UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

BAKALÁŘSKÁ PRÁCE

2025 Vasilii Kozyrev

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Personalizovaná hudební doporučení pomocí umělé inteligence a zpracování přirozeného jazyka

Bakalářská práce

2025 Vasilii Kozyrev

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2024/2025

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Vasilii Kozyrev**
Osobní číslo:	**I21175**
Studijní program:	**B0688A140009 Informační technologie**
Téma práce:	**Personalizovaná hudební doporučení pomocí umělé inteligence a zpracování přirozeného jazyka.**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Tato bakalářská práce se zaměřuje na návrh a vývoj systému, který generuje hudební doporučení na základě vstupů od uživatele zadaných v přirozeném jazyce. Hlavním cílem je vytvořit model, který analyzuje textové popisy nálady, emocí nebo preferencí uživatele a následně navrhuje vhodné skladby. Jádrem řešení model implementovaný v jazyce Python. Backend aplikace je postaven na frameworku Spring Boot a zajišťuje komunikaci s frontendem (vytvořeným v Reactu), databázemi a externími API. Aplikace integruje hudební API (např. Spotify) pro získávání metadat skladeb a umožňuje uživatelům doporučovat vhodné písně.

Rozsah pracovní zprávy: **min. 30 stran**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. CHOLLET, François. *Deep learning with Python*. Second edition. Shelter Island: Manning, [2021]. ISBN 978-1-61729-686-4.
2. TUNSTALL, Lewis; WERRA, Leandro von a WOLF, Thomas. *Natural language processing with transformers: building language applications with Hugging Face*. Revised color edition. Sebastopol, CA: O'Reilly, 2022. ISBN 978-1-0981-3679-6.
3. AGGARWAL, Charu C. *Recommender systems: the textbook*. Cham: Springer, [2016]. ISBN 978-3--319-29657-9.

Vedoucí bakalářské práce: **Ing. Jan Panuš, Ph.D.**
Katedra informačních technologií

Datum zadání bakalářské práce: **15. prosince 2024**
Termín odevzdání bakalářské práce: **16. května 2025**

L.S.

prof. Ing. Petr Doležel, Ph.D. v.r.
děkan

Ing. Jan Panuš, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 28. února 2025

Prohlašuji:

Práci s názvem “Personalizovaná hudební doporučení pomocí umělé inteligence a zpracování přirozeného jazyka” jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 22. 08. 2025 Vasilii Kozyrev

PODĚKOVÁNÍ

Na tomto místě chci poděkovat panu Janu Panušovi za vedení a poskytnuté rady při zpracování této práce, jeho připomínky pomohly ujasnit směřování projektu a soustředit se na podstatné. Největší dík patří mým rodičům za stálou podporu a motivaci – i na dálku, jejich víra ve mne byla oporou po celou dobu a bez ní by tato práce nevznikla. Děkuji za trpělivost v obdobích, kdy se práce protahovala, a za povzbuzení v momentech, kdy se nedařilo. Tato práce je i jejich zásluhou.

ANOTACE

Tato bakalářská práce se zabývá návrhem a implementací webové aplikace, která umožňuje generovat hudební playlisty na základě textového zadání uživatele. Cílem práce je propojit umělou inteligenci s hudebními službami a nabídnout uživatelům personalizovaná doporučení. Backend aplikace je postaven na frameworku Spring Boot, frontend je realizován pomocí React. Pro práci s hudebními daty je využito rozhraní Spotify API a přihlášení je řešeno výhradně přes Spotify OAuth 2.0. Zpracování přirozeného jazyka a převod požadavků do parametrů zajišťuje samostatná mikroslužba v Pythonu (FastAPI), která funguje jako LLM orchestrátor. Tato mikroslužba běží lokálně nad platformou Ollama a provádí inferenci velkého jazykového modelu (Qwen 2.5 7B Instruct) i výpočet vektorových reprezentací (e5-multilingual-small). Kandidáti skladeb jsou získáváni přes Spotify Search API, následně probíhá semantické porovnání, reranking a diverzifikace výsledků pomocí metody MMR. V práci je rovněž popsána architektura systému a způsoby komunikace mezi jednotlivými částmi. Výsledkem je funkční aplikace, která uživatelům poskytuje moderní nástroj pro objevování nové hudby.

KLÍČOVÁ SLOVA

doporučovací systém, zpracování přirozeného jazyka, semantické vyhledávání a reranking, webová aplikace, personalizace doporučení.

TITLE

Personalized music recommendation using Artificial Intelligence and Natural Language Processing

ANNOTATION

This bachelor’s thesis focuses on the design and implementation of a web application that enables the generation of music playlists based on a user’s textual input. The aim of the thesis is to connect artificial intelligence with music services and offer users personalized recommendations. The application’s backend is built on the Spring Boot framework, and the frontend is implemented using React. For working with music data, the Spotify API is used, and authentication is handled exclusively via Spotify OAuth 2.0. Natural language processing and the conversion of requests into parameters are provided by a standalone Python microservice (FastAPI) that serves as an LLM orchestrator. This microservice runs locally on

the Ollama platform and performs inference of a large language model (Qwen 2.5 7B Instruct) as well as the computation of vector representations (e5-multilingual-small). Candidate tracks are retrieved via the Spotify Search API, they are then semantically compared, re-ranked, and diversified using the Maximal Marginal Relevance (MMR) method. The thesis also describes the system architecture and the communication between its components. The result is a functional application that provides users with a modern tool for discovering new music.

