Universitatea de Stat din Moldova
Facultatea Matematică şi Informatică
Departamentul Informaticii Aplicate

Teză de an

Spring Security: Implementarea aplicațiilor sigure în Java

A efectuat:
studentul grupei IA2303
Gutu Nicolae

Conducător ştiinţific:
Lector al Dep. de Informatică
Ciornei Oleg

Chişinău, 2025

Оглавление

Введение

В настоящее время большинство веб-сервисов требует от пользователей прохождения аутентификации и авторизации для безопасного доступа к персонализированным данным и функциям. С увеличением объёмов информации и с усложнением методов кибератак необходимость в надёжных системах защиты становится критически важной. Последствия таких угроз могут включать в себя утечку конфиденциальной информации, нарушение прав пользователей и даже остановку работы системы. В связи с этим защита от подобных рисков становится обязательным этапом в процессе проектирования и реализации современных веб-приложений.

Одним из самых распространённых и мощных инструментов обеспечения безопасности в Java-приложениях является Spring Security — модуль, входящий в экосистему Spring Framework. Он позволяет реализовать тонкую настройку доступа, шифрование данных, проверку подлинности и интеграцию с различными протоколами безопасности. В дополнение к нему активно используется технология JWT (JSON Web Token), обеспечивающая безопасный способ передачи информации между клиентом и сервером без хранения состояния на стороне сервера.

Цель данной курсовой работы — разработать защищённое серверное приложение с реализацией регистрации, аутентификации и авторизации пользователей на базе Spring Boot, Spring Security и JWT. Также в ходе работы будет продемонстрирован механизм разграничения прав доступа по ролям и внедрение базовых мер защиты от атак.

Рис.1 Spring Security

Глава 1. Обзор существующего положения в изучаемой области или исследование соответствующей области.

1.1. Безопасность веб-приложений: основные угрозы и задачи

Современные веб-приложения являются неотъемлемой частью цифровой инфраструктуры — они используются для работы с персональными данными, проведения финансовых операций, взаимодействия между организациями и конечными пользователями. При этом они всё чаще становятся целью кибератак, так как открыты для взаимодействия через интернет и нередко содержат уязвимости.

Основные задачи обеспечения безопасности включают:

- **Защиту конфиденциальности данных** (например, паролей, токенов).
- **Предотвращение несанкционированного доступа** к функционалу и данным.
- **Обеспечение целостности и подлинности данных** при передаче и хранении.
- **Контроль над действиями пользователей** в системе.

Наиболее распространённые угрозы:

1) **Brute Force атаки** — подбор пароля с помощью автоматизированных программ. Простой пароль или неограниченное количество попыток входа делают систему уязвимой.
2) **SQL-инъекции** — внедрение вредоносного SQL-кода в поля ввода, позволяющее злоумышленнику получить или изменить данные в базе.
3) **XSS (межсайтовый скриптинг)** — внедрение скриптов, исполняемых в браузере жертвы, что может привести к краже сессий или токенов.
4) **CSRF (межсайтовая подделка запроса)** — обман системы авторизации, когда пользователь совершает действия от своего имени, не подозревая об этом.
5) **Утечка токенов / сессионных данных** — в случае неправильного хранения или передачи токенов злоумышленник может получить доступ от имени пользователя.

Нарушение безопасности может привести к:

1) Утечке персональных и конфиденциальных данных.
2) Потере доверия пользователей.
3) Юридической ответственности компании.
4) Финансовым потерям и репутационным рискам.

Поэтому безопасность веб-приложений — это не просто техническая опция, а ключевой компонент архитектуры программных систем, который необходимо проектировать с самого начала.

1.2. Аутентификация и авторизация: различие и модели управления доступом

В системах информационной безопасности два ключевых процесса — **аутентификация** и **авторизация**. Хотя они тесно связаны, они выполняют разные задачи и реализуются независимо.

Аутентификация (authentication) — это процесс подтверждения личности пользователя. Он проверяет, действительно ли пользователь является тем, за кого себя выдаёт. Чаще всего для этого используется пара логин/пароль, однако также применяются более надёжные методы, такие как двухфакторная аутентификация, токены, биометрия и другие.

Авторизация (authorization) — это процесс, следующий за аутентификацией. Он определяет, какие действия и ресурсы доступны пользователю в рамках его прав. Иными словами, авторизация отвечает на вопрос: «что разрешено делать пользователю, прошедшему аутентификацию».

Например, пользователь может успешно войти в систему (аутентификация), но при этом не иметь доступа к административной панели (отсутствие авторизации на этот ресурс).

Модели управления доступом

Для реализации авторизации применяются различные модели контроля доступа, каждая из которых имеет свои особенности, преимущества и недостатки. Наиболее распространённые из них:

1. MAC (Mandatory Access Control) — мандатная модель

Это жёстко централизованная модель, в которой доступ пользователей к ресурсам регулируется на основе правил, установленных системным администратором или политиками безопасности. Изменить эти правила пользователь не может. Чаще всего используется в системах с высоким уровнем конфиденциальности (например, военные или государственные структуры).

- **Преимущества:** высокий уровень защиты, строгое соблюдение правил.
- **Недостатки:** низкая гибкость, сложность в администрировании.

2. DAC (Discretionary Access Control) — дискреционная модель

В этой модели владельцы объектов сами управляют доступом к своим ресурсам. Например, пользователь может вручную предоставить или закрыть доступ к своим файлам.

- **Преимущества:** простота, гибкость, индивидуальный контроль.
- **Недостатки:** более высокий риск ошибок со стороны пользователей, сложность масштабирования.

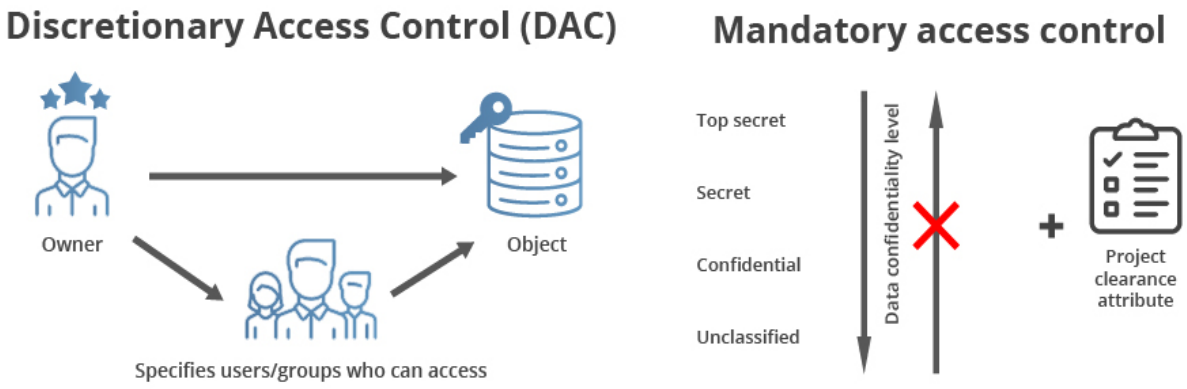


Рис.2 Mandatory Access Control & Discretionary Access Control models

3. RBAC (Role-Based Access Control) — ролевая модель

Одна из самых популярных моделей, особенно в веб-приложениях. Доступы настраиваются не для каждого пользователя отдельно, а через роли. Пользователям назначаются роли, каждая из которых содержит определённый набор прав.

- **Преимущества:** удобство масштабирования, централизованное управление доступом, лёгкость интеграции с системами.
- **Недостатки:** требует предварительного проектирования ролей, не подходит при сильно индивидуальных правах.

1.3. Spring Security: назначение и основные возможности

Spring Security — это мощный и гибкий фреймворк из экосистемы Spring, предназначенный для обеспечения безопасности приложений, разработанных на языке Java. Он предоставляет обширный инструментарий для реализации аутентификации, авторизации, защиты от атак и безопасного хранения данных.

Функциональные предназначения фрэеймворка:

1) **Организовать аутентификацию** пользователей с помощью различных источников (встроенная БД, LDAP, OAuth, JWT и др.).
2) **Реализовать авторизацию** на основе ролей и прав;
3) **Применять разграничение доступа** на уровне URL-адресов, методов контроллеров, сервисов и даже выражений SpEL.
4) **Шифровать пароли** с помощью различных алгоритмов (в том числе BCrypt).
5) **Настраивать цепочку фильтров**, которые перехватывают запросы до передачи в контроллеры.
6) **Защищать приложение** от типичных атак (CSRF, Clickjacking, Session Fixation и др.).

