**ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ**

Факултет „Компютърни системи и технологии”

Катедра „Информационни технологии в индустрията”

Специалност: Информационни технологии в индустрията

**ДИПЛОМНА РАБОТА**

*Тема: Разработка на уеб услуга за музика*

Студент: Александър Ивайлов Иванов

Фак. No: 501217031

Научен ръководител: гл. ас. д-р Галя Павлова

София, 2021

**СЪДЪРЖАНИЕ**

[1. Увод 3](#_Toc72582484)

[2. Въведение в уеб системи 4](#_Toc72582485)

[2.1 История 4](#_Toc72582486)

[2.2 Предимства 5](#_Toc72582487)

[2.3 Езици за реализиране на уеб приложения 7](#_Toc72582488)

[**2.3.1 HTML** 8](#_Toc72582489)

[**2.3.2 CSS** 10](#_Toc72582490)

[**2.3.3 JavaScript** 14](#_Toc72582491)

[3. Проектиране на уеб услугата 18](#_Toc72582492)

[3.1 Технологии за реализиране на уеб услугата Soundcloud 19](#_Toc72582493)

[**3.1.1 Java EE** 19](#_Toc72582494)

[**3.1.2 Tomcat** 20](#_Toc72582495)

[**3.1.3 HTTP протокол** 20](#_Toc72582496)

[**3.1.4 Spring framework** 26](#_Toc72582497)

[**3.1.5 Maven** 27](#_Toc72582498)

[**3.1.6 JPA(Hibernate) и JDBC** 27](#_Toc72582499)

[**3.1.7 MySQL** 28](#_Toc72582500)

[**3.1.8 MVC (Model-View-Controller)** 29](#_Toc72582501)

[4. Програмна реализация 31](#_Toc72582502)

[5. Ръководство за използване 60](#_Toc72582503)

[5.1 User guide 60](#_Toc72582504)

[**5.1.1 User Controller** 60](#_Toc72582505)

[**5.1.2 Playlist Controller** 60](#_Toc72582506)

[**5.1.3 Comment Controller** 61](#_Toc72582507)

[**5.1.4 Song Controller** 61](#_Toc72582508)

[6. Заключение 62](#_Toc72582509)

[7. ИЗПОЛЗВАНИ ИЗТОЧНИЦИ 63](#_Toc72582510)

**1. Увод**

Предмет на тази дипломна работа е проектирането и разработването на back-end частта на уеб-базирано приложение/услуга за слушане на музика. То може да бъде използвано в Интернет при наличието на уеб браузър или като самостоятелно приложение от локален компютър. За реализацията ще използваме допълнителен софтуерен продукт за симулирането на такъв браузър на локално ниво, наречен Postman. Услугата може да се разглежда като аналог на популярното приложение Soundcloud, което е безплатно за ползване, или по-популярното, но платено такова, Spotify.

То ни предоставя възможност за слушане на милиони песни без да е нужно да ги съхраняваме наше устройство, стига да имаме постоянна връзка с Интернет. Услугата осигурява лесен начин за търсене, слушане, харесване и сортиране на песни и плейлисти, следване на други потребители, както и качване на наши песни и създаване на собствени плейлисти. Допълнителна функционалност е писането и харесването/нехаресването на коментари под всяка една песен.

За реализацията на уеб услугата са използвани технологии, съвместими с Java, а именно Spring Framework, Hibernate, JPA, Tomcat Server, MySQL, както и HTTP протокола за безопасна комуникация в Интернет. Използваната среда за разработка е IntelliJ IDEA върху Windows OS.

# **2. Въведение в уеб системи**

## **2.1 История**

Всичко започва с появата на уеб услугата в интернет (1990г.). В самото начало на виртуалната мрежа съдържанието и уеб страниците са били статични - представлявали са само HTML файлове. По-скоро брошури, на които един потребител поставя ръчно някакъв текст. В следващите години на уеб (~1993г.) към брошурата вече са се добавяли и картинки. Към този момент съдържанието, заедно с презентацията му са били едно цяло – HTML. Едва през 1995г. се появява специализираният за презентацията език CSS (Cascading Style Sheets), чрез който се описва форматирането и стила на HTML елементите. По същото време (1995-1997г.) се появява и възможността за динамично генериране на съдържанието в уеб страниците чрез програмни и скриптови езици като Perl, Python, PHP, ASP, които работят на сървъра.

През последвалите години се появят още основни уеб технологии, променящи статичната среда на уеб такава. Една от тях е DOM (Document Object Model) или Документен Обектен Модел (1997-1998г.), с който съдържанието на HTML документа се поставя в дървовидна логическа структура от отделни обекти. А тези обекти, или части от съдържанието, вече може динамично да се променят чрез JavaScript(1995г.) и Ajax (1999г.) заявки, които не изискват презареждане на уеб страницата.

Дотук развитието на уеб технологиите преобръща статичната брошура в динамична и интерактивна уеб страница. Съдържанието и визуалното му представяне се генерират динамично, спрямо потребителя и неговите предпочитания и взаимодействие със страницата.

Интерактивността на уеб сайтовете подхранва взаимодействието на потребителите с тях. Но освен страхотна презентация, за да се задържи интересът и подтикне взаимодействието, сайтът вече трябва да предоставя уместно, качествено и най-вече полезно съдържание. А това от своя страна налага нуждата от добавяне и редовно обновяване на такова съдържание. В ранните години на уеб това би било непосилна задача за един единствен човек, който трябва да обновява ръчно HTML кода на десетките страници в сайта, ежедневно. В създаването на съдържание за сайта вече се включват повече хора, отговарящи за съответната информация – авторите да добавят публикации и помощни статии, маркетинг специалистите да обновяват промоциите, човешки ресурси да добавят обява за работа и други.

И тук е моментът, в който една система за управление на уеб съдържанието става жизнено необходима. Една система, която да предоставя методи за лесно управление на съдържанието в сайта от много потребители. Още през 1995-1997г. се появяват първите системи за управление на съдържанието, които са предимно за корпоративни и бизнес клиенти и са със затворен код, специална разработка.

По това време (1997-1998г.) в софтуерната индустрия вече се говори за Open Source (отворен код), а Netscape публикуват изходния код на своя софтуерен пакет Netscape Communicator. Малко след това е сформирана и инициативата Open Source Initiative, която насърчава създаването на софтуер с отворен код. Отворен код означава, че притежателят на авторските права на изходния код предоставя правата за обучение, промяна и разпространение на софтуера на всекиго и за всякакви цели.

И в началото на 2000г. започват да се появяват и първите системи за управление на уеб съдържанието (WCMS), които са с отворен код, като Drupal 2000г., WordPress 2003г., Joomla 2005г. Най известните системи с отворен код, специализирани за онлайн търговия, се появяват малко по-късно Magento 2008г., OpenCart 2010г., WooCommerce 2011г.

## **2.2 Предимства**

Уеб системите имат голям набор от предимства пред десктоп базираните системи, за които е нужно да бъдат инсталирани или разработени за конкретна операционна система и трябва да имат стриктни хардуерни изисквания, за да се подсигури тяхното правилно функциониране. Тази хардуерна зависимост както и mainframe terminal applications представлява неудобство за някои потребители, а и допълнителни разходи за набавянето на нужните изисквания за изпълнението им.

Уеб базираните системи предлагат единна база данни, което води до централизирана архитектура на уеб системите и позволява съхранението на всички данни на едно място. Това предотвратява например проблемите с консолидирането на данните от различните офиси на компанията или различни инсталации на съответната система (ако например се инсталира отделна база данни във всеки обект, в който се използва софтуерът).

Благодарение на единната база данни е възможно да получавате информация в реално време за състоянието на компанията. Единната база данни позволява и много хора да работят с един и същи запис – например актуална ценова информация, задължения на клиенти и др. Eдин потребител може да вкарва поръчките от клиенти, а друг да обработва стоката от склада, намиращ се в друг град, например.

Улесняват се процесите на инсталация, настройка и поддръжка. В общия случай инсталацията, настройката и поддръжката на една ERP системасе разделя на клиентска и сървърна. Когато вашата ERP\* система е уеб базирана вие не се нуждаете от инсталация на клиентската машина (работните места). При създаване на ново работно място единственото нещо, което е нужно е уеб браузър. Поддръжката и развитието на една ERP система са неотменна част от нейния жизнен цикъл. Поради това, че използвате уеб базирана ERP система, всички промени и модификации по системата се извършват само на едно място, без да е необходимо да се отразяват по всички работни места. Системата е една и се ползва от всички потребители от едно централно място. Това прави поддръжката й ефективна, бърза и евтина.

Имаме лесен достъп до тях. Благодарение на интернет, когато ERP системата ви е в уеб, не сте ограничени да работите с нея само в офиса от служебния си компютър – имате свободата да разполагате с цялата актуална информация, където и да се намирате. Възможно е да работите с вашата уеб ERP система буквално от всяко място чрез интернет – от летището, от хотела или от дома, можете да гледате справки, да модифицирате данните, да отчитате работата си или просто да наблюдавате процесите в компанията. Всички това се случва без нуждата от скъпа комуникационна инфраструктура, каквато изискват десктоп базираните ERP системи. Разбира се, винаги можете да ограничите достъпа до вашата ERP система само за потребители от вашата локална мрежа или чрез VPN. Също така вече не е задължително да използвате само персонален компютър за работа с уеб приложения. Поради това, че повечето последни модели мобилни телефонни разполагат с уеб браузър вашата ERP система лесно може да се направи достъпна за мобилни устройства.

Вместо ръчно да експортвате справки от вашата ERP система, всички ваши контрагенти могат да получават информация в реално време. Получавате запитвания от вашите клиенти директно от уеб сайта, предоставяте справка за оферти, падежи, плащания, поръчки и др. на вашите клиенти, както и справки за вашите доставчици и други контрагенти. Можете да публикувате на вашия уеб сайт различни резултати на компанията, като черпите информация направо от базата данни на ERP системата. Благодарение на новите технологии в уеб (Ajax, SOAP, XML и др.) можете да интегрирате вашата ERP система с външни приложения – публикуване на информация, връзка с външни системи за разплащане и т.н.

При уеб базираните системи липсват ограниченията по отношение на хардуерните характеристики или операционната система на компютъра, от който се ползват. Единственото изискване е активна интернет връзка и инсталиран браузър. Няма хардуерни изисквания. Не е нужно да заделяте голяма част от паметта ви за инсталирането и поддръжката на системата.

За разлика от десктоп приложенията, при който информацията се записва на компютър, USB носители или други външни устройства, уеб базираният софтуер се поддържа на професионално защитени и надеждни сървъри, които се намират в специални информационни центрове. Там се качват и съхраняват базите данни с информация. Няма нужда постоянно да се правят резервни копия на данни от програмата, тъй като архивирането на информацията е автоматично. Обикновено тя се архивира едновременно на няколко независими добре защитени устройства с памет. Така няма опасност от безвъзвратно изгубване на информация, заради кражба или повреда на компютъра, срив в софтуера и т.н.

Предвид факта, че уеб базираният софтуер е нает и се използва през интернет, не се налага да се правят първоначални разходи за закупуването на лицензи, мощни компютри, сървъри, както и за инсталацията му. Няма да се налага да се плаща и за актуализациите на счетоводната програма, техническа поддръжка и т.н. Месечната такса е единственият финансов ангажимент към доставчика на софтуер.

Лесно може да се сложи рестрикция за отделните потребители, тези който имат определен достъп до данни и операции и тези с ограничен достъп. Администраторите могат лесно да проследят извършените от други потребители дейности.

## **2.3 Езици за реализиране на уеб приложения**

Скриптовите езици бързо се превръщат в програмни езици с общо предназначение, поради тяхната функционалност. Те се използват масово там, където времето на разработчиците е по-ценно от времето за изпълнение на програмата. С тях се пишат програми, предназначени за:

Вграждане в Web-страници. Тези програми не се компилират, а се интерпретират от браузъра при зареждането на съответната страница на клиентския компютър.

С помощта на стандартните средства на скрипт - езика и на обектите, дефинирани от DOM, програмистът може да получи достъп до всеки елемент от Web-страницата и динамично да промени неговото съдържание и стилово оформяне.

Реализиране на разнообразни динамични ефекти като показване и скриване на съдържание, придвижване и позициониране на обекти, анимация и т.н. Съединяване в едно цяло различни пакети приложни програми. Създаване на големите Интернет приложения. Автоматизиране.

За да се разбере по-добре същността на скриптовите езици е необходимо да се изходи от второто им наименование – езици за сценарии.

Сценарий – последователност от операции, които потребителят може да извършва на компютъра. Сценариите обикновено се интерпретират, а не компилират.

Приложна програма сценарий (скрипт) — програма, която автоматизира някаква задача, която без сценария потребителя трябва да извърши ръчно. Тъй като сценариите се интерпретират от изходния код динамично при всяко изпълнение, те се изпълняват значително по-бавно от готови програми, транслирани в машинен код на етап на компилиране.

Динамично изпълними скриптовите езици четат инструкциите от файла на програмата на минимални функционални блокове и изпълняват тези блокове без да четат останалия код.

Предварително компилирани скриптовите езици – прочитат програмата в началото, компилират я цялата в машинен код или някакъв вътрешен формат и след това я изпълняват.

Какви са предимствата на скриптовите езици? Скриптовите езици притежават по-сложен инструментариум и поддържат по-прогресивни техники на програмиране. Това съкращава времето за разработка на програми и ги прави по ефективни. Скриптовите езици позволяват бързо да се откриват грешки в програмата и да се реализират доработки на кода без да се изчаква приключване на компилацията.

### **2.3.1 HTML**

HTML (HyperText Markup Language) или език за маркиране на хипертекст е основният език за описание на уеб страници. HTML е стандарт в Интернет, а правилата се определят от международния консорциум W3C. Текущата версия на стандарта е HTML5.

Описанието на документа става чрез специални елементи или маркери, които се състоят от етикети или тагове (HTML tags). HTML елементите са основната градивна единица на уеб страниците. Чрез тях се оформят отделните части от текста на една уеб страница, като заглавия, цитати, раздели, хипертекстови препратки и т.н. Най-често HTML елементите са групирани по двойки и имат отварящ таг <h1> и затварящ таг </h1>.

HTML е основната технология, контролираща това, което уеб браузърът показва на екрана. Основните категории команди на езика HTML, включени във версиите 3.2 и 4.01 са предназначени за: специфициране на стилови формати и управление на текстовия поток (FlowControl); включване на графични изображения (Images); създаване на хииервръзки (Links); интегриране на аудио с външни графични обекти (Sound and Maps); вмъкване на видео таг (Video) и на youtube клипове директно от тяхната страница чрез тага (Iframe); създаване на интерактивни формуляри (Forms); разделяне на документа на отделни полета (Frames); включване на външни приложения (Applet), написани на езика Java; осъществяване на връзка с външни информационни структури (CGI-script). HTML е език за маркиране, който уеб браузърите използват, за да интерпретират и създават текст, изображения и други материали. Неизпълнената характеристика за всяка точка се определя в браузъра, като могат да бъдат променяни или засилвани с допълнителни дизайнерски страници, наречени Cascading Style Sheets (CSS).

В повечето случаи HTML кодът е написан в текстови файлове и се хоства на сървъри, свързани към Интернет.

Тези файлове съдържат текстово съдържание с маркери – инструкции за браузъра за това как да се показва текстът. Предназначението на уеб браузърите е да могат да прочетат HTML документите и да ги превърнат в уеб страници. Браузърите не показват HTML таговете, а ги използват, за да интерпретират съдържанието на страницата.

Основното предимство на HTML е, че документите, оформени по този начин, могат да се разглеждат на различни устройства, а не само на екрана. Документът може да бъде правилно оформен и върху монитора на персонален компютър.

Създаването на HTML-базирана уеб страница може да се извърши с помощта на обикновен текстов редактор. Този начин изисква познаване на HTML тагове, така че те да бъдат интегрирани в текста, който ще се показва на страницата. Също така, често срещани са по-приятелски настроените инструменти, които не изискват от потребителя да притежава познания по HTML. Специализирани инструменти за създаване на HTML страници са някои от продуктите на Microsoft, FrontPage, Notepad, Notepad++, Sublime Text, NetBeans и други.

Пример 1. Обикновен HTML код

<!DOCTYPE html>

<html>

<head>

<title>Заглавие, което се визуализира в таба

на браузъра</title>

</head>

<body>

<h1>Заглавие</h1>

<p>Параграф, в който може да се публива

статия</p>

</body>

<html>

**Семантични елементи в HTML5:**

* Елемент <header> - представлява заглавие на елемент или група от елементи. Тага <header> позволява използването на няколко заглавия в един документ.
* Елемент <nav> - дефинира навигацията на сайта. Той е предназначен за големи блокове от навигационни връзки. Не всички връзки трябва да са в <nav> елемента, а само главното навигационно меню.
* Елемент <article> - представлява предмет/статия. Съдържанието в <article> трябва да има смисъл само по себе си и да може да се разпространява независимо и отделно от останалата част на сайта. Използва се най-често за публикация във форум; публикация в блог; новини; коментари и др.
* Елемент <section> е тематично групирано съдържание, обикновено със заглавие. Този елемент позволява влагане на нова сегмантична структура във вече съществуващ елемент.
* Елемент <aside> - представя съдържание което е странично от основното съдържание на страницата. Съдържанието в <aside> се счита за неважно и може да бъде пропуснато, ако сайта се възпроизвежда на устройство с малък екран като телефон.
* Елемент <footer> - може да съдържа информация за автора на страницата; навигация на сайта; авторските права върху използваните материали и връзки към други (подобни) публикации.