KEYWORDS

recommender systém, natural language processing, semantic search and re-ranking, web application, personalization systetem

OBSAH

SEZNAM TABULEK

SEZNAM OBRÁZKŮ

SEZNAM ZKRATEK A ZNAČEK

AI – Artificial Intelligence

API – Application Programming Interface

BoW – Bag of Words

BPE – Byte Pair Encoding (subword tokenization)

BPM – Beats Per Minute (tempo)

DCG – Discounted Cumulative Gain

ER – Entity–Relationship (data model)

F1 – Harmonic mean of precision and recall

HTTP – Hypertext Transfer Protocol

ILD – Intra-List Diversity

IR – Information Retrieval

IDCG – Ideal DCG

JSON – JavaScript Object Notation

JPA – Java Persistence API

JWT – JSON Web Token

LLM – Large Language Model

LSTM – Long Short-Term Memory (recurrent neural network)

MMR – Maximal Marginal Relevance

MRR – Mean Reciprocal Rank

nDCG@k – Normalized DCG at rank k

NER – Named Entity Recognition

NLP – Natural Language Processing

OAuth 2.0 – Authorization framework

PKCE – Proof Key for Code Exchange (OAuth 2.0 extension)

POS – Part of Speech (tagging)

QA – Question Answering

RAG – Retrieval-Augmented Generation

REST – Representational State Transfer

RNN – Recurrent Neural Network

ROUGE – Recall-Oriented Understudy for Gisting Evaluation

SQL – Structured Query Language

STS – Semantic Textual Similarity

ÚVOD

Dynamický rozvoj streamovacích platforem přinesl uživatelům bezprecedentní dostupnost hudby, avšak současně i problém s jejím efektivním objevováním. Tradiční doporučovací přístupy, opřené převážně o chování uživatelů a základní metadata, často narážejí na limity v situacích, kdy je požadavek formulován přirozeným jazykem, neostře či kontextově (např. „*něco klidného na učení*“, „*energická hudba na běh*“). Tato bakalářská práce reaguje na uvedenou výzvu návrhem a implementací webové aplikace, která z textového zadání uživatele vytváří personalizované hudební playlisty.

Hlavním cílem práce je nabídnout srozumitelný a prakticky využitelný postup, jenž propojí uživatelův popis v přirozeném jazyce s výsledným playlistem tak, aby byla zachována relevance vůči zadanému záměru i rozmanitost doporučených skladeb. Mezi dílčí cíle patří (i) formulace požadavků uživatele do jednoznačně strukturované podoby, (ii) získání kandidátů z externího hudebního katalogu, (iii) semantické porovnání a řazení kandidátů s ohledem na uživatelský záměr a (iv) řízená diverzifikace výsledků, aby se předešlo monotónním výstupům.

Zvolený přístup je koncipován jako vícefázová pipeline. Nejprve je textové zadání interpretováno do interní reprezentace zohledňující požadovanou náladu, tempo či další charakteristiky. Následuje vyhledání kandidátů v externím katalogu a jejich převod do jednotného vektorového prostoru pro účely semantického srovnání. Poté probíhá reranking, který kombinuje odhad relevance s penalizací přílišné podobnosti mezi položkami, čímž je dosaženo vyváženého kompromisu mezi přesností a rozmanitostí. Výsledkem je playlist včetně doprovodných metadat, připravený k okamžitému použití.

Důraz je kladen na transparentnost a přenositelnost řešení: jednotlivé kroky pipeline jsou oddělené a dobře zdokumentované, což usnadňuje budoucí rozšiřování (např. o další zdroje hudebních dat, nové strategie řazení či alternativní metody práce s významem textu). Práce zároveň diskutuje omezení navrženého postupu, zejména závislost na kvalitě a pokrytí externího katalogu, limity interpretace jemných nuancí hudebních preferencí a praktické aspekty provozu při vyšší zátěži.

1. Základy doporučovacích systémů

Doporučovací systém se snaží odhadnout, které položky dávají pro konkrétního uživatele smysl „právě teď“, a předložit je v pořadí podle očekávané relevance. Na rozdíl od klasického vyhledávání je spíše proaktivní: nečeká na přesný dotaz, ale nabídne obsah na základě historie, kontextu a podobných uživatelů či položek [3].

Relevance v doporučování není jen přesná shoda s minulostí. Je to také pocit, že doporučení zapadá do situace a je užitečné v daném okamžiku (např. jiné skladby na běh a jiné na učení). Systém proto vyvažuje několik cílů současně: přesnost (trefnost), rozmanitost (méně opakování, více objevování) a novost (příjemná překvapení), aby se předešlo „bublině“ opakovaných návrhů [3], [4].

1.1 Typy dat a signálů

Úspěšný systém skládá obraz uživatele z více typů informací a každou bere s určitou opatrností:

- **Explicitní zpětná vazba** (hodnocení, „líbí/nelíbí“, uložení) je srozumitelná, ale vzácná a často se týká jen malé části katalogu [3].
- **Implicitní chování** (poslech, přeskočení, opakování, doba interakce) je hojné, ale šumové. Stejná událost může v různém kontextu znamenat něco jiného [3].
- **Obsah a metadata** (žánr, interpret, rok, textové popisy) pomáhají tam, kde chybí interakce. Dnes se navíc využívají naučené reprezentace (vektory), které zachycují významové podobnosti a usnadňují porovnávání položek a dotazů [1].
- **Kontext** (čas, zařízení, aktivita) doplňuje obrázek o situaci, protože preference jsou dynamické a různá nastavení vedou k různým volbám [3].

Klíčovým trendem je učení reprezentací: modely se učí takové vektory pro uživatele, položky i textové dotazy, aby blízké vektory odpovídaly podobným významům. Díky tomu systém lépe chápe volné formulace (např. „něco klidného na večer“) a umí vyhledat vhodné kandidáty i bez přesných klíčových slov [1], [2]. Zároveň je nutné hlídat kvalitu signálů (např. odlišit krátké „vyzkoušení“ od skutečného zájmu) a doplňovat je kontextem [3], [5].

1.2 Hlavní přístupy

V praxi se setkáme se třemi rodinami metod. Každá řeší jiný kus problému a v reálných systémech se obvykle kombinují.

Content-based doporučování (CB) - systém se dívá na to, co je uvnitř položek (např. žánr, nálada, vektorová reprezentace) a porovnává je s tím, co uživatel v minulosti oblíbil. Pokud

poslouchás klidný indie pop, systém najde další skladby s podobnými rysy a nabídne je jako první. Výhoda: funguje i bez dat od ostatních uživatelů a dobře startuje u nových položek. Slabinou je, že uživateli může hrozit více stejného, tedy málo překvapení a menší šance objevovat něco nového mimo jeho obvyklý okruh. [3], [4].