Фреймворк построен по принципу фильтрации запросов, где каждый HTTP-запрос обрабатывается через цепочку фильтров безопасности. Разработчик может как использовать стандартные фильтры, так и создавать собственные, например, для работы с JWT.

1.4. JWT (JSON Web Token): принцип работы и применение

JWT (JSON Web Token**)** — это URL-безопасный способ передачи данных между участниками. Он используется для аутентификации и авторизации в современных веб-приложениях. JWT-токены позволяют клиенту хранить в себе всю необходимую информацию, подтверждающую подлинность пользователя, что исключает необходимость поддержания состояния на сервере.

Структура токена

JWT состоит из трёх частей, разделённых точками: xxxxx.yyyyy.zzzzz

1) **Header (заголовок):** содержит тип токена и алгоритм подписи (например, HS256).
2) **Payload (полезная нагрузка):** включает в себя claims — данные, такие как username, roles, exp (время истечения), userId и т.д.
3) **Signature (подпись):** создаётся путём хэширования заголовка и payload с использованием секретного ключа, известного только серверу.

Это обеспечивает целостность токена: если кто-то попытается изменить содержимое payload, подпись уже не совпадёт.

Рис.3 JWT Token Structure

Преимущества использования JWT:

- Stateless: не требует хранения информации о сессиях на сервере.
- Хорошо подходит для масштабируемых систем и микросервисов.
- Быстро обрабатывается (не требует дополнительных запросов в БД при каждом вызове).
- Удобно хранить информацию о ролях и пользователе внутри токена.

Недостатки и риски:

- Угрозы при краже токена: если злоумышленник получит доступ к токену, он сможет использовать его до истечения срока.
- XSS-атаки: если токен хранится в небезопасном месте (например, в localStorage), он может быть украден скриптами.
- Отозвать токен досрочно не так просто, как сессионные данные — для этого нужна система blacklist или Refresh Token.

1.5. Хранение пользовательских данных и защита базы данных в современных веб-приложениях

Веб-приложения, взаимодействующие с пользователями, практически всегда опираются на базу данных, где хранятся такие сведения, как логины, email-адреса, пароли, роли, история действий и другая персональная информация. Важнейшими задачами при работе с такими данными являются:

- Обеспечение целостности и доступности информации.
- Конфиденциальность чувствительных данных (особенно паролей).
- Защита от несанкционированного доступа и утечек.

Хранилища данных: СУБД и подходы

Для хранения пользовательских данных используются два основных типа СУБД:

1. Реляционные базы данных (SQL)

Классический подход, основанный на таблицах и строгой схеме данных (с определением типов, связей и ограничений). К наиболее популярным относятся:

- **PostgreSQL** — открытая, мощная, поддерживает ACID, имеет активное сообщество;
- **MySQL / MariaDB** — часто используется в веб-хостинге, быстро работает, прост в использовании;
- **Oracle Database** — корпоративное решение, часто применяется в крупных системах.

2. Документо-ориентированные базы данных (NoSQL)

Подход, не требующий строгой схемы. Например:

- **MongoDB** — хранит данные в формате JSON-подобных документов, гибкий по структуре, удобен для хранения вложенных структур;
- **Couchbase, Redis, Cassandra** — используются для специфических задач: кэширования, потоковой обработки, масштабирования.

На практике, реляционные БД остаются стандартом для систем, в которых требуется строгий контроль структуры и связей между сущностями - то, что необходимо при хранении пользователей, ролей и логики доступа.

Защита паролей: алгоритмы и стандарты

Один из важнейших аспектов — безопасное хранение паролей. Их нельзя хранить в открытом виде, поэтому в индустрии используются специализированные криптографические хэш-функции**:**

- **BCrypt** — один из наиболее надёжных и распространённых методов; адаптивен и включает соль по умолчанию.
- **Argon2** — признан лучшим в конкурсе Password Hashing Competition (PHC), защищает от атак с использованием GPU.
- **PBKDF2** — одобрен NIST, широко используется, хотя и менее устойчив к современным атакам, чем Argon2.

Глава 2. Проект системы.

2.1 Назначение и предполагаемые сценарии использования

Разрабатываемый проект представляет собой серверную часть веб-приложения на базе **Java и Spring Boot**, предназначенную для демонстрации работы модуля безопасности **Spring Security** и реализации механизмов защиты приложений.
Проект реализует **регистрацию, аутентификацию и авторизацию** пользователей с разграничением прав доступа на основе ролей.

Благодаря модульной архитектуре и использованию REST-подхода, данную систему можно адаптировать под множество сценариев применения, в том числе:

1) Платформы для ведения блогов с разграничением доступа для авторов, редакторов и администраторов.
2) Системы бронирования **(**гостиницы, рестораны, сервисы).
3) Внутренние CRM-системы для малого и среднего бизнеса.
4) Панели управления пользователями и ролями в админ-интерфейсах
5) Backend для мобильных приложений и SPA-фронтенда.

Гибкость архитектуры позволяет настраивать уровни доступа, расширять логику, добавлять дополнительные компоненты и методы защиты.

2.2 Файловая структура и описание модулей

Файловая структура проекта организована в соответствии с принципами чистой архитектуры и разделения ответственности. Ниже представлены основные пакеты и их назначение:

Таблица1 Описание модулей

Пакет / файл	**Назначение**
controller	Содержит REST-контроллеры, обрабатывающие входящие HTTP-запросы.
service	Реализует бизнес-логику и взаимодействие между слоями приложения.

repository	Интерфейсы для доступа к базе данных через Spring Data JPA.
model	Сущности, соответствующие таблицам базы данных.
DTO	Классы передачи данных между клиентом и сервером.
security	Настройки доступа к энпоинтам, генерация JWT
config	Конфигурационные классы, в том числе настройки безопасности.
resources/application.properties resources/logback-spring.xml	Файлы с параметрами подключения к БД, JWT, портом и настройка логирования в файл.
filter	Содержит в себе java - класс фильтра токенов
exception	Глобальный обработчик ошибок(404/405/422/500 и прочие исключения генерируемые Java, Spring)
exceptionSecutiry	Обработчик ошибок безопасности (401/403)
log	Хранятся логи сервера

В корне проекта расположен файл запуска (SecurityApp.java)

Файловая структура:

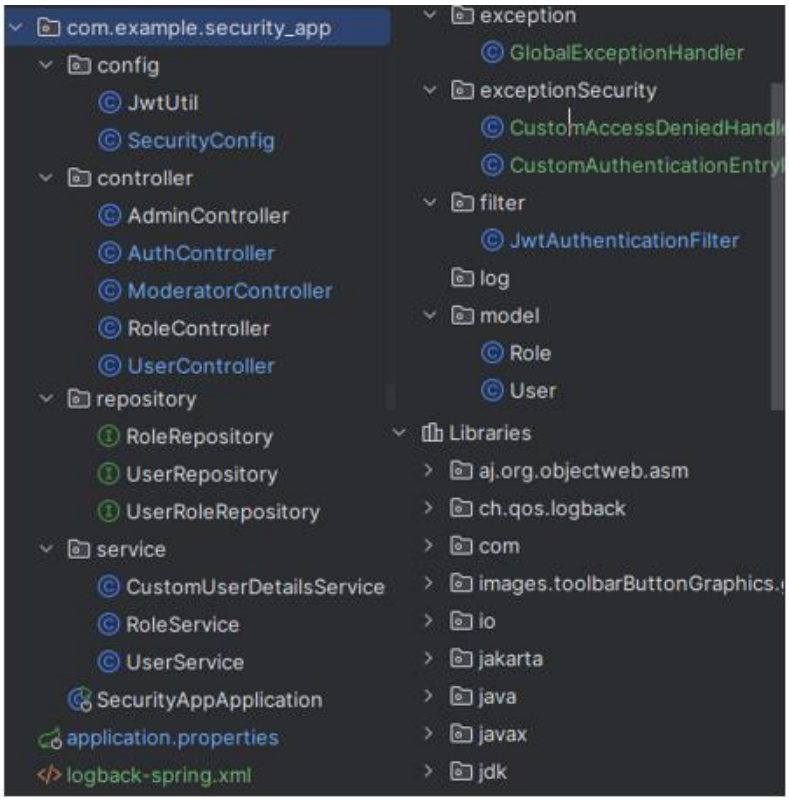


Рис.4 Файловая структура проекта

2.3 Архитектура и логика системы

2.3.1 Архитектурные принципы

Проект построен на основе многослойной архитектуры с чётким разделением ответственности между компонентами. Такой подход упрощает поддержку, тестирование и расширение системы. Архитектурные решения опираются на следующие ключевые принципы:

Принцип единственной ответственности (SRP)

SRP (Single Responsibility Principle) — один из принципов SOLID, предполагающий, что каждый компонент должен выполнять только одну функцию и иметь лишь одну причину для изменения. Это способствует читаемости, сопровождаемости и устойчивости архитектуры при доработках.