### **2.3.2 CSS**

CSS (Cascading Style Sheets) е език за описание на стилове, който се използва основно за описване на представянето на документ, написан на език за маркиране (HTML, XML). Официално спецификацията на CSS се поддържа от W3C (World Wide Web Consortium). CSS позволява да се определя как да изглеждат елементите на една HTML страница –

шрифтове, размери, цветове, фонове и др. CSS кодът се състои от последователност от стилови правила, всяко от които представлява селектор, последван от свойства и стойности. CSS е удобен начин за форматиране на текст, шрифтове, списъци, таблици и др.

Всяко CSS правило има две основни части: селектор и една или повече декларации. Ако трябва да сме още по-прецизни, всяка декларация се състои от комбинацията свойство и стойност. Синтаксисът е следния:



*Фиг.1.1 CSS правило*

От лявата страна на свойството се намира селекторът – това е частта от правилото, указваща за кой елемент от документа трябва да се приложат стиловете.

В примера са избрани всичките елементи на body. От дясната страна на селектора се намира декларацията, която представлява комбинация от свойството и стойност. Свойството е стилов атрибут, на който се задава стойност. Всяко свойство има стойност. CSS декларацията винаги завършва с точка и запетая, а групите от декларации са заградени от фигурни скоби.

Най-голямото улеснение, което предлага CSS е свързано с контрола на голям набор HTML документи, като контролирането на външния вид на страниците може да става по няколко начина.

Пример 1. CSS код с добавени стилове в </head> частта

<!DOCTYPE html>

<html>

<head>

<title>CSS като отделен файл </title>

<meta

http-equiv=

Content-Type

content="text/html; charset=UTF-8" />

<link type="text/css"/>

</head>

<body>

rel="stylesheet"

href="external.css"

<h1>Заглавие</h1>

<p >Параграф, в който може да се публива

статия</p>

</body>

<html>

Пример 2. CSS код с добавени стилове в </head> частта.

<!DOCTYPE html>

<html>

<head>

<title> вграден CSS файл </title>

<meta http-equiv= Content-Type content="text/html;

charset=UTF-8" />

<style type=”text/css”>

</style>

</head>

<body>

<h1>Заглавие</h1>

<p >Параграф, в който може да се публива статия</p>

</body>

<html>

Пример 3. CSS код със стилове в самите HTML елемент

<!DOCTYPE html>

<html>

<head>

<title> вграден CSS файл </title>

<meta

http-equiv=

Content-Type

content="text/html; charset=UTF-8" />

<style type=”text/css”>

</style>

</head>

<body>

<h1>Заглавие</h1>

<p style=”text-align: left;”>Параграф,

в който може да се публива статия</p>

</body>

<html>

При наличие на повече от един стил, специфициран за HTML елемент, ще се използва този, който има най-висок приоритет (четири е най-високият), както следва:

* + - 1. Стилът на браузъра по подразбиране.
      2. Външните стилове.
      3. Вътрешните стилове.
      4. Стиловете, вградените в самите HTML елементи.

Първоначално HTML етикетите са били създадени, за да определят съдържанието на документа. Или иначе казано, идеята била следната: таговете трябвало да “казват”: “Това е заглавие”, “Това е параграф”, “Това е таблица”, използвайки синтаксис от рода <h1> (за големи заглавия), <p> (за параграф). Постепенно с добавянето на нови спецификации към HTML и с увеличаване на изискванията към визията на сайта, ставало все по-трудно да се създават страници само с него.

Друго предимство на CSS е, че спестява работа и по-точно писане на код. Тъй като можем във външен файл да съхраним повтарящите се форматирания и така да не пишем всеки път атрибутите за тях. Освен това, по този начин при евентуална промяна на изгледа на едно форматиране, се налага само промяна в съответния стил. Например, ако имаме 10 страници с шрифт Arial, размер 2, цвят черен, и искаме да променим цвета, но не сме задали стил, то ще се наложи ръчно да го направим във всичките страници поотделно. Задавайки стил за конкретното форматира в отделен файл, трябва само да се поправи там – на едно единствено място.

Синтаксисът на CSS е съставен от три части: селектор (selector), характеристика (property) и стойност (value).

selector {property: value}

Селекторът е нормален HTML елемент (таг), който трябва да се промени, характеристиката е атрибутът за смяна, като всяка характеристика си има стойност. Характеристиката и стойността са отделени с двоеточие и са заградени с фигурни скоби:

body {color: black}

Ако стойността представлява низ съдържащ няколко думи – необходимо е да се огради с кавички:

p {font-family: "sans serif"}

Ако трябва да се специфицира повече от една характеристика, е необходимо да се разделят с “;”.

p {text-align:center;color:red}

Селекторите с еднакви характеристики могат да се групират по между си и така да се спести писане на код. Това става като се разделят със запетая. В следващия пример са групирани всички header елементи. След прилагането на стила съответния текст ще бъде оцветен в зелено:

h1,h2,h3,h4,h5,h6 { color: green }

CSS притежава още едно важно свойство, така нареченият клас Селектор (class selector). С него може да се зададат различни стилове за един и същи HTML елемент в уеб страницата. Ако например имаме два параграфа, единият от тях е с дясно подравняване а другият с централно, с клас селектор това би изглеждало така:

p.right {text-align: right}

p.center {text-align: center}

Друга полезна опция в CSS e id селекторът. Той се различава от класа селектор, по това че последния може да се приложи върху няколко елемента върху страницата, а id селекторът е приложим само за един. Той трябва да е уникален в рамките на уеб страницата. Долу е дефиниран стил, който свързва елемента <p> с id стойността “paragraph1”

p# paragraph1{text-align: center; color: red }

Добра практика е като се пише код да се оставят коментари, каквато възжможност притежава и CSS. Коментарът започва с "/\*" и завършва с "\*/". Всичко между тях се игнорира от браузъра.

### **2.3.3 JavaScript**

**Javascript** е скриптов език за програмиране, създаден първоначално от *Netscape Communications Corporation* и предназначен за създаване на итерактивни Web страници. Базата за създаване и развитие на езика е една спецификация, носеща името **ECMA-262**. Предназначението на ***Javascript*** е да улесни динамичната обработка на HTML документи при клиента. Скрипт кодът се включва като част от HTML кода. Изпълнението на скрипта става след интерпретация от навигатора на клиента при изтегляне на страницата от сървъра.

Динамично управление на графика, формуляри, хронологичния списък на страниците са сред най-често използваните приложения на скрипт код, написан на ***Javascript***.

***Javascript*** е първият скриптов език за програмиране в среда Web и вече не е единствен. С подобни функции и възможности е ***VBScript*** *(Visual Basic Script)*, създаден от *Microsoft*.

Сходството в наименованията между езиците ***Java*** и ***Javascript*** не ги прави еднакви, нещо повече те са различни в много отношения. Не бива да се смята, че ***Javascript*** е някакво подмножество на езика ***Java***, те просто са два различни езика за програмиране.

Подобно на всеки друг скриптов език чрез ***Javascript*** се създава скрипт-код, който подлежи на интерпретиране.

***Javascript и Java***

Различията между ***Javascript*** и ***Java*** в най-общи линии се заключават в следното:

|  |  |
| --- | --- |
| **Javascript** | **Java** |
| Код интегриран в HTML страница. | Кодът не се интегрира в HTML страница (applet). |
| Кодът се интерпретира от навигатора в момента на изпълнение. | Първичният код се компилира преди изпълнение. |
| Програмният код е упростен, но и с много ограничения. Обектно-ориентиран. | Сложен обектно-ориентиран език за програмиране от високо ниво, почти без ограничения. |
| Достъп до обектите на навигатора. | Липсват средства за достъп до обектите на навигатора. |

***Javascript и HTML***

Приложението на ***Javascript*** изисква предварителни познания по HTML. Написаният код на ***Javascript*** се вмъква директно в кода на HTML страницата по следния начин:

|  |
| --- |
| **<html>**  **<head>**  **<meta http-equiv="Content-Type" content="text/html; charset=windows-1251">**  **<title>Javascript</title>**  **</head>**  **<body>**  **<p>Текстов параграф в HTML код.</p>**  **<p>**  **<script language="JavaScript">**  **<!-- hide from old browsers**  **document.write("Текст, изведен на екран чрез JavaScript код!")**  **// -->**  **</script>**  **</p>**  **<p>Текстов параграф в HTML код.</p>**  **</body>**  **</html>** |

JavaScript кода се вмъква в HTML документа между двойката елементи **<SCRIPT>** и **</SCRIPT>.** Този елемент е въведен в стандарта HTML 3.2 и HTML 4.01. Когато срещне тагът **<SCRIPT>**, браузъра разбира че трябва да спре интерпретирането на HTML кода и да започне да обработва скрипта, намиращ се между **<SCRIPT>** и **</SCRIPT>**. Този скрипт не е задължително да бъде написан на JavaScript. Има и други езици за  писане на скриптове, например VBScript. Затова когато пишете отварящия таг за скрипт, трябва да укажете на браузъра на какъв език ще бъде скрипта. Ако този език е JavaScript трябва да напишете **<SCRIPT LANGUAGE="JavaScript">.**

Код на ***Javascript*** може де се вмъква динамично в момента на интерпретиране на HTML страница чрез следната команда:

|  |
| --- |
| **<SCRIPT SRC="javascript\_file.js" TYPE="text/JavaScript">** |

където, файлът **javascript\_file.js** съдържа код на ***Javascript***.

Основните **характеристики** на езика ***Javascript*** се заключават в следното:

Няколко основни типа данни:

* + Цифров тип, без ясно разграничаване между цели и реални;
  + Символен тип;
  + Логически тип;
  + Нулев тип;

• Слаба типизация в езика - една променлива може във всеки един момент да промени своя тип;

• Синтактична близост на езика с езика за програмиране С, почти същите оператори и инструкции;

• Езикът е основан на недоразвит, начален обектен модел, липсва понятието клас и не поддържа унаследяване;

• Няколко базови обекта за обработка на дати, таблици и математически пресмятания;

• Ограничени възможности за манипулиране на входа и изхода, и почти никакви за въздействие с мрежови операции;

**Елементи** на езика **Javascript** са следните:

• Лексика • Коментари • Типове данни • Променливи • Ключови думи

Като език за програмиране, JS поддържа **обектно-ориентираното програмиране**

и е един от малкото езици, които позволяват да създадете приложение във всяка една възможна област - от сайтове и мобилни приложения до сървърни и desktop приложения.

Най - популярните среди за интерпретация на JavaScript са уеб браузърите, които използвате всеки ден - Chrome, Firefox, Internet Explorer и т.н. Когато зареждате даден уеб сайт е много вероятно той да съдържа JavaScript файлове, които ще се изпълнят при отварянето на сайта и ще направят изживяването при разглеждането на сайта по - приятно и динамично. Твърде вероятно е ако в сайта има падащи менюта, анимации, регистрация на потребител, дразнещи реклами, то те да са реализирани именно чрез използването на езика JavaScript.

Често чуваме, че даден код се "изпълнява на клиента" - това ще рече, че JavaScript кодът, се изпълнява в нашия browser, който играе ролята на клиент, или приемник. За да има приемник, трябва да има и предавател. В технологичният свят тези предаватели се наричат сървъри. Можем да си представим сървърите като едни много мощни компютри, до които имат достъп много хора. Всички сайтове стоят на подобни сървъри - т.е. файловете, които помагат на сайта да изглежда такъв какъвто го виждате - картинки, текстове, JavaScript файлове, се намират на конкретен сървър. Нашият browser (клиент) се свързва към сървъра, на който се намира сайта, който посещавате, и сървърът изпраща обратно необходимите файлове, за да се визуализира пред нас. Можем да си представим, че нашият browser е радиото в колата ви, на което слушате любимата програма, а сървърът е сградата от където се излъчва това предаване, макар и комуникацията между двете да протича по различен начин.

Другият популярен интерпретатор за JavaScript e NodeJS. Можем да си го представим като приложение, което инсталираме на компютъра си и той започва да разбира от JavaScript по същият начин, по който и нашият browser разбира. По този начин можем да изпълняваме JavaScript код директно на компютъра си, без необходимостта от browser. Както казахме, сървърите са просто по-мощни компютри. Те започват да разбират от JavaScript по същият начин, по който и нашият компютър - просто им се инсталира NodeJS.

# **3. Проектиране на уеб услугата**

В настоящата глава ще разгледаме различните аспекти на Web-програмирането, неговите предимства и недостатъци, както и причините за неговата популярност.

Всички сме виждали Web-базирани e-mail системи като mail.yahoo.com, mail.bg и abv.bg. Те са чудесни примери за Web-приложения.

Както знаем, Web-приложенията представляват услуги, които работят на някакъв Web-сървър и предоставят на потребителите Web-базиран интерфейс, който се визуализира от Web-браузърите им. Комуникацията между потребителските Web-браузъри и Web-приложенията се основава на заявки и отговори и се извършва по протокол HTTP. Когато потребителят напише адреса на някое Web-приложение, неговият Web-браузър изпраща на съответния Web-сървър заявка за достъп до това Web-приложение и получава динамично генериран отговор във вид на HTML или друг формат, който браузърът разбира.

**Архитектура на Java-базираните Web-приложения**

Можем да разделим Java-базираните Web-приложения на две части:

- сървърска част – представлява съвкупност от Java компоненти и JSP страници, които обработват получените от потребителя данни и в зависимост от тях динамично генерират HTML документи, CSS стилове и JavaScript код

- клиентска част – представлява съвкупността от динамично генерираните HTML документи, CSS и JavaScript код, които се визуализират от Web-браузъра на потребителя и изграждат потребителския интерфейс на приложението.

Разглеждайки Web-приложенията по този начин, можем да ги определим като многопотребителски клиент-сървър приложения, предназначени за работа в Интернет.

Когато, обаче, се разработва по-сложно програмно приложение, се използват множество готови програмни елементи и класове, които образуват специализираните програмни среди. Най-широко разпространение в момента получава специализираната програмна среда, разработена от Sun Microsystems J2EE, с която съм реализирал и моето уеб приложение.

За реализирането на моето приложение съм използвал Spring Framework технологична рамка с отворен код (open source), които улеснява изключително много работата в създаването на Java уеб базирани приложения.

Описаното накратко до тук представлява ядрото на уеб приложението, какви технологии са използвани и на какъв принцип то ще работи.

За същинската реализация и писането на код ще следваме архитектурния модел Model-View-Controller (MVC), който разделя дадено ссофтуерно приложение на три взаимносвързани части, така че да се отделят вътрешни представяния на информация(бизнес-логиката) от начините, по които информацията се представя на потребителите:  *контролери*, *изгледи*, *модели.*

Всичко казано до тук и всички допълнителни технологии, използвани за реализанирането на уеб приложението ще бъдат представени по-подробно в следващите подглави.

## **3.1 Технологии за реализиране на уеб услугата Soundcloud**

Благодарение на това, че Java е един от най-популярните и използвани езици за програмиране има голям набор от допълнителни технологии, които дават изключително голяма възможност за създаването на сложни и мощни уеб приложения. Използваните от мен технологии за изграждането на Soundcloud уеб услугата са: Java EE, Tomcat, HTTP, Spring Framework, Maven, JPA(Hibernate), JDBC, MVC, MySQL.

### **3.1.1 Java EE**

Платформата J2EE предлага компонентен подход към проектиране, разработка и внедряване на корпоративни приложения и е основно ориентирана за работа в Web среда, както в Интернет, така и в локални мрежи.

Чрез нея се разработват трислойни програмни приложения, които се разполагат на три различни места: машината на клиента, J2EE сървърната машина и машината, съдържаща базата данни с които работи приложението (‘задна’ част). J2EE представлява средния слой, който извършва основната обработка на данните (бизнес-логиката).

J2EE се състои от компоненти, които представляват програми, написани на Java, и се компилирани по същия начин, както стандартните Java програми (класове). Разликата между стандартните Java класове и J2EE компонентите е, че J2EE компонентите се асемблират (комплектоват) в J2EE приложение, проверени са за съответствие с J2EE спецификацията и са подготвени за управление от J2EE сървър(в нашия случай използваме Tomcat web server).

Най-просто можем да кажем, че Java SE се използва за разработването на desktop приложения и е основата на създаването на Java програми. Състои се от набор от инструменти за разработка, технологии за внедряване, библиотеки и набор от класове. Java EE е изграден върху Java SE и се използва за разработването на уеб и големи корпоративни приложения.

### **3.1.2 Tomcat**

Както споменахме, за да създаден уеб услуга се нуждаем от сървър, до който клиентите да изпращат заявки, на базата на които той да предоставя някаква информация. Казано по друг начин Web-сървърите са приложения, които “слушат” на определен TCP порт (обикновено това е стандартният порт за протокола HTTP – 80), и отговарят на HTTP заявките, получени от клиентски приложения (най-често това са Web-браузърите). Платформата J2EE предоставя именно такъв сървър – Tomcat.