Collaborative filtering (CF) - místo obsahu sledujeme vzorce v chování napříč uživateli. Pokud lidé se „skoro stejným vkusem“ poslouchají ještě něco, co ty zatím neznáš, je to dobrý kandidát na doporučení. Existují dvě oblíbené podoby:

- **Sousedské metody**: hledají podobné uživatele nebo podobné položky a přenášejí preference mezi “sousedy”. Intuitivně: “Uživatelé jako ty si pouštějí i X, zkusíme ti ho nabídnout” [3], [4].
- **Latentní faktory (jednoduše)**: systém se naučí krátké „otisky“ uživatelů a položek (vektory), které se dají smysluplně porovnávat. Pokud je “otisk” uživatele blízko “otisku” skladby, je šance, že se trefíme. Technické detaily rozkladů necháváme stranou. Důležité je, že takové vektory zachycují vzorce chování, i když chybí dobrá metadata [3], [4].
- **Hybridní přístupy**: spojují silné stránky CB a CF. V praxi se často uplatňuje vícestupňový postup: nejprve rychlé vytažení kandidátů (podle podobnosti obsahu či vzorců chování), poté přesnější řazení s využitím více signálů a nakonec post-processing s pravidly a diverzifikací. Tato kombinace zajišťuje jak relevanci, tak objevitelnost. [3], [4]

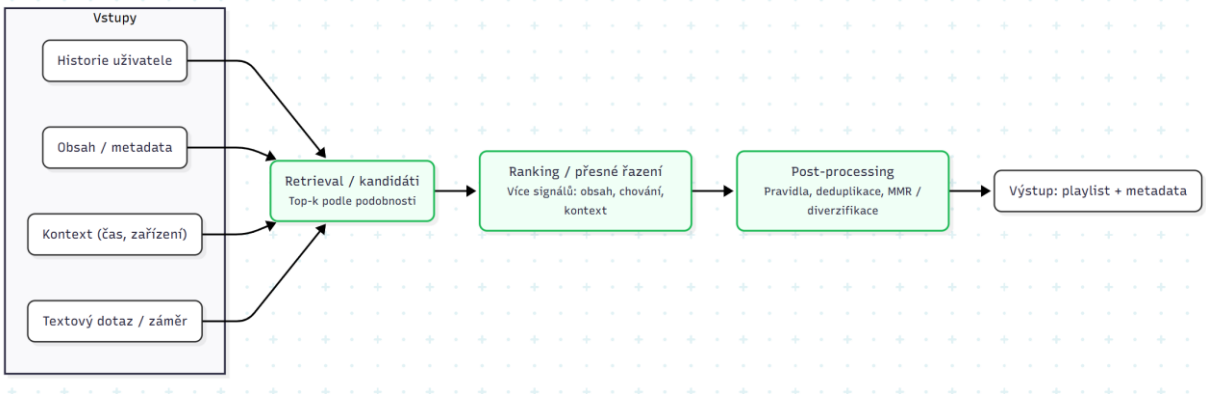
Pokud máme málo interakcí, ale kvalitní metadata, je vhodné použít obsahově založené doporučování (CB) jako základ. Naopak v případě, že máme k dispozici velké množství uživatelských interakcí, ale metadata jsou slabší, uplatní se lépe kolaborativní filtrování (CF). Chceme-li tyto přístupy bezpečně zkombinovat a využít silné stránky obou, je vhodné zvolit hybridní metodu [3], [4].

1.3 Typické problémy, zkreslení a obranné strategie

Kvalitu doporučení neurčují jen použité algoritmy, ale i povaha dat a provozní realita. V praxi se opakovaně projevuje nedostatek historie, řídkost interakcí, zvýhodňování populárních položek a šum v implicitních signálech. Účinný přístup proto spojuje modelové postupy s jednoduchými kroky v aplikaci: krátký onboarding, práce s obsahem, řízená diverzifikace a základní kontrola anomálií udržují systém stabilní a férový [3], [4], [5].

- **Cold start (nový uživatel/položka)**: bez historie je těžké hádat vkus. Pomáhá krátký onboarding (rychlé volby nálad/žánrů), využití obsahu (podobnost k již známým položkám) a opatrné zapojení trendů. U nových skladeb je dobré opřít se o metadata a naučené vektory podobnosti, aby se neviditelné položky neztratily [3], [4].
- **Sparsita (málo interakcí v obřím katalogu):** většina kombinací uživatelů a položek je prázdná, proto se učí krátké vektory uživatelů a položek, které dovolují smysluplné porovnávání i bez mnoha explicitních dat. Pomáhá také práce s obsahem a chytré doplňování negativních příkladů (věci, které uživatel pravděpodobně nechce), aby se model učil rozlišovat [3], [4].
- **Popularity bias a zpětnovazební smyčky:** co je populární, to dostává další expozice, a tím se stává ještě populárnějším. Řešením je diverzifikace (nepředkládat stále totéž), řízené zkoušení novinek (malá část výsledků je vyhrazena pro objevování) a případné vážení popularity v řazení, aby se neztratil "long tail" [3], [5].
- **Šum v implicitních datech a kontext:** Přeskočení nemusí znamenat „nelíbí se", uživatel třeba jen neměl čas. Proto se signály váží (např. dokončený poslech má větší váhu než krátký dotek) a kombinují s kontextem (čas, zařízení) [3].
- **Jazyk a zkreslení:** u dotazů ve volném textu záleží na formulaci. Pomáhá normalizace (sjednocení pojmů, práce se synonymy) a jednoduchý post-processing (kontrola, že výsledek opravdu odpovídá záměru). Je také vhodné hlídat přenos zkreslení z tréninkových dat a udržet výstupy vysvětlitelné [2], [5].
- **Robustnost a bezpečnost:** Platformy čelí i pokusům uměle nafukovat popularitu (shilling) nebo manipulaci metadat. Základní obranou je detekce anomálií, validace zdrojů a limity pro neobvyklé chování [3], [4].