В проекте соблюдение SRP выражается через чёткое разделение слоёв:

1) **Контроллеры (controller)** — принимают и обрабатывают HTTP-запросы, не содержат бизнес-логики;
2) **Сервисы (service)** — реализуют бизнес-логику и управляют последовательностью операций;
3) **Репозитории (repository)** — отвечают исключительно за работу с базой данных;

И т.д.

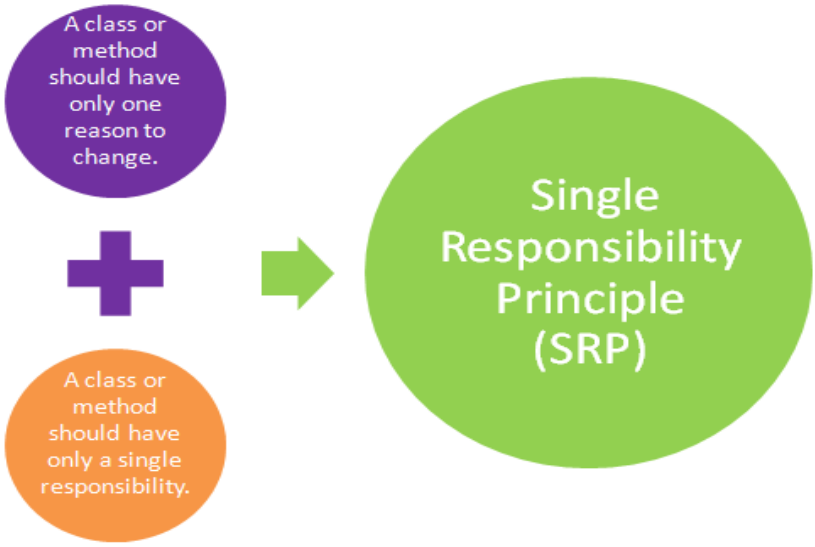


Рис.5 Single Responsibility Principle

REST-архитектура

Взаимодействие клиента и сервера построено по принципам REST:

1) Каждый ресурс представлен в виде URL (например, /users, /auth/login).
2) Используются стандартные HTTP-методы: GET, POST, PUT, DELETE.
3) Система не хранит состояние между запросами (stateless).
4) Чёткое разделение между сущностями, их действиями и URL-структурой.

Такая архитектура облегчает разработку frontend-приложений (SPA, мобильные клиенты), а также даёт хорошую масштабируемость и расширяемость.

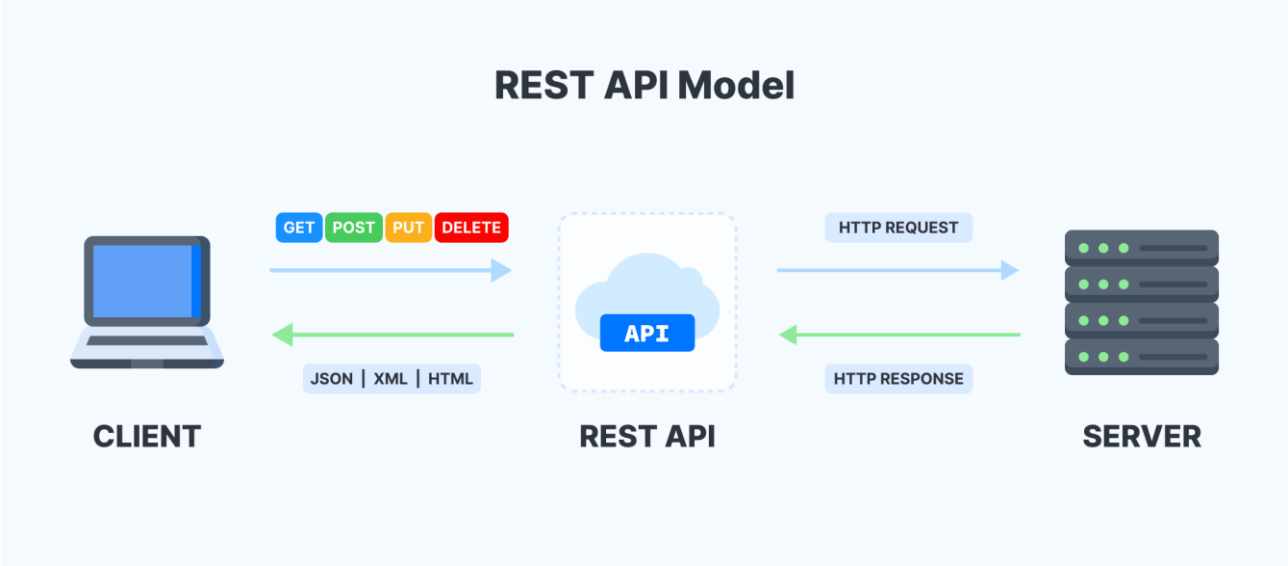


Рис.6 REST-API Model

Stateless-подход

Приложение работает по принципу безсессионного хранения состояния. Сервер не хранит данные о текущем пользователе — вместо этого аутентификационная информация передаётся в каждом запросе в виде JWT-токена.

Преимущества:

1) Высокая масштабируемость (возможность обрабатывать запросы на разных серверах).
2) Простота отказоустойчивой архитектуры.
3) Отсутствие нагрузки на память сервера из-за хранения сессий.

2.3.2 Структура контроллеров и маршрутов

Веб-приложения, работающие с пользовательскими данными, нуждаются в чёткой и надёжной системе безопасности. В данном проекте безопасность реализована на базе Spring Security с использованием технологии JWT (JSON Web Token) и модели RBAC (Role-Based Access Control).

Разделение контроллеров по ролям (RBAC)

В дополнение к SRP в проекте реализован подход (Role-Based Access Control) согласно которому контроллеры разделяются не только по функциям, но и по ролям пользователей, которым они предназначены. Это обеспечивает логическое и техническое разделение областей доступа.

- USER — стандартный пользователь;
- MODERATOR — имеет доступ к управлению контентом и пользователями;
- ADMIN — обладает полным доступом ко всем функциям системы.

В реализации это выражается следующим образом:

1) UserController обслуживает маршруты /users/** (только для USER);
2) ModeratorController обрабатывает /moderator/** (доступен MODERATOR);
3) AdminController работает с /admin/** (доступен ADMIN).
4) AuthController работает с /auth/** (доступен всем пользователям, в том числе и не аутентифицированым)

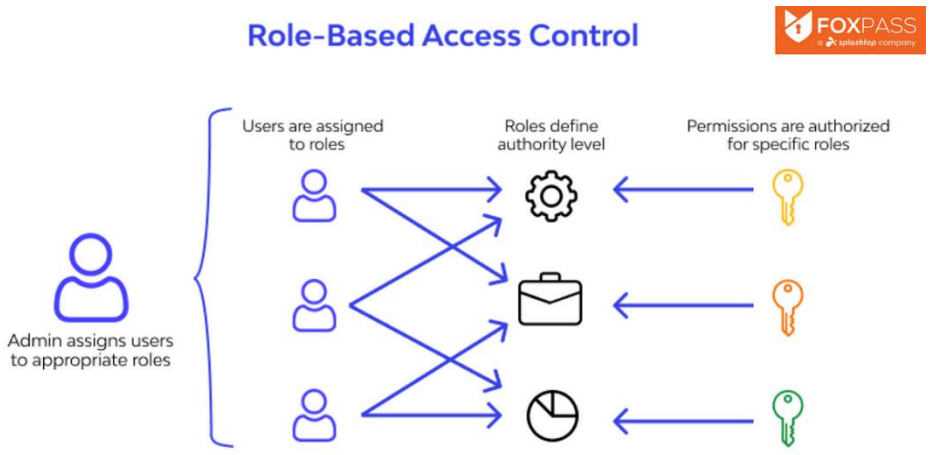


Рис.7 Role-Based Access Control

2.3.3 Безопасность и контроль доступа

Разделение данных и логики

Передача данных между слоями реализуется с использованием объектов **DTO** (Data Transfer Object), что позволяет отделить бизнес-логику от представления и обеспечить защиту от утечек лишней информации.

Механизм аутентификации и авторизации в проекте построен на использовании JWT-токенов**,** обработке их через Spring Security**,** и применении ролевой модели для контроля доступа.