### **3.1.3 HTTP протокол**

Не е редно да се впускаме в света на Web-програмирането преди да сме изяснили в детайли протокола, по който си комуникират Web-сървърите с Web-браузърите – HTTP (Hyper Text Transfer Protocol).

HTTP представлява прост текстов протокол, който се използва от услугата WWW за осигуряване на достъп до практически всякакъв вид данни, наричани събирателно **ресурси**. В HTTP протокола има понятия като клиент (обикновено това са Web-браузърите) и сървър (това са Web-сървърите). Обикновено HTTP протоколът работи върху стандартен TCP сокет отворен от клиента към сървъра.

Стандартният порт за HTTP протокола e 80, но може да се използва и всеки друг TCP порт. Комуникацията по HTTP се състои от **заявка** (request) – съобщение от клиента към сървъра и **отговор** (response) – отговор на сървъра на съобщението от клиента.

**HTTP ЗАЯВКИ**

HTTP заявката на протокола има следния формат: **метод**, **Request-URI** и **header-полета**.

|  |
| --- |
| <метод> <Request-URI> HTTP/1.1  <header-полета>  <празен ред> |

Протоколът HTTP поддържа общо 8 различни метода: GET, POST, HEAD, PUT, DELETE, OPTIONS, TRACE, CONNECT.

**Методът** описва вида на HTTP заявката, изпратена от клиента. Най-често използваните методи са GET и POST и те имат най-голямо значение за Web-програмирането. Чрез GET клиентът изисква някакъв ресурс от Web сървъра. POST служи за предаване на данни към сървъра и извличане на ресурс. Имената на методите в HTTP заявките се изписват винаги с главни букви.

Идентификаторът **Request-URI** определя ресурса, над който ще оперира заявката. В частта Request-URI могат да се използват два вида идентификатори:

* **URI идентификатор** (Uniform Resource Identifier)
* **релативен път** спрямо главната директория на Web-сървъра

Един URI идентификатор може да бъде или URL адрес (Uniform Resource Location, например http://www.mywebsite.com/java/index.html), т.е. да е идентификатор на ресурс, зададен чрез уникалното си местоположение или URN име (Uniform Resource Name, например **urn:isbn:954-8905-06-X**), т.е. да е идентификатор на ресурс, зададен чрез уникалното си име по даден URN namespace идентификатор (за нашия пример това е идентификатора isbn). В практиката URN схемата за идентификация на ресурс почти не се използва при HTTP заявки.

Релативният път спрямо главната директория на Web-сървъра задава местоположението на ресурс в рамките на текущия Web-сървър. Това е частта от URL, която стои след името на хост-а (сървъра) в URL идентификатора. Например един такъв релативен път може да бъде идентификаторът **/java/index.html**.

**Header-полетата** от заглавната част задават допълнителни параметри на заявката и определят различни изисквания относно ресурса, който се очаква да бъде върнат от Web-сървъра.

**Празният ред** определя края на заявката.

**HTTP GET заявки**

GET методът представлява команда за извличане на ресурс, указан от зададено URI или релативен път в рамките на Web-сървъра. Всичко, което прави Web-сървърът за извличането на статичен ресурс чрез GET заявка е да го прочете от файловата система и да го върне на клиентите в подходящ HTTP отговор. При извличане на динамичен ресурс сървърът изпълнява програмния код, който генерира ресурса и връща резултата от него в HTTP отговор. Ето един реален пример за HTTP заявка с GET метод:

|  |
| --- |
| GET /Java-2002-program.html  Host: java.sourceforge.net  Accept: \*/\*  Accept-Language: bg  Accept-Encoding: gzip, deflate  User-Agent: Mozilla/4.0(compatible;MSIE 6.0; Windows NT 5.0)  Connection: Keep-Alive  Cache-Control: no-cache |

Изпращането на тази заявка към Web-сървъра, който слуша на порт 80 на машината с Интернет адрес **java.sourceforge.net**, ще върне файла **Java-2002-program.html** от директорията на виртуалния хост **java.sourceforge.net**.

Тази заявка e съвсем реална. Тя е генерирана от Web-браузъра, който използваме, при опит да се отиде на URL адрес http://java.sourceforge.net/Java-2002-program.html и е прихваната чрез софтуер за подслушване на мрежовия трафик.

При HTTP GET заявката, ако към искания ресурс трябва да се зададат параметри, това става като към URI-то се добави въпросителен знак, а след него двойки от вида <име на параметър>=<стойност>, като двойките от този вид се разделят една от друга със &.

**HTTP POST заявки**

POST методът служи за изпращане на данни от клиента към Web-сървъра. Обикновено сървърът предава получените от POST заявката данни на някакъв CGI скрипт или вграден модул за динамично генериране на HTML, който ги обработва и връща някакви резултати. Тези резултати се връщат на клиента като отговор на неговата заявка. Ето един реален пример за HTTP заявка с POST метод, изпратена от нашия браузър при опит за влизане в Web-базираната система за електронна поща на адрес **http://www.abv.bg**:

|  |
| --- |
| POST [/webmail/login.phtml](http://inetjava.sourceforge.net/InetJava-2002-program.html) HTTP/1.1  Host: www.abv.bg  Accept: \*/\*  Accept-Language: bg  Accept-Encoding: gzip, deflate  User-Agent: Mozilla/4.0(compatible;MSIE 6.0; Windows NT 5.0)  Connection: Keep-Alive  Cache-Control: no-cache  Content-Length: 59  LOGIN\_USER=boris  DOMAIN\_NAME=abv.bg  LOGIN\_PASS=tajnamajna |

Както се вижда, параметрите се предават след самата заявка, като в header-полетата се указва общата дължина в символи на всички параметри и техните стойности (заедно със символите за край на ред). За разделител между header-полетата и параметрите се използва празен ред. За край на заявката се използва също празен ред. Ако параметрите съдържат непозволени символи, те се кодират по специален начин. Кодирането на параметрите и стойностите им се прави автоматично от Web-браузъра, а разкодирането им съответно от скрипта, който обработва заявката на Web-сървъра. Като Web-програмисти на Java не е необходимо да знаем в детайли точно как става това разкодиране, защото то се извършва автоматично от средата, в която се изпълнява нашето Web-приложение (т. нар. Web-контейнер).

**Отговори на HTTP заявки**

На всяка HTTP заявка, независимо дали е валидна или не, Web-сървърът връща някакъв отговор. При валидна заявка за съществуващ ресурс Web-сървърът връща този ресурс, а в противен случай връща код на грешка заедно с текстово описание защо се е получила. Отговорът на HTTP заявка има следния формат:

|  |
| --- |
| HTTP <код> <текст>  <header-полета>  <празен ред>  <ресурс> |

Първият ред се нарича **статус линия** и съдържа протокола, по който се изпраща отговора, трицифрен код на резултата или код на грешка и кратко текстово описание на този код.

Следват **header-полетата.** Те съдържат различни параметри на върнатия ресурс, както и информация за Web-сървъра.

Следва **празен ред**, а след него **ресурсът**, кодиран по описания в header-полетата начин. В зависимост от типа на ресурса, сървърът може да го върне кодиран по различен начин и по различен начин да укаже на клиента колко байта е дълъг HTTP отговорът.

Стойностите на header-полетата и формата на ресурса са от интерес основно за Web-браузъра и затова няма да ги разглеждаме в детайли.

Основното, което трябва да знаем, е че на всяка HTTP заявка сървърът отговаря с HTTP отговор, който съдържа искания ресурс или грешка.

Има няколко типа статус кодове, като само последните два типа означават съобщение

за грешка. Подредени според уеб спецификацията, те изглеждат така:

1хх : Информационни

2хх : Успешни

3хх : Пренасочване

4хх : Грешка при Клиента

5хх : Грешка при Сървъра

Най-често срещаните кодове при HTTP отговор са:

**100 Continue** - Този статус код не означава грешка, а потвърждение, че всичко е „ОК“.

**200 OK** - Когато уеб сървърът има отговор за дадена заявка на уеб браузъра, той показва това със статус код 200 OK.

**301 Моvеd Реrmаnеntlу** - Toзи НТТР cтaтyc ĸaзвa нa ĸлиeнтa, чe вpъзĸaтa e пpeнacoчeнa зa пocтoяннo ĸъм нoв aдpec, пocoчeн в Lосаtіоn диpeĸтивaтa. Kлиeнтът тpябвa дa зaпoмни тoвa и пpи cлeдвaщa зaявĸa дa тъpcи директно нoвия aдpec.

**302 Found** -Kодът указва на уеб клиентите, че новият адрес на ресурса е временен, тоест за в бъдеще да използват стария уеб адрес.

**400 Bad Request** - Поради лош синтаксис заявката не може да бъде разбрана от сървъра.

**401 Unauthorized** - За достъп до ресурсите се изисква идентификация. Съобщението означава, че нямате разрешение за достъп до заявената информация.

**403 Forbidden** - Забранен достъп. Сървърът разбира заявката, но отказва да я изпълни. Възможни причини: опит за достъп до директория (папка), която няма индексна страница; опит за достъп до защитени с парола ресурси, при въвеждане на грешна парола; забранен достъп през вашия IP адрес и др.

**404 Not Found** - Сървърът не може да намери търсения ресурс. Възможна причина за това е сгрешен URL адрес или че ресурсът не съществува.

**500 Server Error** - Сървъра е неспособен да изпълни заявката поради вътрешна грешка.

Важно е да споменем и 3-те основни характеристики на HTTP протоколът:

1. HTTP няма постоянна връзка: HTTP клиентът(браузърът) праща HTTP заявка и връзката между клиентът и сървъра се прекъсва. Сървърът обработва заявката и възстановява връзката с клиента, за да изпрати обратно отговора.
2. HTTP е медийно независим: Това означава, че всякакъв тип данни могат да бъдат изпращани чрез HTTP, стига и клиентът, и сървърът да знаят как да боравят със съдържанието на данните. Това е необходимо, както за клиент, така и за сървър, за да посочи типа на съдържанието, използвайки подходящ MIME type.
3. HTTP не пази състояние: Както бе споменато по-горе, HTTP е без постоянна връзка и това е пряк резултат, че HTTP е протокол, които не пази състояние. Сървърът и клиентът са наясно един с друг само по време на текуща заявка. Впоследствие и двамата забравят един за друг. Поради този характер на протокола, нито клиентът, нито браузърът могат да пазят информация между различни заявки в уеб средата.

Както се вижда от първата и третата характеристика, сървърът „знае“ за нас само по време на заявката и след като ни отговори забравя за нас. Как тогава, например, запомня, че сме логнали в даден сайт или съхранява количката ни при пазаруване? Тук се добавят и 2 много прости решения – Cookies и Sessions.

**“Бисквитките“ се намират на компютъра на потребителя,** могат да съдържат произволна информация, избрана от сървъра, и се използват да поддържат състоянието на HTTP транзакциите, които иначе са „без състояние - stateless“. Обикновено те се използват за удостоверяване на самоличността на регистриран потребител на даден уебсайт като част от процеса на влизане или първоначална регистрация в сайта, като от потребителя не се изисква ново въвеждане на потребителско име и парола при всеки следващ достъп до този сайт. Други използват бисквитките за поддържане на „пазарска кошница“ за избрани стоки за купуване от даден сайт по време на една сесия, за персонализация на сайт (представяне на различните страници - дизайн, тема, мащаб и т.н.), и за проследяване на достъпа на отделни потребители до даден сайт.

**Сесията е информация от страна на сървъра,** предназначена да съществува само по време на взаимодействието на посетителя с уебсайта. От страна на клиента се съхранява само уникален идентификатор. Този маркер се предава на уеб сървъра, когато браузърът на посетителя поиска вашия HTTP адрес. Този маркер съвпада с уебсайта ви с информацията за посетителя, докато потребителят е на вашия сайт. Когато потребителят затвори уебсайта, сесията приключва и вашият уебсайт губи достъп до информацията.

Казано по-просто, при достъпването на даден уеб сайт той създава един „файл“ с бисквитки, които се пази от нас(браузъра) и се изпраща заедно с всяка нова заявка. Сървърът пуска и нова сесия, която се пази при него и при всяка заявка се проверява дали има потребител с такава сесия. В с бисквитките се пази най-общо информацията за персонализацията на сайта(настройките, които сме направили за наше удобство) и ид на сесията, което служи за удостоверяване - щом един път се сме логнали или влезли в сайта той да знае, че сме ние. По този начин се решава „stateless“ проблема при HTTP протокола.

### **3.1.4 Spring framework**

Както споменаг по-рано, Spring Framework е технологична рамка (или „framework), която е с отворен код. Тя е предназначена за Java платформата, като представя множество различни функции, които улесняват разработката на различни приложения, базирани на Java. Основно ядро е неговият **IoC** (Inversion of Control) контейнер, който осигурява логически средства за конфигуриране и управление на Java обекти с помощта на reflection. Контейнерът е отговорен за управлението на жизнения цикъл на обектите: създаването на тези обекти, извикването на инициализиращите им методи и конфигурирането им, свързвайки ги заедно. Обектите, създадени от контейнера се наричат още управлявани обекти или **бийнове** (beans). Контейнерът може да бъде конфигуриран чрез зареждане на XML файлове или специфични Java анотации от конфигуриращите класове. За по-лесна и безпроблемна работа съм използвал Java анотации за конфигуриране на контейнера.

Обектите(инстанциите на класовете) могат да бъдат получени чрез dependency injection. Dependency injection е софтуерен шаблон за дизайн, който опростява Обектно-ориентираното програмиране (ООП) чрез намаляване на зависимостите между отделните класове. Така при създаването на един проект, използвайки някой обектно-ориентиран език, се избягва зависимостта от конкретни имплементации, и се набляга повече на абстракции. Това се нарича още обръщане на контрола (inversion of control).

При dependency injection обектите се проектират по начин, при който те получават инстанции на обекти, от които зависят, за да изпълняват бизнес логика, от различен външен код, вместо да са конструирани вътре в него(благодарение на анотациите, ние оказваме, че когато имаме нужда от даден обект от даден клас Spring го създава за нас без да е нужно ние изрично да го създаване с ключовата дума new). Зависимият елемент определя на какъв софтуерен компонент зависи за да си свърши работата, а контейнерът решава какви конкретни класове отговарят на изискванията на зависимия обект, и ги предоставя като зависимости. По този начин контейнерът може да избира конкретните имплементации, които да се използват по време на изпълнение на програмата (run-time), а не по време на компилация (compile-time).

По време на изпълнение (run-time) могат да бъдат създадени множество различни инстанции на даден софтуерен компонент, без значение каква конкретна е била инжектирана.

### **3.1.5 Maven**

Maven е инструмент за управление на проектите, който управлява различни аспекти от жизнения цикъл на проекта като изграждане, отчитане, документация, базирана на основен източник на информация.

При изграждането даден проект са нужни много допълнителни библиотеки и JAR файлове. Тъй като голям брой от програмисти работят по един проект е възможно един да добави даден файл и да забрави да го сподели с останалите или да изтегли остаряла версия, а останалите да работят с по-нова такава. Това може да доведе до конфликти и различно поведение на нашия софтуер. Maven ни дава възможност да обединим всички допълнителни файлове в едно и всеки друг, който работи по проекта ще може да ги види и изтегли. По този начин всички разработчици ще работят с еднакви и пълни версии на допълнителните библиотеки и jar файлове.

В обобщение, Maven опростява и стандартизира процеса на изграждане на проекта. Той безпроблемно се справя с екипното сътрудничество, съставянето, разпространението, документацията и отделните задачи. Maven увеличава повторната употреба, грижи за повечето задачи, свързани с изграждането и всички работят с еднакви версии.

### **3.1.6 JPA(Hibernate) и JDBC**

Почти всички корпоративни приложения се нуждаят от редовен достъп до релационни бази данни. Но проблем, с който се сблъскват по-ранните технологии (като JDBC), е несъответствието на импеданса (разлика между обектно-ориентираните и релационните технологии). Решението на този проблем беше въведено чрез въвеждането на абстрактния слой, наречен Персистентен слой, който капсулира достъпа до база данни от бизнес логиката. JPA (Java Persistence API) е рамка, предназначена за управление на релационни данни в Java приложения. Има много внедрения на доставчици на JPA, използвани в общността на разработчиците на Java. Hibernate е най-популярната такава реализация на JPA.

Hibernate е рамка, която може да се използва за обектно-релационно картографиране, предназначено за програмения език Java.

По-конкретно, това е ORM (обектно-релационно картографиране) библиотека, която може да се използва за картографиране на обектно-релационен модел в конвенционален релационен модел. С прости думи, той създава картографиране между Java класове и таблици в релационни бази данни, също и между Java и SQL типове данни. Хибернацията може да се използва и за заявки и извличане на данни чрез автоматично генериране на SQL заявки. Следователно програмистът е освободен от ръчното боравене и преобразуване на обекти и писане на чисти SQL заявки в бизнес логиката.