1.4 Architektura a vícestupňová pipeline



Obrázek 1: Architektura doporučování

V praxi se osvědčila architektura rozdělená do tří fází:

- **Generování kandidátů (retrieval):** Cílem je z milionů položek vybrat hrstku s vysokým potenciálem relevance (např. top-k vektorově nejbližších). Reprezentace uživatele **u** a položky **v** se porovnávají jednoduchou, ale účinnou podobnostní funkcí, například kosinovou podobností

$$\cos(\theta) = \frac{u \times v}{\|u\| \times \|v\|}$$

  V této fázi je klíčová rychlost a škálovatelnost, modely a indexy jsou navrženy tak, aby umožnily velmi rychlé dotazy [3].

- **Řazení (ranking):** Menší množinu kandidátů je možné skórovat bohatší funkcí, která kombinuje obsahové, kolaborativní a kontextové rysy. Přístupy mohou být pointwise (předpověď pravděpodobnosti interakce), pairwise (preferenční dvojice) nebo listwise (optimalizace přímo na metrikách pořadí) [3]. Hluboké učení zde dovoluje modelovat nelineární efekty a interakce rysů [1].
- **Post-processing:** Aplikuje se deduplikace verzí, pravidla čerstvosti, omezení opakování téhož interpreta a diverzifikace, aby výsledek nebyl monotónní. Tato fáze často řídí uživatelský flow a obchodní pravidla platformy [3].

Striktní oddělení fází zjednodušuje výměnu komponent, ladění kompromisů (přesnost/rozmanitost/rychlost) a transparentní vyhodnocování dopadů změn [3]. Všemi fázemi se táhne princip učení reprezentací, jenž spojuje vektorový pohled na uživatele, položky i textové dotazy [1], [2].

2. Doporučování hudby a specifika domény

Tato kapitola převádí obecné principy doporučovacích systémů z kap. 1 do specifického prostředí hudby. Vysvětluje, jak se struktura katalogu (skladba–album–interpret, více verzí téhož titulu, regionální dostupnost) a dlouhý ocas nabídky promítají do návrhu systému a proč je nezbytné pracovat s deduplikací, normalizací identit a *fallback* strategiemi při nedostupnosti konkrétních verzí. Zároveň upřesňuje roli přirozeného jazyka: uživatelské požadavky bývají vágní, směřují k náladě, aktivitě či období a vyžadují převod do jednoduchých atributů a sémantických reprezentací, aby na ně mohl navázat retrieval a řazení (viz schéma pipeline v kap. 1).

Důraz je kladen na relevance playlistu jako celku: vedle tematické shody jednotlivých skladeb je posuzována i jejich lokální soudržnost (přechody, tempo, energie) a vyvážená rozmanitost, aby výsledek nebyl monotónní ani roztříštěný. Kapitola v tomto pořadí rozpracuje charakter hudebních dat, typologii uživatelského záměru, převod textu na hudební atributy, doménové aspekty získávání kandidátů, pravidla řazení s ohledem na *flow* a praktická východiska pro diverzifikaci a hodnocení, na něž později navážou samostatné kapitoly o NLP, sémantickém vyhledávání a diverzifikaci.

2.1 Charakter hudebních dat

Hudební katalog není jen seznam skladeb, ale propletená síť **„***track* → *album* → *interpret***“** s více verzemi téhož titulu (live, remaster, radio edit, cover) a s regionální dostupností, která se mění podle licencí. Správná práce s těmito zvláštnostmi přímo ovlivňuje kvalitu kandidátů, řazení i finální soudržnost playlistu [6], [11].

Identita a hierarchie. Jednu „skladbu“ lze chápat v několika vrstvách: (i) nahrávka/recording (konkrétní zvuková stopa), (ii) vydání/release (album, singl, reedice) a (iii) přiřazení k interpretovi (hlavní interpret, feat, kompilace). Pro doporučování je praktické držet kanonickou identitu na úrovni nahrávky a k ní připojit dostupné verze vydání, tím se snižuje riziko duplicit a kolizí názvů [11], [12].

Vícenásobné verze téže skladby. Běžné situace: live vs. studio, remaster vs. původní, „radio edit“ vs. full length, explicit vs. clean. Aby výsledek nepůsobil jako opakování téhož, je vhodné zavést jednoduchá **pravidla výběru preferované verze**:

- pokud je dotaz obecný, preferovat standardní/studio verzi.
- při jednoznačném kontextu (např. „live“) akceptovat odpovídající verzi.

- pokud preferovaná verze není dostupná v regionu, zvolit nejbližší ekvivalent (stejný interpret, podobná délka, stejná nahrávka) a tuto volbu dále už neopakovat.
- vyvarovat se zařazení dvou verzí téže nahrávky do jednoho playlistu, pokud to nestanoví dotaz [6], [11].

Regionální dostupnost a licence. Katalog se liší podle teritoria, některé verze nemusí být legálně přehratelné. Systém proto potřebuje fallback: přiřadit ekvivalentní verzi, která je dostupná, zachovává intenci dotazu (tempo/nálada/styl), a nenaruší plynulost sousedních skladeb. Nezbytností je průběžně logovat nahrazení, aby bylo možné ladit pravidla post-processingu [6].

„Long-tail" a popularita. Poslechovost je extrémně nerovnoměrná: malý počet hitů a dlouhý ocas zřídka hraných titulů. Bez korekcí se doporučování stáčí k popularitě a ztrácí objevitelnost. Doménově dává smysl omezit opakování téhož interpreta a otevřít prostor pro méně známé položky, pokud stále plní záměr dotazu [6], [3].

Kvalita a konzistence metadat. Tituly mohou mít různé zápisy, přepisované feat/remix informace, chybějící nebo nejednotné žánrové tagy. Praktická normalizace zahrnuje: standardizaci názvu (odstranit závorkové sufixy typu „(Remastered 2011)" pro účely deduplikace), kanonizaci interpretů (sjednocení aliasů), toleranci délky skladby (např. ±2–3 s pro shodu nahrávky) a preferenci spolehlivých identifikátorů, jsou-li k dispozici [11], [7].

