**Лекция 1:**

Стек технологий:

PostgreSQL (СУБД);

Backend:

Spring Boot 2.7.3+ (Backend java framework);

Maven (Система сборки проектов);

Spring Data JPA;

String MVC;

String Security (реализуется на jwt – JSON Web Tokens - jwt.io);

Java 8+;

Hibernate;

Frontend:

Angular 9+;

TypeScript;

Приложение:

Микросервисная архитектура, поддерживает stateless подход, на клиенте SPA (Single Page Appliaction) и клиент работает с сервером через REST API ендпойнты.

//Комментарий:

**Stateless** - состояние не хранится на сервере. Т.е. в каждом новом запросе мы передаём свой логин/пароль (если в приложении есть авторизация), а также данные для запроса.

**Statefull** - на сервере открывается сессия. Благодаря этому нам не надо постоянно отправлять на сервер логин/пароль. И не надо отсылать какие-то дополнительные данные (например выбранную локализацию), они хранятся в сессии.

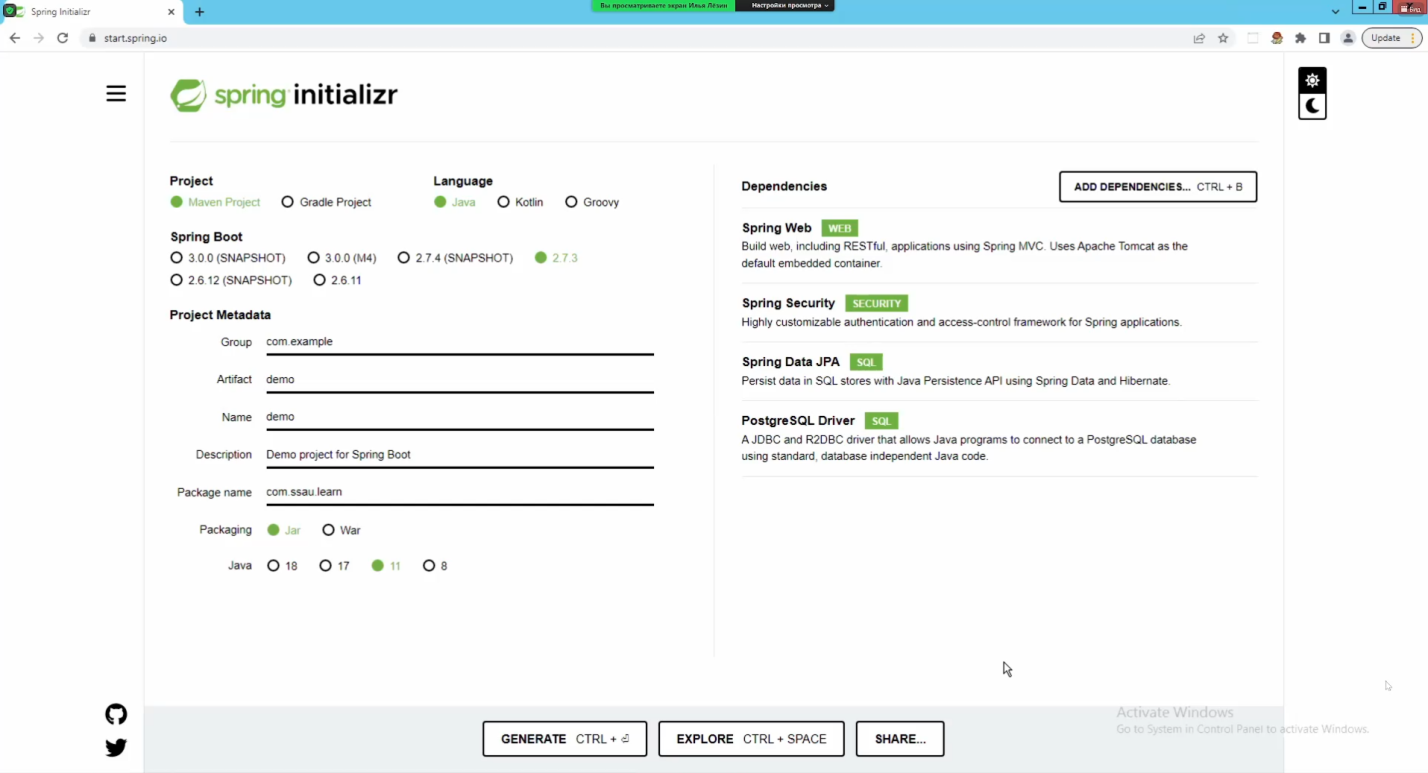
Подробнее: [веб сервер - Что такое Stateless и Statefull? - Stack Overflow на русском](https://ru.stackoverflow.com/questions/1242990/%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-stateless-%D0%B8-statefull?ysclid=lcxrp6mm8266643027)

Юнит тесты + интеграционные тесты.

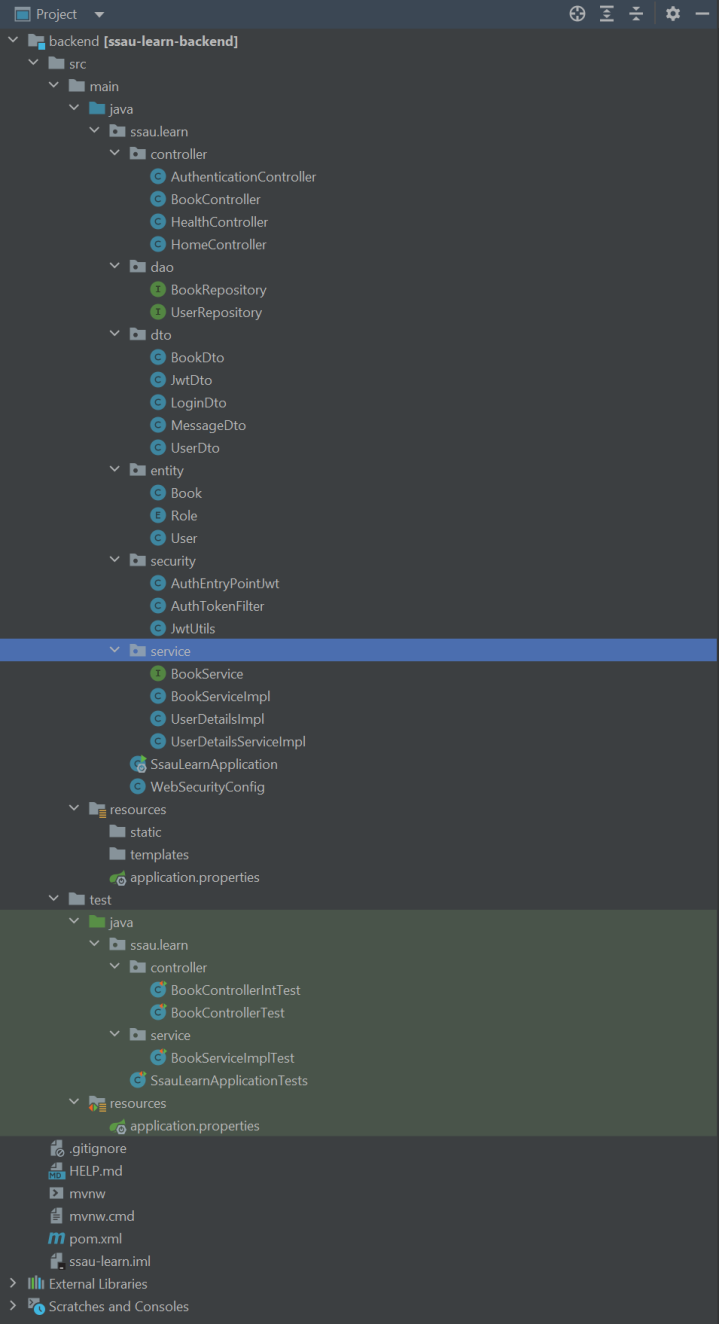
jUnit + Mockito

Ролевая модель пользователей (User + Admin)

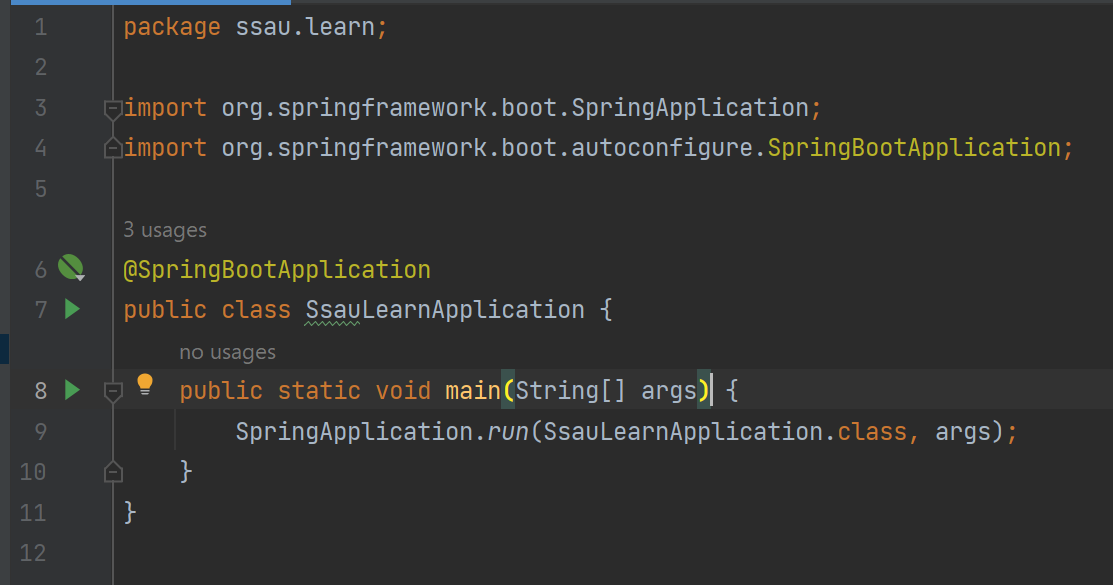
Генерация нового проекта с готовыми зависимостями в pom.xml для maven:



Общая структура:



Стандартный класс запуска spring boot приложения:



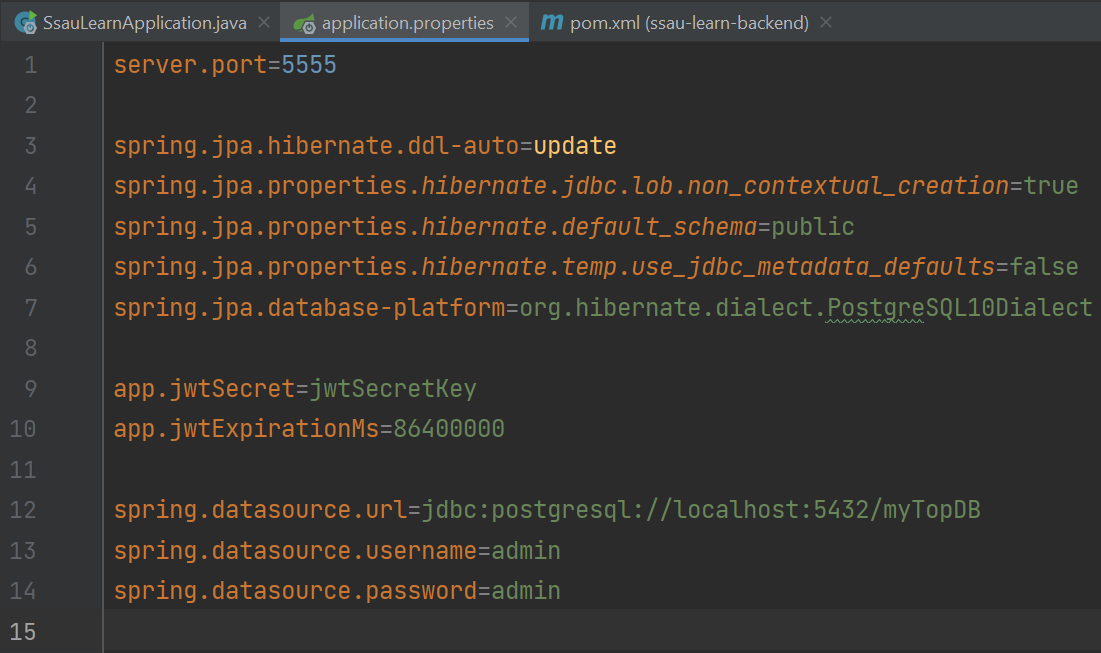
На строчках 3 – 4 указаны импорты аннотации SpringBootApplication и класса SpringApplication.