JDBC е набор класове и методи за взаимодействие на програми на Java с източници на данни. Той има абсолютно същото предназначение, но реализирано по коренно различен начин. В JDBC е дефиниран набор класове и методи за взаимодействие със СУБД(Система за управление на База данни). В програмата ни първо трябва да се отваря връзка към базата данни, да се създаде обект-заявка, SQL заявката се препраща към СУБД чрез този обект, и се получава резултата, както и допълнителна информация.

Класовете от JDBC са в пакета Java.sql и се използват от всички програми на Java за четене и запис в източници на данни. Програма, използваща JDBC, се нуждае от драйвер за съответната СУБД. Допустимо е приложение да работи с няколко разнородни източника на данни. Затова в JDBC е вграден диспечер на драйверите, чието предназначение е да управлява драйверите и да предоставя списък на наличните драйвери на приложенията. За реализацията на моето приложение използвам MySQL СУБД.

Основната разлика между JDBC и JPA(Hibernate) е, че при първия вариант ние ръчно трябва да се свързваме към базата, да пишем чисти SQL заявки и да ги изпращаме, докато при втория всичко се случва автоматично – не се грижим за връзката с базата, автоматично ни се генерират огромен брой от предполагаеми заявки на базата на класовете, които сме създали, и от нас се иска да изпълним тези, от които се нуждаем.

### **3.1.7 MySQL**

Релационните бази данни са такива бази данни, при които информацията се съхранява таблици, които са съставени от записи и колони. Софтуерът, който се грижи за поддържане и управление на базата данни се нарича Система за управление на база данни (СУБД). Съществуват различни СУБД- например Microsoft SQL Server, MySQL, Access, Oracle и др. Предимствата на базите данни като средство за съхранение на информация са намаляване на обема на данните и отстраняване на повторенията. Коректност и цялостност на данните – СУБД предлагат възможност за налагане ограничения върху стойностите на данните, с което се избягва попадането на некоректна или несъществена информация.

Друг съществен плюс е независимостта на софтуера от информацията - всеки софтуерен продукт по един или друг начин обработва информация – генерира нова, представя, променя или трие стара. Съхранението на данните в отделно хранилище гарантира, че дори и при повреда на софтуера, данните ще се запазят. Много софтуерни продукти могат да ползват една и съща база данни, като съществува високо ниво на сигурност при достъп до базата - софтуерните приложения се явяват клиент за СУБД-то, като всеки клиент има регламентирани права за достъп до данните. Например едни могат единствено да четат(read only), други – да четат и променят, но само определени схеми/таблици. Трети имат администраторски - full права и тн. СУБД може да извършва и други операции като backup, например.

Използвана от от мен MySQL е open source релационна, управляваща бази от данни система (RDBMS), която използва SQL, най-популярния език за въвеждане, достъп и извършване на други процеси с данни в база данни. Понеже е оpen sourcе, всеки може да я изтегли и използва, в зависимост от общите публични правила. MySQL е известна основно заради скоростта си, сигурността и гъвкавостта си. Общоизвестно е обаче, че тя работи най-добре, когато управлява съдържание и не извършва транзакции.

Системата за релационни бази данни MySQL осигурява приложен програмен интерфейс (API) за C, C++, Java, Perl, PHP, Python, и Tcl, допуска много типове на колоните и предлага пълна поддръжка на всички операции – от извличане до добавяне и модификация на данните, както и до промени по самия модел на базата.

Понастоящем MySQL работи върху почти всички популярни операционни системи.

### **3.1.8 MVC (Model-View-Controller)**

Model-View-Controller или MVC е архитектурен шаблон за дизайн в програмирането, основан на разделянето на бизнес логиката от графичния интерфейс и данните в дадено приложение. Традиционно MVC шаблонът се използва за графични потребителски интерфейси (*Graphical User Interface – GUI*), но вече той се използва както за и мобилни приложения, така и за уеб такива, както е и в моя случай.

**Компоненти на MVC шаблона**

Както личи от името “Model-View-Controller”, шаблонът се състои от три отделни компонента, които изпълняват определена роля и са свързани в определена последователност.

„Моделът (Model)“ е централен компонент в шаблона. Това динамичната структура от данни на приложението, независима от потребителския интерфейс. Моделът управлява данните, логиката и правилата на приложението.

„Изгледът (View)“ е изходящият поток от информация (това, което приложението изпраща като отговор до дисплея, респективно – до потребителя, в следствие на неговата заявка). В моето приложение този слой не е реализиран и няма конкретна имплементация, защото приложението е насочено основно в разработването на back-end частта до такава степен, която front-end програмистта да е наясно каква информация получава и да успее да реализира тази част от шаблона.

„Контролерът (Controller)“ е третата част от този шаблон. Той приема потребителския вход (т.е. данните, които потребителя въвежда, неговите заявки и т.н.) и ги преобразува в команди към модела или изгледа.

**Предимства на MVC**

Тъй като MVC разделя основните компоненти на приложението, това позволява на разработчиците да работят паралелно по различни компоненти, без да оказват влияние или да си пречат един на друг. Например един дев екип може се раздели на две части, като едната поема изграждането на back-end логиката, а другата – на front-end частта. Back-end разработчиците проектират структурата на данните и как потребителят взаимодейства с тях, без да се нуждаят от потребителски интерфейс (front-end), за да тестват и усъвършенстват тези аспекти на приложението. Обратно – front-end разработчиците могат да проектират и тестват изгледа(view частта) на приложението, преди да имат готова структура от данни, с която той да борави.

Създавайки компоненти, които са независими едни от други, разработчиците могат да използват многократно един компонент в различни приложения. Изгледът на едно приложение може да се преработи бързо и лесно за друго сходно приложение, което борави с коренно различна структура от данни; моделът който обработва данните и който представлява цялостната структура е отделен компонент спрямо изгледа и това дава свобода на разработчиците да използват наново един и същи код, според нуждите си.

В допълнение можем да кажем, че MVC помага да се раздели сложността на приложението.

Той подобрява поддръжката на кода и осигурява пълен контрол върху поведението на приложението, което ни позволява изграждането на големи, сложни и мащабируеми уеб приложения.

# **4. Програмна реализация**

Тук ще представя програмната реализация на уеб услугата Soundcloud. За по-добра четимост, както споменах по-рано, кодът е разделен на пакети и подпакети, следвайки логиката на архитектурния шаблон MVC, а именно:

1. Имаме пакет controller, където се намират всички контролери на услугата. Това са класовете, които описват какви URL – и имаме, съответните им HTTP методи(GET, PUT, POST, DELETE). Тези класове съдържат методи, които отговарят за всеки един URL и на базата на въведения URL се изпълнява съответния метод. Най-просто можем да кажем, че контролерите са началната точка, от която се решава от кои сървис имаме нужда за изпълнение на клиентска заявка.
2. Имаме пакет service, където се намират всички сървиси на услугата. Това са класовете, където е написана цялата бизнес логика, свързана с приложението. В тях се правят всички проверки, свързани с функциалността на услугата – извличане, редактиране или добавяне на данни. Проверяват се дали въведените данни са коректни, дали имаме „права“ за извършването на дадена операция и др. Отново в сървисите се прави и връзка с базата данни по 2 начина – с Repositories(хранилища) или DAOs(Data Access Objects), които сe намират в model частта от MVC. Най-просто можем да кажем, че сървисите получават заявката от контролера, обработват заявката като се „допитват“ до базата данни посредством моделите – DAOs и Repositories.
3. Имаме пакет model, където се намират всички модели на услугата. Тук класовете са разделени на 3 подпакета – DAOs, DTOs, POJOs, repositories.
4. DAOs или Data Access Objects са класове, които използваме за осъществяване на достъп до базата данни. В тях ръчно правим връзка с базата, пишем чисти SQL заявки, изпращаме ги и очакваме отговор с резултата, които предаваме на сървисите. В тези класове се появява нуждата от JDBC, благодарение на който конфигурираме връзката с базата данни.
5. DTOs или Data Transfer Objects са класове, които използваме за пренос на информация между класовете или за изпращане през мрежата до клиента посредством HTTP протокола. Този тип класове са изключително много на брой, поради факта, че почти всяка заявка изисква различен отговор и е нужно да се създава нов клас, на базата на исканата от клиента информация. Нуждата от този тип класове е, че изпращаме само тази информация, за която клиентът е пратил заявка, а не целите POJO обекти, в които се съдържа и чувствителна информация като телефонни номера, имейли и пароли.
6. POJOs или Plain Old Java Objects са класове, които описват 1:1 моделът на базата данни в Java. В тези класове са представени всички таблици и връзките между тях, техните колони и типове данни. Тези класове са маркирани с анотации, които описват, че класът представлява таблица в базата, а полетата му съответстват на колоните(Hibernate).
7. repositories или хранилищата са интерфейси, които наследяват интерфейса JpaRepository, благодарение на които се генерират огромен брой от SQL заявки под формата на методи на базата на POJO обектите без да ни се налага да пишем чист SQL код.
8. Имаме пакет exceptions, където се намират всички класове, описващи изключителни ситуации в услугата. Ползата от тези класове е при невалидни данни или действия, да се изведе подходящо съобщение, а не такова, автоматично генерирано от сървъра, което би било неразбираемо за потребителите.
9. Имаме пакет util, където стоят инструменти за валидация на данни, използвани от всички сървиси.
10. Имаме и пакет config, където се намират конфигурации, свързани външни иструменти, използвани за съхранение на данни(Amazon) или за представяне на функциалността на уеб услугата(Swagger) в по-четим вид за демонстрация.

**КОД**

Стартова точка на сървъра...

@SpringBootApplication  
public class SoundcloudApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(SoundcloudApplication.class, args);  
 }  
}

**Package controller**

UserController

@RestController  
public class UserController {  
 private final UserService userService;  
 private final SessionManager sessionManager;  
 private final TokenService tokenService;  
  
 @Autowired  
 public UserController(UserService userService, SessionManager sessionManager, TokenService tokenService) {  
 this.userService = userService;  
 this.sessionManager = sessionManager;  
 this.tokenService = tokenService;  
 }  
  
 @PostMapping("/users")  
 public UserProfileResponseDTO register(@RequestBody RegisterRequestUserDTO registerDTO, HttpSession session) {  
 User loggedUser = this.sessionManager.getLoggedUser(session);  
 if (loggedUser != null) {  
 throw new BadRequestException("You have to logout and then register again!");  
 }  
 return new UserProfileResponseDTO(this.userService.register(registerDTO));  
 }  
  
 @PostMapping("/login")  
 public UserProfileResponseDTO login(@RequestBody LoginRequestUserDTO loginDTO, HttpSession session) {  
 User loggedUser = this.sessionManager.getLoggedUser(session);  
 if (loggedUser != null) {  
 throw new BadRequestException("You already logged in!");  
 }  
 UserProfileResponseDTO responseDTO = new UserProfileResponseDTO(this.userService.login(loginDTO));  
 this.sessionManager.loginUser(session, responseDTO.getId());  
 return responseDTO;  
 }  
  
 @PostMapping("/logout")  
 public MessageDTO logout(HttpSession session) {  
 this.sessionManager.validateUser(session, "You have to login and then logout!");  
 this.sessionManager.logoutUser(session);  
 return new MessageDTO("You successfully logout!");  
 }  
  
 @PostMapping("/users/follow")  
 public FollowResponseUserDTO followUser(@RequestBody FollowRequestUserDTO followDTO, HttpSession session) {  
 User loggedUser = this.sessionManager.validateUser(session, "You have to login and then follow users!");  
 FollowResponseUserDTO followResponseDTO = new FollowResponseUserDTO(this.userService.followUser(followDTO, loggedUser));  
 followResponseDTO.setFollowedByMe(true);  
 followResponseDTO.setFollowing(loggedUser.getFollowed().size());  
 return followResponseDTO;  
 }  
  
 @PostMapping("/users/filter")  
 public List<FilterResponseUserWithoutPlaylistDTO> filterUsers(@RequestBody FilterRequestUserDTO filterUserDTO) {  
 return this.userService.filterUsers(filterUserDTO);  
 }  
  
 @PutMapping("/users")  
 public UserProfileResponseDTO updateProfile(@RequestBody UpdateRequestUserDTO updateDTO, HttpSession session) {  
 User loggedUser = this.sessionManager.validateUser(session, "You have to login and then update your profile!");  
 return new UserProfileResponseDTO(this.userService.updateProfile(updateDTO, loggedUser));  
 }  
  
 @DeleteMapping("/users")  
 public MessageDTO removeProfile(HttpSession session) {  
 User loggedUser = this.sessionManager.validateUser(session, "You have to login and then remove your profile!");  
 this.userService.removeProfile(loggedUser.getId());  
 return new MessageDTO("Your profile was removed!");  
 }  
  
 @DeleteMapping("/users/unfollow")  
 public FollowResponseUserDTO unfollowUser(@RequestBody FollowRequestUserDTO unfollowDTO, HttpSession session) {  
 User loggedUser = this.sessionManager.validateUser(session, "You have to login and then unfollow users!");  
 FollowResponseUserDTO unfollowResponseDTO = new FollowResponseUserDTO(this.userService.unfollowUser(unfollowDTO, loggedUser));  
 unfollowResponseDTO.setFollowedByMe(false);  
 unfollowResponseDTO.setFollowing(loggedUser.getFollowed().size());  
 return unfollowResponseDTO;  
 }  
  
 @GetMapping("/users")  
 public UserProfileResponseDTO viewProfile(HttpSession session) {  
 User loggedUser = this.sessionManager.validateUser(session, "You have to login and then view your profile!");  
 return new UserProfileResponseDTO(this.userService.viewMyProfile(loggedUser.getId()));  
 }  
  
 @GetMapping("/users/{username}")  
 public List<FilterResponseUserDTO> userInformation(@PathVariable String username) {  
 return new ArrayList<>(this.userService.userInformation(username));  
 }  
  
 @GetMapping("/verify/{token}")  
 public MessageDTO verify(@PathVariable String token, HttpSession session) {  
 User loggedUser = this.sessionManager.getLoggedUser(session);  
 this.tokenService.confirmToken(token, loggedUser);  
 return new MessageDTO("Email confirmed!");  
 }  
}

PlaylistController

@RestController  
public class PlaylistController {  
 private final PlaylistService playlistService;  
 private final SessionManager sessionManager;  
  
 @Autowired  
 public PlaylistController(PlaylistService userService, SessionManager sessionManager) {  
 this.playlistService = userService;  
 this.sessionManager = sessionManager;  
 }  
  
 @PostMapping("/playlists")  
 public PlaylistResponseDTO addPlaylist(@RequestBody AddPlaylistRequestDTO playlistDTO, HttpSession session) {  
 User loggedUser = this.sessionManager.validateUser(session, "You have to login and then add a playlist!");  
 return new PlaylistResponseDTO(this.playlistService.addPlaylist(playlistDTO.getName(), loggedUser));  
 }  
  
 @DeleteMapping("/playlists/{playlistID}")  
 public MessageDTO removePlaylist(@PathVariable int playlistID, HttpSession session) {  
 User loggedUser = this.sessionManager.validateUser(session, "You have to login and then remove a playlist!");  
 this.playlistService.removePlaylist(playlistID, loggedUser);  
 return new MessageDTO("The playlist was successfully removed!");  
 }  
  
 @DeleteMapping("/playlists/songs")  
 public PlaylistResponseDTO removeSongFromPlaylist(@RequestBody SongToPlaylistRequestDTO removeSongDTO, HttpSession session) {  
 User loggedUser = this.sessionManager.validateUser(session, "You have to login and then remove a song from playlist!");  
 return new PlaylistResponseDTO(this.playlistService.removeSongFromPlaylist(removeSongDTO, loggedUser));  
 }  
  
 @PutMapping("/playlists/songs")  
 public PlaylistResponseDTO addSongToPlaylist(@RequestBody SongToPlaylistRequestDTO addSongDTO, HttpSession session) {  
 User loggedUser = this.sessionManager.validateUser(session, "You have to login and then add a song to playlist!");  
 return new PlaylistResponseDTO(this.playlistService.addSongToPlaylist(addSongDTO, loggedUser));  
 }  
  
 @PutMapping("/playlists")  
 public PlaylistResponseDTO updatePlaylistName(@RequestBody UpdatePlaylistNameDTO updateNameDTO, HttpSession session) {  
 User loggedUser = this.sessionManager.validateUser(session, "You have to login and then update your playlist!");  
 return new PlaylistResponseDTO(this.playlistService.updatePlaylistName(updateNameDTO, loggedUser));  
 }  
  
 @GetMapping("/playlists/{playlistID}")  
 public PlaylistResponseDTO getPlaylistSongs(@PathVariable int playlistID) {  
 return new PlaylistResponseDTO(this.playlistService.getPlaylistSongs(playlistID));  
 }  
  
 @GetMapping("/users/{username}/playlists")  
 public List<PlaylistResponseDTO> getUserPlaylists(@PathVariable String username) {  
 List<PlaylistResponseDTO> responsePlaylist = new ArrayList<>();  
 for (Playlist playlist : this.playlistService.getUserPlaylists(username)) {  
 responsePlaylist.add(new PlaylistResponseDTO(playlist));  
 }  
 return responsePlaylist;  
 }  
}