Praktický příklad. Dotaz „*klidný indie večer*" vrátí dvě verze téže skladby: „*Song A (Remastered 2019)*" a „*Song A (Album Version)*". Obě odpovídají náladě i tempu. Systém zvolí jednu kanonickou nahrávku (např. album verzi) a remaster vyřadí jako duplicitu, aby playlist nepůsobil repetitivně. Pokud album verze v regionu chybí, přijde na řadu remaster, ale v další části playlistu se stejná nahrávka již znovu neobjeví [6], [11].

2.2 Uživatelský záměr (intent) v hudbě

Textové zadání často míří na náladu, aktivitu nebo éru („*klidné na učení*", „*energické 2000s*"), případně obsahuje negace („*bez rapu*"), neurčitost („*spíš klidné*") a vícejazyčné prvky. Aby šlo s dotazem pracovat, převádí se na sadu jednoduchých vlastností (škála nálady/energie, tempo, období, žánrové tagy) a volitelně na kompaktní reprezentaci pro sémantické porovnání. Tuto transformaci usnadňuje moderní NLP: práce se synonymií, mírnými negacemi a parafrázemi snižuje citlivost na formulaci a přibližuje dotaz skutečnému záměru [2], [5]. V doméně hudby dává smysl mírně preferovat „měkké" škály (např. 5–7 bodů pro náladu/energii) před ostrými

filtry: uživatelské formulace jsou často vágní a tvrdé hranice vedou k vyloučení jinak vhodných skladeb [10].

2.3 Relevance v hudbě: od skladby k playlistu

V hudebním doporučování nestačí, aby jednotlivé skladby odpovídaly dotazu. Uživatel posuzuje i to, zda playlist drží pohromadě a příjemně plyne. Relevance se proto skládá z několika dimenzí: tematická shoda s úmyslem dotazu, lokální soudržnost mezi sousedními skladbami, rozmanitost bez ztráty jednotícího tématu a přiměřená novost, která podporuje objevování [6], [7], [8].

Tematická shoda. Zadání typu „klidné indie na večer“ vyžaduje, aby převládaly skladby s nízkou energií, vhodným tempem a odpovídajícím stylem nebo érou. Pokud je dotaz vágní, vyplatí se tolerovat širší pásmo hodnot a doladit výsledek až v řazení, jinak hrozí příliš úzký výběr [6], [8].

Lokální soudržnost a „flow“. I relevantní skladby mohou působit rušivě, pokud mezi nimi dochází k prudkým skokům tempa/energie nebo se často opakuje jeden interpret. Jednoduché lokální heuristiky (omezit skoky, hlídat rozestupy interpretů, vyhnout se dvěma verzím téže nahrávky za sebou) výrazně zlepšují vnímanou kvalitu [7], [3].

Rozmanitost a pokrytí. Příliš homogenní seznam rychle unaví. Naopak přiměřená variace v rámci zadaného tématu udržuje pozornost a rozšiřuje obzory. V hudbě to typicky znamená pracovat s osami žánr/éra/interpret/tempo a zabránit nadměrné redundanci (např. kvótami a penalizací příliš podobných položek) [4], [6].

Novost a serendipita. Uživateli prospívá objevit něco nového, ale ne „*mimo mísu*“. V praxi proto mírně podporujeme méně známé kandidáty, pokud stále naplňují záměr dotazu a nenaruší soudržnost playlistu [3], [8].

Níže je tabulka, která tyto dimenze převádí do konkrétních návrhových pravidel a náznaku, jak je sledovat v hodnocení.

Tabulka 1: Dimenze relevance a důsledek pro návrh pravidel

Dimenze	Co znamená v praxi	Pokud se zanedbá…	Příklad pravidla/heuristiky	Poznámka k měření
Tematická shoda (intent)	Většina skladeb odpovídá náladě/stylu/éře z dotazu	„Mimo téma“, nízká spokojenost	Tolerovat pásma (např. energie 0.2–0.5) a doladit v řazení	nDCG@k pro tematické tagy [6], [8]
Lokální soudržnost (flow)	Plynulé přechody tempa/energie; nepřeskakují se „vrstevně“ odlišné skladby	Rozbitý poslech, vyšší skip-rate	Limit na změnu tempa/energie mezi sousedy; min. rozestup interpretů	lokální delta-tempo/energie, párové skóre přechodů [7], [3]
Rozmanitost	Variace v rámci tématu (žánr/éra/interpret/tempo)	Monotónnost, „bublina“	Kvóty (max. X % jednoho žánru/éry), penalizace příliš podobných položek	ILD/coverage@k [4], [6]
Novost/serendipita	Přiměřený podíl méně známých, ale tematicky vhodných skladeb	„Jen hity“, slabé objevování	Mírně zvýhodnit long-tail kandidáty s dobrou shodou; horní limit na opakovanost	Novelty/Popularity-Adjusted Precision [3], [8]
Verze/dostupnost	Preferovaná (kanonická) verze a regionálně přehratelné položky	Duplicity, nedostupné odkazy	Výběr preferované verze; fallback na dostupný ekvivalent; zákaz dvou verzí v playlistu	chybovost na dostupnosti, míra duplicit [11]

2.4 Od textu k hudebním atributům (text → struktura)

Cílem je převést volnou větu uživatele do podoby, se kterou umí systém pracovat: **jednoduché atributy** (nálada, tempo, éra, vokály, explicit/clean…) a kompaktní vektor pro sémantické porovnání v retrievalu. Prakticky postupujeme v několika krocích.

