



Geekbrains

# **Разработка веб-приложения для фиксации состава и длительности тренировок**

Специализация

Разработчик. Веб-разработка на Java

Соловьева Виктория Александровна

Москва  
2024

## Оглавление

Введение.....	3
Глава 1. Основы разработки веб-приложений .....	4
1.1 Основные компоненты веб-приложения .....	4
1.2 Архитектура веб-приложений.....	8
1.3 Основные компоненты Spring MVC.....	12
Глава 2. Разработка веб-приложения фиксации состава и длительности тренировок .....	14
2.1 Порядок работы над приложением.....	14
2.2 Проектирование архитектуры веб-приложения.....	16
2.3 Разработка веб-приложения фиксации состава и длительности тренировок .....	17
Заключение .....	27
Список литературы .....	28

## **Введение**

Люди, занимающиеся спортом, нередко сталкиваются с потерей мотивации продолжать регулярные тренировки. Одним из способов, который может помочь человеку, желающему сделать спорт частью своей жизни, может стать фиксация даты, длительности, состава тренировок. Психологи рекомендуют делить поставленную задачу на части и отмечать выполнение каждой «маленькой» подзадачи. В таком случае, завершение каждой подзадачи для человека вызывает радость и может оказывать положительное влияние на желание продолжать работу. Наиболее актуально ведение таких записей для людей, занимающихся спортом без спортивного тренера. Таким образом, фиксация в приложении выполнения отдельных элементов тренировки, может помочь пользователю сохранить желание заниматься спортом регулярно. Кроме того, записи о прошедших тренировках, их длительности, составе и дате позволяют отслеживать динамику спортивной жизни пользователя.

**Цель проектной работы** – разработать веб-приложение, позволяющее фиксировать выполненные элементы во время спортивных тренировок, вести записи о прошедших тренировках, их составе и длительности.

### **Задачи проектной работы:**

1. Рассмотреть основные компоненты веб-приложения.
2. Рассмотреть существующие виды архитектуры веб-приложений.
3. Рассмотреть особенности компонента фреймворка Spring – Spring MVC (Model-View-Controller).
4. Определение порядка разработки приложения.
5. Проектирование архитектуры веб-приложения.
6. Разработка веб-приложения фиксации состава и длительности тренировок.

# **Глава 1. Основы разработки веб-приложений**

## **1.1 Основные компоненты веб-приложения**

Разработка веб-приложений — это процесс создания программного обеспечения, которое работает на веб-серверах и доступно через веб-браузеры. Он включает несколько ключевых аспектов и технологий. Ниже представлены основные концепции и шаги, которые нужно учитывать при разработке веб-приложений.

### **Основные компоненты веб-приложения:**

#### **1. Клиентская сторона (Frontend):**

HTML: Язык разметки для создания структуры веб-страниц.

CSS: Язык стилей для оформления и размещения элементов на веб-странице.

JavaScript: Скриптовый язык для добавления интерактивности и динамики на веб-странице.

Фреймворки и библиотеки: React, Angular, Vue.js и другие для упрощения разработки сложных пользовательских интерфейсов.

#### **2. Серверная сторона (Backend):**

Языки программирования: Java, Python, PHP, Node.js и другие для разработки серверной логики.

Фреймворки: Spring, Django, Laravel, Express.js и другие для ускорения разработки серверной части.

Серверы: Apache, Nginx, Node.js и другие для обработки запросов клиентов и отправки ответов.

#### **3. База данных:**

СУБД: MySQL, PostgreSQL, MongoDB, SQLite и другие для хранения данных.

ORM (Object-Relational Mapping): Hibernate, Sequelize и другие для упрощения работы с базой данных.

#### **4. API (Application Programming Interface):**

REST: Стандарт для построения веб-сервисов.

GraphQL: Альтернатива REST для гибкого запроса данных.

## **Шаги разработки веб-приложения**

### **1. Определение требований**

Сбор требований: Понимание потребностей и ожиданий пользователей.

Документирование: Создание спецификаций и пользовательских историй.

### **2. Проектирование**

Архитектура: Определение структуры приложения (монолит, микросервисы и т.д.).

Проектирование базы данных: Создание схемы базы данных и определение сущностей.

Прототипы и макеты: Создание интерфейсов пользователя для визуализации приложения.

### **3. Настройка среды разработки**

Инструменты: Выбор и настройка IDE, систем контроля версий (Git), систем сборки и управления зависимостями.

Серверы и базы данных: Настройка локальных или удаленных серверов и баз данных для разработки и тестирования.

### **4. Реализация**

Фронтенд: Разработка интерфейсов пользователя с использованием HTML, CSS и JavaScript.

Бэкенд: Реализация серверной логики, API и интеграции с базой данных.

Интеграция: Соединение фронтенда и бэкенда, настройка маршрутизации и обработки запросов.

### **5. Тестирование**

Модульное тестирование: Проверка отдельных компонентов на корректность работы.

Интеграционное тестирование: Проверка взаимодействия между компонентами.

Системное тестирование: Полное тестирование приложения на соответствие требованиям.

Приемочное тестирование: Проверка приложения конечными пользователями.

## **6. Развертывание**

Настройка продакшн-среды: Подготовка серверов, базы данных и сетевой инфраструктуры.

CI/CD: Настройка непрерывной интеграции и доставки для автоматизации развертывания.

Мониторинг и логирование: Внедрение систем мониторинга и логирования для отслеживания работы приложения.

## **7. Поддержка и обслуживание**

Мониторинг: Постоянное отслеживание состояния приложения и производительности.