Полный цикл выглядит следующим образом:

1) Пользователь отправляет логин и пароль на эндпоинт /auth/login.
2) Сервер проверяет данные, и, если они корректны — генерирует JWT-токен**,** содержащий информацию о пользователе, его идентификаторе и роли.
3) Клиент сохраняет токен и добавляет его в заголовок каждого запроса (Authorization: Bearer <токен>).
4) Запрос перехватывается кастомным фильтром JwtAuthenticationFilter, встроенным в цепочку Spring Security.
5) Фильтр извлекает токен, проверяет подпись и срок действия, после чего устанавливает пользователя в контекст безопасности.
6) На основании информации о роли, содержащейся в токене, происходит проверка доступа к маршруту
7) В случае успеха запрос попадает в контроллер, в противном случае возвращается ошибка 401 Unauthorized или 403 Forbidden.

Обработчики ошибок

Для повышения надёжности взаимодействия с API в проекте также предусмотрен глобальный обработчик ошибок, который обеспечивает возврат стандартизированных JSON-ответов при возникновении исключений. Это позволяет унифицировать формат ошибок, улучшить читаемость откликов со стороны клиента и облегчить отладку.

Дополнительные меры безопасности

В дополнение к базовой авторизации и аутентификации реализованы дополнительные механизмы защиты:

1. **Защита от Brute Force-атак.**
   После каждой неудачной попытки входа создаётся временное "окно ожидания", в течение которого повторная попытка невозможна. При превышении определённого порога неудачных попыток входа аккаунт временно блокируется.
   Это затрудняет автоматический подбор пароля злоумышленником.
2. **Логирование активности.**
   Система ведёт отдельный лог-файл, в котором фиксируются успешные/неспешный попытки входа в аккаунт, блокировка и т.д

2.3.4 Обоснование выбора архитектурных решений и технологий

Основой проекта стали **Spring Boot** и **Spring Security**, так как темой курсовой работы является разработка безопасного веб-приложения на Java с демонстрацией механизмов авторизации и аутентификации. Spring Boot обеспечивает удобную структуру проекта, автоматическую конфигурацию и быструю интеграцию компонентов, а Spring Security — гибкую и расширяемую систему безопасности, подходящую для демонстрации современных механизмов защиты.

Почему использован JWT

В качестве механизма аутентификации выбран JWT (JSON Web Token), поскольку он позволяет реализовать stateless-подход, удобно интегрируется с REST API и позволяет хранить в себе информацию о пользователе и его ролях

Почему выбрана ролевая модель (RBAC)

Для разграничения прав доступа применена ролевая модель, где действия пользователя определяются назначенной ему ролью. Такой подход легко управляется, масштабируется и соответствует типичным требованиям к защите данных. Использована иерархия: более высокая роль включает в себя полномочия нижестоящих.

2.4 Особенности реализации и обобщение

В ходе проектирования и разработки особое внимание уделялось вопросам безопасности, модульности и логичности архитектуры. Использование Spring Boot и Spring Security позволило сосредоточиться на ключевых аспектах работы с пользователями и контролем доступа, не перегружая приложение лишней инфраструктурой.

Структура контроллеров отражает разграничение ответственности между ролями, что повышает читаемость и надёжность кода. Применение JWT-токенов и модели RBAC позволило реализовать гибкую и современную схему авторизации, хорошо подходящую для REST-приложений.

Проект наглядно демонстрирует, как с помощью современных Java-технологий можно построить надёжную и расширяемую архитектуру серверного приложения с акцентом на безопасность.

Глава 3. Описание реализации подсистемы

3.1 Стартовая конфигурация проекта

Проект инициализирован через *Spring Initializr*, где были выбраны следующие модули:

- Spring Web;
- Spring Security;
- Spring Data JPA;
- PostgreSQL Driver;
- Lombok;
- Validation API.

Для управления зависимостями используется Maven.

В файле *application.properties* заданы параметры подключения к базе данных, порт сервера, настройки JWT и безопасности:

```
spring.application.name=security-app
spring.datasource.url=jdbc:postgresql://localhost:5432/springSecurityApp
spring.datasource.username=
spring.datasource.password=
spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
logging.level.org.springframework.security=DEBUG
logging.level.org.springframework.web=DEBUG
```

Рис.8 application.properties file

3.2 Структура базы данных

В проекте реализованы две основные сущности: User и Role. Связь между пользователем и ролями — многие ко многим. Репозитории созданы на основе JpaRepository, обеспечивая удобную работу с базой данных без написания SQL-запросов.

	id [PK] bigint	password character varying (255)	username character varying (50)
1	1	$2a$10$42EA5jGYDVhzOyvc4oznUeMym49zkm75445LjAOmwBw80mFF...	admin1
2	2	$2a$10$KvGxts7sDbnjlubIM40ILuVJkI3YBk5rE7UMIfTYOy5r.FjHyaKbS	moderator1

Рис.9 Users Table

	id [PK] bigint	name character varying (255)
1	1	ROLE_ADMIN
2	2	ROLE_USER
3	3	ROLE_MODERATOR

Рис.10 Roles Table

	id [PK] bigint	role_id bigint	user_id bigint
1	1	1	1
2	2	2	1
3	3	3	1
4	4	2	2
5	5	3	2

Рис.11 User_Roles table

3.3 Реализация безопасности

В проекте безопасность реализована с помощью Spring Security и JWT-токенов, что позволяет обеспечить гибкую и расширяемую архитектуру контроля доступа. Основные задачи, решаемые в данном разделе:

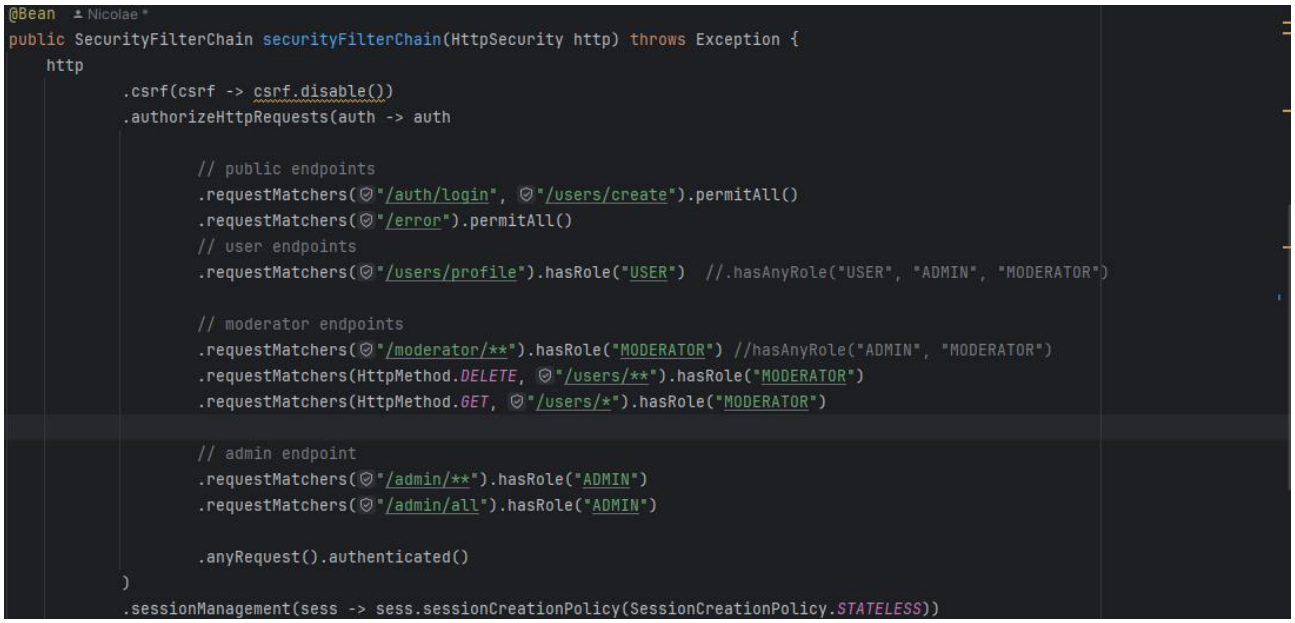
1) Фильтрация и проверка токена в каждом запросе.
2) Разграничение прав пользователей по ролям.
3) Отключение хранения сессий.
4) Централизованная конфигурация точек доступа.
5) Корректная обработка ошибок для ясного отображения ошибки на клиенте, удобства отладки кода и скрытия чувствительной информации от пользователя (для предотвращения утечки данных)

Конфигурация Spring Security

Настройка безопасности сосредоточена в классе *SecurityConfig*, где с помощью *SecurityFilterChain* определяются правила доступа и порядок фильтров.