1. **Jazyková normalizace**. Krátce očistíme vstup: sjednotíme jazyk a diakritiku, zjemníme synonymii a triviální varianty („*klidné*“ ≈ „*chill*“, „*na učení*“ ≈ „*study*“), uchováme ale negace a zeslabovače („spíš“, „raději“), ty budou později řídit škály a tolerance [2], [13].
2. **Rozpoznání záměru a entit.** Ze věty vytáhneme typ dotazu (mood/activity/žánr/éra/seed), konkrétní výrazy (např. „*indie*“, „*2000s*“), negace („*bez rapu*“, „*bez vokálů*“) a doplňkové preference (explicit/clean). Klíčové je zachytit i měkké formy („*spíš klidné*“, „*mírně energické*“), protože právě ty rozhodují o šířce výsledku [2], [14].
3. **Mapování na škály a slovník**. Výrazy převádíme na interní škály (energie 0–1, tempo v BPM, valence od nízké po vysokou) a na standardizované tagy (žánr/éra). Vágní výrazy dávají intervaly, nikoli ostré hranice (např. „*klidné*“ → energie 0.2–0.4, tempo 60–90 BPM) [7], [15].
4. **Ošetření neurčitosti a konfliktů**. Pokud je požadavek nejednoznačný nebo obsahuje konflikt („*rychlé a uklidňující*“), upřednostníme hlavní osu (z kontextu věty) a druhou použijeme jako jemné boosty. Negace překládáme do zákazů (žánr ≠ rap) nebo do atributů (vokály = instrumentální) [2], [13].
5. **Vytvoření dvou výstupů pro vyhledávání:** (i) **Symbolické atributy**: jednoduchý JSON/struktura se škálami a tagy (energie, BPM, éra, explicit/clean…). (ii) **Sémantická reprezentace (vektor)**: embedding věty pro porovnání významu s kandidáty, aby systém „pochopil“ dotaz i bez přesných klíčových slov [2], [16].
6. **Lehká validace před retrievalem**. Zkontrolujeme, že nechystáme duplicity (např. „*bez vokálů*“ vs. „*female vocal*“), doplníme defaulty (když chybí tempo, odvodíme ho z nálady/aktivity) a přidáme informaci o síle požadavku („*spíš*“ → širší interval) [6], [11].

Obrázek 2: Od věty k atributům a reprezentaci s příkladem

2.5 Získání kandidátů v hudební doméně

Cílem fáze kandidátů je z rozsáhlého katalogu rychle vytáhnout malou sadu skladeb, které pravděpodobně naplní záměr dotazu a jsou technicky „čisté“ (přehratelné, bez duplicit). Aby byl následný ranking účinný, je nutné udržet rovnováhu mezi pokrytím a přesností: příliš úzký výběr zmenší manévrovací prostor, příliš široký zbytečně zatíží další kroky [3], [4]. Kandidáti

se typicky kombinují ze tří zdrojů: sémantická podobnost k textu/atributům (embeddingy pro významovou blízkost), atributové vyhledávání/filtry (tempo, energie, éra, vokály, explicit/clean) a seed-expanze kolem interpretů/playlistů. Skóre z jednotlivých kanálů se sjednocuje a váhově kalibruje, aby žádný proud nedominoval jen objemem výsledků [15], [16].

Hudební specifika vyžadují provést část „sanity-checků“ už zde: sjednotit identitu nahrávky a potlačit více verzí téhož titulu (album vs. remaster vs. radio edit), hned nahradit regionálně nedostupné položky nejbližším ekvivalentem, respektovat nastavení explicit/clean a vyloučit kandidáty s chybějícími klíčovými metadaty (např. délka). Praktické je také pohlídat, aby se balík nepřesytil jedním interpretem – jinak bude výsledek monotónní bez ohledu na kvalitu řazení [11], [6], [7].

Šířku záběru řídíme podle formulace dotazu: vágní výrazy mapujeme na širší intervaly a dotvarujeme je až v rankingu. U přesných zadání volíme užší pásma a krotit seed-expanzi. Pro provozní rychlost se sémantický kanál obvykle realizuje přes aproximativní vyhledávání sousedů (ANN).

Tabulka 2: Opatření ve fázi kandidátů

Problém (hudba)	Jak se projeví v kandidátech	Opatření v retrievalu
Duplicity / více verzí téhož titulu	Vrací se album+remaster+radio edit téže nahrávky	Kanonická identita podle nahrávky; povolit jen „preferovanou“ verzi
Nedostupnost v regionu	Kandidát nejde přehrát	Okamžitý fallback na nejbližší dostupnou verzi (stejný interpret, podobná délka)
Explicit/clean nesoulad	Kandidát neodpovídá nastavení uživatele	Vyloučit nevyhovující verzi; pokud existuje clean varianta, preferovat ji
Kolize názvů/aliasy interpretů	Vrátí se nesprávný „podobný“ interpret	Normalizace identit (aliasy), doplňkový filtr podle žánru/éry
Chybějící klíčová metadata	Nelze odhadnout tempo/energie, vadí v řazení a „flow“	Minimální práh kvality (vyřadit), případně doplnit z externího zdroje
Přesycení jedním interpretem	Balík kandidátů je monotónní	Limit „max N položek na interpreta“ už při generování

Přiliš úzký match (přefiltrováno)	Málo kandidátů, ztráta objevitelnosti	Rozšířit pásma škál (energie/BPM), přimíchat seed-expanzi
Příliš obecný match	Kandidáti tematicky ujíždějí	Posílit atributové filtry/boosty hlavní osy dotazu (mood/activity/éra)

2.6 Řazení s ohledem na hudební kritéria

Cíl je seřadit kandidáty tak, aby trefily záměr a zároveň měly plynulé přechody. Nejdřív spočteme pro každou skladbu základní skóre jako směs tří složek: shoda s dotazem, preference uživatele, hudební rysy (tempo/energie/valence/hlasitost). Všechny složky si převedeme na škálu 0–1 (normalizace), aby se dalo smysluplně míchat. Poměry (váhy) lze naladit na validační sadě nebo zvolit rozumný start (např. 0,5 / 0,3 / 0,2). [3], [4], [5].