Обновления и исправления: Регулярное обновление приложения, исправление багов и внедрение новых функций.

Анализ обратной связи: Сбор и анализ отзывов пользователей для улучшения приложения.

## **Основные инструменты и технологии**

Клиентская сторона

HTML/CSS: Основы верстки и стилей.

JavaScript: Базовые и продвинутые концепции языка.

Фреймворки и библиотеки: React, Angular, Vue.js для построения сложных интерфейсов.

Инструменты сборки: Webpack, Babel, Gulp для оптимизации и сборки проектов.

Серверная сторона

Языки программирования: Java, Python, PHP, Node.js и другие.

Фреймворки: Spring (Java), Django (Python), Laravel (PHP), Express.js (Node.js) и другие.

API: Разработка RESTful сервисов, использование GraphQL.

Базы данных

Реляционные: MySQL, PostgreSQL, SQLite.

NoSQL: MongoDB, Cassandra.

Инструменты ORM: Hibernate, Sequelize.

## **1.2 Архитектура веб-приложений**

Архитектура веб-приложений может варьироваться в зависимости от масштабов, целей и требований проекта. Вот несколько наиболее распространенных архитектур веб-приложений:

### **1. Однослойная (Monolithic) архитектура**

В монолитной архитектуре все компоненты приложения тесно связаны и работают в одном процессе или сервере. Эта архитектура включает пользовательский интерфейс, бизнес-логику и доступ к данным в одном приложении.

Преимущества:

Простота разработки и развертывания.

Единое управление и мониторинг.

Подходит для небольших и средних приложений.

Недостатки:

Трудности масштабирования и изменения.

Сложность в управлении большим кодом.

Уязвимость к сбоям: ошибка в одном компоненте может повлиять на все приложение.

### **2. Многоуровневая (Multitier) архитектура**

Часто называется трехуровневой архитектурой (Three-tier architecture), эта модель разделяет приложение на три основные части:

Презентационный уровень (Presentation tier): Отвечает за отображение данных и взаимодействие с пользователем.

Уровень бизнес-логики (Business Logic tier): Обработывает логику приложения.

Уровень данных (Data tier): Отвечает за хранение и управление данными.

Преимущества:

Разделение обязанностей.

Улучшенная масштабируемость.

Легкость в поддержке и обновлении.



Недостатки:

Повышенная сложность и затраты на разработку.

Необходимость в четком управлении взаимодействием между уровнями.

### **3. Микросервисная (Microservices) архитектура**

Микросервисная архитектура разбивает приложение на множество небольших, независимых сервисов, каждый из которых выполняет конкретную функцию.

Преимущества:

Масштабируемость и гибкость.

Легкость в развертывании и обновлении отдельных сервисов.

Улучшенная устойчивость: сбой одного сервиса не приводит к сбою всего приложения.

Недостатки:

Повышенная сложность управления и оркестрации.

Необходимость в надежной системе для коммуникации между сервисами.

Потенциальные проблемы с согласованностью данных.

### **4. Архитектура клиент-сервер (Client-Server)**

Классическая модель, в которой клиентская часть (фронтенд) и серверная часть (бэкенд) работают отдельно и взаимодействуют через сеть.

Преимущества:

Четкое разделение обязанностей.

Легкость в обновлении клиентской и серверной частей независимо друг от друга.

Масштабируемость на уровне клиента и сервера.

Недостатки:

Возможные проблемы с производительностью из-за сетевой задержки.

Сложности в синхронизации изменений между клиентом и сервером.

### **5. Serverless архитектура**

Serverless архитектура позволяет разработчикам запускать код без управления серверами. В этой архитектуре инфраструктура управляется

сторонним поставщиком услуг, таким как AWS Lambda, Azure Functions или Google Cloud Functions.

Преимущества:

Автоматическое масштабирование.

Оплата только за фактическое использование ресурсов.

Упрощенное управление инфраструктурой.

Недостатки:

Ограниченная контроль над инфраструктурой.

Возможные проблемы с задержками при холодном старте.

Зависимость от конкретного поставщика услуг.

## **6. Архитектура на основе событий (Event-Driven)**

Эта архитектура включает компоненты, которые обмениваются информацией через события. Компоненты публикуют события, на которые подписываются другие компоненты.

Преимущества:

Высокая степень асинхронности и масштабируемости.

Разделение обязанностей.

Легкость в добавлении новых функций.

Недостатки:

Повышенная сложность разработки и отладки.

Необходимость в управлении событиями и подписчиками.

Возможные проблемы с согласованностью данных.

## **7. Прогрессивные веб-приложения (PWA)**

PWA — это тип веб-приложений, которые используют современные веб-технологии для обеспечения аналогичного пользовательского опыта как у нативных мобильных приложений. Они работают в браузере, но могут быть установлены на устройстве пользователя и работать офлайн.

Преимущества:

Улучшенный пользовательский опыт.

Доступность с любого устройства с браузером.

Возможность работы офлайн.

Недостатки:

Ограниченные возможности по сравнению с нативными приложениями.

Зависимость от поддержки браузера.

Выбор архитектуры веб-приложения зависит от множества факторов, включая требования к масштабируемости, производительности, управляемости и других. Важно оценить все плюсы и минусы каждой архитектуры и выбрать ту, которая лучше всего соответствует вашим целям и ресурсам.

### 1.3 Основные компоненты Spring MVC

Spring MVC (Model-View-Controller) — это компонент фреймворка Spring, который предоставляет мощные возможности для создания веб-приложений с разделением логики на модель, представление и контроллер. Это помогает организовать код и упростить поддержку и расширение приложения.