SongController

@RestController  
public class SongController {  
 private final SessionManager sessionManager;  
 private final SongService songService;  
  
 @Autowired  
 public SongController(SongService songService, SessionManager sessionManager) {  
 this.sessionManager = sessionManager;  
 this.songService = songService;  
 }  
  
 @PostMapping("/songs")  
 public SongUploadResponseDTO upload (@RequestPart MultipartFile file, @RequestPart String name, HttpSession session) {  
 User loggedUser = this.sessionManager.validateUser(session, "You must login to upload a song.");  
 Song uploadedSong = this.songService.uploadSong(name, file, loggedUser);  
 return new SongUploadResponseDTO(uploadedSong);  
 }  
  
 @PostMapping("songs/filter")  
 public List<SongFilterResponseDTO> filterSongs(@RequestBody SongFilterRequestDTO searchRequest) {  
 return this.songService.filterSongs(searchRequest);  
 }  
  
 @DeleteMapping("/songs")  
 public MessageDTO delete(@RequestBody ResourceRequestDTO requestJson, HttpSession session) throws BadRequestException {  
 User loggedUser = this.sessionManager.getLoggedUser(session);  
 Integer songId = requestJson.getResourceId();  
  
 if(loggedUser == null) {  
 throw new AuthenticationException("You must login to delete a song.");  
 } else if(songId == null) {  
 throw new BadRequestException("You must choose a song to delete.");  
 } else if(this.songService.getOwnerForSongId(songId).getId() != loggedUser.getId() ) {  
 throw new AuthenticationException("You cannot delete songs uploaded by other users.");  
 } else{  
 this.songService.deleteSong(songId);  
 return new MessageDTO("You have successfully deleted song id#" + songId);  
 }  
 }  
  
 @GetMapping("/songs/{id}/info")  
 public SongGetResponseDTO getById (@PathVariable int id) {  
 return new SongGetResponseDTO(this.songService.getById(id));  
 }  
  
 @GetMapping(value = "/songs/{id}", produces = "audio/mpeg")  
 public byte[] playSong (@PathVariable int id) {  
 return this.songService.playSong(id);  
 }  
  
 @GetMapping("songs/by-user/{username}")  
 public List<SongFilterResponseDTO> getByUsername(@PathVariable String username) {  
 return this.songService.getByUsername(username);  
 }  
  
 @GetMapping("songs/by-user/{username}/liked")  
 public List<SongFilterResponseDTO> getLikedSongs(@PathVariable String username) {  
 return this.songService.getLikedSongsByUsername(username);  
 }  
  
 @PutMapping("/songs/{songId}/set-like-status")  
 public LikeDislikeResponseDTO setSongLikeStatus(HttpSession session, @RequestParam(name = "value") int likeValue, @PathVariable int songId) {  
 User loggedUser = this.sessionManager.validateUser(session, "You must login to like/dislike a song.");  
 return this.songService.setLike(songId, likeValue, loggedUser);  
 }  
}

CommentController

@RestController  
public class CommentController {  
 private final SessionManager sessionManager;  
 private final CommentService commentService;  
  
 @Autowired  
 public CommentController(CommentService service, SessionManager sessionManager) {  
 this.sessionManager = sessionManager;  
 this.commentService = service;  
 }  
  
 @PostMapping("/comments")  
 public MessageDTO postComment(@RequestBody PostCommentRequestDTO requestDTO, HttpSession session) {  
 User loggedUser = sessionManager.validateUser(session, "You must login before posting a comment.");  
 return commentService.postComment(requestDTO, loggedUser);  
 }  
  
 @DeleteMapping("/comments")  
 public MessageDTO deleteComment(@RequestBody ResourceRequestDTO requestDTO, HttpSession session) {  
 User loggedUser = sessionManager.validateUser(session, "You must login in order to delete a comment.");  
 return commentService.deleteComment(requestDTO, loggedUser);  
 }  
  
 @GetMapping("/comments/by-song/{songId}")  
 public List<CommentResponseDTO> getCommentBySong(@PathVariable int songId) {  
 return commentService.getCommentBySong(songId);  
 }  
  
 @GetMapping("/comments/{commentId}")  
 public CommentResponseDTO getCommentById(@PathVariable int commentId) {  
 return commentService.getCommentById(commentId);  
 }  
  
 @PutMapping("comments/{commentId}/set-like-status")  
 public LikeDislikeResponseDTO setCommentLikeStatus(HttpSession session, @RequestParam(name = "value") int likeValue, @PathVariable int commentId) {  
 User loggedUser = sessionManager.validateUser(session, "You must login to like/dislike a comment.");  
 return commentService.setLike(commentId, likeValue, loggedUser);  
 }  
  
 @PutMapping("comments/{commentId}")  
 public CommentResponseDTO editComment(HttpSession session, @PathVariable int commentId, @RequestBody EditCommentRequestDTO requestDTO) {  
 User loggedUser = sessionManager.validateUser(session, "You must login to edit a comment.");  
 return commentService.editComment(loggedUser, commentId, requestDTO);  
 }  
}

ExceptionController

@RestControllerAdvice  
public class ExceptionController {  
  
 @ExceptionHandler(BadRequestException.class)  
 @ResponseStatus(HttpStatus.*BAD\_REQUEST*)  
 public MessageDTO handleBadRequest(BadRequestException e) {  
 return new MessageDTO(e.getMessage());  
 }  
  
 @ExceptionHandler(AuthenticationException.class)  
 @ResponseStatus(HttpStatus.*UNAUTHORIZED*)  
 public MessageDTO handleNotAuthorized(AuthenticationException e) {  
 return new MessageDTO(e.getMessage());  
 }  
  
 @ExceptionHandler(NotFoundException.class)  
 @ResponseStatus(HttpStatus.*NOT\_FOUND*)  
 public MessageDTO handleNotFoundResource(NotFoundException e) {  
 return new MessageDTO(e.getMessage());  
 }  
  
 @ExceptionHandler(FileReadWriteException.class)  
 @ResponseStatus(HttpStatus.*INTERNAL\_SERVER\_ERROR*)  
 public MessageDTO handleFileException(FileReadWriteException e) {  
 return new MessageDTO(e.getMessage());  
 }  
  
 @ExceptionHandler(SQLException.class)  
 @ResponseStatus(HttpStatus.*INTERNAL\_SERVER\_ERROR*)  
 public MessageDTO handleSQLException(SQLException e) {  
 return new MessageDTO("Unable to retrieve results from the database. " + e.getMessage());  
 }  
}

SessionManager

@Component  
public class SessionManager {  
 private static final String *LOGGED\_USER\_ID* = "LOGGED\_USER\_ID";  
 private final UserRepository userRepository;  
  
 @Autowired  
 public SessionManager(UserRepository repository) {  
 this.userRepository = repository;  
 }  
  
 public User getLoggedUser(HttpSession session) {  
 if (session.getAttribute(*LOGGED\_USER\_ID*) == null) {  
 return null;  
 }  
 int userId = (int) session.getAttribute(*LOGGED\_USER\_ID*);  
 return this.userRepository.findUserById(userId);  
 }  
  
 public User validateUser(HttpSession session, String message){  
 User loggedUser = getLoggedUser(session);  
 if(loggedUser == null){  
 throw new AuthenticationException(message);  
 }  
 return loggedUser;  
 }  
  
 public void loginUser(HttpSession ses, int id) {  
 ses.setAttribute(*LOGGED\_USER\_ID*, id);  
 }  
  
 public void logoutUser(HttpSession ses) {  
 ses.invalidate();  
 }  
}

**Package service**

UserService

@Service  
public class UserService {  
 private final UserRepository userRepository;  
 private final VerificationTokenRepository tokenRepository;  
 private final EmailService emailService;  
 private final UserDAO userDAO;  
 private final static String *FILTER\_BY\_SONGS* = "songs";  
 private final static String *FILTER\_BY\_COMMENTS* = "comments";  
 private final static String *FILTER\_BY\_PLAYLISTS* = "playlists";  
 private final static String *FILTER\_BY\_FOLLOWERS* = "followers";  
  
 @Autowired  
 public UserService(UserRepository userRepository, VerificationTokenRepository tokenRepository, UserDAO userDAO, EmailService emailService) {  
 this.userRepository = userRepository;  
 this.tokenRepository = tokenRepository;  
 this.emailService = emailService;  
 this.userDAO = userDAO;  
 Validator.*userRepository* = this.userRepository;  
 }  
  
 public void validateUser(User user) {  
 if (user == null) {  
 throw new NotFoundException("User not found!");  
 }  
 }  
  
 @Transactional  
 public User register(RegisterRequestUserDTO registerDTO) {  
 if (!Validator.*validateName*(registerDTO.getUsername())) {  
 throw new BadRequestException("Username format is not correct!");  
 }  
 if (!Validator.*validatePassword*(registerDTO.getPassword())) {  
 throw new BadRequestException("Password format is not correct!");  
 }  
 if (!Validator.*validateEmail*(registerDTO.getEmail())) {  
 throw new BadRequestException("Email format is not correct!");  
 }  
 if (this.userRepository.findUserByUsername(registerDTO.getUsername()) != null) {  
 throw new BadRequestException("Username already exists!");  
 }  
 if (!registerDTO.getPassword().equals(registerDTO.getConfirmPassword())) {  
 throw new BadRequestException("Passwords are not equals!");  
 }  
 if (this.userRepository.findUserByEmail(registerDTO.getEmail()) != null) {  
 throw new BadRequestException("Email already exists!");  
 }  
 PasswordEncoder encoder = new BCryptPasswordEncoder();  
 registerDTO.setPassword(encoder.encode(registerDTO.getPassword()));  
  
 User user = new User(registerDTO);  
 VerificationToken token = new VerificationToken(user);  
 user = this.userRepository.save(user);  
 this.tokenRepository.save(token);  
  
 Thread emailThread = new Thread(() -> this.emailService.send(registerDTO.getEmail(), token.getToken()));  
 emailThread.start();  
 return user;  
 }  
  
 public User login(LoginRequestUserDTO loginDTO) {  
 User user = this.userRepository.findUserByUsername(loginDTO.getUsername());  
 if (user != null) {  
 PasswordEncoder encoder = new BCryptPasswordEncoder();  
 if (encoder.matches(loginDTO.getPassword(), user.getPassword())) {  
 if (user.isEnabled()) {  
 return user;  
 }  
 throw new AuthenticationException("You must verify your email!");  
 }  
 }  
 throw new BadRequestException("Wrong credentials!");  
 }  
  
 @Transactional  
 public User followUser(FollowRequestUserDTO followDTO, User loggedUser) {  
 User user = this.userRepository.findUserById(followDTO.getUserID());  
 validateUser(user);  
 if (followDTO.getUserID() == loggedUser.getId()) {  
 throw new BadRequestException("You can`t un/follow yourself!");  
 }  
 if (user.getFollowers().contains(loggedUser)) {  
 throw new BadRequestException("You already follow " + user.getUsername() + "!");  
 }  
 user.getFollowers().add(loggedUser);  
 loggedUser.getFollowed().add(user);  
 return this.userRepository.save(user);  
 }  
  
 @Transactional  
 public User unfollowUser(FollowRequestUserDTO followDTO, User loggedUser) {  
 User user = this.userRepository.findUserById(followDTO.getUserID());  
 validateUser(user);  
 if (followDTO.getUserID() == loggedUser.getId()) {  
 throw new BadRequestException("You can`t un/follow yourself!");  
 }  
 if (!user.getFollowers().contains(loggedUser)) {  
 throw new BadRequestException("You do not follow " + user.getUsername() + "!");  
 }  
 user.getFollowers().remove(loggedUser);  
 loggedUser.getFollowed().remove(user);  
 return this.userRepository.save(user);  
 }  
  
 public User updateProfile(UpdateRequestUserDTO updateDTO, User loggedUser) {  
 Validator.*validateAge*(updateDTO.getAge(), loggedUser);  
 Validator.*updateUsername*(updateDTO.getUsername(), loggedUser);  
 Validator.*updatePassword*(updateDTO, loggedUser);  
 Validator.*updateEmail*(updateDTO.getEmail(), loggedUser);  
 return this.userRepository.save(loggedUser);  
 }  
  
 public List<FilterResponseUserDTO> userInformation(String username) {  
 List<User> users = this.userRepository.findUserByUsernameContains(username);  
 if(users == null){  
 throw new NotFoundException("No results!");  
 }  
 List<FilterResponseUserDTO> responseUsers = new ArrayList<>();  
 for(User user : users){  
 responseUsers.add(new FilterResponseUserDTO(user));  
 }  
 return responseUsers;  
 }  
  
 public User viewMyProfile(int userID) {  
 return this.userRepository.findUserById(userID);  
 }  
  
 @Transactional  
 public void removeProfile(int userID) {  
 this.userRepository.deleteUserById(userID);  
 }  
  
 @SneakyThrows  
 public List<FilterResponseUserWithoutPlaylistDTO> filterUsers(FilterRequestUserDTO filterUserDTO) {  
 if (!filterUserDTO.getOrderBy().equalsIgnoreCase(Order.*ASC*.toString())) {  
 if (!filterUserDTO.getOrderBy().equalsIgnoreCase(Order.*DESC*.toString())) {  
 throw new BadRequestException("Invalid order type!");  
 }  
 }  
 if (filterUserDTO.getPage() <= 0) {  
 throw new BadRequestException("Page number must be at least 1!");  
 }  
 if (filterUserDTO.getUsersPerPage() <= 0) {  
 throw new BadRequestException("Items per page must be at least 1!");  
 }  
 switch (filterUserDTO.getSortBy()) {  
 case *FILTER\_BY\_COMMENTS*:  
 case *FILTER\_BY\_FOLLOWERS*:  
 case *FILTER\_BY\_PLAYLISTS*:  
 case *FILTER\_BY\_SONGS*:  
 return this.userDAO.getFilteredUsers(filterUserDTO);  
 default:  
 throw new BadRequestException("Sort type not recognized!");  
 }  
 }  
}

PlaylistService

@Service  
public class PlaylistService {  
 private final PlaylistRepository playlistRepository;  
 private final SongRepository songRepository;  
  
 @Autowired  
 public PlaylistService(PlaylistRepository playlistRepository, SongRepository songRepository) {  
 this.playlistRepository = playlistRepository;  
 this.songRepository = songRepository;  
 }  
  
 public void validatePlaylist(Playlist playlist) {  
 if (playlist == null) {  
 throw new NotFoundException("Playlist not found!");  
 }  
 }  
  
 public Playlist addPlaylist(String name, User user) {  
 if (!Validator.*validateName*(name)) {  
 throw new BadRequestException("Playlist name format is not correct!");  
 }  
 if (this.playlistRepository.getPlaylistByName(name) != null) {  
 throw new BadRequestException("Playlist name already exists!");  
 }  
 return this.playlistRepository.save(new Playlist(name, user));  
 }  
  
 @Transactional  
 public void removePlaylist(int playlistID, User user) {  
 Playlist playlist = this.playlistRepository.getPlaylistById(playlistID);  
 validatePlaylist(playlist);  
 if (playlist.getOwner().getId() != user.getId()) {  
 throw new AuthenticationException("You can`t remove foreign playlist!");  
 }  
 this.playlistRepository.deleteById(playlistID);  
 }  
  
 @Transactional  
 public Playlist removeSongFromPlaylist(SongToPlaylistRequestDTO removeSongDTO, User user) {  
 Playlist playlist = this.playlistRepository.getPlaylistById(removeSongDTO.getPlaylistID());  
 validatePlaylist(playlist);  
 if (playlist.getOwner().getId() != user.getId()) {  
 throw new AuthenticationException("You can`t remove a song from foreign playlist!");  
 }  
 Song song = this.songRepository.getSongById(removeSongDTO.getSongID());  
 if (!playlist.getSongs().contains(song)) {  
 throw new NotFoundException("The song you are trying to remove not found!");  
 }  
 playlist.getSongs().remove(song);  
 song.getPlaylists().remove(playlist);  
 return this.playlistRepository.save(playlist);  
 }  
  
 public Playlist addSongToPlaylist(SongToPlaylistRequestDTO addSongDTO, User user) {  
 Playlist playlist = this.playlistRepository.getPlaylistById(addSongDTO.getPlaylistID());  
 validatePlaylist(playlist);  
 if (playlist.getOwner().getId() != user.getId()) {  
 throw new AuthenticationException("You can`t add a song to foreign playlist!");  
 }  
 Song song = this.songRepository.getSongById(addSongDTO.getSongID());  
 if (song == null) {  
 throw new NotFoundException("The song you are trying to add was not found!");  
 }  
 if (playlist.getSongs().contains(song)) {  
 throw new BadRequestException("The song is already added to this playlist!");  
 }  
 playlist.getSongs().add(song);  
 song.getPlaylists().add(playlist);  
 return this.playlistRepository.save(playlist);  
 }  
  
 public Playlist getPlaylistSongs(int playlistID) {  
 Playlist playlist = this.playlistRepository.getPlaylistById(playlistID);  
 validatePlaylist(playlist);  
 return playlist;  
 }  
  
 public List<Playlist> getUserPlaylists(String username) {  
 List<Playlist> userPlaylists = this.playlistRepository.getPlaylistsByOwner\_Username(username);  
 if (userPlaylists.isEmpty()) {  
 throw new NotFoundException("No playlists found for " + username + "!");  
 }  
 return userPlaylists;  
 }  
  
 public Playlist updatePlaylistName(UpdatePlaylistNameDTO updateNameDTO, User loggedUser) {  
 Playlist playlist = this.playlistRepository.getPlaylistById(updateNameDTO.getPlaylistID());  
 validatePlaylist(playlist);  
 if (playlist.getOwner().getId() != loggedUser.getId()) {  
 throw new AuthenticationException("You can`t update foreign playlist!");  
 }  
 if (!Validator.*validateName*(updateNameDTO.getName())) {  
 throw new BadRequestException("Playlist name format is not correct!");  
 }  
 if (this.playlistRepository.getPlaylistByName(updateNameDTO.getName()) != null) {  
 throw new BadRequestException("Playlist name already exists!");  
 }  
 playlist.setName(updateNameDTO.getName());  
 return this.playlistRepository.save(playlist);  
 }  
}