Класс помечается аннотацией SpringBootApplication.

Содержит метод main.

В методе main вызывается статический метод run класса SpringApplication, куда передаётся описатель текущего запускаемого класса и аргументы командной строки.

Файл application.proterties:



Строка 1 – настройка порта сервера.

Строка 3 – автоматическая генерация и перегенерация таблиц сущностей в СУБД при использовании Hibernate (используется автоматически при использовании Spring Data JPA)

Строка 4 – настройка для того чтобы Spring Data не тормозил при запуске.

Строка 5 – указание схемы с БД, используемой по умолчанию.

Строка 7 – указание диалекта, используемого для генерации SQL скриптов.

Строка 9 – параметр, задающий секрет, используемый для генерации JWT токенов.

Строка 10 – время жизни генерируемых токенов.

Строка 12 – указание url используемой СУБД. Состоит из указания протокола – jdbc, указания СУБД – postgresql, Указания IP или Доменного имени сервера – localhost:5432, указание пути к БД – myTopDB.

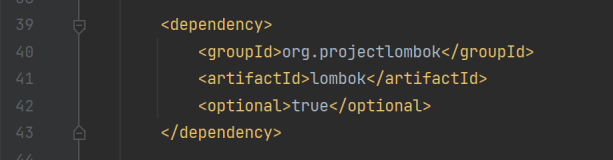
Строки 13 – 14 – имя пользователя и пароль для доступа к СУБД, соответственно.

Сущностные классы.

Класс User:



Строки 3 – 4 – аннотации Lombok. Lombok – это библиотека, содержащая аннотации, позволяющие генерировать часть java кода в классы, над которыми указана аннотация lombok. Библиотека подключается в файле pom.xml:



Строка 6 – импорт для Entity.

Строка 8 – аннотация, указывающая, что данный класс User является сущностью.

Строка 9 – аннотация Table – позволяет указать дополнительные параметры для таблицы сущности, генерируемой Hibernate, например имя таблицы или ограничения.

Строки 11 – 12 – указание ограничений на столбцы генерируемой таблицы сущности.

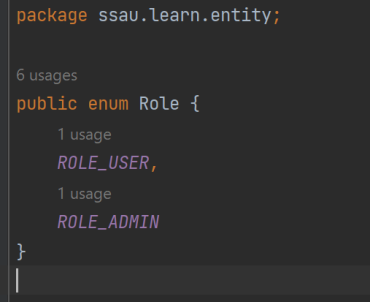
Строка 14 – аннотация Data и Lombok – позволяет автоматически генерировать методы get и set для всех приватных полей класса User и так же переопределяет некоторые другие методы:

Строка 15 – аннотация генерирует пустой конструктор без параметров по умолчанию. Конструктор нужен для сериализации и десериализации класса сущности.

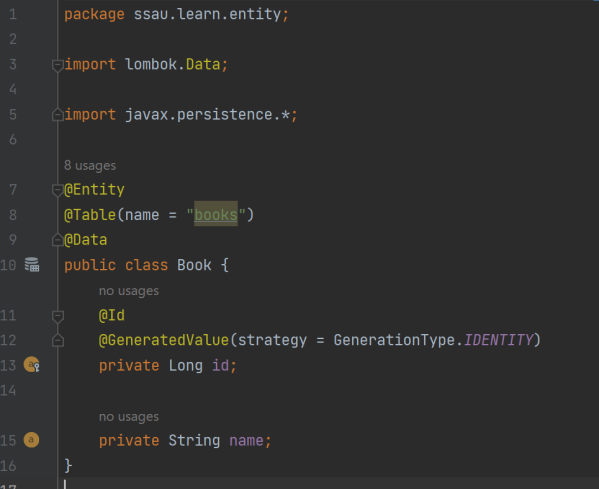
Строка 17 – аннотация Id – указывает поле, которое будет служить идентификатором сущности.

Строка 18 – аннотация указывающая на то, что значение для поля будет генерироваться СУБД автоматически. IDENTITY – автоинкрементное поле.

Класс (enum) с ролями пользователей:

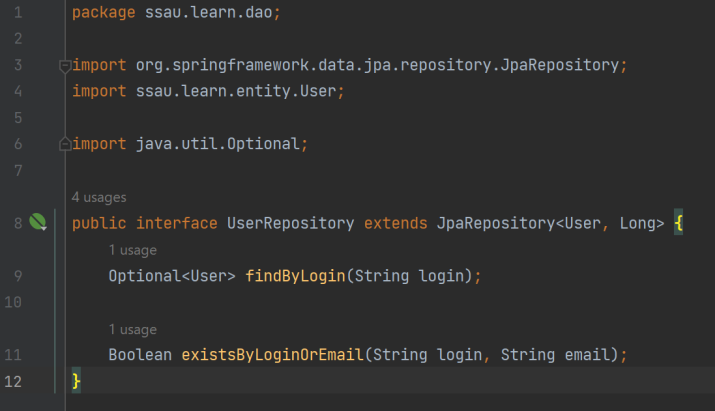
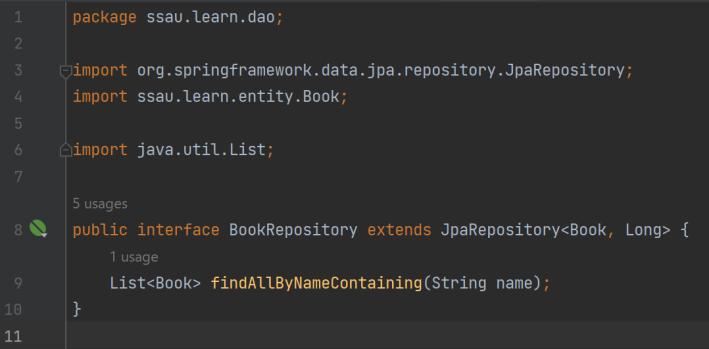


Класс Book (аналогичен классу User):



Spring Data JPA – это подфреймворк внутри spring boot, отвечающий за работу с данными. В качестве движка использует Hibernate.

Для каждого сущностного класса были созданы интерфейсы репозиториев:



Во время запуска spring boot автоматически находит все эти интерфейсы и генерирует для них байткод автоматически. Т.е. не нужно писать их реализацию.

Чтобы это всё работало нужно расширить Generic интерфейс JpaRepository с указанием типов, как в примере. При этом объекты, реализующие эти интерфейсы, можно импортировать в различные классы, как БИНЫ spring. Эти классы будут иметь стандартный набор методов для работы с данными, унаследованный от интерфейса JpaRepository. Внутри кастомного интерфейса можно доопределить недостающие методы по специальному шаблону, если стандартных методов недостаточно.

JPA репозиторий обеспечивает работу с таблицами БД, позволяя отображать записи из СУБД в сущности и наоборот.

**Лекция 2:**

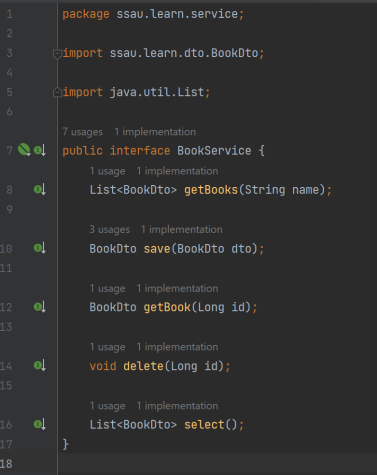
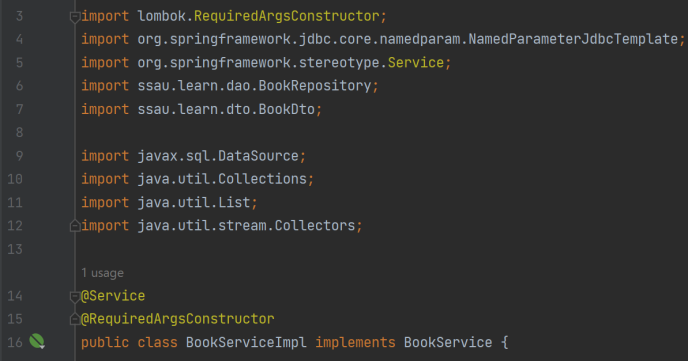
Сервисы.

Сервисы обычно делятся на 2 части – интерфейс сервиса и его имплементация.

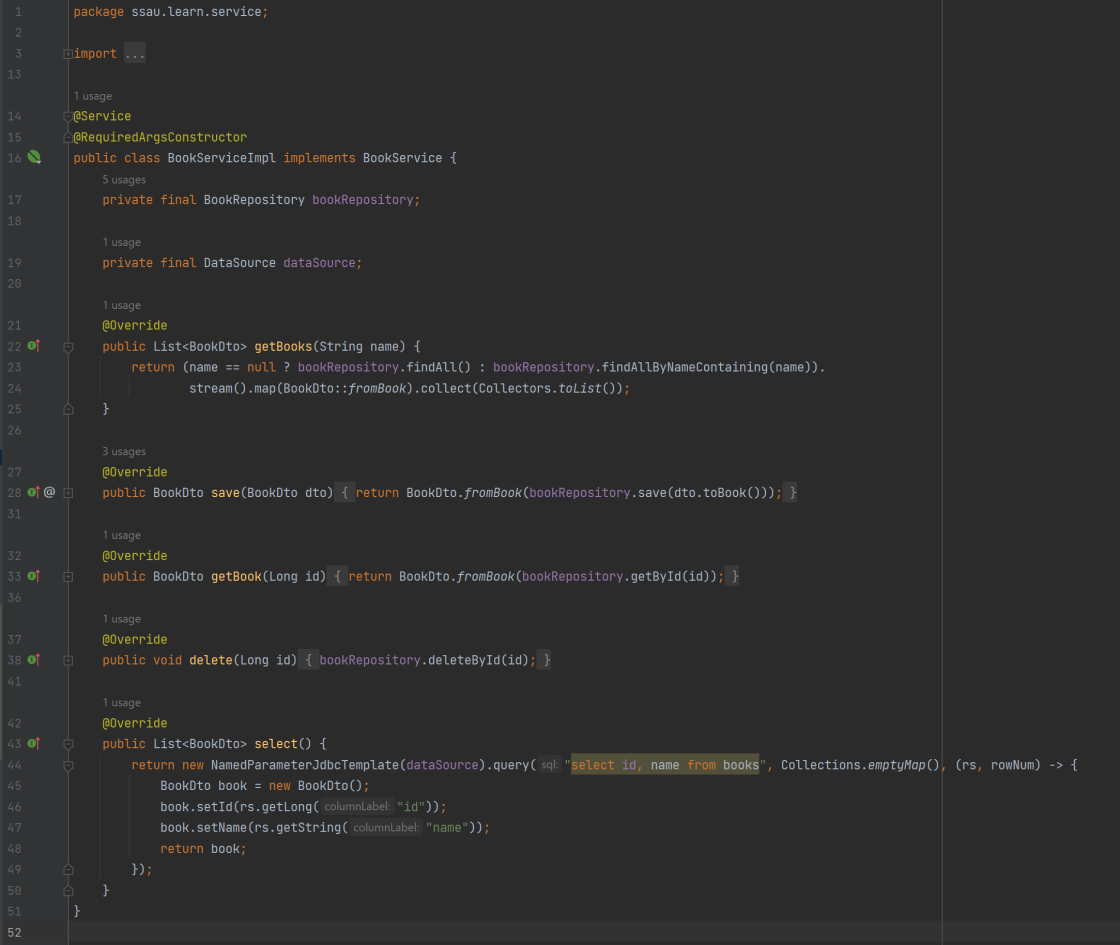
Сервисы описывают слой бизнес логики приложения. Они нужны для того чтобы логика работы с данными не отличалась в зависимости от того – какой интерфейс обмена данными был использован при обращении к серверу.