#### Основные компоненты Spring MVC

##### **Model (Модель):**

Модель отвечает за бизнес-логику приложения и управление данными. Она включает объекты данных, которые обрабатываются и манипулируются в приложении.

##### **View (Представление):**

Представление отвечает за отображение данных пользователю. В Spring MVC представления могут быть реализованы с использованием различных технологий, таких как JSP, Thymeleaf, FreeMarker и другие.

##### **Controller (Контроллер):**

Контроллеры обрабатывают пользовательские запросы, взаимодействуют с моделью и выбирают представление для отображения ответа пользователю. Контроллеры обычно аннотируются с помощью `@Controller` или `@RestController`.

#### **Работа Spring MVC:**

- **Запрос пользователя:**

Пользователь отправляет HTTP-запрос на сервер (например, через браузер).

- **Обработка запроса DispatcherServlet:**

DispatcherServlet — это фронт-контроллер Spring MVC. Он принимает все запросы, направленные к приложению, и направляет их в соответствующий контроллер.

- **Выбор подходящего контроллера:**

DispatcherServlet использует HandlerMapping для поиска подходящего контроллера, который должен обработать запрос.

- **Выполнение логики контроллера:**

Контроллер выполняет необходимую бизнес-логику, взаимодействует с моделью (например, с сервисами и репозиториями) и подготавливает данные для представления.

- **Выбор представления:**

Контроллер возвращает имя представления или объект представления, который будет использоваться для отображения данных пользователю.

ViewResolver отвечает за разрешение логического имени представления в физическое представление.

- **Отображение данных:**

Данные из модели передаются в представление, которое генерирует HTML или другой формат ответа для отправки пользователю.

Spring MVC помогает разделить обязанности в веб-приложении, что упрощает его разработку, тестирование и сопровождение. Использование MVC-шаблона позволяет сосредоточиться на отдельных аспектах приложения, улучшая модульность и повторное использование кода.

## Глава 2. Разработка веб-приложения фиксации состава и длительности тренировок

### 2.1 Порядок работы над приложением

#### 1.1. Порядок проведения работ

Порядок работы над приложением:

1. Определение требований к приложению.
2. Проектирование архитектуры и компонентов приложения.
3. Подготовка инструментов и среды разработки.
4. Написание кода для реализации требований.
5. Отладка и оптимизация.

#### Требования к приложению:

Функциональные требования	<ol style="list-style-type: none"><li>1. Создание тренировки (фиксация упражнений во время тренировки, расчёт длительности тренировки, отображение таймера).</li><li>2. Просмотр таблицы состоявшихся тренировок с основной информацией о них (дата, длительность, состав).</li><li>3. Просмотр пользователей приложения.</li></ol>
Требования к безопасности	<ol style="list-style-type: none"><li>1. Аутентификация и авторизация пользователей.</li><li>2. Данные о паролях шифруются.</li></ol>
Требования к интерфейсу	<ol style="list-style-type: none"><li>1. Простой, дружелюбный, понятный интерфейс.</li></ol>
Требования к производительности	<ol style="list-style-type: none"><li>1. Приложение выдерживает нагрузки и обеспечивает быстрый доступ к данным.</li></ol>

Технологии:

Язык программирования	Java
Фреймворк	Spring
База данных	MySQL
Шаблонизатор	Thymeleaf
Обеспечение безопасности	SpringSecurity
Среда разработки	IntelliJ IDEA

## 2.2 Проектирование архитектуры веб-приложения

В качестве архитектуры веб-приложения выбрана однослойная (монолитная) архитектура, которая подходит для небольших и средних приложений.

В рамках паттерна MVC приложение делится на следующие компоненты:

1. Model – включает классы Role, User, Training.
2. View – отображение данных осуществляется с использованием шаблонизаторов, которые позволяют вставлять данные из контроллеров в HTML-шаблоны.
3. Controller – в приложении реализованы 2 контроллера: TrainingController и UserController.
4. Repository – в приложении используются 3 репозитория: RoleRepository, UserRepository и TrainingRepository.
5. Service – бизнес-логику приложения реализуют классы: TrainingService, User Service.



## 2.3 Разработка веб-приложения фиксации состава и длительности тренировок

Model – включает классы Role, User, Training.

Класс Training (тренировка) содержит информацию об идентификаторе тренировки, дате, составе, времени начала и времени окончания занятия для расчёта общей длительности тренировки, длительности тренировки (Рис. 1).

```

    @Data
    @Entity
    @AllArgsConstructor
    @NoArgsConstructor
    @Table(name = "trainings")
    public class Training {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "id")
        private Long id;

        @Column(name = "createdDate")
        private LocalDate createdDate;

        @Column(name = "timeStart")
        private Long timeStart;

        @Column(name = "timeFinish")
        private Long timeFinish;

        @Column(name = "Duration")
        private Long duration;

        @Column(name = "title")
        private String title;

        @Column(name = "body")
        private String body;

        @ManyToOne(cascade = {CascadeType.MERGE, CascadeType.PERSIST}, fetch = FetchType.EAGER)
        @JoinColumn(name = "user_id")
        private User user;
    }

```

Рис.1

Класс User содержит информацию об идентификаторе пользователя, имени, логине и пароле, ролях и пройденных тренировках (Рис. 2).

```

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    private String login;

    private String password;

    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinTable(
        name = "users_roles",
        joinColumns = {@JoinColumn(name = "USER_ID", referencedColumnName = "ID")},
        inverseJoinColumns = {@JoinColumn(name = "ROLE_ID", referencedColumnName = "ID")})
    private List<Role> roles = new ArrayList<>();
    @OneToMany(mappedBy = "user", cascade = {CascadeType.MERGE, CascadeType.PERSIST},
        fetch = FetchType.EAGER)
    private List<Training> trainings = new ArrayList<>();

    no usages
    public void addTrainingToUser(Training training) {
        training.setUser(this);
        this.getTrainings().add(training);
    }
}