- CSRF отключён, так как используется stateless-подход с JWT.
- Настройка SessionCreationPolicy.STATELESS отключает серверные сессии.
- Каждому набору URL соответствует определённая роль.
- Добавляется кастомный фильтр, который выполняется до стандартного фильтра авторизации.

Это позволяет чётко разграничивать доступ к эндпоинтам и не допустить попадания пользователя в «чужую» зону.



Puc.12 securityConfig.java: filterChain

JWT (JSON Web Token)

JWT — это стандарт, который используется для безопасной передачи информации между сторонами в виде JSON-объекта. Он широко применяется в веб-приложениях для реализации аутентификации и авторизации без хранения сессий на сервере**.**

Зачем используется JWT?

В контексте безопасных приложений JWT позволяет:

- Аутентифицировать пользователя один раз, а затем передавать его токен в заголовках последующих запросов
- Избежать необходимости хранить сессии на сервере (статусность)
- Передавать внутри токена полезную информацию (например, логин, роли)

Безопасность

- Подделать токен без знания секретного ключа невозможно (если подпись проверяется)
- Токен может быть подписан с использованием **HMAC SHA-256** или **RSA**
- JWT может содержать **время жизни (exp)**, чтобы предотвращать "вечную" авторизацию

Как работает в приложении

1) Пользователь отправляет логин и пароль на сервер
2) Если данные верны — генерируется JWT и возвращается клиенту
3) Клиент сохраняет токен и передаёт его в заголовке Authorization: Bearer ...
4) При каждом запросе фильтр JWT проверяет токен:
    - Валиден ли
    - Не просрочен ли
    - Какие роли и права в нём записаны
5) Если всё в порядке — запрос пропускается в контроллер

```
// Секретный ключ для подписи токена и время жизни
private static final String SECRET_KEY = "superSecretKeyForJWTsuperSecretKeyForJWT";  1 usage
private static final long EXPIRATION_TIME = 86400000; // (1 день в миллисекундах)  1 usage

private static final Key key = Keys.hmacShaKeyFor(SECRET_KEY.getBytes()); // Хэш ключа  2 usages

// Генерация токена
public static String generateToken(String username, List<String> roles) {  1 usage  Nicolae *
    return Jwts.builder()
            .setSubject(username)
            .claim( s: "roles", roles)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
            .signWith(key, SignatureAlgorithm.HS256)
            .compact();  // Сбор строки токена
}
```

Рис.13 Генерация JWT

JwtAuthenticationFilter

Одним из ключевых элементов безопасности в проекте является фильтр *JwtAuthenticationFilter*, реализующий обработку каждого входящего запроса. Его задача — перехватывать HTTP-запросы, извлекать из них JWT-токен, проверять его подлинность и в случае успеха — устанавливать информацию о пользователе в контекст безопасности Spring.

Реализация контроля доступа до попадания запроса в контроллер — это хорошая практика и важная мера защиты. Она позволяет:

- Предотвратить вызов даже пустых методов контроллера
- Исключить любые побочные эффекты или утечки
- Централизовать логику авторизации и не дублировать проверку в каждом методе

Фильтр работает следующим образом:

1) Получает заголовок Authorization из запроса.
2) Проверяет, что заголовок присутствует и начинается с префикса "Bearer “ (что соответствует стандарту передачи токена).
3) Извлекает сам JWT (всё, что после "Bearer ").

```
// Извлекаем заголовок Authorization
String authHeader = request.getHeader( s: "Authorization");
if (authHeader == null || !authHeader.startsWith("Bearer ")) {
    filterChain.doFilter(request, response);
    return;
}

// Получаем сам токен (убираем "Bearer ")
String token = authHeader.substring( beginIndex: 7);
System.out.println("🔍 JWT фильтр сработал. Токен: " + token);
```

Рис.14 JwtAuthenticationFilter realization, points 1-3

4) Проверяет, что пользователь ещё не аутентифицирован в текущем контексте (SecurityContext).
5) Загружает данные пользователя через UserDetailsService.

```
// Извлекаю username из токена
String username = jwtUtil.extractUsername(token);

// Проверяю, не аутентифицирован ли уже этот пользователь
if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

    // Загружаю пользователя из базы
    UserDetails userDetails = userDetailsService.loadUserByUsername(username);
```

Рис.15 JwtAuthenticationFilter realization, points 4-5

6) Проверяет валидность токена (срок действия, подпись, соответствие пользователю).

```
public boolean validateToken(String token, UserDetails userDetails) {  1 usage   Nicolae
    final String username = extractUsername(token);
    return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
}
```

Рис.16 Метод валидации токена

7) Если всё успешно — создаёт объект Authentication и помещает его в SecurityContext.

```
UsernamePasswordAuthenticationToken authToken =
        new UsernamePasswordAuthenticationToken(userDetails,  credentials: null, authorities);

// Устанавливаю аутентификацию в SecurityContext
authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
SecurityContextHolder.getContext().setAuthentication(authToken);
```

Рис.17 Создание и помещение объекта аутентификации в securityContext

3.4 Регистрация и аутентификация

Эти функции обрабатываются в проекте через специальный контроллер — AuthController, отвечающий за этап взаимодействия пользователя с системой.

Регистрация пользователя

Регистрация реализована через метод *register*, который доступен всем. На вход контроллер принимает JSON-объект с данными нового пользователя — например, username и password.

Прежде чем сохранить пользователя в базу данных, выполняются следующие проверки и действия:

1) Проверяется уникальность username.
2) Проверяется корректность пароля (валидация).
3) Пароль не хранится в открытом виде, а хешируется с помощью BCryptPasswordEncoder.
4) Пользователю назначается роль по умолчанию (USER).
5) Данные сохраняются через UserRepository.

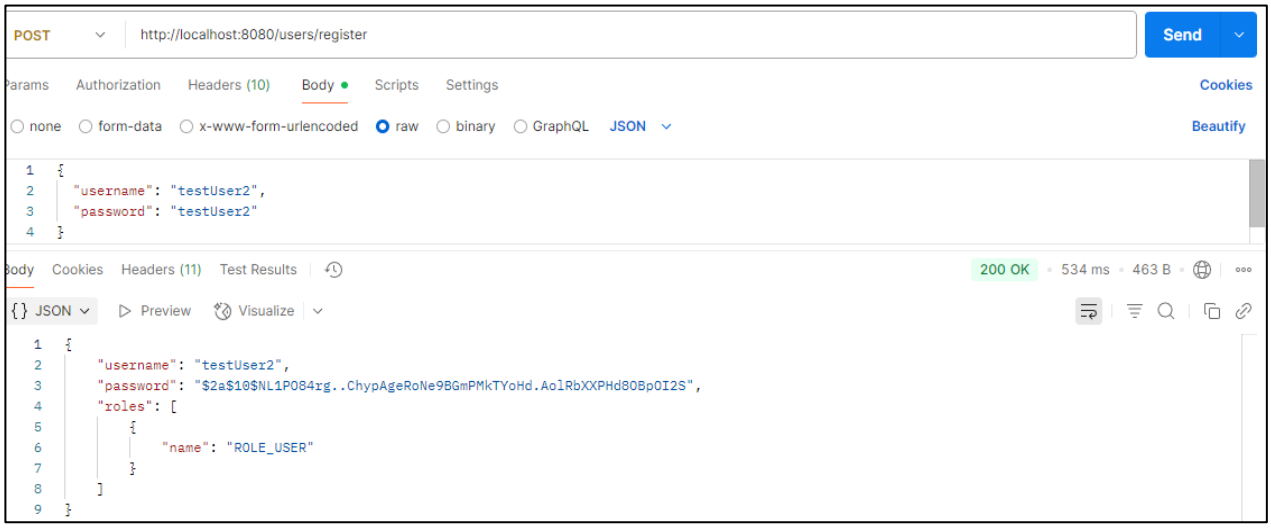


Рис.18 Пользователь был добавлен в бд с ролью «USER».

Аутентификация (вход в систему)

Аутентификация реализуется в методе *login*, который также открыт для всех пользователей. На вход принимаются логин и пароль. Данные пользователя проверяются и в случае успеха генерируется JWT-токен, в который включается информация о пользователе и его роли, после чего токен возвращается клиенту, который сохраняет его и использует при последующих запросах.