Пример сервиса для работы с книгами:

Интерфейсная часть:

  (Импорты реализации)

Реализация:



Строка 14 – аннотация, указывающая спрингу, что необходимо создать БИН этого класса. @Service – аналог аннотации @Component, но указывается над сервисами.

Строка 15 – RequiredArgsConstructor – аннотация создающая конструктор для всех полей класса, требующих инициализацию при создании объекта класса. В примере инициализируются все поля с модификатором final. Спринг подставит значения автоматически при создании объекта сервиса и сохранении его в контексте приложения.

Далее идёт реализация методов с использованием проинициализированных полей. Напрямую используется репозиторий, а так же поле dataSource для создания JdbcTemplate, который позволяет выполнять SQL запрос напрямую по заданному шаблону и определять метод обработки ответа от СУБД.

В качестве типов возвращаемых значений в методах сервиса указан специальный объект – BookDto (Data Transfer Object). Он нужен для того чтобы его можно было передать, в качестве ответа, пользователю, поскольку напрямую передавать сущностные классы (Entity) в ответ пользователю – считается плохой практикой.

Применение DTO позволяет отделить представление данных, с которыми работает пользователь, от внутреннего представления данных внутри приложения и внутри СУБД.

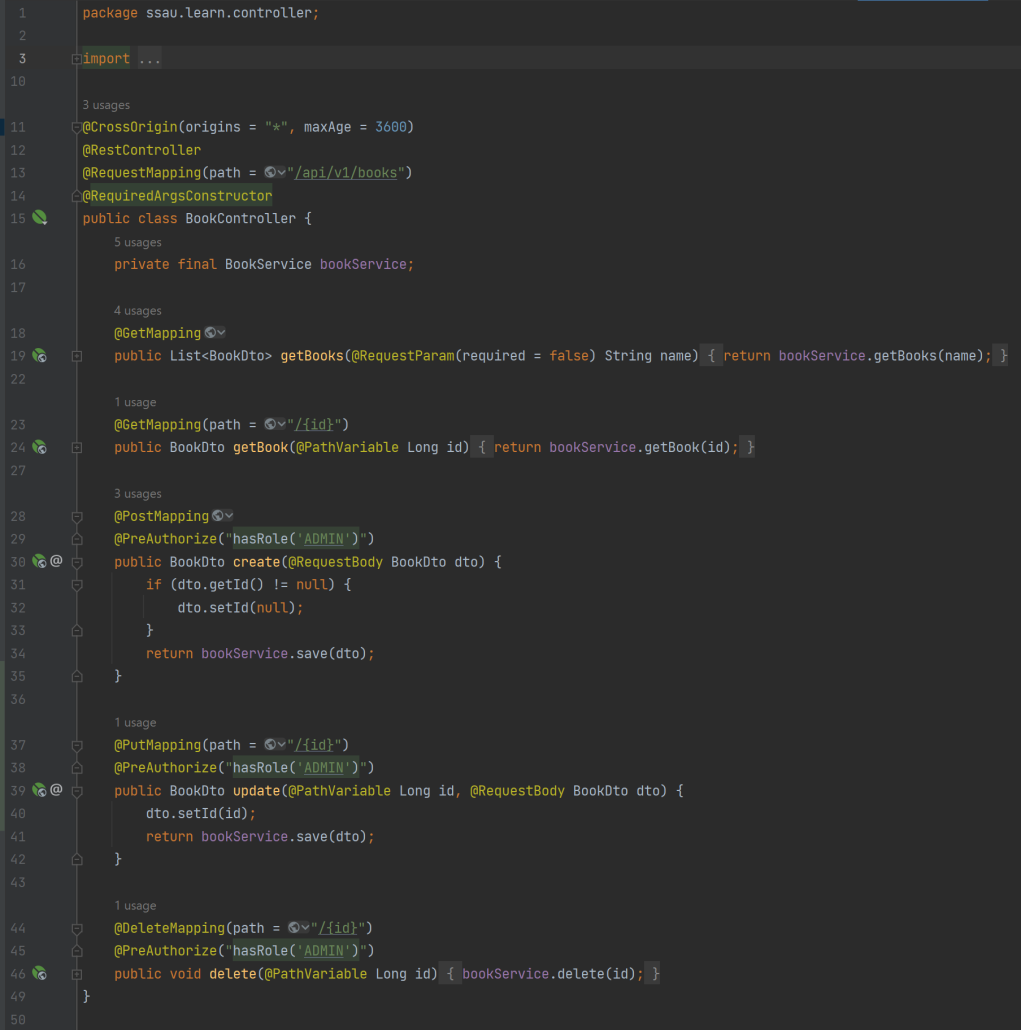
Т.о. контроллеры работают с объектами DTO, а сервисы работают непосредственно с данными в базе данных, выполняя преобразования DTO → Entity и Entity → DTO.

Контроллеры.

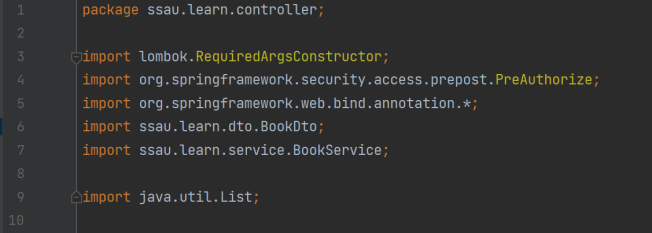
Контроллеры нужны для описания методов обработки запросов, приходящих снаружи.

В нашем примере используются REST контроллеры, обрабатывающие HTTP запросы.

BookController:



Импорты:



Строка 11 – нужна для настройки доступа внешнего HTTP приложения. Разрешает доступ любому HTTP клиенту к эндпойнтам контроллера.

Строка 12 – указывает спрингу что мы хотим создать контроллер с типом взаимодействия rest.

Строка 13 – RequestMapping – описывает общий путь для всех методов обработки http запросов, объявленных в классе контроллера.

Строка 14 – аннотация Lombok. Аналогично предыдущим примерам – генерирует конструктор для класса контроллера, в котором будет проинициализировано поле bookService созданным ранее объектом, взятым из контекста спринга.

Строка 18 – аннотация @GetMapping говорит что метод, над которым она стоит, занимается обработкой http запросов по общему пути, т.к. не указан параметр path, типа GET.

Строка 19 – метод указывает на то, что запрос может содержать необязательный параметр name (@RequestParam). Сам метод getBooks возвращает все книги из БД, при использовании параметра name только те книги, в названии которых содержится строка name.

Строки 23 – 24 – аналогичны предыдущим с той разницей что путь запроса дополняется обязательным параметром id, который используется в методе для обработки запроса. В методе помечен как @PathVariable – т.е. переменная пути запроса.

Метод getBook возвращает только одну книгу, по указанному id.

Строка 28 – @PostMapping – определяет обработчик http метода post по общему пути.

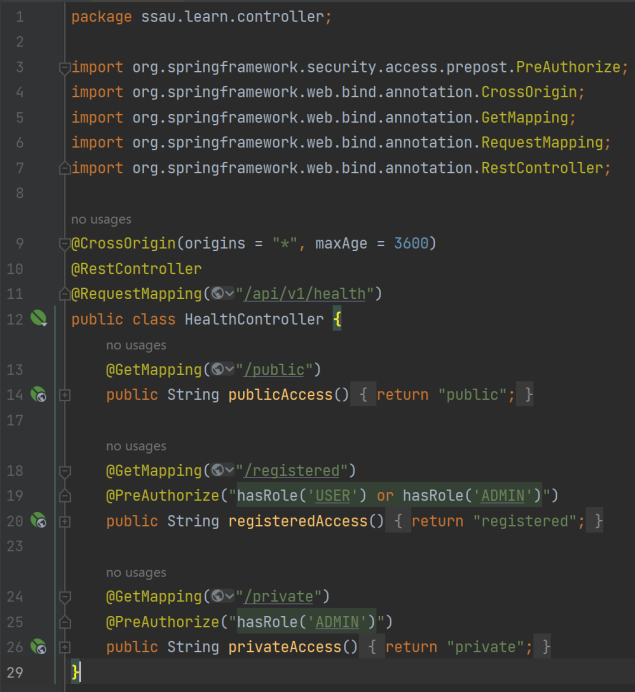
Строка 29 – @PreAuthorize("hasRole('ADMIN')") – аннотация говорит о том, что пользователь, который использует этот метод должен быть авторизован и иметь роль администратора.

Строка 30 – метод create – создаёт новую книгу в базе данных. Принимает в кацестве параметра @RequestBody – это тело http post запроса. Оно автоматически преобразуется в объект BookDto при помощи маппера объектов – библиотеки Jackson (используется спрингом по умолчанию).

Все остальные методы по аналогии.

HealthController:

Более простой пример контроллера. Он показывает статус приложения и уровень доступа пользователя к приложению.



Аналогично предыдущему классу задаёт общий путь для всех методов – /api/v1/health/…

Содержит только GET методы.

Метод publicAccess() возвращает строку “public” всем, кто обращается к этому методу, если приложение работает.

Метод registeredAccess() возвращает строку “ registered” только зарегистрированным пользователям с ролями USER или ADMIN.

Метод privateAccess() возвращает строку “private” только зарегистрированным пользователям с ролью ADMIN.

**Лекция 3:**

Используем jwt аутентификацию для stateless приложения. Это позволяет нам не хранить информацию о пользователе в сессии, а получать её из токена, каждый раз, когда пользователь делает авторизованный запрос на сервер.

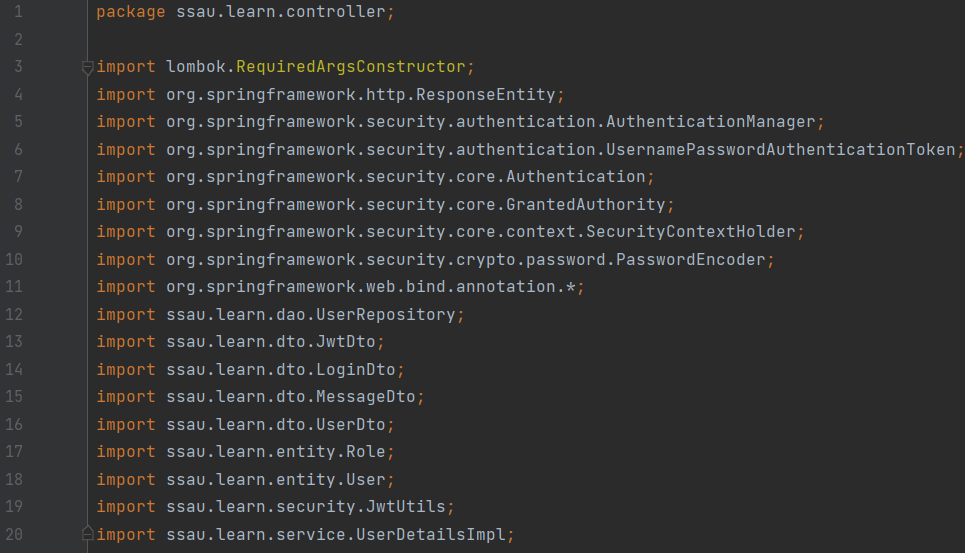
Все запросы, требующие информацию о пользователе, выполняются с добавлением заголовка Authorization, со значением jwt.