```

Рис.2

Для управления ролями в приложении создан класс Role (Рис. 3).

```

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "roles")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany(mappedBy = "roles")
    private List<User> users;
}

```

Рис.3

Для обработки запросов в приложении реализованы контроллеры:

1. TrainingController (Рис. 4).

```

@Slf4j
@AllArgsConstructor
@org.springframework.stereotype.Controller
public class TrainingController {

    private TrainingService trainingService;
    private UserService userService;

    no usages
    @GetMapping("/home")
    String home() { return "home"; }

    no usages
    @GetMapping("/createTraining")
    public String createTrainingForm(Model model) {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        User user = userService.findByLogin(authentication.getName());
        Training training = new Training();
        training.setCreatedDate(LocalDate.now());
        training.setTimeStart(System.currentTimeMillis());
        model.addAttribute(attributeName: "training", training);
        model.addAttribute(attributeName: "user", user);
        return "createTraining";
    }

    @PostMapping("/createTraining")
    public String createTrainings(Model model, @ModelAttribute Training training) {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        User user = userService.findByLogin(authentication.getName());
        training.setTimeFinish(System.currentTimeMillis());
        long duration = (training.getTimeFinish() - training.getTimeStart()) / 1000 / 60;
        training.setDuration(duration);
        training.setUser(user);
        trainingService.createTraining(training);
        List<Training> trainingsList = trainingService.getTrainingsByUser(user);
        model.addAttribute(attributeName: "trainings", trainingsList);
        return "storyTraining";
    }

    no usages
    @GetMapping("/storyTraining")
    public String diaryStory(Model model) {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        User user = userService.findByLogin(authentication.getName());
        List<Training> noteList = trainingService.getTrainingsByUser(user);
        model.addAttribute(attributeName: "trainings", noteList);
        return "storyTraining";
    }

    no usages
    @GetMapping("/deleteTraining/{id}")
    public String deleteTraining(@PathVariable Long id) {
        Training training = trainingService.getTrainingById(id);
        User user = training.getUser();
        user.getTrainings().remove(training);
        userService.updateUsers(user.getId(), user);
        trainingService.deleteTraining(id);
        return "redirect:/storyTraining";
    }

    no usages
    @GetMapping("/storyTrainingUpdate/{id}")
    public String updateTraining(@PathVariable("id") Long id, Model model) {
        Training training = trainingService.getTrainingById(id);
        model.addAttribute(attributeName: "training", training);
        return "storyTrainingUpdate";
    }

    no usages
    @PostMapping("/storyTrainingUpdate/{id}")
    public String updateNote(@PathVariable Long id, @ModelAttribute Training training) {
        trainingService.getTrainingById(training.getId());
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        User user = userService.findByLogin(authentication.getName());
        trainingService.updateTraining(id, training);
        return "redirect:/storyTraining";
    }
}

```

Рис.4

## 2. UserController (Рис.5)

```

@Controller
public class UserController {
    4 usages
    private UserService userService;
    no usages
    @Autowired
    public UserController(UserService userService) { this.userService = userService; }
    no usages
    @GetMapping("/")
    public String home() { return "home"; }
    no usages
    @GetMapping("/login")
    public String loginForm() { return "login"; }
    no usages
    @GetMapping("register")
    public String showRegistrationForm(Model model) {
        User user = new User();
        model.addAttribute(attributeName: "user", user);
        return "register";
    }
    no usages
    @PostMapping("/register/save")
    public String registration(@ModelAttribute("user") User user, Model model) {
        User existing = userService.findByLogin(user.getLogin());
        if (existing != null) {
            model.addAttribute(attributeName: "error", attributeValue: "Account already registered");
            model.addAttribute(attributeName: "user", user);
            return "register";
        }
        userService.saveUser(user);
        return "redirect:/register?success";
    }
    no usages
    @GetMapping("/users")
    public String listRegisteredUsers(Model model) {
        List<User> users = userService.findAllUsers();
        model.addAttribute(attributeName: "users", users);
        return "users";
    }
}

```

Рис. 5

В приложении используются 3 репозитория: RoleRepository, UserRepository и TrainingRepository (Рис. 6).

```

@Repository
public interface RoleRepository extends JpaRepository<Role, Long> {
    1 usage
    Role findByName(String name);
}

public interface TrainingRepository extends JpaRepository<Training, Long> {
    1 usage
    List<Training> findByUser(User user);
}

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    2 usages
    User findByLogin(String login);
}

```

Рис.6

Бизнес-логику приложения реализуют классы:

1. TrainingService (Рис.7).

```
@Service
@RequiredArgsConstructor
public class TrainingService {
    private final TrainingRepository trainingRepository;

    1 usage
    public Training createTraining(Training training) {
        training.setCreatedDate(LocalDate.now());
        training.setTimeStart(System.currentTimeMillis());
        return trainingRepository.save(training);
    }

    no usages
    public List<Training> getAllNotes() {
        return trainingRepository.findAll();
    }

    3 usages
    public Training getTrainingById(Long id) {
        return trainingRepository.findById(id).orElseThrow(() -> new RuntimeException("Training not found"));
    }

    1 usage
    @Transactional
    public void deleteTraining(Long id) {
        trainingRepository.deleteById(id);
    }

    2 usages
    public List<Training> getTrainingsByUser(User user) {
        return trainingRepository.findByUser(user);
    }

    1 usage
    @Transactional
    public Training updateTraining(Long id, Training training) {
        Training trainingById = trainingRepository.findById(id).orElseThrow(() -> new RuntimeException("Training not found"));
        trainingById.setTitle(training.getTitle());
        training.setBody(training.getBody());
        return trainingRepository.save(trainingById);
    }
}
```