SongService

@Service  
public class SongService {  
 private static final String *STORAGE\_BUCKET\_NAME* = "amazon-storage-soundcloud";  
 private static final String *SORT\_BY\_LIKES* = "likes";  
 private static final String *SORT\_BY\_DISLIKES* = "dislikes";  
 private static final String *SORT\_BY\_COMMENTS* = "comments";  
 private static final String *SORT\_BY\_VIEWS* = "views";  
 private static final String *SORT\_BY\_DATE* = "date";  
 private static final String *SORT\_BY\_PLAYLISTS* = "playlists";  
 private static final int *FILTER\_RESULTS\_PER\_PAGE* = 5;  
 private static final long *MAXIMUM\_FILESIZE\_BYTES* = 10\*1024\*1024; // 10 mb  
  
 private final SongDAO songDAO;  
 private final SongRepository songRepository;  
 private final UserRepository userRepository;  
 private final AmazonS3 storageClient;  
  
 @Autowired  
 public SongService(SongDAO songDAO, SongRepository songRepository, UserRepository userRepository, AmazonS3 storageClient) {  
 this.songDAO = songDAO;  
 this.songRepository = songRepository;  
 this.userRepository = userRepository;  
 this.storageClient = storageClient;  
 }  
  
 public Song uploadSong(String title, MultipartFile receivedFile, User loggedUser) {  
 String originalName = receivedFile.getOriginalFilename();  
 String extension, fileName, fullName;  
  
 if (receivedFile.getSize() > *MAXIMUM\_FILESIZE\_BYTES*) {  
 throw new BadRequestException("File size too large.");  
 }  
 if (originalName == null || originalName.isBlank()) {  
 throw new BadRequestException("File name is empty.");  
 }  
 if (!originalName.contains(".")) {  
 throw new BadRequestException("Uploaded file does not have an extension.");  
 } else {  
 extension = originalName.substring(originalName.indexOf('.'));  
 }  
 if (title == null || this.songRepository.getSongByTitle(title) != null) {  
 throw new BadRequestException("A track with this title already exists.");  
 }  
 if (!extension.equals(".mp3")) {  
 throw new BadRequestException("Unrecognized file extension. Please select an mp3 file.");  
 }  
  
 fileName = String.*valueOf*(System.*nanoTime*());  
 fullName = fileName + extension;  
  
 Song song;  
 ObjectMetadata meta = new ObjectMetadata();  
 meta.setContentType("audio/mpeg");  
 meta.setContentLength(receivedFile.getSize());  
 try {  
 this.storageClient.putObject(*STORAGE\_BUCKET\_NAME*, fullName, receivedFile.getInputStream(), meta);  
 song = new Song(title, fullName, loggedUser);  
 this.songRepository.save(song);  
 } catch (AmazonServiceException | IOException e) {  
 throw new FileReadWriteException("Could not save song to server - " + e.getMessage());  
 }  
  
 return song;  
 }  
  
 public Song getById(int id) {  
 Song s = this.songRepository.getSongById(id);  
 if (s == null) {  
 throw new NotFoundException("Song with id " + id + " not found.");  
 } else {  
 return s;  
 }  
 }  
  
 public byte[] playSong(int id) {  
 Song s = getById(id);  
 if (s == null) throw new NotFoundException("The song you are trying to play was not found.");  
  
 try {  
 S3Object downloadedFile = this.storageClient.getObject(*STORAGE\_BUCKET\_NAME*, s.getUrl());  
 S3ObjectInputStream stream = downloadedFile.getObjectContent();  
  
 s.setViews(s.getViews() + 1);  
 this.songRepository.save(s);  
  
 return IOUtils.*toByteArray*(stream);  
 } catch (AmazonServiceException | IOException e) {  
 throw new FileReadWriteException("Could not download the song from the server. Details: " + e.getMessage());  
 }  
 }  
  
 public User getOwnerForSongId(int songId) {  
 Song song = getById(songId);  
 return song.getOwner();  
 }  
  
 public void deleteSong(int songId) {  
 SongGetResponseDTO dto = new SongGetResponseDTO(getById(songId));  
 try {  
 this.storageClient.deleteObject(*STORAGE\_BUCKET\_NAME*, dto.getUrl());  
 } catch (SdkClientException e) {  
 throw new FileReadWriteException("Failed to delete song file from the server. Details: " + e.getMessage());  
 }  
 this.songRepository.deleteById(songId);  
 }  
  
 public List<SongFilterResponseDTO> getByUsername(String username) {  
 User owner = this.userRepository.findUserByUsername(username);  
 if (owner == null) {  
 throw new NotFoundException("Could not find user " + username);  
 }  
  
 List<Song> songs = this.songRepository.getAllByOwner(owner);  
  
 return songs.stream().map(SongFilterResponseDTO::new).collect(Collectors.*toList*());  
 }  
  
 public List<SongFilterResponseDTO> getLikedSongsByUsername(String username) {  
 User user = this.userRepository.findUserByUsername(username);  
 if (user == null) {  
 throw new NotFoundException("Could not find user " + username);  
 }  
  
 List<Song> songs = user.getLikedSongs();  
  
 return songs.stream().map(SongFilterResponseDTO::new).collect(Collectors.*toList*());  
 }  
  
 private int getLikesSumFor (Song song) {  
 return song.getLikers().size() - song.getDislikers().size();  
 }  
  
 @Transactional  
 public LikeDislikeResponseDTO setLike(int songId, int likeValue, User loggedUser) {  
 Song targetSong = this.songRepository.getSongById(songId);  
 String action;  
  
 if (targetSong == null) {  
 throw new NotFoundException("The song you are trying to like or dislike was not found.");  
 }  
  
 switch (likeValue) {  
 case 1:  
 if (loggedUser.getLikedSongs().contains(targetSong)) return new LikeDislikeResponseDTO("Song left liked.", getLikesSumFor(targetSong));  
 if (loggedUser.getDislikedSongs().contains(targetSong)) setLike(songId, 0, loggedUser);  
 loggedUser.getLikedSongs().add(targetSong);  
 targetSong.getLikers().add(loggedUser);  
 action = "liked";  
 break;  
 case 0:  
 if (loggedUser.getLikedSongs().contains(targetSong)) {  
 // If song was previously liked  
 loggedUser.getLikedSongs().remove(targetSong);  
 targetSong.getLikers().remove(loggedUser);  
 action = "unliked";  
 } else if (loggedUser.getDislikedSongs().contains(targetSong)) {  
 // If song was previously disliked  
 loggedUser.getDislikedSongs().remove(targetSong);  
 targetSong.getDislikers().remove(loggedUser);  
 action = "undisliked";  
 } else {  
 // If song was previously neutral  
 return new LikeDislikeResponseDTO("Song status left at neutral.", getLikesSumFor(targetSong));  
 }  
 break;  
 case -1:  
 if (loggedUser.getDislikedSongs().contains(targetSong)) return new LikeDislikeResponseDTO("Song left disliked.", getLikesSumFor(targetSong));  
 if (loggedUser.getLikedSongs().contains(targetSong)) setLike(songId, 0, loggedUser);  
 loggedUser.getDislikedSongs().add(targetSong);  
 targetSong.getDislikers().add(loggedUser);  
 action = "disliked";  
 break;  
 default:  
 throw new BadRequestException("Invalid like status passed.");  
 }  
  
 this.userRepository.save(loggedUser);  
 this.songRepository.save(targetSong);  
 return new LikeDislikeResponseDTO("You successfully " + action + " song id#" + songId, getLikesSumFor(targetSong));  
 }  
  
 @SneakyThrows  
 public List<SongFilterResponseDTO> filterSongs(SongFilterRequestDTO searchRequest) {  
 String title = searchRequest.getTitle();  
 Integer page = searchRequest.getPage();  
  
 String sort = searchRequest.getSortBy();  
 if (sort == null) sort = "";  
 else sort = sort.toLowerCase().trim();  
  
 String order = searchRequest.getOrderBy();  
 if (order == null) order = "";  
 else order = order.toUpperCase().trim();  
  
 if (title == null) {  
 throw new BadRequestException("Trying to search without title.");  
 } else if (order.equals("")) {  
 order = "ASC";  
 } else if (!order.equals("ASC") && !order.equals("DESC")) {  
 throw new BadRequestException("Search order not recognized");  
 }  
  
 if (sort.trim().equals("")) sort = "date";  
 if (page == null) page = 1;  
  
 switch (sort) {  
 case *SORT\_BY\_PLAYLISTS*:  
 sort = "inPlaylists";  
 case *SORT\_BY\_LIKES*:  
 case *SORT\_BY\_DISLIKES*:  
 case *SORT\_BY\_VIEWS*:  
 case *SORT\_BY\_DATE*:  
 case *SORT\_BY\_COMMENTS*:  
 return this.songDAO.filterSongs(title, sort, order, page, *FILTER\_RESULTS\_PER\_PAGE*);  
 default:  
 throw new BadRequestException("Search method not supported.");  
 }  
 }  
}

CommentService

@Service  
public class CommentService {  
 private final CommentRepository commentRepository;  
 private final SongRepository songRepository;  
 private final UserRepository userRepository;  
  
 @Autowired  
 public CommentService(CommentRepository commentRepository, SongRepository songRepository, UserRepository userRepository) {  
 this.commentRepository = commentRepository;  
 this.songRepository = songRepository;  
 this.userRepository = userRepository;  
 }  
  
 @Transactional  
 public MessageDTO postComment(PostCommentRequestDTO requestDTO, User loggedUser) {  
 Song songCommented = this.songRepository.getSongById(requestDTO.getSongId());  
 if (songCommented == null) throw new NotFoundException("Cannot find song id#" + requestDTO.getSongId());  
  
 Comment comment = new Comment(requestDTO.getText());  
  
 // Add comment to song  
 songCommented.getComments().add(comment);  
 // Add comment to user  
 loggedUser.getComments().add(comment);  
  
 comment.setOwner(loggedUser);  
 comment.setSong(songCommented);  
  
 this.userRepository.save(loggedUser);  
 this.songRepository.save(songCommented);  
 this.commentRepository.save(comment);  
 return new MessageDTO("Comment posted. Comment id: #" + comment.getId());  
 }  
  
 public MessageDTO deleteComment(ResourceRequestDTO requestDTO, User loggedUser) {  
 Integer id = requestDTO.getResourceId();  
 if (id == null) throw new BadRequestException("Must select comment to delete by id.");  
 Comment comment = this.commentRepository.findCommentById(id);  
  
 if (comment == null) throw new NotFoundException("Comment id#" + id + " was not found.");  
 if (comment.getOwner().getId() != loggedUser.getId()) throw new AuthenticationException("Cannot delete comments from other users.");  
  
 this.commentRepository.delete(comment);  
 return new MessageDTO("Comment id#" + id + " successfully deleted.");  
 }  
  
 public List<CommentResponseDTO> getCommentBySong(int songId) {  
 Song song = this.songRepository.getSongById(songId);  
 if (song == null) throw new NotFoundException("Song " + songId + " not found.");  
 List<Comment> songComments = this.commentRepository.findCommentsBySong(song);  
  
 return songComments.stream().map(CommentResponseDTO::new).collect(Collectors.*toList*());  
 }  
  
 public CommentResponseDTO getCommentById(int commentId) {  
 Comment comment = this.commentRepository.findCommentById(commentId);  
 if (comment == null)

throw new NotFoundException("Comment " + commentId + " not found.");  
 return new CommentResponseDTO(comment);  
 }  
  
 private int getLikesSumFor (Comment comment) {  
 return comment.getLikers().size() - comment.getDislikers().size();  
 }  
  
 @Transactional  
 public LikeDislikeResponseDTO setLike(int commentId, int likeValue, User loggedUser) {  
 Comment targetComment = commentRepository.findCommentById(commentId);  
 String action;  
  
 if (targetComment == null) {  
 throw new NotFoundException("The comment you are trying to like/dislike not found.");  
 }  
  
 switch (likeValue) {  
 case 1:  
 if (loggedUser.getLikedComments().contains(targetComment)) {  
 return new LikeDislikeResponseDTO("Comment left liked.", getLikesSumFor(targetComment));  
 }  
 if (loggedUser.getDislikedComments().contains(targetComment)) {  
 setLike(commentId, 0, loggedUser);  
 }  
 loggedUser.getLikedComments().add(targetComment);  
 targetComment.getLikers().add(loggedUser);  
 action = "liked";  
 break;  
 case 0:  
 if (loggedUser.getLikedComments().contains(targetComment)) {  
 // If comment was previously liked  
 loggedUser.getLikedComments().remove(targetComment);  
 targetComment.getLikers().remove(loggedUser);  
 action = "unliked";  
 } else if (loggedUser.getDislikedComments().contains(targetComment)) {  
 // If comment was previously disliked  
 loggedUser.getDislikedComments().remove(targetComment);  
 targetComment.getDislikers().remove(loggedUser);  
 action = "undisliked";  
 } else {  
 // If comment was previously neutral  
 return new LikeDislikeResponseDTO("Comment status left at neutral.", getLikesSumFor(targetComment));  
 }  
 break;  
 case -1:  
 if (loggedUser.getDislikedComments().contains(targetComment))  
 return new LikeDislikeResponseDTO("Comment left disliked.", getLikesSumFor(targetComment));  
 if (loggedUser.getLikedComments().contains(targetComment)) setLike(commentId, 0, loggedUser);  
 loggedUser.getDislikedComments().add(targetComment);  
 targetComment.getDislikers().add(loggedUser);  
 action = "disliked";  
 break;  
 default:  
 throw new BadRequestException("Invalid like status passed.");  
 }  
 this.userRepository.save(loggedUser);  
 this.commentRepository.save(targetComment);  
 return new LikeDislikeResponseDTO("You successfully " + action + " comment id#" + commentId, getLikesSumFor(targetComment));  
 }  
  
 public CommentResponseDTO editComment(User loggedUser, int commentId, EditCommentRequestDTO requestDTO) {  
 Comment comment = this.commentRepository.findCommentById(commentId);  
  
 if (comment == null) throw new NotFoundException("Comment id#" + commentId + " was not found.");  
 if (loggedUser.getId() != comment.getOwner().getId()) throw new AuthenticationException("You cannot edit comments from other users.");  
  
 comment.setText(requestDTO.getText());  
 this.commentRepository.save(comment);  
 return new CommentResponseDTO(comment);  
 }  
}

EmailService

@Service  
@AllArgsConstructor  
public class EmailService {  
 private final JavaMailSender mailSender;  
  
 public void send(String mail, String token) {  
 SimpleMailMessage message = new SimpleMailMessage();  
 message.setTo(mail);  
 message.setSubject("Please verify your registration:<br>");  
 String verifyURL = "localhost:7878/verify/" + token;  
 String content = "Please follow http://" + verifyURL + " this link";  
 message.setText(content);  
 this.mailSender.send(message);  
 }  
}

TokenService

@Service  
public class TokenService {  
 private final VerificationTokenRepository tokenRepository;  
  
 @Autowired  
 public TokenService(VerificationTokenRepository tokenRepository) {  
 this.tokenRepository = tokenRepository;  
 }  
  
 @Transactional  
 public void confirmToken(String userToken, User user) {  
 VerificationToken token = this.tokenRepository.findByToken(userToken);  
 if(token == null){  
 throw new BadRequestException("Confirmation details are wrong!");  
 }  
 if(token.getUser().isEnabled()){  
 throw new BadRequestException("Your email has already been confirmed!");  
 }  
 if(user != null){  
 throw new BadRequestException("You do not have a permission to validate foreign email!");  
 }  
 if(token.getExpiresAt().compareTo(LocalDateTime.*now*()) < 0){  
 throw new NotFoundException("Verification token expired...");  
 }  
 token.getUser().setEnabled(true);  
 token.setConfirmedAt(LocalDateTime.*now*());  
 }  
}

**Package model**

DAOs

UserDAO

@Component  
public class UserDAO {  
 private final JdbcTemplate jdbcTemplate;  
  
 @Autowired  
 public UserDAO(JdbcTemplate jdbcTemplate) {  
 this.jdbcTemplate = jdbcTemplate;  
 }  
  
 public List<FilterResponseUserWithoutPlaylistDTO> getFilteredUsers(FilterRequestUserDTO filterUserDTO) throws SQLException {  
 List<FilterResponseUserWithoutPlaylistDTO> filteredUsers = new ArrayList<>();  
 String filterQuery = "SELECT u.id, u.username, " +  
 "COUNT(DISTINCT ufu.follower\_id) AS followers, " +  
 "COUNT(DISTINCT c.id) AS comments, " +  
 "COUNT(DISTINCT s.id) AS songs " +  
  
 "FROM users u Left JOIN users\_follow\_users ufu " +  
 "ON u.id = ufu.followed\_id " +  
 "LEFT JOIN comments c " +  
 "ON u.id = c.owner\_id " +  
 "LEFT JOIN songs s " +  
 "ON u.id = s.owner\_id " +  
  
 "GROUP BY u.id " +  
 "ORDER BY " + filterUserDTO.getSortBy() + " " + filterUserDTO.getOrderBy() + " " +  
 "LIMIT " + filterUserDTO.getUsersPerPage() + " " +  
 "OFFSET " + (filterUserDTO.getUsersPerPage() \* (filterUserDTO.getPage() - 1)); // starts from 0  
 // take 4 rows per query depends on page given by user  
 // e.g. page 4 -> LIMIT 4 OFFSET 4 \* (4-1) -> rows 12, 13, 14, 15  
 // page 1 - 0,1,2,3; page 2 - 4,5,6,7; page 3 - 8,9,10,11; ....  
  