Так же в токене или на сервере хранится информация о ролях пользователя, для которого был сгенерирован токен. Пользователь имеющий определённый набор ролей может получить доступ только к тем эндпоинтам, которые определены для него набором ролей. (См. пример с контроллером HealthController, аннотация PreAuthorize)

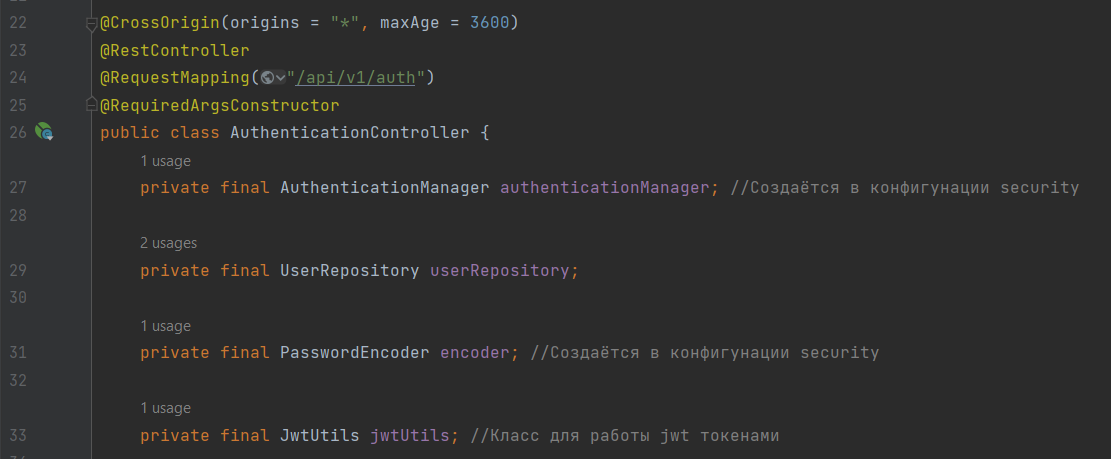
Далее будут куски кода с комментариями и иногда с пояснениями.

AuthenticationController – контроллер логина и регистрации пользователей:

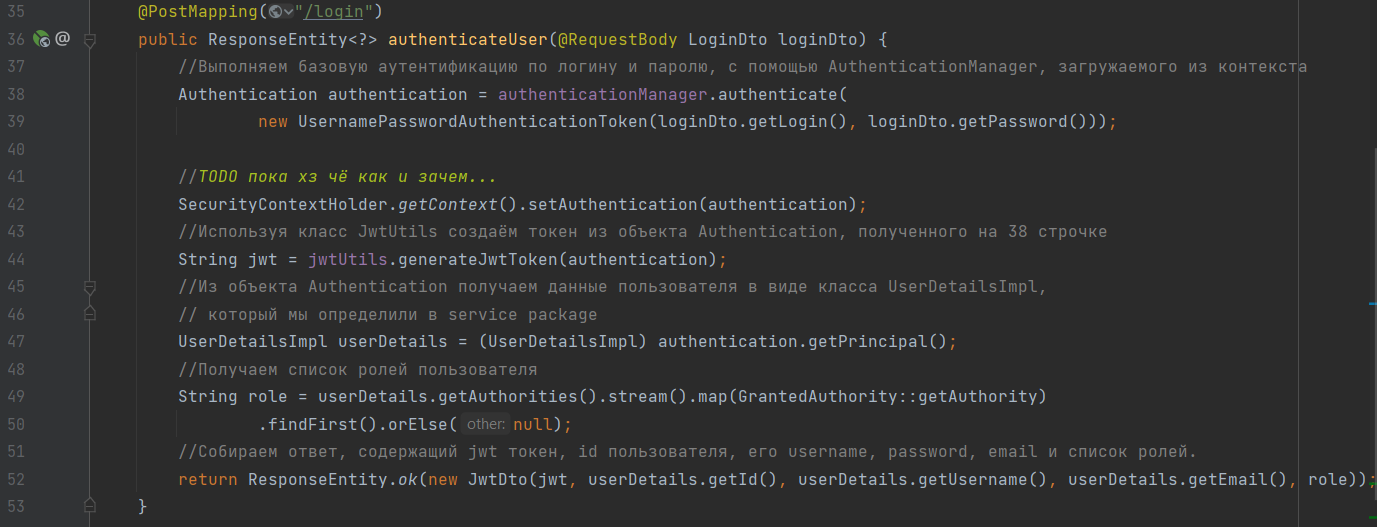
Импорты:



Сигнатура класса контроллера и его поля:



Метод логина:

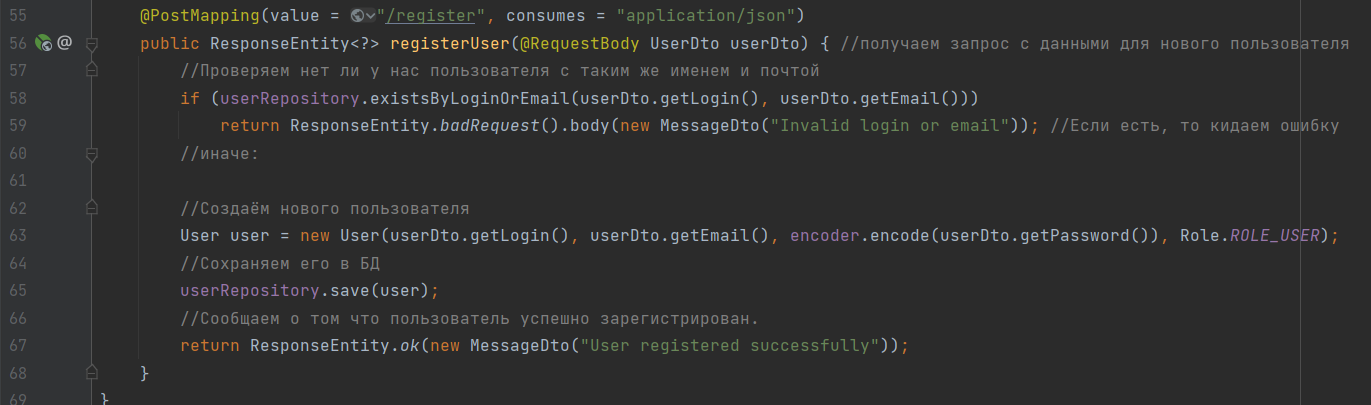


В TODO идёт сохранение пользователя, если он прошёл логин, в контекст спринга, но не особо понятно зачем, т.к. для генерации токена мы используем authentication, который мы здесь же и создали, а для остальных запросов authentication создаст фильтр, если ему придёт верный токен.

SecurityContextHolder — хранит в себе аутентификацию обычно в ThreadLocal переменной.

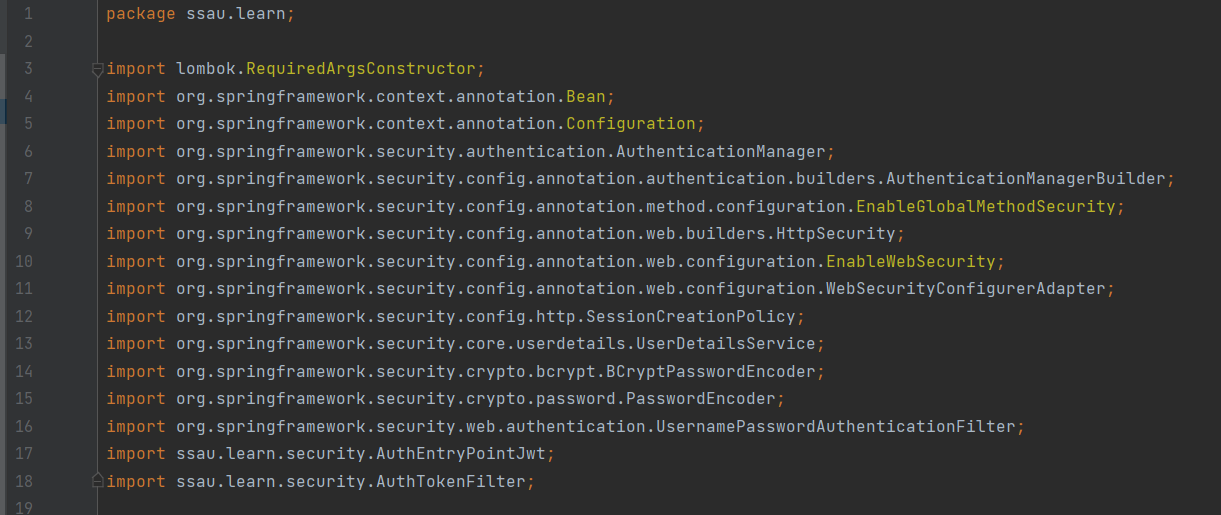
Проверил, можно и без SecurityContextHolder обойтись, если только вам не нужно в этом же методе получать юзера из контекста. В данном случаи ненужно.

Метод регистрации нового пользователя:

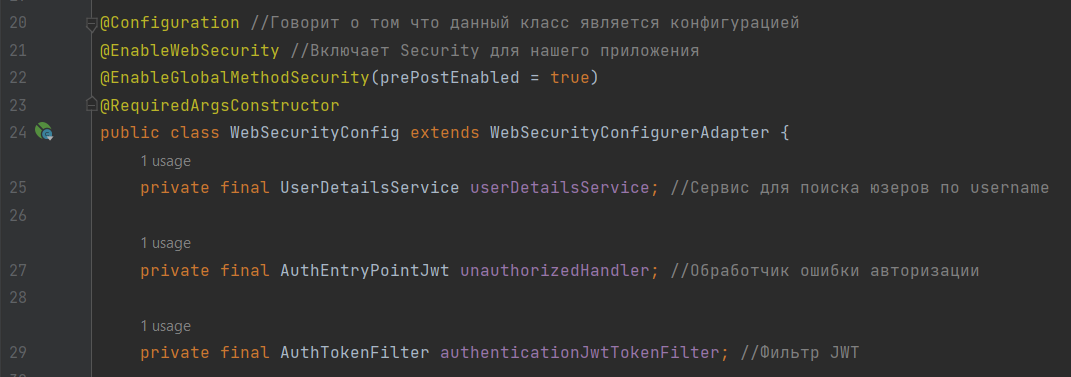


WebSecurityConfig – класс конфигурации spring security:

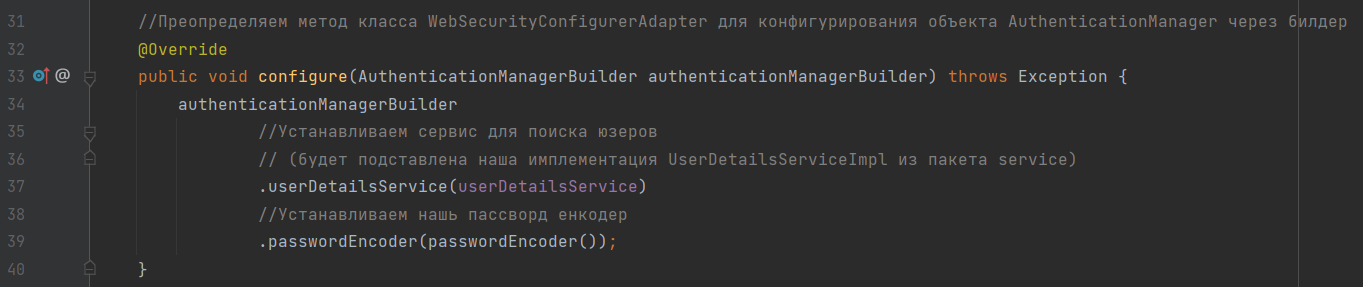
Импорты:



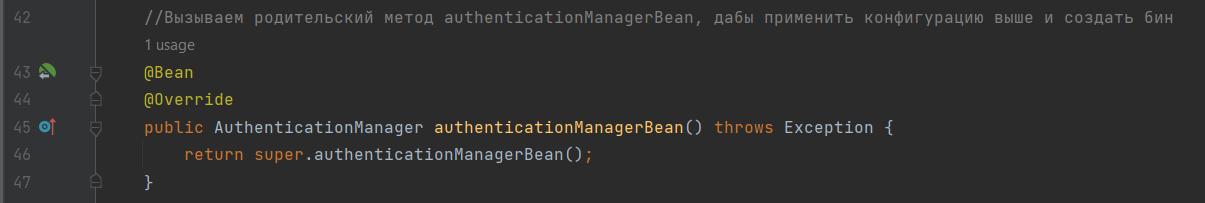
Сигнатура класса и поля:



Метод настройки AuthenticationManager:



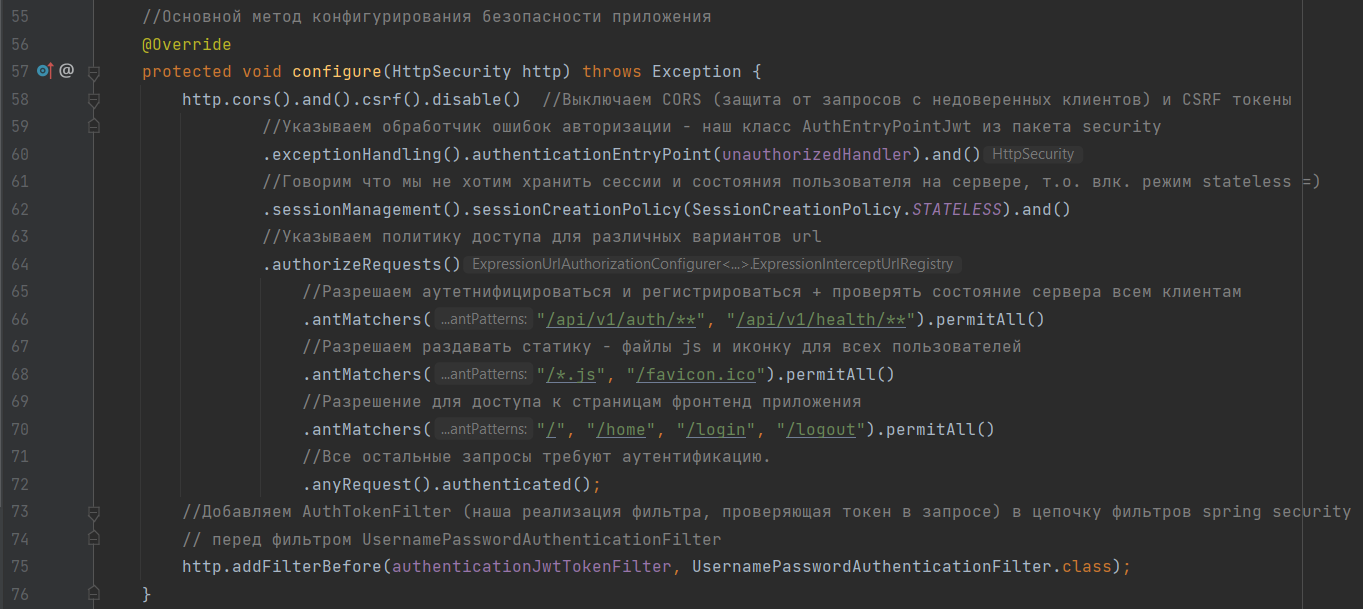
Метод инициализации бина AuthenticationManager:



Метод инициализации бина PasswordEncoder:



Основной метод конфигурации:



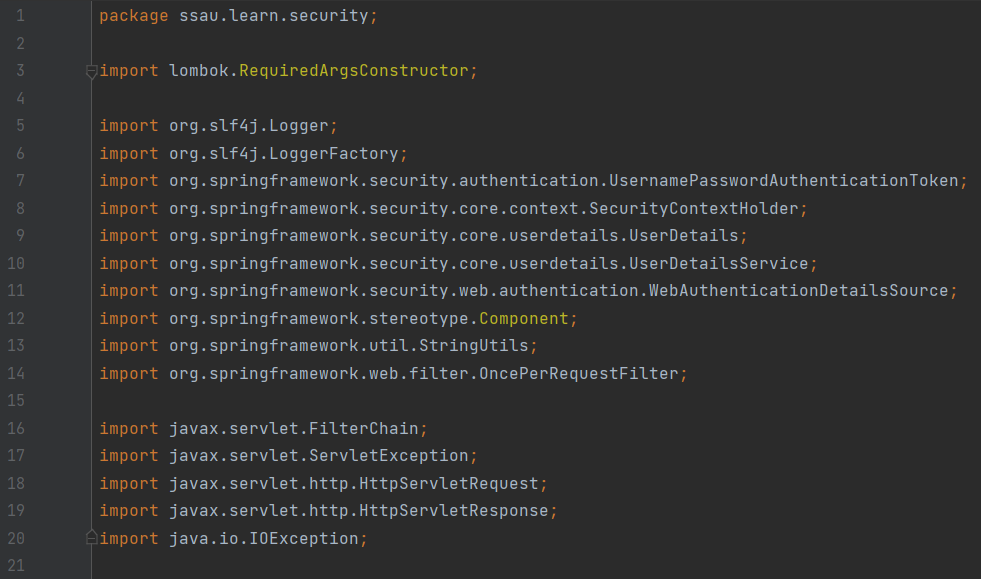
//Примечание:

CORS — это способ для браузеров понять, разрешено ли конкретному коду javascript, выполняемому на нем (например, My React Code — http://localhost:3000), запрашивать ресурс с сервера (My Spring Boot Rest Api, запущенный на сервере — http://localhost:8000). Если они не разрешены серверами, то браузеры просто выдают исключение CORS.

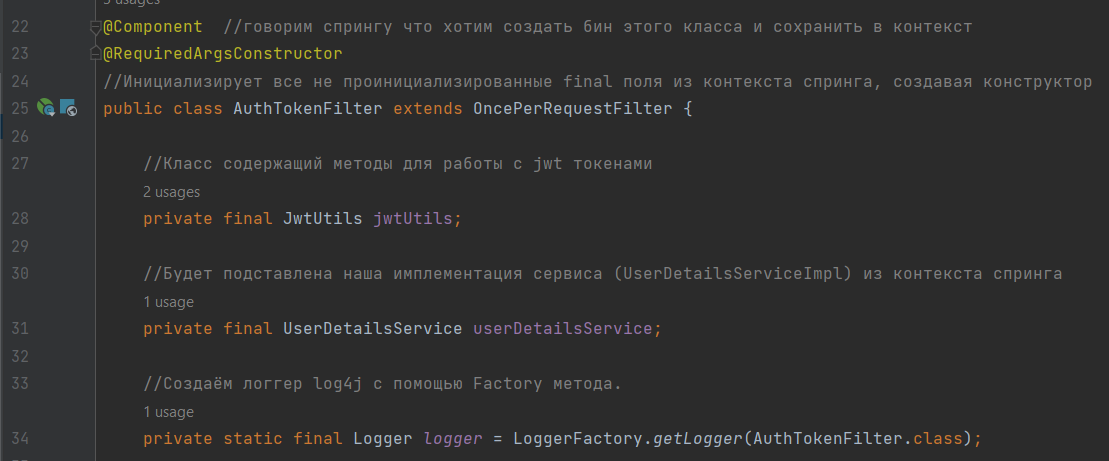
CSRF токены передаются сервером в заголовке ответа клиенту. Если клиент не передаст этот токен в следующем запросе, то сервер будет считать, что он не знает пользователя, пославшего запрос.

AuthTokenFilter – проверяет http запрос на наличие токена аутентификации:

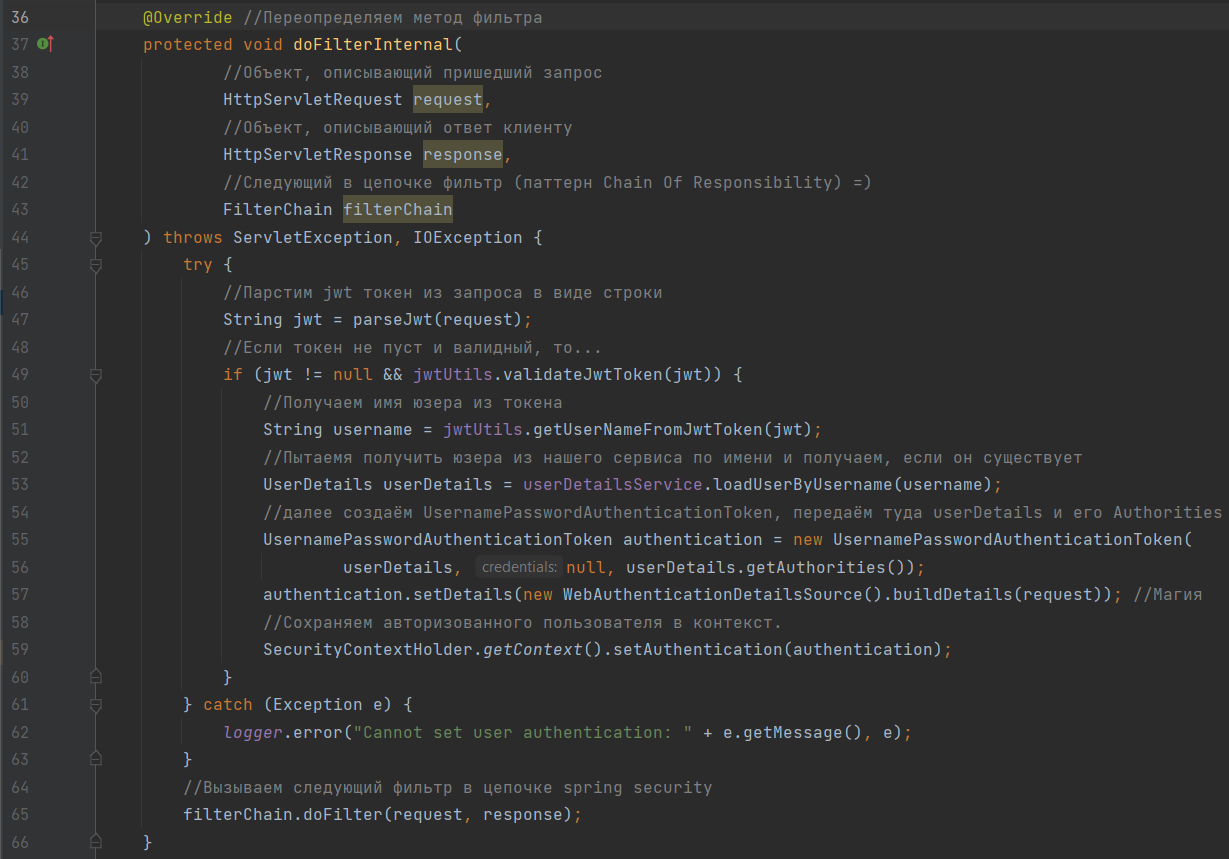
Импорты:



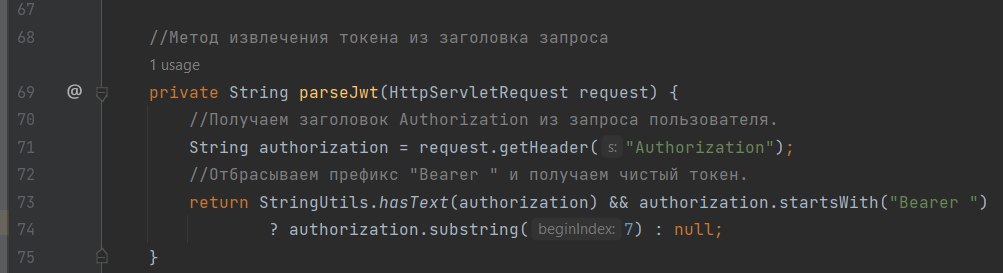
Сигнатура класса и поля:



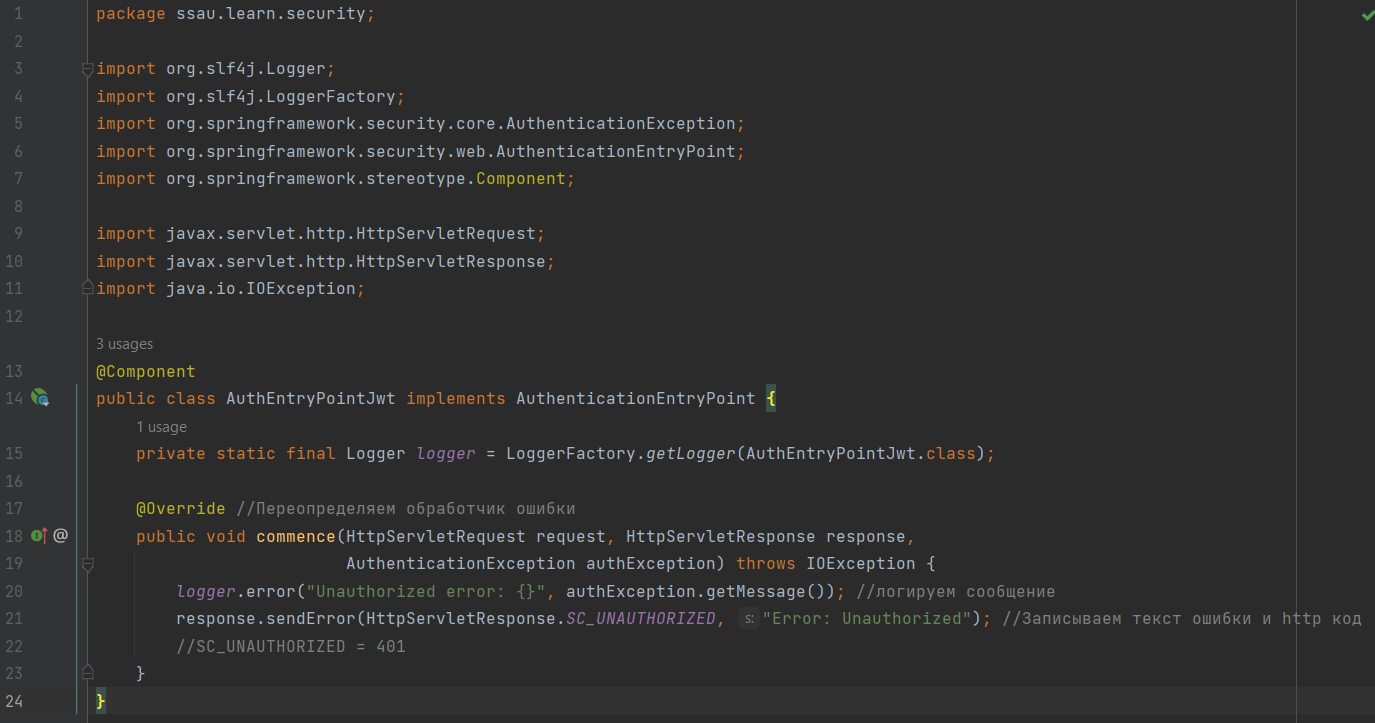
Основной метод фильтра:



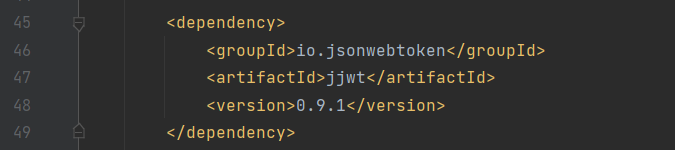
Метод извлечения токена:



AuthEntryPointJwt – обрабатывает ошибку аутентификации:

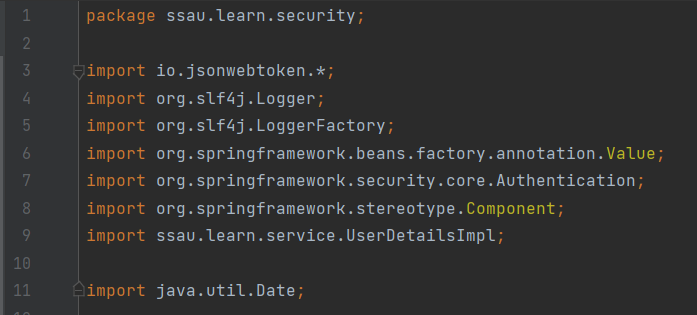


Используемая библиотека для работы с jwt:

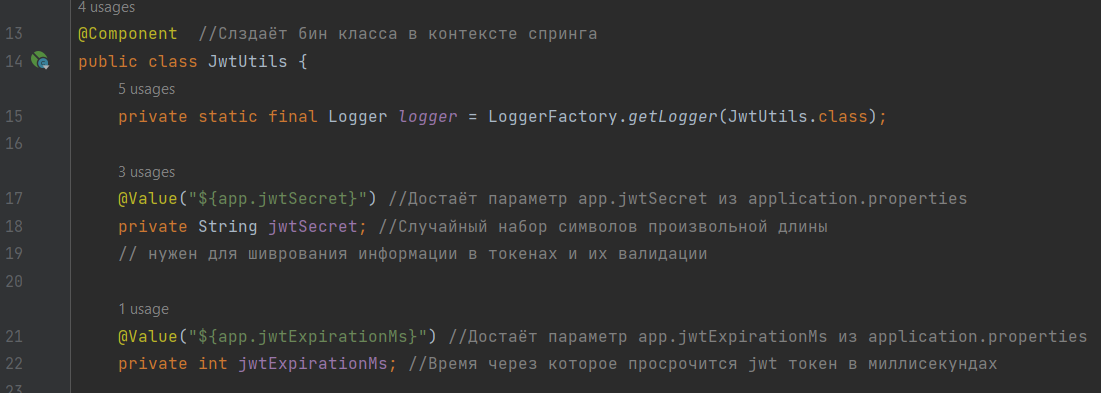
 (из файла pom.xml)

Класс JwtUtils:

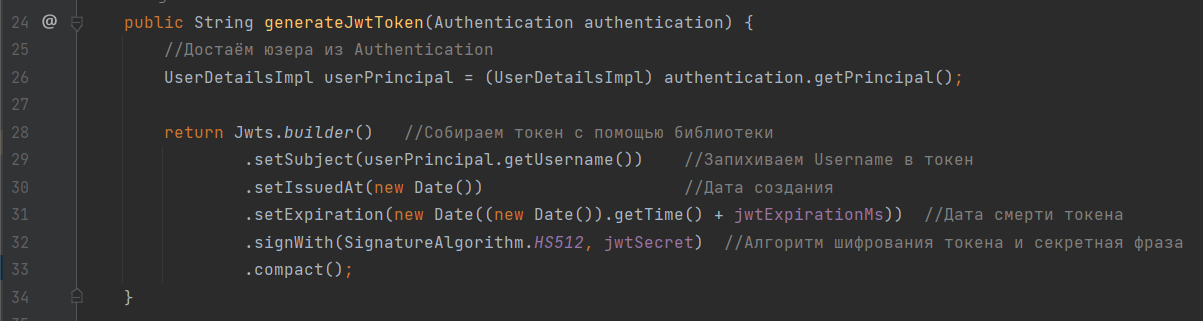
Импорты:



Сигнатура класса и поля:



Метод generateJwtToken:

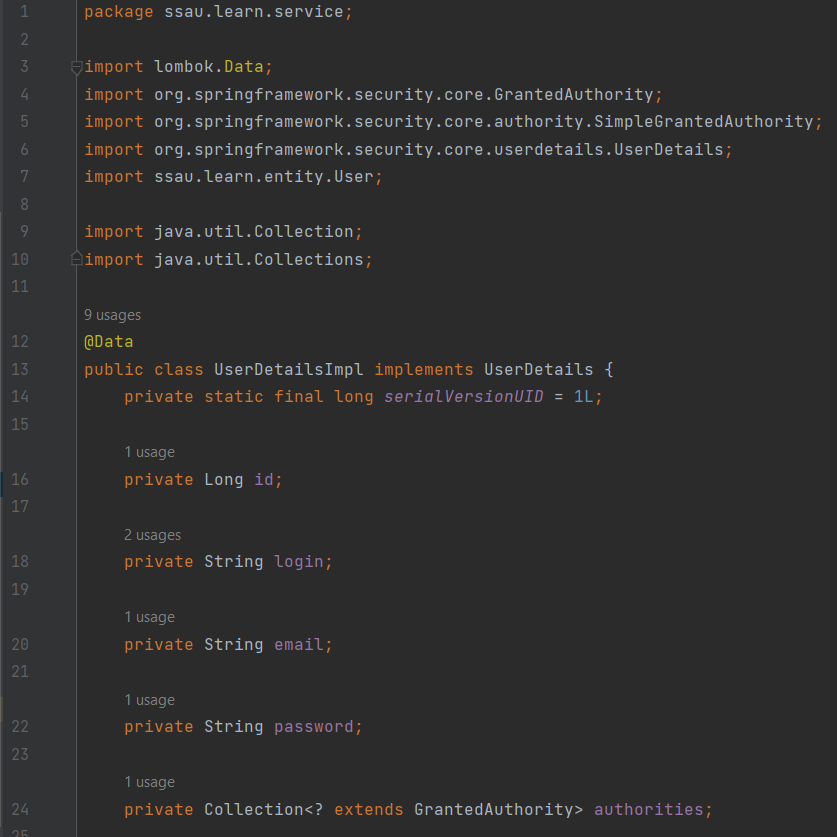


Остальные методы говорят сами за себя.