Рис.7

2. User Service (Рис.8).

```

public class UserService {
    8 usages
    private UserRepository userRepository;
    3 usages
    private RoleRepository roleRepository;
    2 usages
    private PasswordEncoder passwordEncoder;
    no usages
    public UserService(UserRepository userRepository, RoleRepository roleRepository, PasswordEncoder passwordEncoder) {
        this.userRepository = userRepository;
        this.roleRepository = roleRepository;
        this.passwordEncoder = passwordEncoder;
    }
    1 usage
    public void saveUser(User user) {
        User createdUser = new User();
        createdUser.setName(user.getName());
        createdUser.setLogin(user.getLogin());
        createdUser.setPassword(passwordEncoder.encode(user.getPassword()));
        Role role = roleRepository.findByName("ROLE_ADMIN");
        if (role == null) {
            role = roleCheck();
        }
        createdUser.setRoles(List.of(role));
        userRepository.save(createdUser);
    }
    1 usage
    private Role roleCheck() {
        Role role = new Role();
        role.setName("ROLE_ADMIN");
        return roleRepository.save(role);
    }
    no usages
    public User findUserById(Long id) {
        return userRepository.findById(id).orElseThrow(() -> new RuntimeException("User not found."));
    }
    5 usages
    public User findByLogin(String login) { return userRepository.findByLogin(login); }
    1 usage
    public List<User> findAllUsers() { return userRepository.findAll(); }
    no usages
    public void deleteUser(Long id) { userRepository.deleteById(id); }
    1 usage
    @Transactional
    public User updateUser(Long id, User user) {
        User updateUser = userRepository.findById(id).orElseThrow(() -> new RuntimeException("User not found"));
        updateUser.setName(user.getName());
        updateUser.setLogin(user.getLogin());
        updateUser.setTrainings(user.getTrainings());
        return userRepository.save(updateUser);
    }
}

```

Рис.8

Для обеспечения безопасности используется Spring Security.

Класс CustomUserDetailsService используется для интеграции с Spring Security и обеспечивает аутентификацию пользователей на основе данных из базы данных. Он загружает пользователя по его логину, преобразует его роли в формат, понятный Spring Security, и возвращает объект UserDetails, который используется для проверки аутентификации и авторизации (Рис.9).

```

@Configuration
@EnableWebSecurity
public class SecurityConfig {
    1 usage
    @Autowired
    private UserDetailsService userDetailsService;
    1 usage
    @Bean
    public static PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }
    no usages
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests((authorize) ->
                authorize.requestMatchers(...patterns: "/register/**").permitAll()
                    .requestMatchers(...patterns: "/index").permitAll()
                    .requestMatchers(...patterns: "/users", "/note-delete/**", "/*").hasRole("ADMIN")
            ).formLogin(
                form -> form
                    .loginPage("/login")
                    .loginProcessingUrl("/login")
                    .defaultSuccessUrl("/home")
                    .permitAll()
            ).logout(
                logout -> logout
                    .logoutRequestMatcher(new AntPathRequestMatcher(pattern: "/logout"))
                    .permitAll()
            );
        return http.build();
    }
    no usages
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .userDetailsService(userDetailsService)
            .passwordEncoder(passwordEncoder());
    }
}

```

Рис.9

Класс CustomUserDetailsService реализует интерфейс UserDetailsService из Spring Security. Он загружает пользователя по его логину (email), преобразует его роли в формат, понятный Spring Security, и возвращает объект UserDetails, который используется для проверки аутентификации и авторизации (Рис.10).

```

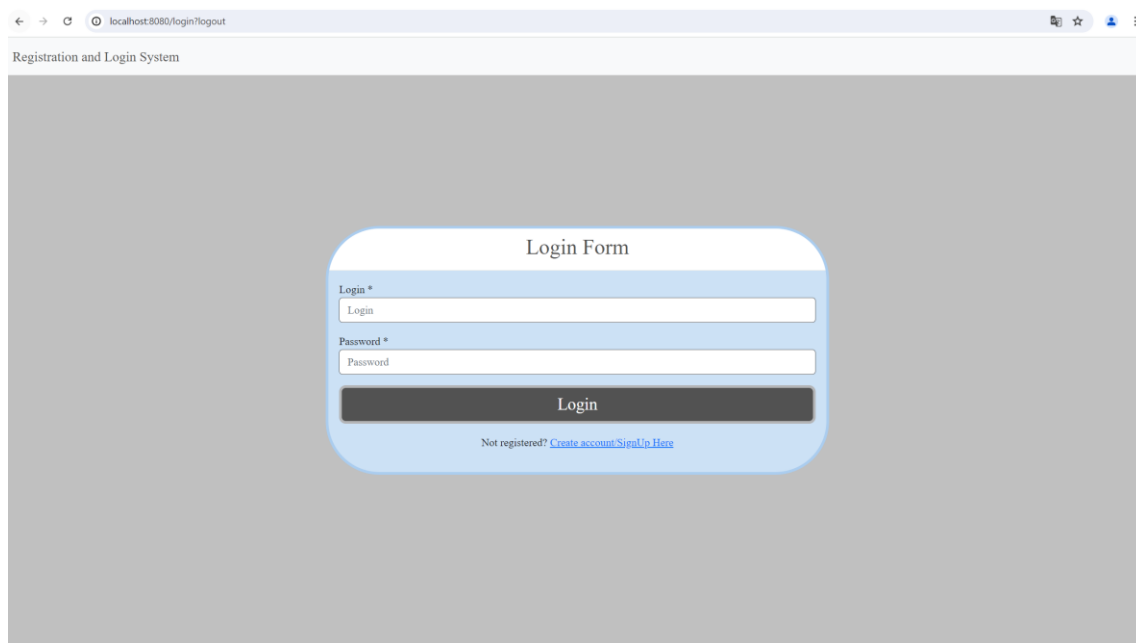
@Component
@RequiredArgsConstructor
public class CustomUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;
    no usages
    @Override
    public UserDetails loadUserByUsername(String userLogin) throws UsernameNotFoundException {
        User user = userRepository.findByLogin(userLogin);
        if(user != null){
            Set<GrantedAuthority> authorities = user.getRoles().stream() Stream<Role>
                .map(role -> new SimpleGrantedAuthority(role.getName())) Stream<SimpleGrantedAuthority>
                .collect(Collectors.toSet());
            return new org.springframework.security.core.userdetails.User(user.getLogin(),user.getPassword(), authorities);
        } else {
            throw new UsernameNotFoundException("Invalid username or password.");
        }
    }
}