Poté seznam skládáme sekvenčně. Pro první pozici vybereme skladbu s nejvyšším skóre (případně tu, která má vhodný „úvodní" charakter, spíš klidnější, když dotaz míří na klidnou náladu). Pro další pozice vybíráme kandidáta, který má vysoké skóre a zároveň nenaruší *flow*: malý skok tempa/energie oproti poslední vybrané skladbě a rozestup téhož interpreta. Prakticky přičítáme k základnímu skóre malé tresty za velké skoky a za opakování. V praxi se osvědčí dívat se na okno několika posledních skladeb (nejen na jednu), aby se netvořily mini-smyčky téhož interpreta či stylu. Při shodném skóre rozhodne pořadí jednoduché pravidlo: preferuj novější/neslyšenou položku, případně tu s lepším přechodem (menší skok tempa). [7], [6]

Skóre můžeme vypočítat pomocí tohoto příkladu*: shoda* = 0,80, *preference* = 0,60, *rysy* = 0,70.

$$skóre(\mathrm{C}) \;=\; 0{,}80 \cdot 0{,}50 + 0{,}60 \cdot 0{,}30 + 0{,}70 \cdot 0{,}20 = 0{,}72$$

Skóre pro kandidáty A a B vypočítáme pomocí stejného principu. Předpokládejme, že naše výsledky jsou: A = 0.78, B = 0.75, C = 0.72. První vybereme **A**. Na druhou pozici porovnáme skok tempa/energie vůči A: B má větší skok (−0,05 bodu), C menší (−0,01 bodu). Upravená skóre: B→0,70, C→0,71. Vezmeme **C**. Pro třetí pozici zůstává **B**, pokud by B byl stejný interpret jako C, přidá se drobný trest (např. −0,03) a počká si o pár pozic dál. Tím vznikne pořadí, které je věrné tématu a zároveň **se** dobře poslouchá. [3], [4], [7]

Nakonec proběhnou pevné kontroly: nikdy neřadit dvě verze téže nahrávky vedle sebe, respektovat explicit/clean a regionální dostupnost. Tyto „*pojistky*" chrání před nejčastějšími artefakty dat, ale nezastupují samotné skórování a plynulost, jen je doplňují. [6], [11]

2.7 Diverzifikace a serendipita

Diverzifikace brání tomu, aby playlist působil jednotvárně: i když všechny skladby splňují zadání, přílišná podobnost unaví. Cílem je proto udržet téma a zároveň do seznamu vnést drobné rozdíly podél několika os, zejména interpret, žánr, éra, tempo/energie. V praxi stačí několik jednoduchých pravidel: nepokládat dvě verze téže nahrávky vedle sebe, držet rozestupy téhož interpreta a nelepit za sebe skladby s identickým profilem tempa/energie. Tato pravidla mají velký efekt i bez složitých modelů [4], [6], [7].

Když potřebujeme řízeně vyvážit "**trefnost × odlišnost"**, použijeme princip MMR v lidské podobě: *nová skladba má být dost relevantní a zároveň se nemá příliš podobat tomu, co už v seznamu je*. Prakticky si rozdělíme body na dvě části, např. 70 % relevance a 30 % odlišnost.

Pro výběr další skladby použijeme „*MMR-light*":

$$score(i) = \lambda \cdot rel(i) - (1 - \lambda) \cdot \max_{j \in S} \mathrm{sim}(i, j)$$

kde S je už vybraný seznam; rel(i)∈[0,1] je relevance kandidáta k zadání; sim(i,j)∈[0,1] je podobnost k některé z již vybraných skladeb (např. kosinová podobnost embeddingů nebo podobnost podle atributů/žánru/éry); λ∈[0,1] určuje poměr relevance: odlišnost (v hudbě se osvědčuje 0,6–0,8). [3], [4], [8]

Mini-příklad: Kandidát A má relevanci 0,9 a podobnost k dosavadnímu seznamu 0,8 → 0,7·0,9 − 0,3·0,8 = **0,39**. Kandidát B má relevanci 0,8 a podobnost 0,2 → 0,7·0,8 − 0,3·0,2 = **0,50**. Vezmeme **B**, protože lépe rozšíří seznam, ale stále drží téma. Poměr „relevance : odlišnost" (**λ**) se ladí validační. Pro hudbu se osvědčuje 0,6–0,8 ve prospěch relevance, aby rozmanitost nepřebila záměr [3], [4], [8].

Diverzifikace přirozeně pomáhá long-tailu: méně známé, ale tematicky vhodné skladby dostanou šanci. Současně je potřeba hlídat, aby „rozmanitost pro rozmanitost" nezpůsobila skoky, které rozbíjejí poslouchatelnost. Praktická pojistka: u osy tempo/energie nastavíme jen mírné rozvolnění (např. nepřekročit určitý skok mezi sousedy), zatímco u interpreta a žánru můžeme být odvážnější, změna jména nebo podžánru obvykle neruší tolik jako náhlý přechod ze 70 BPM na 150 BPM [6], [7].

2.8 Hodnocení kvality doporučení

Kvalitu je vhodné posuzovat několika metrikami, protože neexistuje jedno číslo, které by zachytilo trefnost, soudržnost i objevování.

- **nDCG@k (pořadí záleží):** myslete na to, že skladby na začátku mají větší váhu. Když nejlepší odpovídající skladby dáme dopředu, nDCG je blízko 1, když je „dobré“ věci až na konci, nDCG klesá. Je to prakticky „součet hvězdiček se slevou“ pro nižší pozice [3], [4].
- **ILD (Intra-List Diversity):** průměrná nepodobnost dvojic v playlistu. 1 znamená velmi různorodý seznam, 0 naopak „všechno stejné“. Příklad se třemi skladbami*:* podobnosti jsou 0,9; 0,2; 0,3 → nepodobnosti 0,1; 0,8; 0,7 → průměr ≈ 0,53. Čím výš, tím pestřejší výsledek (ale téma musí zůstat). [4], [6]
- **Plynulost přechodů (flow):** průměrný skok tempa/energie mezi sousedními skladbami. Příklad tempa*:* 80 → 82 → 120 → 124 BPM dává skoky 2, 38, 4; průměr ≈ 14,7 BPM a je vidět, že skok „120“ ruší. Čím menší průměr (a menší maximum), tím příjemnější poslech. [7]
- **Coverage a novelty:** coverage říká, kolik různých interpretů/žánrů pokryjeme. Novelty zhruba hodnotí, nakolik nové položky se dostaly do seznamu (nejen notorické hity). Je vhodné držet rovnováhu s relevancí, aby objevování neodjelo mimo záměr. [6], [8]