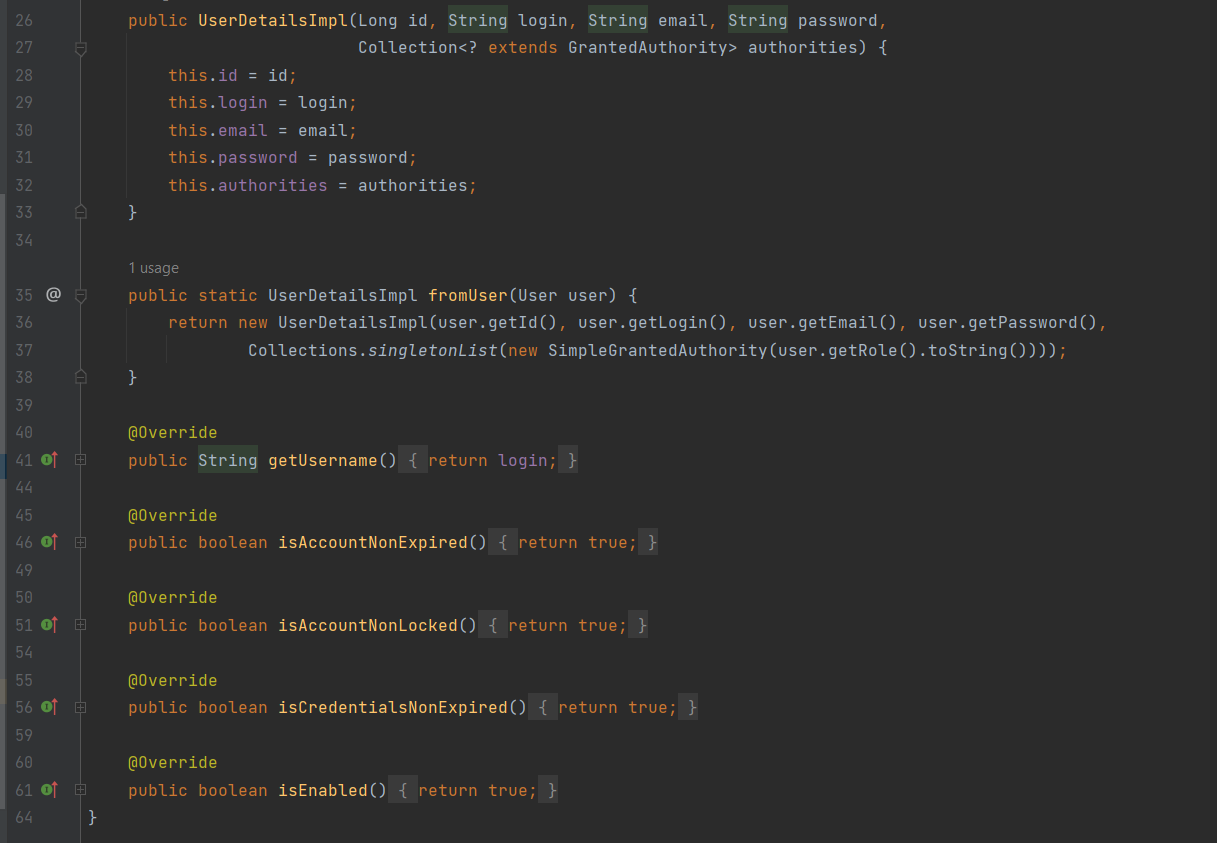
Класс UserDetailsImpl – расширяет базовый класс UserDetails, который требует хранить имя пользователя, пароль пользователя и его роли. Расширяется для добавления информации о пользователе и расширения возможностей.

Импорты, сигнатура класса и поля:

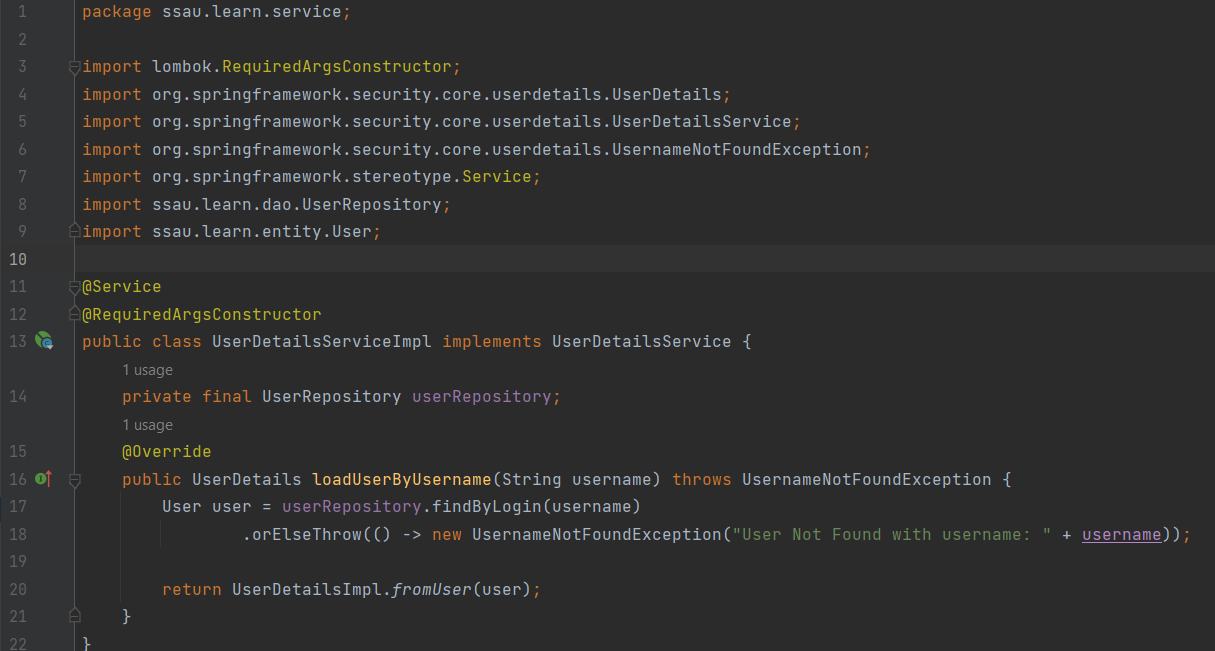


GrantedAuthority – интерфейс, определяющий роли пользователя в spring security.

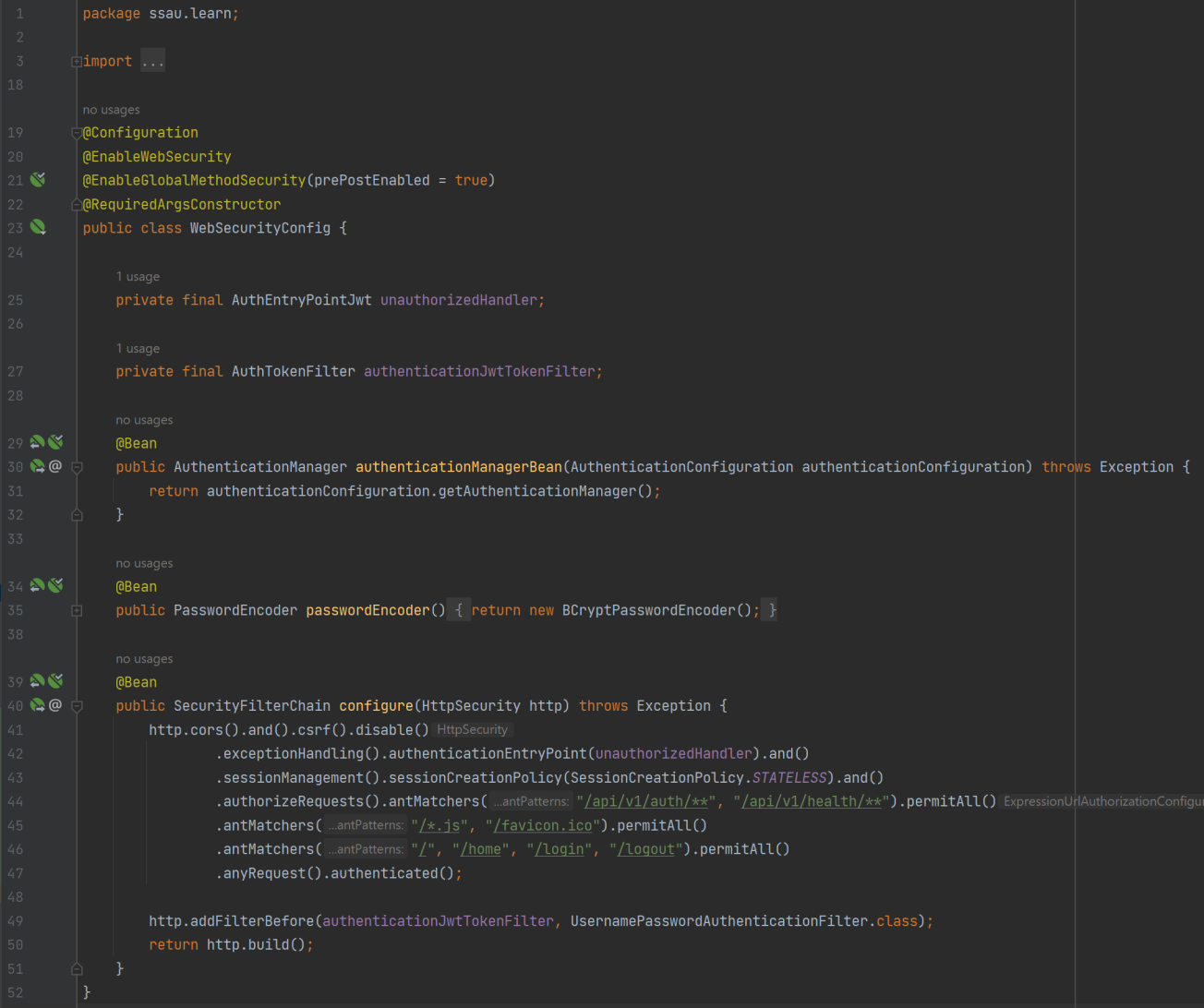
Конструктор и методы:



Класс UserDetailsServiceImpl – реализует интерфейс UserDetailsService, нужен для того чтобы спринг понимал где ему искать пользователей:



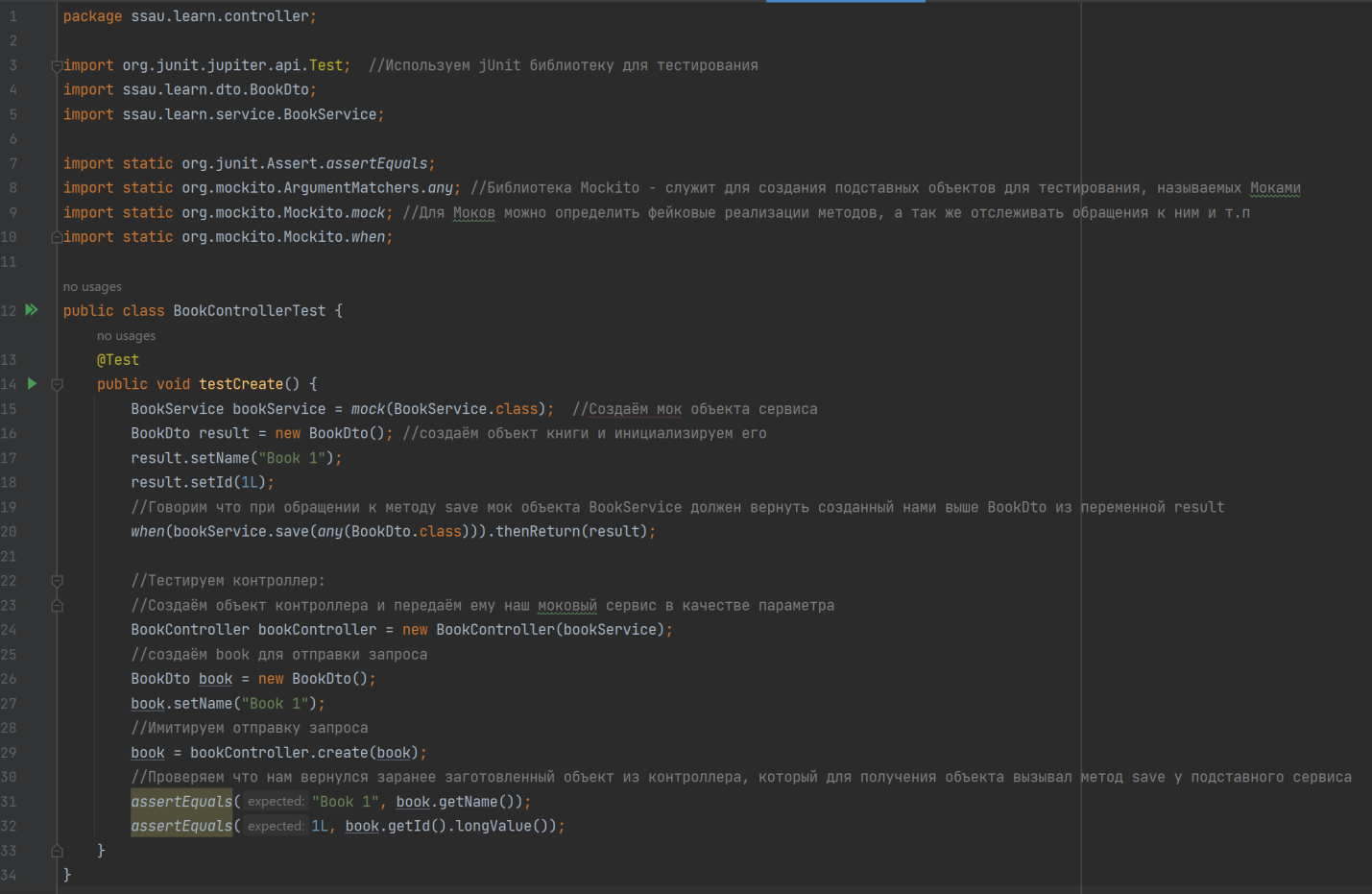
//Дополнание – новый способ конфигурирования spring security без WebSecurituConfigurerAdapter:



**Лекция 4:**

Примеры тестов.

BookControllerTest:



// скипнул остальные тетсы, т.к. на экзамене тестов скорее всего не будет.

**Лекция 5:**

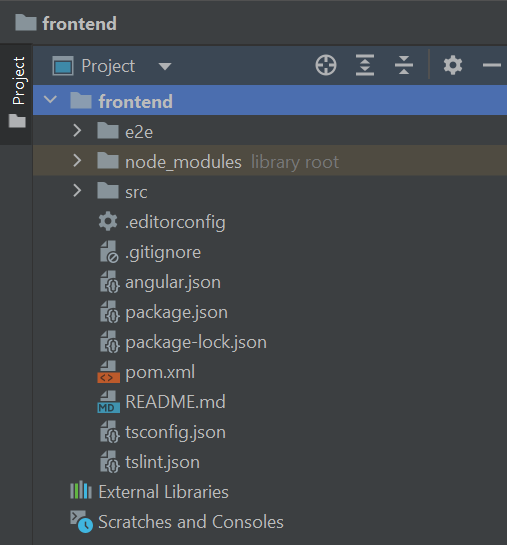
Фронтенд Angular + TS

Генерация нового проекта через angular cli: 

Для скачивания зависимостей фронтенд приложения используем утилиту NPM из NodeJS:



Структура проекта, верхний уровень:



Папка e2e нужна для end to end тестов (в лабораторке не используется).

В папке node\_modules хранятся библиотеки, которые используются при разработке фронтенд приложения на angular.

Папка src хранит исходный код приложения.

**Доп Инфа**

Про связи в JPA

Ориг. Статья: [Как связать Entity в JPA? – Devcolibri](https://devcolibri.com/%d0%ba%d0%b0%d0%ba-%d1%81%d0%b2%d1%8f%d0%b7%d0%b0%d1%82%d1%8c-entity-%d0%b2-jpa/)

В этом уроке речь пойдет о связях между таблицами базы данных на уровне Entity.

### Теоретический материал

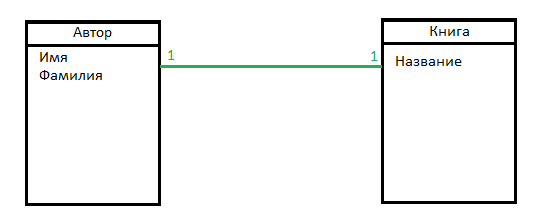
Что такое связь между Entity?

**Entity** – это сущность которая является отображением в базе данных. Связь между Entity – это зависимость одной сущности от другой. Очень часто используются они в построении больших БД.

### Теперь рассмотрим какие есть виды связей

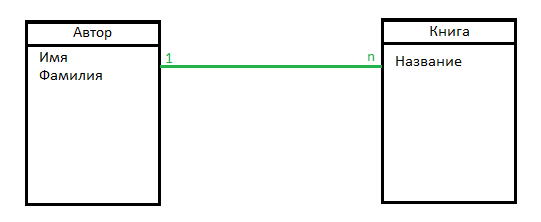
Допустим есть сущность**A** и сущность **B** как сделать между ними различные типы связей:

1. **OneToOne** – один к одному

На рисунку выше видно что есть две сущности **Автор** и **Книга.**Так как сущности имеют связь OneToOne, то с сущности Книга можно получить Автора этой книги, а с сущности Автор можно получить его книгу.

2. **OneToMany** – один ко многому

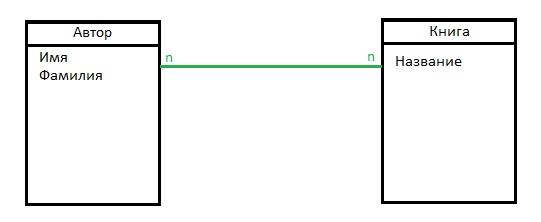
3. **ManyToOne** – много к одному



OneToMany и ManyToOne обычно применяет вместе, что это дает?

Со стороны Автора мы сможем получить список книг которые он написал – это связь OneToMany, а со стороны Книги мы можем получить автора который написал данную  книгу – это связь ManyToOne.

4. **ManyToMany** – много ко многому



В этой связи со стороны Автора мы можем получить все книги автора, а со стороны Книги мы можем получить список авторов, которые написали книгу.

### Реализация

Теперь реализуем все выше рассмотренные виды связей.

Для дальнейших примеров нам нужно реализовать сущность **Автор** и **Книга**.

Создаем класс **Author**.java, который будет отображать сущность **Автор**.

package com.devcolibri.admin.entity;

import javax.persistence.\*;

@Entity

@Table(name = "author")

public class Author {

@Id

private Integer id;

@Column(name = "name")

private String name;

@Column(name = "last\_name")

private String lastName;

public Author() {

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

}

Создаем класс **Book**.java, который будет отображать сущность **Книга**.

package com.devcolibri.admin.entity;

import javax.persistence.\*;

@Entity

@Table(name = "book")

public class Book {

@Id

Integer id;

@Column(name = "name")

private String name;

public Book() {

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

Обязательно должны быть Getter & Setter для того чтобы JPA мог инициализировать данные поля.

### [OneToOne](file:///D:\%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%202.0\SpringDataJPA\%D0%9A%D0%B0%D0%BA%20%D1%81%D0%B2%D1%8F%D0%B7%D0%B0%D1%82%D1%8C%20Entity%20%D0%B2%20JPA%20%E2%80%93%20Devcolibri.html#onetoone)

В сущность Author добавляем следующее:

@OneToOne(optional = false)

@JoinColumn(name="book\_id", unique = true, nullable = false, updatable = false)

private Book book;

public Book getBook() {

return book;

}

public void setBook(Book book) {

this.book = book;

}

а в сущность Book:

@OneToOne(optional = false, mappedBy="book")

public Author author;

public Author getAuthor() {

return author;

}

public void setAuthor(Author author) {

this.author = author;

}

Обратите внимание на **mappedBy**, он указывает на имя атрибута сущности Author для связи с ним.

### [OneToMany](https://devcolibri.com/%d0%ba%d0%b0%d0%ba-%d1%81%d0%b2%d1%8f%d0%b7%d0%b0%d1%82%d1%8c-entity-%d0%b2-jpa/onetomany) и [ManyToOne](https://devcolibri.com/%d0%ba%d0%b0%d0%ba-%d1%81%d0%b2%d1%8f%d0%b7%d0%b0%d1%82%d1%8c-entity-%d0%b2-jpa/manytoone)

В сущность Author добавляем следующее:

@ManyToOne(fetch = FetchType.EAGER, cascade = {CascadeType.MERGE, CascadeType.PERSIST})

@JoinColumn(name = "book\_id", nullable = false)

private Book book;

public Book getBook() {

return book;

}

public void setBook(Book book) {

this.book = book;

}

а в сущность Book:

@OneToMany(fetch = FetchType.EAGER, mappedBy = "book")

private Set<Author> users;

public Set<Author> getUsers() {

return users;

}

public void setUsers(Set<Author> users) {

this.users = users;

}

### [ManyToMany](https://devcolibri.com/%d0%ba%d0%b0%d0%ba-%d1%81%d0%b2%d1%8f%d0%b7%d0%b0%d1%82%d1%8c-entity-%d0%b2-jpa/manytomany)

В сущность Author добавляем следующее:

@ManyToMany

@JoinTable(name="author\_book",

joinColumns = @JoinColumn(name="author\_id", referencedColumnName="id"),

inverseJoinColumns = @JoinColumn(name="book\_id", referencedColumnName="id")

)

private Set<Book> books;

public Set<Book> getBooks() {

return books;

}

public void setBooks(Set<Book> books) {

this.books = books;

}

а в сущность Book:

@ManyToMany(fetch = FetchType.EAGER, mappedBy = "books")

private Set<Author> users;

public Set<Author> getUsers() {

return users;

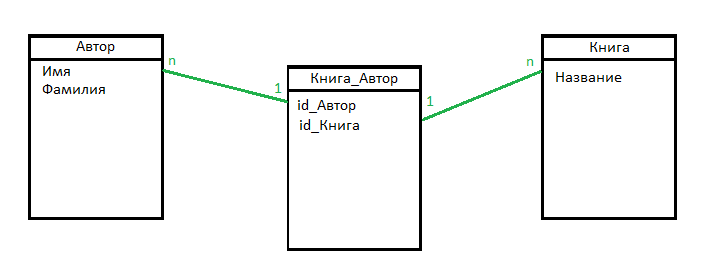
}

public void setUsers(Set<Author> users) {

this.users = users;

}

При такой связи будет получена новая промежуточная таблица:



### CascadeType

Определяет набор каскадных операций, которые распространяются на соответствующие сущности:

**cascade**=**ALL** – на все операции

**cascade**={**PERSIST**, **MERGE**, **REMOVE**, **REFRESH**, **DETACH**} – на определенные операции, те что есть в перечислении будут учитываться.

@ManyToOne(fetch = FetchType.EAGER, cascade = {CascadeType.MERGE, CascadeType.PERSIST})

@JoinColumn(name = "book\_id", nullable = false)

private Book book;

В зависимости от **cascade** будет либо учитываться связь между сущностями для выполнения операции, либо нет.

### FetchType

Определение стратегии для извлечения данных из базы данных.

**EAGER** – стратегия, которая предусматривает получение полной связи между сущностями, и последующих обращениях к связям не будет выполнять запрос на получение данных, так как данные изначально были получены полностью.

@OneToMany(fetch = FetchType.EAGER, mappedBy = "book")

private Set<Author> users;

**LAZY**– стратегия, которая не предусматривает получение полной связи сущностей, и при первом обращении к связи будет выполнятся запрос на получение данных с БД.

@OneToMany(fetch = FetchType.LAZY, mappedBy = "book")

private Set<Author> users;