```

Рис. 10

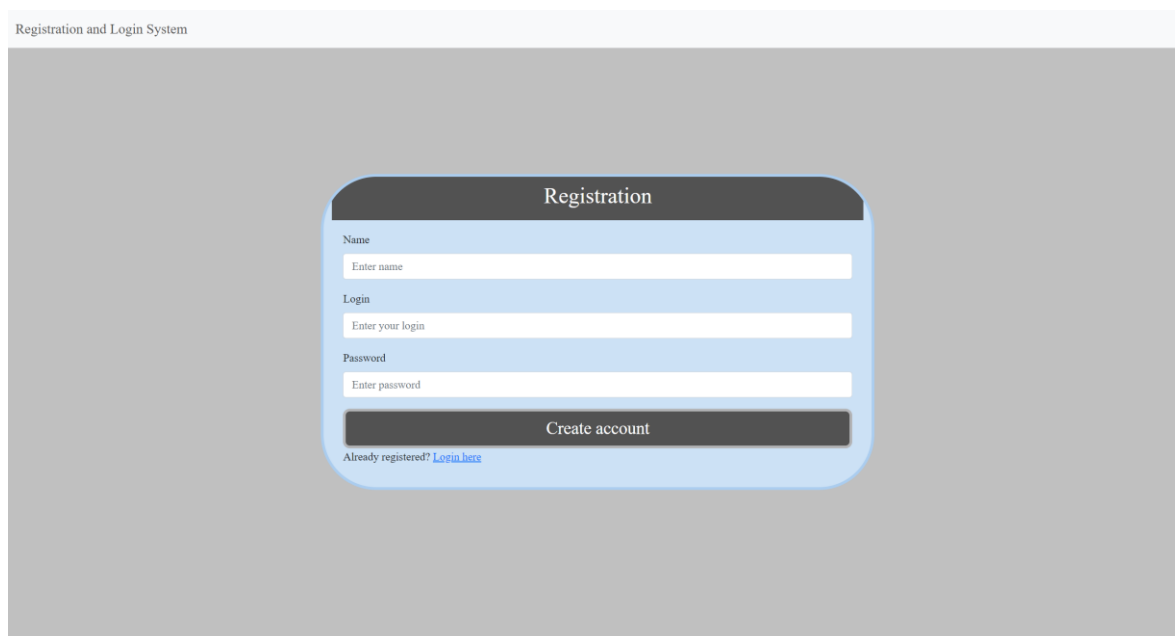
Для отображения данных приложения был использован HTML и разработаны соответствующие файлы. Для страницы авторизации (Рис. 11) – «login.html».



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/login/logout'. The page title is 'Registration and Login System'. The main content area features a light blue rounded rectangle titled 'Login Form'. Inside this form, there are two input fields: 'Login \*' with the placeholder text 'Login' and 'Password \*' with the placeholder text 'Password'. Below these fields is a dark grey button labeled 'Login'. At the bottom of the form, there is a link that reads 'Not registered? [Create account](#) [Sign In Here](#)'.

Рис.11

Для страницы регистрации (Рис. 12) – «register.html».



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/register'. The page title is 'Registration and Login System'. The main content area features a light blue rounded rectangle titled 'Registration'. Inside this form, there are three input fields: 'Name' with the placeholder text 'Enter name', 'Login' with the placeholder text 'Enter your login', and 'Password' with the placeholder text 'Enter password'. Below these fields is a dark grey button labeled 'Create account'. At the bottom of the form, there is a link that reads 'Already registered? [Login here](#)'.

Рис.12

Для главной страницы (Рис. 13) – «home.html».