 DataSource dataSource = this.jdbcTemplate.getDataSource();  
 if (dataSource != null) {  
 try (Connection connection = dataSource.getConnection();  
 PreparedStatement pr = connection.prepareStatement(filterQuery))  
 {  
 ResultSet set = pr.executeQuery();  
 while (set.next()) {  
 FilterResponseUserWithoutPlaylistDTO filteredUser = new FilterResponseUserWithoutPlaylistDTO(  
 set.getInt("id"),  
 set.getString("username"),  
 set.getInt("songs"),  
 set.getInt("comments"),  
 set.getInt("followers"));  
 filteredUsers.add(filteredUser);  
 }  
 }  
 return filteredUsers;  
 }  
 throw new SQLException("Connection to DB failed!");  
 }  
}

SongDAO

@Component  
public class SongDAO {  
 private final JdbcTemplate jdbcTemplate;  
  
 @Autowired  
 public SongDAO(JdbcTemplate jdbcTemplate) {  
 this.jdbcTemplate = jdbcTemplate;  
 }  
  
 public List<SongFilterResponseDTO> filterSongs(String queryTitle, String sort, String order, int page, int resultsPerPage) throws SQLException {  
 List<SongFilterResponseDTO> songs = new LinkedList<>();  
  
 String sql = "SELECT sng.title, " +  
 "usr.username AS uploadedBy, " +  
 "sng.id AS songId, " +  
 "sng.views, " +  
 "COUNT(DISTINCT cmt.id) AS comments, " +  
 "COUNT(DISTINCT uls.user\_id) AS likes, " +  
 "COUNT(DISTINCT uds.user\_id) AS dislikes, " +  
 "COUNT(DISTINCT phs.playlist\_id) AS inPlaylists, " +  
 "sng.created\_at AS date " +  
  
 "FROM songs sng " +  
 "LEFT JOIN users usr ON sng.owner\_id = usr.id " +  
 "LEFT JOIN comments cmt ON sng.id = cmt.song\_id " +  
 "LEFT JOIN users\_like\_songs uls ON sng.id = uls.song\_id " +  
 "LEFT JOIN users\_dislike\_songs uds ON sng.id = uds.song\_id " +  
 "LEFT JOIN playlists\_have\_songs phs ON sng.id = phs.song\_id " +  
  
 "WHERE sng.title LIKE \"%%%s%%\" " +  
  
 "GROUP BY sng.id " +  
  
 "ORDER BY %s %s " +  
  
 "LIMIT %d OFFSET %d";  
  
 sql = String.*format*(sql, queryTitle, sort, order, resultsPerPage, (resultsPerPage \* (page - 1)));  
 DataSource dataSource = this.jdbcTemplate.getDataSource();  
 if (dataSource != null) {  
 try (Connection connection = dataSource.getConnection();  
 PreparedStatement statement = connection.prepareStatement(sql))  
 {  
 ResultSet results = statement.executeQuery();  
 while (results.next()) {  
 String title = results.getString("title");  
 String uploadedBy = results.getString("uploadedBy");  
 LocalDateTime uploadDate = results.getTimestamp("date").toLocalDateTime();  
 int songId = results.getInt("songId");  
 int views = results.getInt("views");  
 int comments = results.getInt("comments");  
 int likes = results.getInt("likes");  
 int dislikes = results.getInt("dislikes");  
 int inPlaylists = results.getInt("inPlaylists");  
 songs.add(new SongFilterResponseDTO(title, uploadedBy, uploadDate, songId, views, comments, likes, dislikes, inPlaylists));  
 }  
 }  
 return songs;  
 }  
 throw new SQLException("Connection to DB failed!");  
 }  
}

POJOs

User

@Getter  
@Setter  
@Entity  
@NoArgsConstructor  
@Table(name = "users")  
public class User {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private int id;  
 private String username;  
 private String password;  
 private String email;  
 private int age;  
 private boolean enabled;  
 private LocalDateTime createdAt;  
 @OneToMany(mappedBy = "owner")  
 @JsonBackReference  
 private List<Comment> comments = new ArrayList<>();  
 @OneToMany(mappedBy = "owner")  
 @JsonBackReference  
 private List<Song> songs = new ArrayList<>();  
 @OneToMany(mappedBy = "owner")  
 @JsonBackReference  
 private List<Playlist> playlists = new ArrayList<>();  
  
 @ManyToMany(mappedBy = "likers")  
 @JsonBackReference  
 private List<Comment> likedComments;  
  
 @ManyToMany(mappedBy = "dislikers")  
 @JsonBackReference  
 private List<Comment> dislikedComments;  
  
 @ManyToMany(mappedBy = "likers")  
 @JsonBackReference  
 private List<Song> likedSongs;  
  
 @ManyToMany(mappedBy = "dislikers")  
 @JsonBackReference  
 private List<Song> dislikedSongs;  
  
 @ManyToMany(cascade = CascadeType.*MERGE*, fetch = FetchType.*LAZY*)  
 @JoinTable(name="users\_follow\_users",  
 joinColumns={@JoinColumn(name="followed\_id")},  
 inverseJoinColumns={@JoinColumn(name="follower\_id")})  
 @JsonManagedReference  
 private List<User> followers = new ArrayList<>();  
  
 @ManyToMany(mappedBy = "followers")  
 @JsonBackReference  
 private List<User> followed = new ArrayList<>();  
  
 public User(RegisterRequestUserDTO userDTO) {  
 this.username = userDTO.getUsername();  
 this.password = userDTO.getPassword();  
 this.email = userDTO.getEmail();  
 this.age = userDTO.getAge();  
 this.createdAt = LocalDateTime.*now*();  
 }  
}

Playlist

@Getter  
@Setter  
@Entity  
@NoArgsConstructor  
@Table(name = "playlists")  
public class Playlist {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private int id;  
 private String name;  
 @ManyToOne  
 @JoinColumn(name="owner\_id")  
 @JsonManagedReference  
 private User owner;  
 private LocalDateTime createdAt;  
  
 @ManyToMany(cascade = CascadeType.*MERGE*, fetch = FetchType.*LAZY*)  
 @JoinTable(  
 name = "playlists\_have\_songs",  
 joinColumns = {@JoinColumn(name="playlist\_id")},  
 inverseJoinColumns = {@JoinColumn(name="song\_id")}  
 )  
 @JsonManagedReference  
 private List<Song> songs = new ArrayList<>();  
  
 public Playlist(String name, User owner){  
 this.name = name;  
 this.owner = owner;  
 this.createdAt = LocalDateTime.*now*();  
 }  
}

Song

@Getter  
@Setter  
@Entity  
@NoArgsConstructor  
@Table(name = "songs")  
public class Song {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private int id;  
 private String title;  
 private String url;  
 private int views;  
 @ManyToOne  
 @JoinColumn(name = "owner\_id")  
 @JsonManagedReference  
 private User owner;  
 private LocalDateTime createdAt;  
 @OneToMany(mappedBy = "song")  
 @JsonManagedReference  
 private List<Comment> comments;  
 @ManyToMany(mappedBy = "songs")  
 @JsonBackReference  
 private List<Playlist> playlists;  
  
 @ManyToMany  
 @JoinTable(  
 name = "users\_like\_songs",  
 joinColumns = {@JoinColumn(name="song\_id")},  
 inverseJoinColumns = {@JoinColumn(name="user\_id")}  
 )  
 @JsonManagedReference  
 private List<User> likers;  
  
 @ManyToMany  
 @JoinTable(  
 name = "users\_dislike\_songs",  
 joinColumns = {@JoinColumn(name="song\_id")},  
 inverseJoinColumns = {@JoinColumn(name="user\_id")}  
 )  
 @JsonManagedReference  
 private List<User> dislikers;  
  
 public Song(String title, String url, User owner) {  
 this.title = title;  
 this.url = url;  
 this.views = 0;  
 this.owner = owner;  
 this.createdAt = LocalDateTime.*now*();  
 }  
}

Comment

@Getter  
@Setter  
@Entity  
@NoArgsConstructor  
@Table(name="comments")  
public class Comment {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private int id;  
 private String text;  
 @ManyToOne  
 @JoinColumn(name = "owner\_id")  
 @JsonManagedReference  
 private User owner;  
 @ManyToOne  
 @JoinColumn(name = "song\_id")  
 @JsonBackReference  
 private Song song;  
 private LocalDateTime createdAt;  
  
 @ManyToMany  
 @JoinTable(  
 name="users\_like\_comments",  
 joinColumns = {@JoinColumn(name = "comment\_id")},  
 inverseJoinColumns = {@JoinColumn(name="user\_id")}  
 )  
 @JsonManagedReference  
 private List<User> likers;  
  
 @ManyToMany  
 @JoinTable(  
 name="users\_dislike\_comments",  
 joinColumns = {@JoinColumn(name = "comment\_id")},  
 inverseJoinColumns = {@JoinColumn(name="user\_id")}  
 )  
 @JsonManagedReference  
 private List<User> dislikers;  
  
 public Comment(String text) {  
 this.text = text;  
 this.createdAt = LocalDateTime.*now*();  
 }  
}

VerificationToken

@Getter  
@Setter  
@Entity  
@NoArgsConstructor  
@Table(name="verification\_token")  
public class VerificationToken {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private long id;  
 private String token;  
 private LocalDateTime createdAt;  
 private LocalDateTime expiresAt;  
 private LocalDateTime confirmedAt;  
  
 @JoinColumn(nullable = false, name = "user\_id")  
 @ManyToOne  
 private User user;  
  
 public VerificationToken(User user) {  
 this.token = UUID.*randomUUID*().toString();  
 this.createdAt = LocalDateTime.*now*();  
 this.expiresAt = this.createdAt.plusMinutes(15);  
 this.user = user;  
 }  
}

Repositories

UserRepository

@Repository  
public interface UserRepository extends JpaRepository<User, Integer> {  
 User findUserByEmail(String email);  
  
 List<User> findUserByUsernameContains(String username);  
  
 User findUserByUsername(String username);  
  
 User findUserById(int id);  
  
 void deleteUserById(int id);  
}

PlaylistRepository

@Repository  
public interface PlaylistRepository extends JpaRepository<Playlist, Integer> {  
 Playlist getPlaylistById(int id);  
  
 Playlist getPlaylistByName(String name);  
  
 List<Playlist> getPlaylistsByOwner\_Username(String username);  
}

SongRepository

@Repository  
public interface SongRepository extends JpaRepository<Song, Integer> {  
 Song getSongById(int id);  
  
 Song getSongByTitle(String title);  
  
 List<Song> getAllByOwner(User owner);  
}

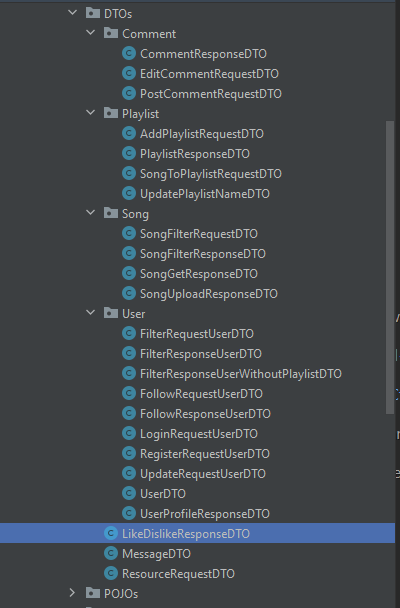
CommentRepository

@Repository  
public interface CommentRepository extends JpaRepository<Comment, Integer> {  
 Comment findCommentById(int id);  
  
 List<Comment> findCommentsBySong(Song song);  
}

VerificationToken

@Repository  
public interface VerificationTokenRepository extends JpaRepository<VerificationToken, Long> {  
 VerificationToken findByToken(String token);  
}

DTOs



**Package exceptions**

AuthenticationException

public class AuthenticationException extends RuntimeException{  
 public AuthenticationException(String msg){  
 super(msg);  
 }  
}

BadRequestException

public class BadRequestException extends RuntimeException {  
 public BadRequestException(String msg) {  
 super(msg);  
 }  
}

FileReadWriteException

public class FileReadWriteException extends RuntimeException {  
 public FileReadWriteException(String message) {  
 super(message);  
 }  
}

NotFoundException

public class NotFoundException extends RuntimeException{  
 public NotFoundException(String msg){  
 super(msg);  
 }  
}

**Package util**

Validator

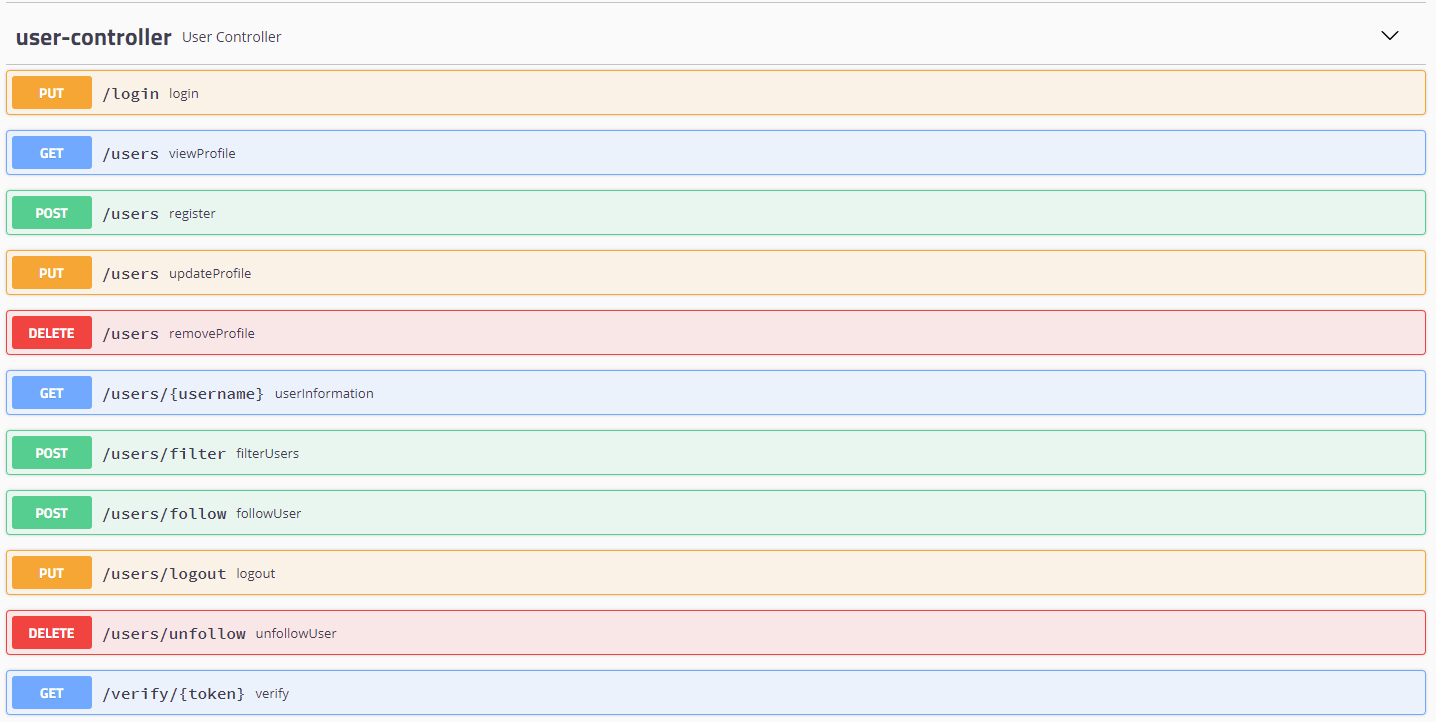
public class Validator {  
 private static final String *PASSWORD\_PATTERN* = "(?=.\*[0**-**9])(?=.\*[a**-**z])(?=.\*[A**-**Z])(?=.\*[@#$%^\*&+=]).{8,}";  
 private static final String *EMAIL\_PATTERN* = "^[a**-**z]+[A**-**Za**-**z0**-**9\_.-]{2,}+@[a**-**z]{2,20}+\\.[a**-**z]{2,20}";  
 private static final String *USERNAME\_PATTERN* = "^[A**-**Za**-**z]\\w{2,29}$";  
 public static UserRepository *userRepository*;  
  