Рис.19 Пример успешной аутентификации

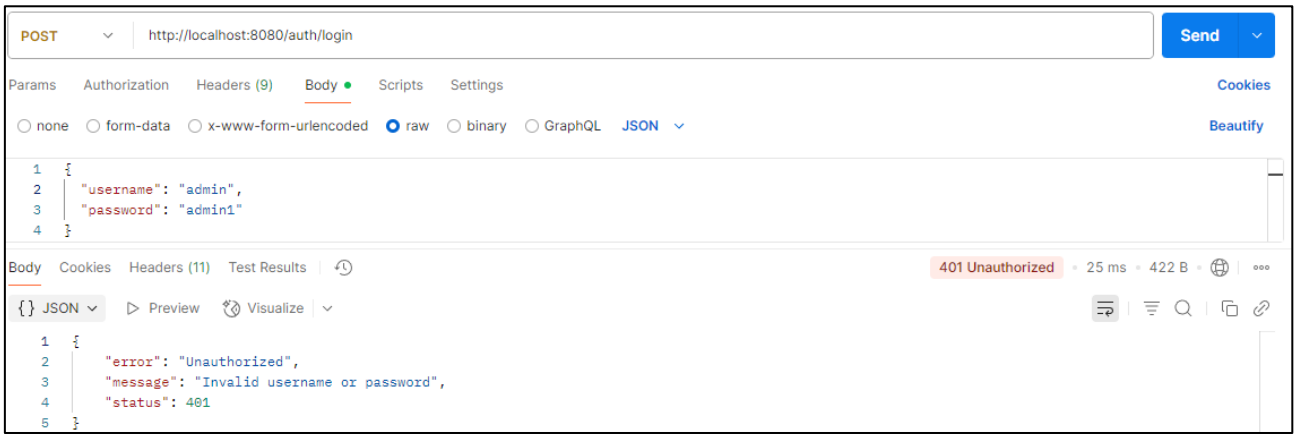


Рис.20 Пример неуспешной аутентификации

Преимущества такой реализации

- Отсутствие сессий на сервере (stateless).
- Чёткое разделение ролей уже на этапе генерации токена.
- Высокая гибкость и простота расширения (например, добавление refresh токенов, 2FA, email-подтверждения)

3.5 Роли и разграничение доступа

В рамках проекта реализованы три уровня ролей:

1) USER — базовый уровень, доступ к личному профилю;
2) MODERATOR — доступ к управлению контентом, пользователями, расширенные функции;
3) ADMIN — полный контроль над системой и пользователями.

Безопасность маршрутов

Проверка прав доступа

После прохождения JwtAuthenticationFilter (описанного ранее), информация о пользователе устанавливается в SecurityContext. Далее, при попадании запроса в фильтр авторизации, Spring проверяет какие роли есть у пользователя и соответствие ролей пользователя требуемым.

- Если хотя бы один пункт не выполняется — доступ блокируется.
- Токен отсутствует/повреждён/ просрочен => **401 Unauthorized**
- Роль пользователя недостаточна => **403 Forbidden**

Заголовок Authorization

Все защищённые запросы должны содержать заголовок *Authorization*
Фильтр JwtAuthenticationFilter извлекает этот заголовок, валидирует токен и только после этого пользователь считается авторизованным. Если заголовок отсутствует — пользователь остаётся неаутентифицированным, и система применяет правила для анонимного пользователя.

Рис.21 Пример наличия токена для запроса на защищенный эндпоинт

3.6 Прочие доработки: блокировка, логирование и обработка ошибок

В дополнение к базовому механизму аутентификации и авторизации в проекте были реализованы функции, усиливающие защищённость приложения и улучшающие его поддержку. К ним относятся:

- Защита от подбора паролей (brute force).
- Временная блокировка пользователя.
- Логирование действий.
- Обработка исключений с информативными ответами.

Блокировка при множественных неудачных входах

Одной из распространённых атак на веб-приложения является подбор пароля путём автоматических повторных попыток входа. Чтобы минимизировать риск такого сценария, в системе реализован механизм временной блокировки аккаунта**.**

Принцип работы:

1) При каждой неудачной попытке входа счётчик ошибок увеличивается, при введении правильного пароля – сбрасывается.
2) До выполнения следующего запроса происходит небольшая пауза (пару секунд), чтоб еще снизить эффективность broot-force атак.
3) Если количество ошибок превышает заданный порог (например, 5), пользователь блокируется.
4) Блокировка может настраиваться по потребности конкретной системы.
5) После блокировки система не даёт войти даже при вводе правильного пароля.

```
private static final int MAX_ATTEMPTS = 5;  1 usage
private static final long LOCKOUT_DURATION = 1 * 60 * 1000; // 3 минуты  2 usages
private static final Logger logger = LoggerFactory.getLogger(AuthController.class);  4 usages
```

Рис.22 Установка кол-ва попыток для входа и длительности блокировки

Логирование действий

Для удобства анализа и обеспечения прозрачности работы системы реализовано логирование событий, связанных с безопасностью:

- Успешные и неуспешные попытки входа
- Блокировки пользователей
- Ошибки авторизации и доступа

Для этого используется встроенный механизм логирования Spring Boot и библиотека Logback. Отдельный лог-файл (security.log или аналогичный) позволяет отслеживать активность без перегрузки основной консоли.

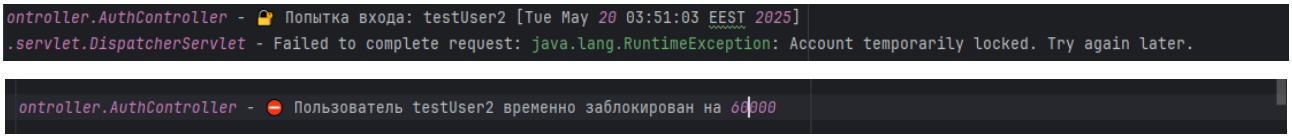


Рис. 23 Блокировка пользователя

Обработка ошибок

Чтобы система корректно реагировала на исключения, реализован обработчик обрабатывающий ошибки безопасности (AccessDeniedException, AuthenticationException), и глобальный обработчик, перехватывающий прочие ошибки и исключения.

Отлавливаемые ошибки:

1) **401 Unauthorized** — ошибка аутентификации. Возникает, если отсутствует JWT-токен, он недействителен или пользователь не прошёл проверку.
2) **403 Forbidden** — отказ в доступе. Возникает, если пользователь аутентифицирован, но не обладает необходимой ролью или правами.
3) **400 Bad Request** — ошибка валидации. Возникает при неверном формате данных, отсутствии обязательных полей и других нарушениях правил DTO.
4) **404 Not Found** — несуществующий маршрут или ресурс.
5) **405 Not Found** — несуществующий маршрут или ресурс.
6) **409 Conflict** — конфликт при регистрации. Например, если пользователь с таким email уже существует.
7) **500 Internal Server Error** — обработка непредвиденных внутренних ошибок, исключений и сбоев логики.

```java
// 404 Not Found
@ExceptionHandler(NoHandlerFoundException.class)  new *
public ResponseEntity<Map<String, Object>> handleNotFound(NoHandlerFoundException ex) {
    Map<String, Object> error = new HashMap<>();
    error.put("status", 404);
    error.put("error", "Not Found");
    error.put("message", "The requested resource was not found");
    error.put("path", ex.getRequestURL());
    return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
}

// 422 Unprocessable Entity (ошибка валидации)
@ExceptionHandler(MethodArgumentNotValidException.class)  new *
public ResponseEntity<Map<String, Object>> handleValidationException(MethodArgumentNotValidExceptio
    String errorMessage = ex.getBindingResult().getFieldErrors().stream()  Stream<FieldError>
            .map(err -> err.getField() + ": " + err.getDefaultMessage())  Stream<String>
            .findFirst()  Optional<String>
            .orElse( other: "Invalid request");

    Map<String, Object> error = new HashMap<>();
    error.put("status", 422);
    error.put("error", "Validation Failed");
    error.put("message", errorMessage);
    return new ResponseEntity<>(error, HttpStatus.UNPROCESSABLE_ENTITY);
}
```

Рис.24 Пример обработки ошибок 404 & 422

Каждая ошибка обрабатывается соответствующим блоком в глобальном контроллере (@ControllerAdvice) и возвращается клиенту в виде JSON-объекта с полями:

```
{
  "timestamp": "xxxx-xx-xx T xx:xx:xx",
  "status": код  ошибки,
  "error": "Причина ошибки",
  "message": "Сообщение с описанием ошибки для клиента"
}
```

Рис.25 JSON-шаблон с возвращаемый обработчиками ошибок

Это позволяет клиенту грамотно обрабатывать ошибки и отображать соответствующие уведомления пользователю.