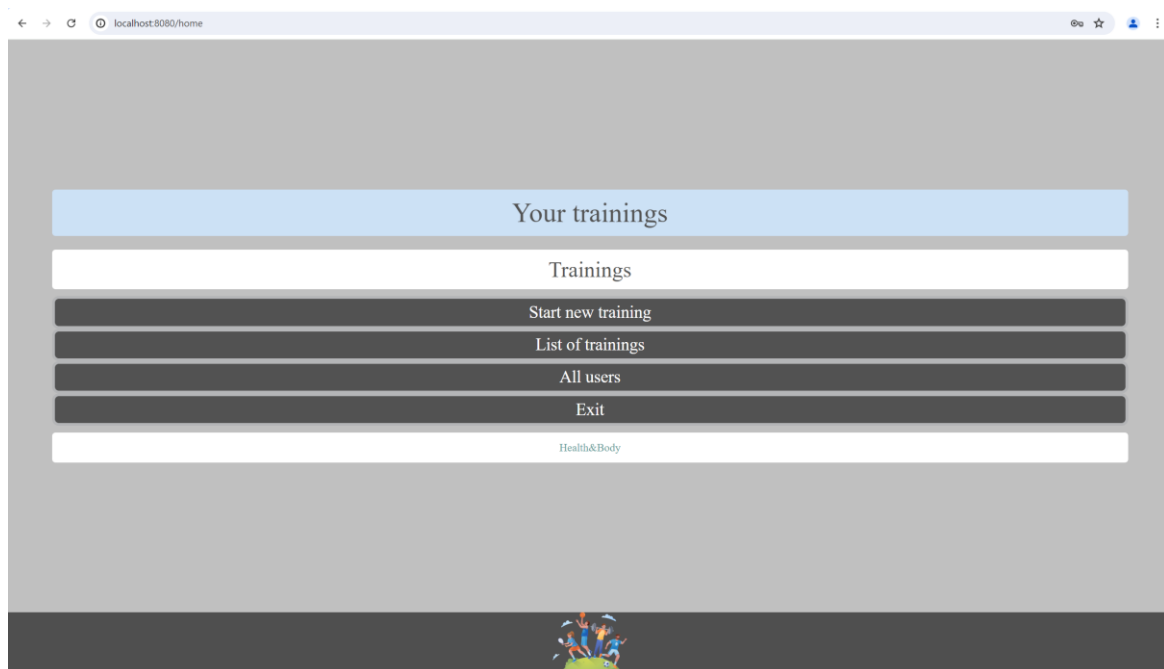


Рис.13

Для страницы тренировки (Рис. 14) – «createTraining.html».

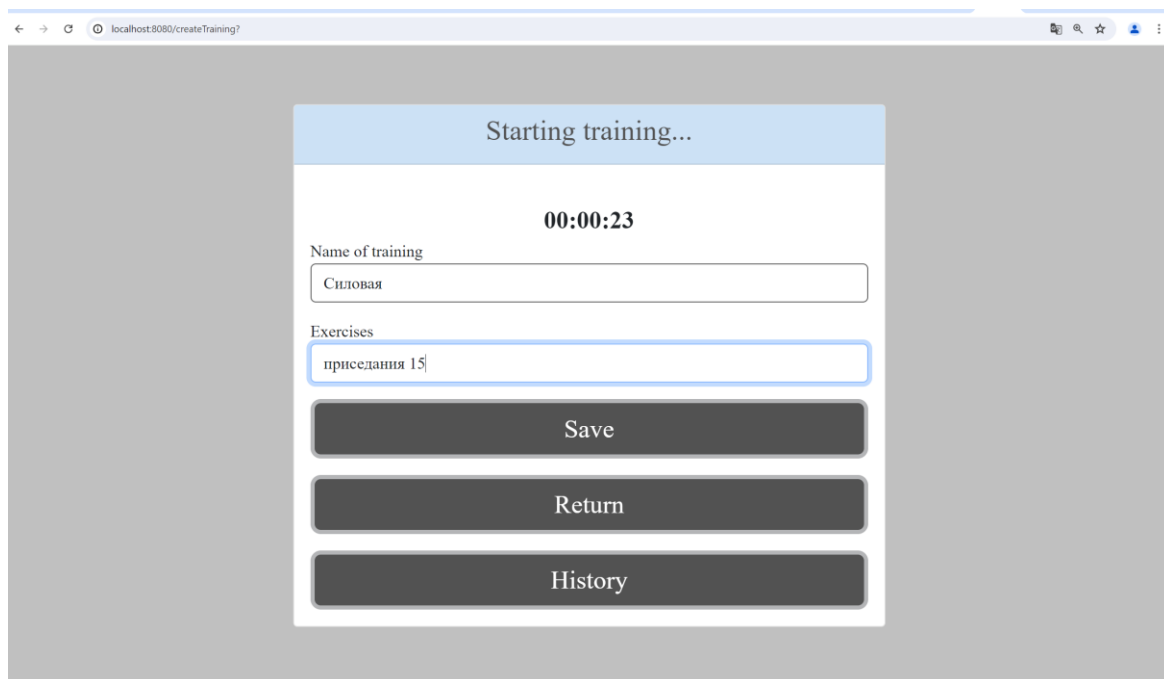


Рис.14

Для страницы со списком тренировок (Рис. 15) – «storyTraining.html» и «storyTrainingUpdate.html».