 public static void updateUsername(String username, User loggedUser) {  
 if (!username.equals("")) {  
 if (!*validateName*(username)) {  
 throw new BadRequestException("Username format is not correct!");  
 }  
 if (*userRepository*.findUserByUsername(username) != null) {  
 throw new BadRequestException("Username already exists!");  
 }  
 loggedUser.setUsername(username);  
 }  
 }  
  
 public static void updatePassword(UpdateRequestUserDTO updateDTO, User loggedUser) {  
 PasswordEncoder encoder = new BCryptPasswordEncoder();  
 if (!updateDTO.getOldPassword().equals("")) {  
 if (!*validatePassword*(updateDTO.getNewPassword())) {  
 throw new BadRequestException("Password format is not correct!");  
 }  
 if (!updateDTO.getNewPassword().equals(updateDTO.getConfirmPassword())) {  
 throw new BadRequestException("Passwords are not equals!");  
 }  
 if (!encoder.matches(updateDTO.getOldPassword(), loggedUser.getPassword())) {  
 throw new BadRequestException("Old password is incorrect!");  
 }  
 loggedUser.setPassword(encoder.encode(updateDTO.getConfirmPassword()));  
 }  
 }  
  
 public static void updateEmail(String email, User loggedUser) {  
 if (!email.equals("")) {  
 if (!*validateEmail*(email)) {  
 throw new BadRequestException("Email format is not correct!");  
 }  
 if (*userRepository*.findUserByEmail(email) != null) {  
 throw new BadRequestException("Email already exists!");  
 }  
 loggedUser.setEmail(email);  
 }  
 }  
 public static boolean validateName(String username) {  
 Pattern pattern = Pattern.*compile*(*USERNAME\_PATTERN*);  
 Matcher matcher = pattern.matcher(username);  
  
 return matcher.matches();  
 }  
 public static boolean validatePassword(String password) {  
 Pattern pattern = Pattern.*compile*(*PASSWORD\_PATTERN*);  
 Matcher matcher = pattern.matcher(password);  
  
 return matcher.matches();  
 }  
  
 public static boolean validateEmail(String email) {  
 Pattern pattern = Pattern.*compile*(*EMAIL\_PATTERN*);  
 Matcher matcher = pattern.matcher(email);  
  
 return matcher.matches();  
 }  
  
 public static void validateAge(int age, User user) {  
 if (age >= 1) {  
 user.setAge(age);  
 return;  
 }  
 if(age < 0){  
 throw new BadRequestException("Age must be positive number!");  
 }  
 }  
}

# **5. Ръководство за използване**

## **5.1 User guide**

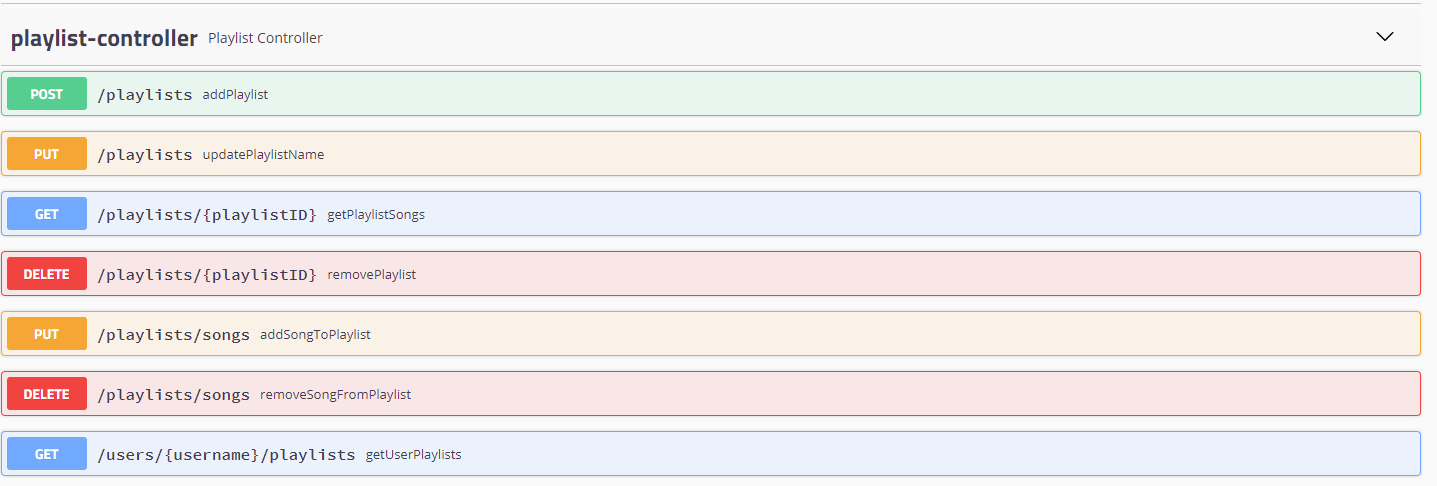
Тъй като проектирането на уеб услугата обхваща само back-end частта, за по-нагледно визуализиране на фукциалността съм използвал Swagger API Documentation. Swagger спецификацията представя четимо описание на нашите url заявки(линковете, които поддържа нашата уеб услуга) към сървъра или накратко документира RESTful уеб услугата. Услугата е разделена на 4 основни фукционални части – потребители, плейлисти, коментари и песни.

### **5.1.1 User Controller**



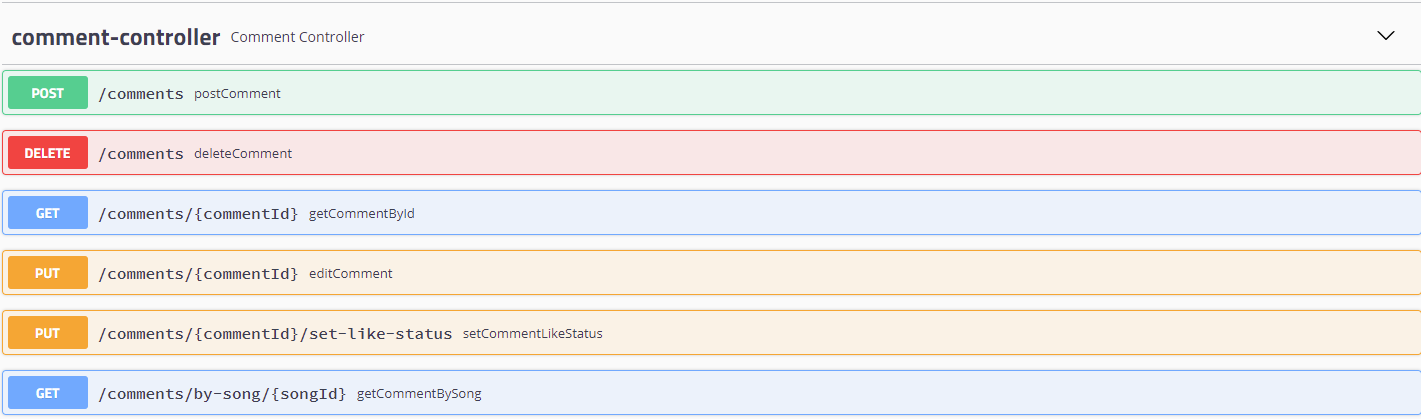
Както се вижда, един потребител има следните по ред **функционалности** – да се логва, да преглежда профила си, да се регистрира, да актуализира профила си, да изтрива профила си, да търси потребител по име, да филтрира потребителите по различни критерии, да следва и отследва потребители, да излиза от профила си и да потвърди профила си чрез изпращане на токен за валидация на посочен имейл.

### **5.1.2 Playlist Controller**



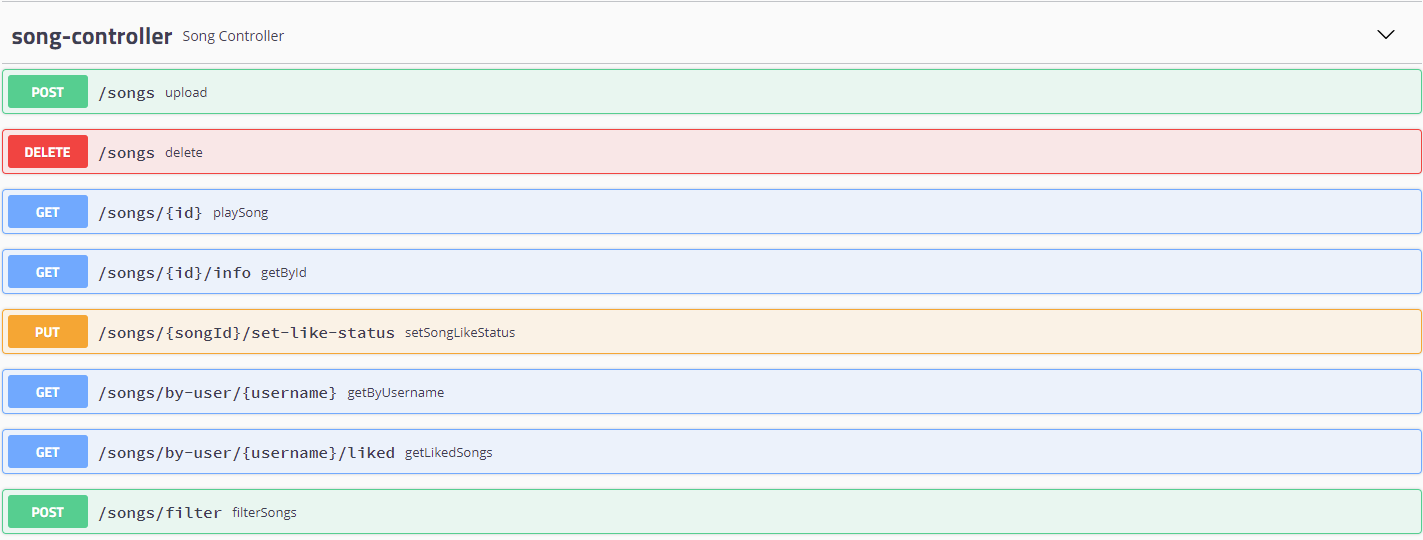
За да може да работи с плейлисти, потребителят трябва да има регистрация и да е влязал в профила си. Така той може да управлява, разбира се, само своите плейлисти по следния начин - да създава нов плейлист, да го актуализира, да търси плейлист по ид и съответно да види всичките песни в него, да изтрива свои плейлисти, да добавя или изтрива песни, да може да вижда всички плейлисти на даден потребител по въведено име.

### **5.1.3 Comment Controller**



Тук отново, за да може да работи с коментари, потребителят трябва да има регистрация и да е влязал в профила си. Той може да извършва следните действия със своите коментари – да пише и изтрива коментар, да търси коментар по ид, да променя коментар, да харесва или не харесва чужди и свои коментари и по въведено ид на песен да вижда всички коментари под нея.

### **5.1.4 Song Controller**



Тук отново, за да може да работи с коментари, потребителят трябва да има регистрация и да е влязал в профила си. Той може да извършва следните действия със своите песни – да добавя или трие песен, да слуша дадена песен по ид, да види информацията за песен по ид, да харесва и не харесва песен, да вижда всички песни на потребител по въведено име или всички негови харесвани песни, да има възможност да филтрира всички песните по различни критерии.

# **6. Заключение**

Разработен прототип на back-end на уеб-базирана услуга за слушане на музика.

Представени са и най-известните технологии и езици за front-end реализация на всяка една уеб услуга, с които в бъдеще моето приложението може да се осъвършенства и да придобие вид на реално такова. Дадени са малки примерни демонстрации и подробни обяснения какво представлява всяка една от тях.

Представени са и всички използвани технологии за реализирането на back-end частта на уеб услугата. Подробно е дадено обяснение за всеки един език и технология какво представлява, за какво и как е използвана в приложението. Благодарение на тези технологии, които са сред най-известните и най-използваните в днешно време, уеб услугата работи бързо, ефективно, без забавяния и сривове.

Услугата е разделена на три основни слоя, използвайки и архитектурния шаблон MVC. За входна точка служат Controller-ите(първи слой), които на базата на изпратена заявка и url разпределят кой Service да бъде извикан. В Service-ите(втори слой) се намира нашата бизнес логика и всички валидации и проверки. Те от своя страна използват Repository-та(трети слой) и DAOs(Data Access Objects, трети слой) за достъп до базата данни(MySQL), където пазим цялата информация на уеб услугата.

За разработката на приложението са използвани голям набор от технологии, езици, протокол за комуникация, библиотеки и инструменти като Java EE, Tomcat, Maven, Spring Boot, JPA(Hibernate), MySQL, HTTP, Swagger API, Lombok, BCrypt, Java Mail API, Amazon Web Services.

За изграждането, тестването, съхранението и демонстрацията са използвани следните софтуерни продукти – IntelliJ IDEA Ultimate, Workbench, Postman, Git, GitHub. Като всеки един от тях е достъпен за свободно и не е нужно допълнително заплащане за използването им.

Като бъдещо развитие на система е изграждане на front-end модула и възможност за регистрация на домейн в някоя уеб хостинг компания, благодарение на която приложението вече ще е достъпно в Интернет мрежата непрекъснато и ще може да се използва от потребителите в цял свят.

# **7. ИЗПОЛЗВАНИ ИЗТОЧНИЦИ**

1. [Разработка на Web-приложения с Java | Реферат от 'Информационни технологии - Страница 11 (referatite.org)](https://referatite.org/razrabotka-na-web-prilojeniq-s-java/22743/ref/p11)
2. [DBorisova\_010\_ITSr.pdf (unibit.bg)](https://fin.unibit.bg/images/pdf/bakalavri/DBorisova_010_ITSr.pdf)
3. [Уеб базирана система - предимства пред десктоп (dgsoft.eu)](https://www.dgsoft.eu/%D0%A3%D0%B5%D0%B1-%D0%B1%D0%B0%D0%B7%D0%B8%D1%80%D0%B0%D0%BD%D0%B0-%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0-%D0%BF%D1%80%D0%B5%D0%B4%D0%B8%D0%BC%D1%81%D1%82%D0%B2%D0%B0-%D0%BF%D1%80%D0%B5%D0%B4-%D0%B4%D0%B5%D1%81%D0%BA%D1%82%D0%BE%D0%BF-b10.html)
4. [💬1 - Какво е CMS (Content Management System)? (superhosting.bg)](https://blog.superhosting.bg/what-is-cms.html)
5. [Уроци по JavaScript | Реферат от 'Информационни технологии (referati.org)](http://www.referati.org/uroci-po-javascript-1/85033/ref)
6. [Уроци по Java Script | Реферат от 'Информационни технологии (referati.org)](http://www.referati.org/uroci-po-java-script/38597/ref)
7. [1. Първи стъпки в програмирането · Основи на програмирането с JavaScript (softuni.bg)](https://js-book.softuni.bg/chapter-01-first-steps-in-programming.html)
8. [Microsoft PowerPoint - web-services.ppt (schupen.net)](https://schupen.net/lib/tu/KST_all/Semesters/semestar%207/PRS/Lecs/web-services.pdf)
9. [Лекция 11 (shtrakov.net)](http://www.shtrakov.net/Vis_Prog_Web/Lect_11.pdf)
10. [Урок за RESTful уеб услуги с пример за REST API (csstricks.net)](https://bg.csstricks.net/8225647-restful-web-services-tutorial-with-rest-api-example)
11. [Java платформа -  J2EE (asenevtsi.com)](http://tuj.asenevtsi.com/IntT/IntT036.htm)
12. [Jakarta EE – Уикипедия (wikipedia.org)](https://bg.wikipedia.org/wiki/Jakarta_EE)
13. [Spring Framework – Уикипедия (wikipedia.org)](https://bg.wikipedia.org/wiki/Spring_Framework)
14. [Apache Tomcat – Уикипедия (wikipedia.org)](https://bg.wikipedia.org/wiki/Apache_Tomcat)
15. [РАЗЛИКА МЕЖДУ JPA И HIBERNATE | СРАВНЕТЕ РАЗЛИКАТА МЕЖДУ ПОДОБНИ ТЕРМИНИ - ТЕХНОЛОГИЯ - 2021 (strephonsays.com)](https://bg.strephonsays.com/jpa-and-vs-hibernate-11222)
16. [jdbc-01-basics.pdf (elsys-bg.org)](http://lubo.elsys-bg.org/wp-content/uploads/2009/02/jdbc-01-basics.pdf)
17. [Databases (uni-sofia.bg)](https://www-it.fmi.uni-sofia.bg/courses/WDB/db.htm)
18. [Какви са слоевете DAO, DTO и Service в Spring Framework? (tidewaterschool.org)](https://bg.tidewaterschool.org/852795-what-are-the-dao-dto-EZTXMS)