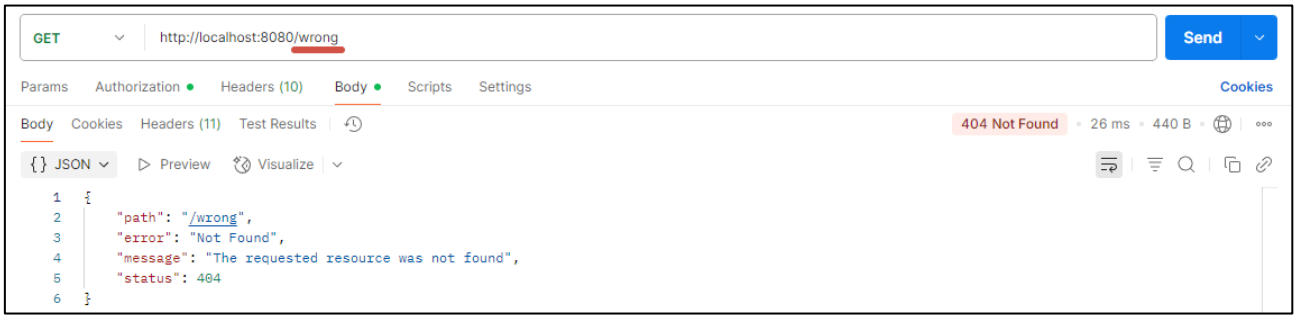


Рис.26 Пример запроса на несуществующий ресурс: 404 Not Found Error

3.7 Тестирование и демонстрация работы

Для проверки и демонстрации корректности работы реализованных функций безопасности, проведена серия тестов, примеры которых показаны ниже.

Проверка доступа по ролям

JWT-токен добавляется в заголовок каждого запроса:

1) Доступ к /users/profile с ролью USER (требует токен с USER-ролью)

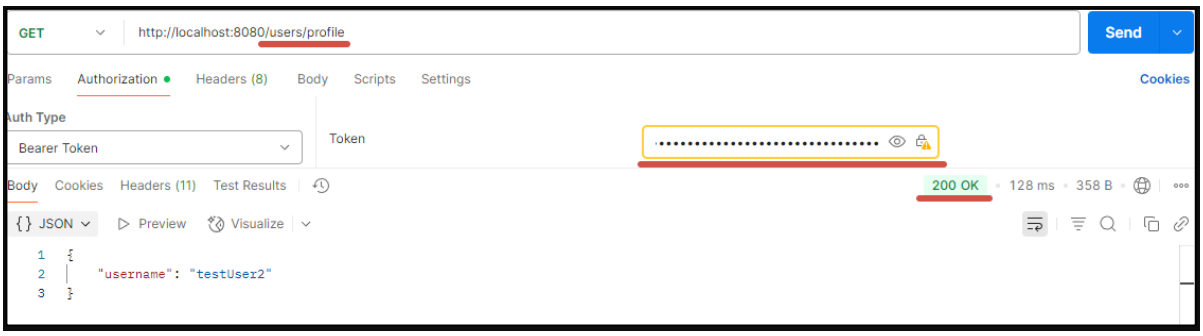


Рис.27 Успешный запрос на эндпоинт для роли USER

2) Доступ к /admin/allUsers с ролью USER (требует токен с ADMIN-ролью)

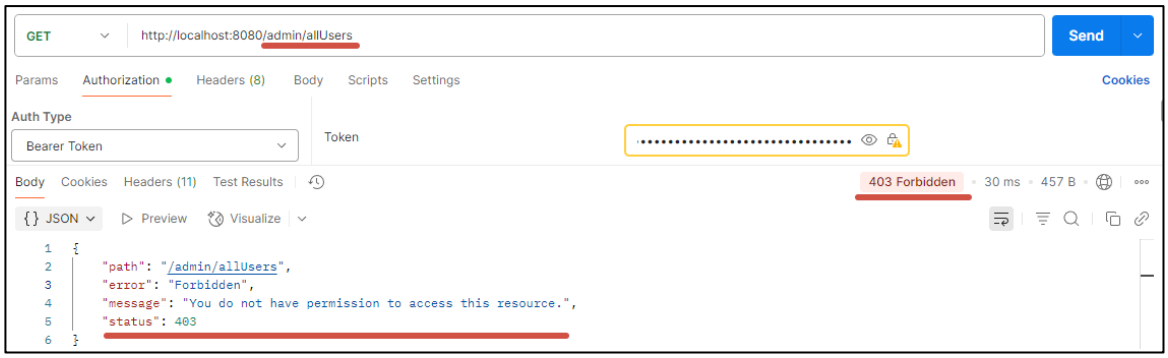


Рис.28 Отказ в доступе, при недостатке прав

3) Доступ на этот же эндпоинт /admin/allUsers с ролью «ADMIN»

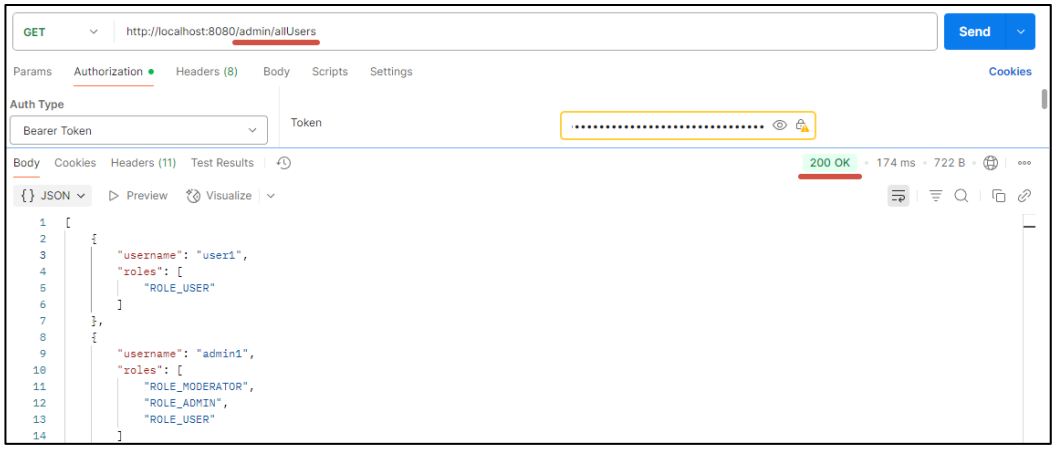


Рис.29 Успешный зарпос с ролью ADMIN на /admin/allUsers

Назначение ролей пользователям

Для назначения роли администратором реализован эндпоинт (/admin/setRole), с передачей ID пользователя и новой роли.

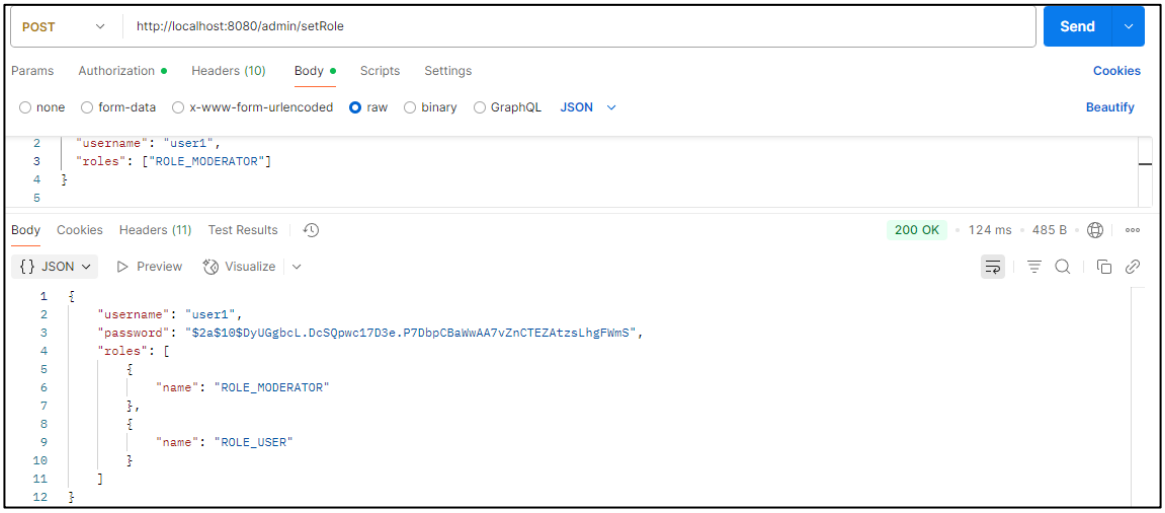


Рис.30 Назначение ролей администратором

Проверка блокировки аккаунта:

После 5 неудачных попыток входа в аккаунт, пользователь блокируется:

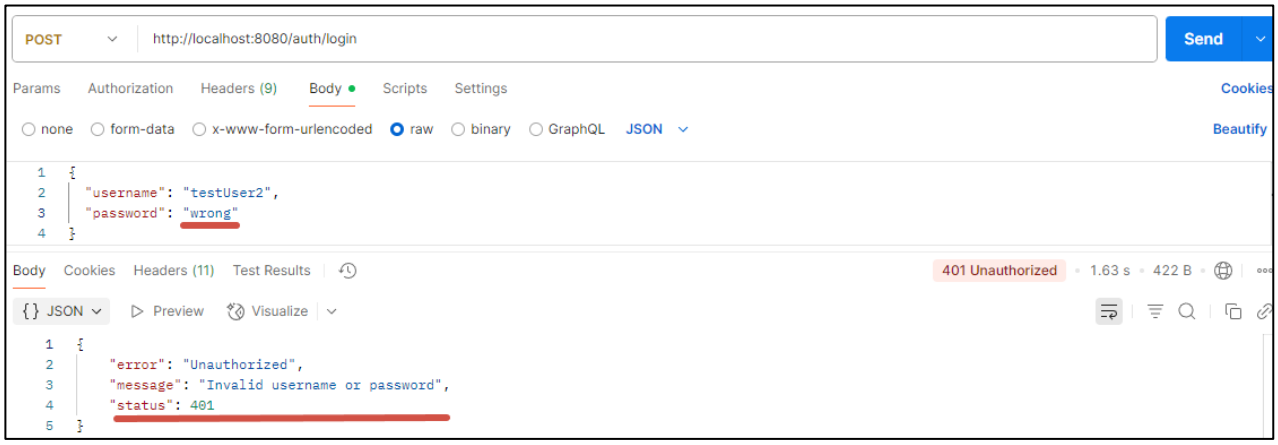


Рис.31 Пользователь не прошел аутентификацию

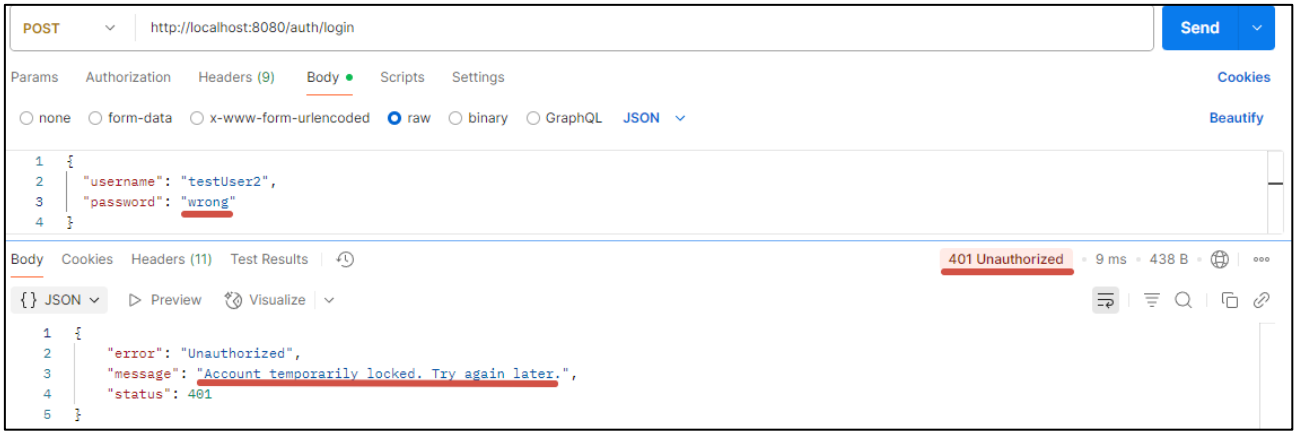


Рис.32 блокировка после 5 неудачных попыток входа

Проверка валидации данных при регистрации

1. Попытка регистрации пользователя с уже существующим именем:

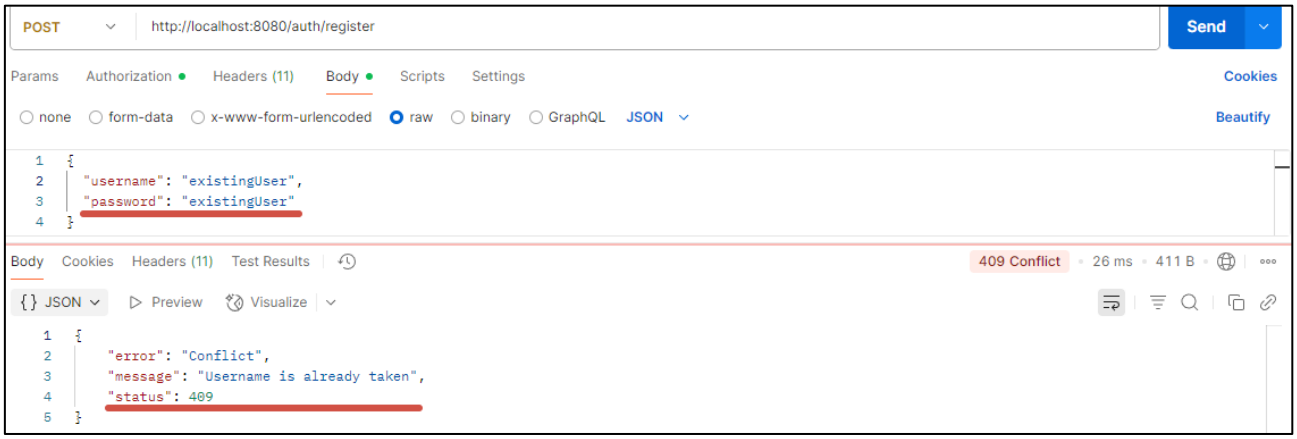


Рис.33 409 Conflict error

2. Попытка регистрации с данными, не прошедшими валидацию:

Рис.34 422 Unprocessable Entity Error

Результаты тестирования

В результате ручного тестирования:

- Корректно работает разграничение доступа по ролям
- Успешно формируется и валидируется JWT
- Срабатывает блокировка аккаунта при превышении лимита
- Маршруты, не соответствующие роли, недоступны
- При отсутствии токена возвращается 401 Unauthorized
- При некорректной роли — 403 Forbidden

Таким образом тестирование подтвердило работоспособность всех компонентов системы безопасности.

Заключение

В ходе выполнения курсовой работы на тему **«Разработка защищённого веб-приложения с использованием Spring Security»** были достигнуты следующие результаты:

1) **Были изучены**:
   - современные подходы к обеспечению безопасности веб-приложений.
   - архитектурные принципы построения backend-систем (SRP, REST, Stateless).
   - особенности работы фреймворков Spring Boot и Spring Security.

2) **Был разработан**:
   - Проект серверного приложения с поддержкой регистрации, входа в систему, распределения ролей и защиты маршрутов
   - Централизованная конфигурация безопасности с кастомным фильтром обработки JWT.

3) **Были реализованы и отлажены**:
   - Механизмы регистрации и аутентификации с хешированием пароля.
   - Фильтрация и проверка JWT-токена на каждом запросе.
   - Разграничение доступа к маршрутам по ролям с уровнем контроля на уровне фильтров.
   - Защита от атак типа brute force через временную блокировку аккаунта;
   - Логирование активности и ошибок авторизации.
   - Глобальная обработка исключений с возвратом структурированных ответов.

4) **Результаты тестирования** подтвердили:
   - Корректную работу всех механизмов безопасности;
   - Соответствие поведения заявленным ролям и доступам;
   - Устойчивость к некорректным действиям и попыткам несанкционированного доступа.

Таким образом, поставленные в начале работы задачи были успешно решены, а выбранный технологический стек позволил реализовать все необходимые функции в рамках заявленной архитектуры.

Библиография

1) **Spring Security Reference Documentation** - *Официальная документация по Spring Security*
   URL: https://docs.spring.io/spring-security/reference/

2) **JWT.io — JSON Web Tokens Introduction** - *Подробное описание структуры, формата и применения JWT*
   URL: https://jwt.io/introduction

3) **Baeldung: Guide to Spring Security** - *Обзор основных аспектов настройки Spring Security*
   URL: https://www.baeldung.com/security-spring

4) **Baeldung: JWT with Spring Security** - *Реализация JWT-аутентификации в Spring Boot*
   URL: https://www.baeldung.com/spring-security-oauth-jwt

5) **Spring Boot Reference Documentation** - *Официальная документация Spring Boot*
   URL: https://docs.spring.io/spring-boot/docs/current/reference/html/

6) **OWASP: Authentication Cheat Sheet** - *Рекомендации по безопасной аутентификации*
   URL: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

7) **OpenAI ChatGPT** — AI language model, используемая для генерации и проверки кода, пояснений, а также для стилистической и логической правки текста.
   URL: https://chat.openai.com/