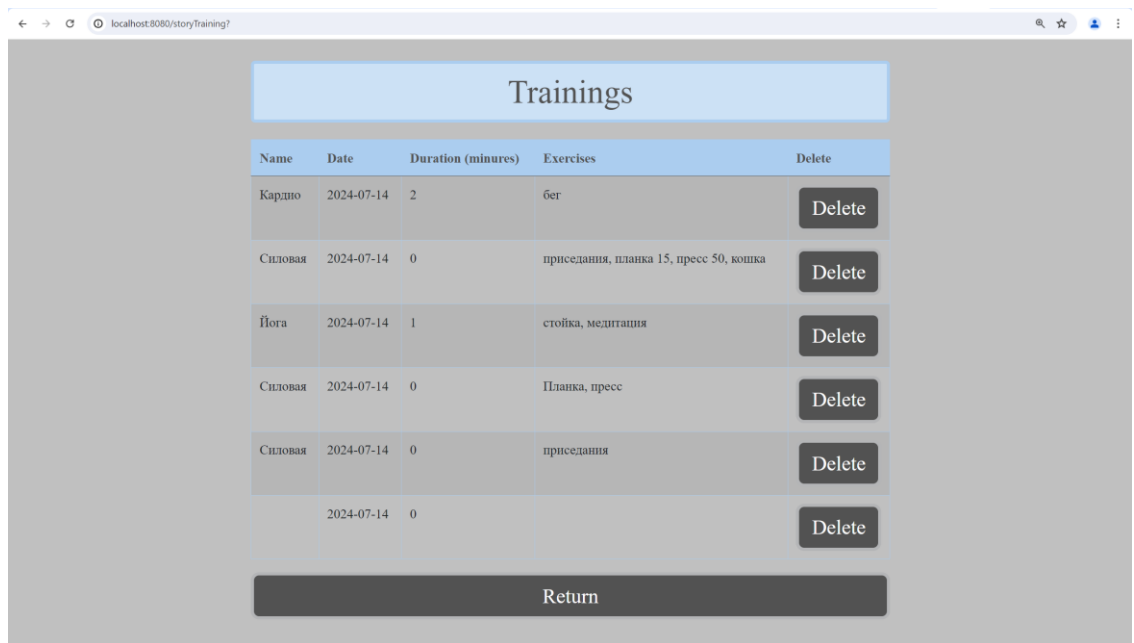


Рис.15

Для страницы со списком пользователей (Рис. 16) – «users.html».

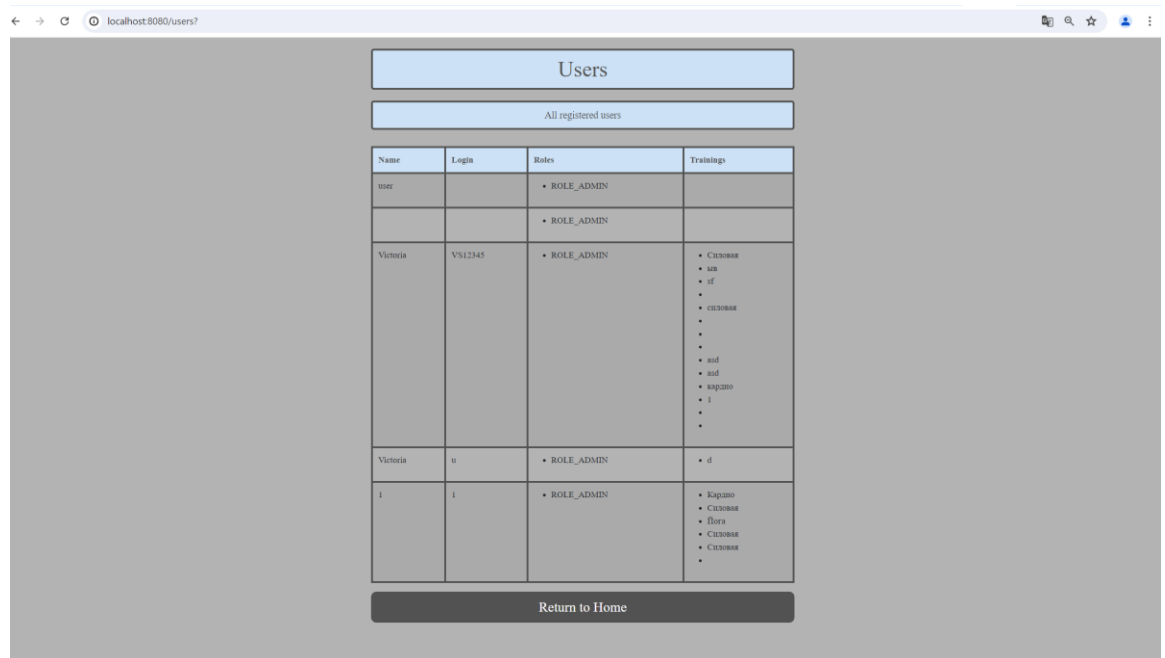


Рис.15

## **Заключение**

В процессе работы над проектом:

1. Рассмотрены основные компоненты веб-приложения, особенности компонента Spring MVC, а также существующие виды архитектуры веб-приложений.

2. Определение порядок и требования к приложению, спроектирована архитектура веб-приложения.

Разработано веб-приложение, позволяющее фиксировать выполненные элементы во время спортивных тренировок, вести записи о прошедших тренировках, их составе и длительности. Данное приложение нацелено на поддержание мотивации к спортивным тренировкам и может помочь сохранить интерес к регулярным занятиям.

### **Список литературы**

1. Лауренциу Спилкэ. Spring быстро. Санкт-Петербург: Питер, 2023.
2. Евгений Борисов. Spring-потрошитель, часть 1 [Видео]. URL: <https://www.youtube.com/watch?v=BmBr5diz8WA>.
3. Евгений Борисов. Spring-потрошитель, часть 1 [Видео]. URL: [https://www.youtube.com/watch?v=cou\\_qomYLNU](https://www.youtube.com/watch?v=cou_qomYLNU).
4. Евгений Борисов. Spring-построитель [Видео]. URL: <https://www.youtube.com/watch?v=rd6wxPzXQvo>.
5. Крейг Уоллс. Spring в действии. Москва: ДМК Пресс, 2022.