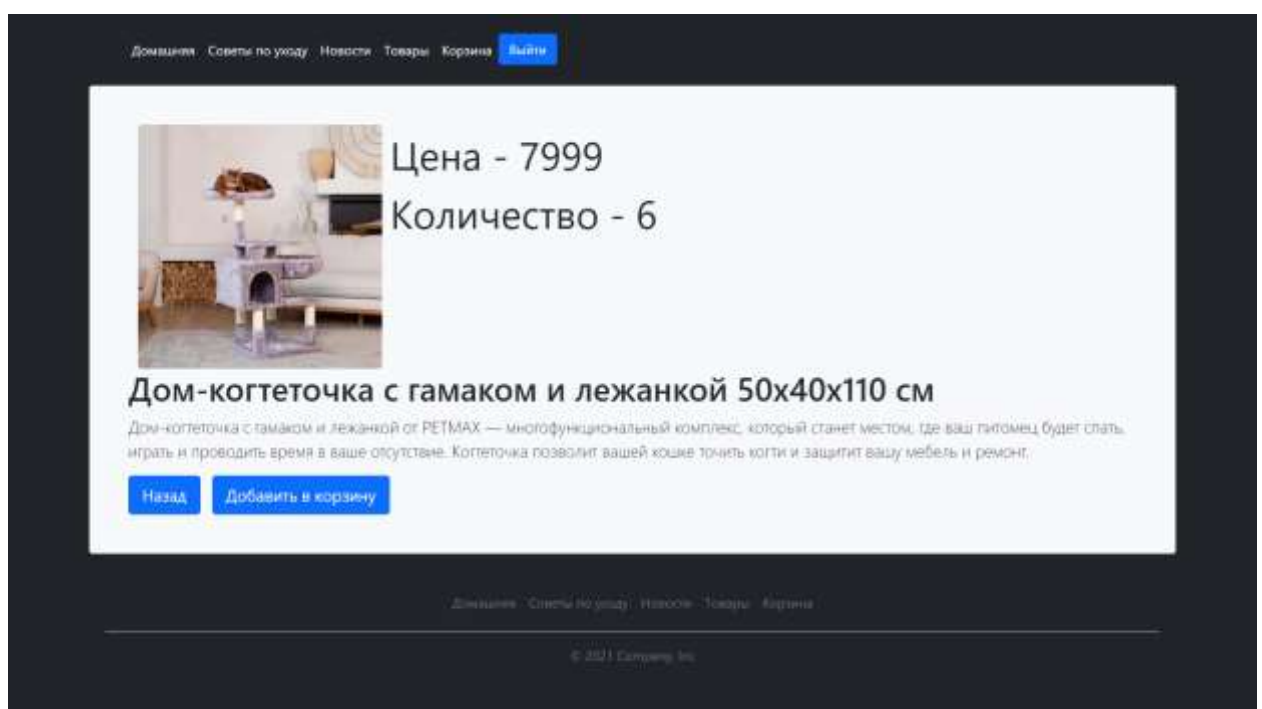


Рисунок 95 – Просмотр товара

Нажмем на кнопку добавить в корзину и перейдем в нее по верхнему навигационному меню.

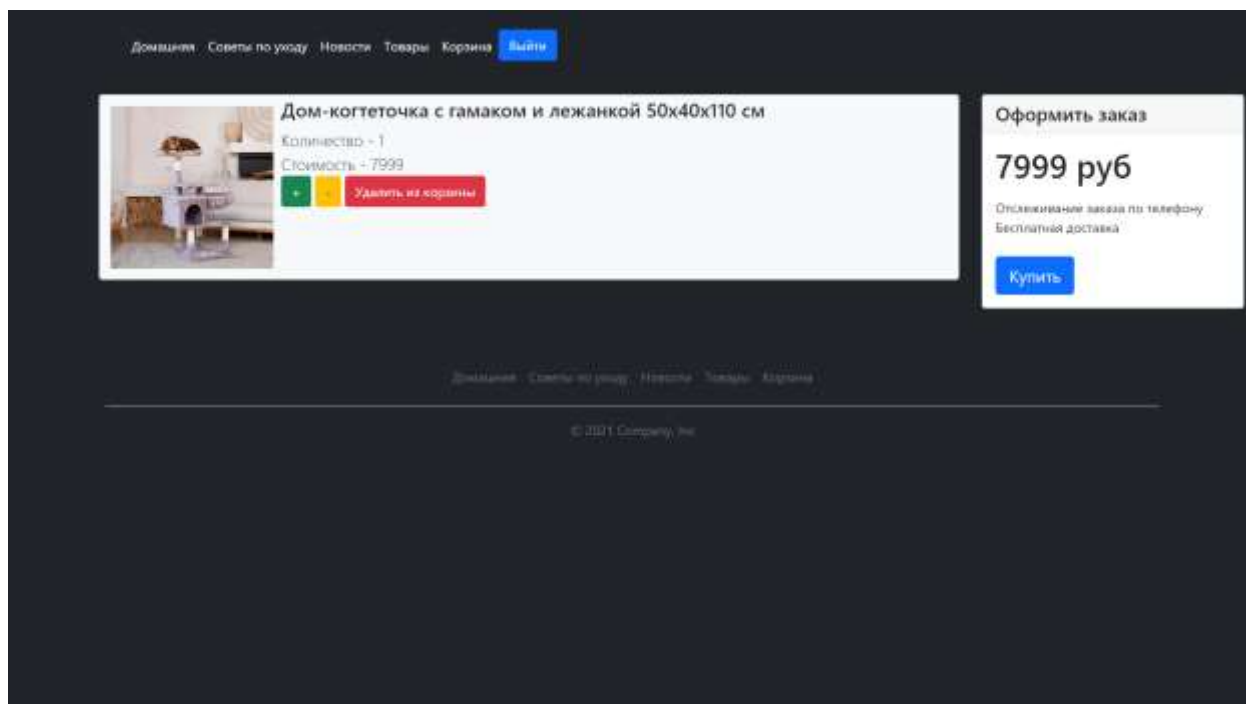


Рисунок 96 – корзина

Вывод: В ходе написания личного проекта я изучил структуру программ для составления баз данных, научился работать с языком Java и освоил на базовом уровне Spring framework, научился подключать базу данных, выводить и вводить в нее данные с сайта на Spring, получил более ясное представление о паттерне программирования MVC, реализовал регистрацию и авторизацию пользователей, а также разграничение функционала в зависимости от прав пользователя, предусмотрел защиту ввода данных пользователем, так же овладел навыком пользования связями в таблицах базы данных 3х типов, значительно повысил свои навыки и умения владения языком разметки html и языком разметки css.