3. Základy zpracování přirozeného jazyka (NLP)

NLP zprostředkovává most mezi lidským jazykem a výpočetní reprezentací: z volných vět vytváří struktury, s nimiž může pracovat algoritmus, a naopak dokáže z dat generovat srozumitelný text. Společným jmenovatelem je zvládnout neurčitost přirozeného jazyka – synonyma, vícevýznamovost, negace či vágní formulace, a převést ji na rozhodnutí či výstupy s jasnou interpretací. Moderní přístup stojí na učení z velkých korpusů a na transformerech, které zachycují kontext a dlouhé závislosti, takže jeden model lze s malými úpravami použít pro různé úlohy od klasifikace přes extrakci informací až po generování [2], [13]. V praxi to znamená, že stejný princip reprezentace jazyka může sloužit jak pro „rozumění“ (např. rozpoznání záměru nebo entit), tak pro „tvorbu“ (shrnutí, odpověď, překlad), pokud je jasně stanoveno, co je požadovaným výstupem a jak se bude hodnotit jeho kvalita [5].

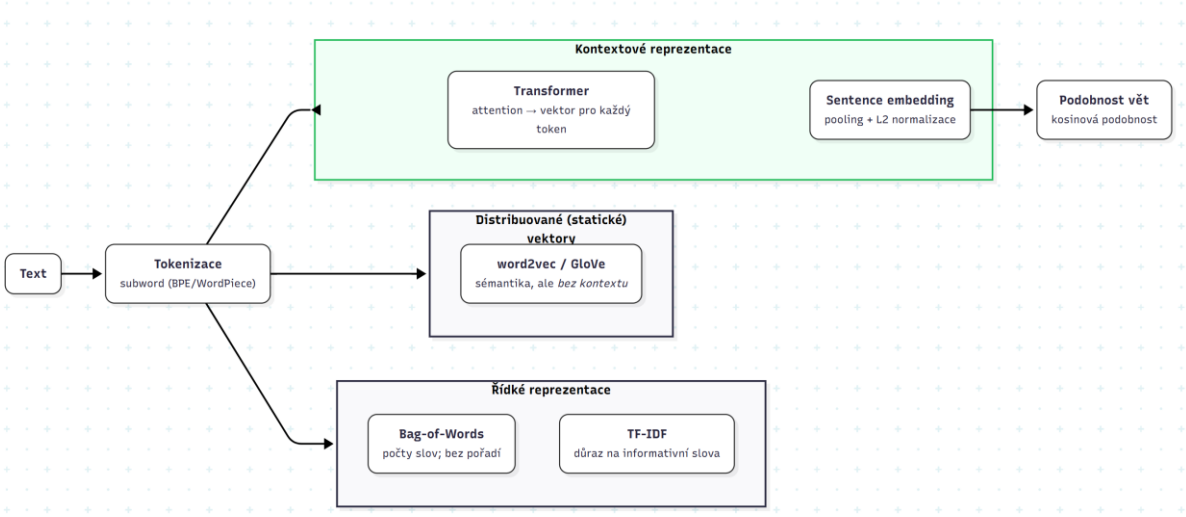
3.1 Reprezentace textu a modely

Každé zpracování začíná tokenizací, tedy rozkladem textu na jednotky, s nimiž model pracuje. Subword přístupy (např. BPE/WordPiece) zvládají tvary i novotvary, protože skládají slova z častých segmentů a tím snižují závislost na pevném slovníku [13], [2]. Tradiční bag-of-words či TF-IDF dávají řídké a srozumitelné vektory, ale ztrácejí pořadí slov a jemný kontext, což je problém u negací nebo frazeologie [5], [15]. Distribuované slovní vektory (word2vec, GloVe) sice přidaly sémantiku, ale bez ohledu na kontext: každé slovo má jeden vektor, i když v různých větách znamená něco jiného [13], [9]. Současným standardem jsou proto kontextové reprezentace z transformerů, kde vektor každého tokenu i celé věty závisí na okolí, to umožňuje porovnávat význam vět (sentence-embeddings) jednoduchou kosinovou podobností a přenášet znalosti mezi úlohami (transfer learning) [2], [16]. Důležitá je i praktická volba: kontextové vektory zpravidla lépe chápou význam a vícejazyčnost, ale jsou náročnější. Jednodušší reprezentace jsou levnější a snadněji vysvětlitelné, zato hůře zachytí nuance [1], [5].

3.2 Od volného textu ke strukturovaným výstupům

Teorie extrakce informací říká, že text je třeba převést na sadu rozhodnutí či struktur, které lze dále použít: třídu, sadu entit, škálované vlastnosti nebo předem dané pole ve schématu. K tomu se nabízí tři komplementární cesty. První je pravidlová a slovníková – vhodná pro jasné vzory, snadno vysvětlitelná, ale málo pružná. Druhá je učení s učitelem, model se naučí mapovat vstupy na výstupy z příkladů, poskytuje stabilní přesnost, vyžaduje však anotovaná data. Třetí je instrukční řízení generativních modelů: výstup má přesně danou formu (např. JSON se jmény polí a rozsahy hodnot) a model je veden, aby jej dodržel (*schema-driven decoding*) [2], [14],

[17]. Nezbytnou součástí teorie je práce s neurčitostí: negace a zeslabovače („*spíš*“, „*ne moc*“) by neměly mizet v předzpracování, ale propisovat se do výstupu. Vágní výrazy je vhodné převádět na intervaly, nikoli ostré hranice. Konfliktní požadavky se řeší prioritou nebo rozlišením mezi „tvrdou“ podmínkou a „měkkou“ preferencí [13], [5]. Validace a normalizace následují bezprostředně po inferenci, aby měl výsledek jednoznačnou interpretaci (povinná pole, rozsahy, standardizované štítky) [2].



Obrázek 3: Reprezentační cesta textu

3.3 Sémantická reprezentace a vyhledávání podle významu

Vedle symbolických struktur je užitečné mít i vektorový obraz významu. Větné embeddingy umožňují porovnávat krátké dotazy a věty bez přesných klíčových slov. Vektorové prostory bývají sdílené napříč jazyky, takže sémanticky blízké výroky leží poblíž, i když jsou v různých řečích [2], [16]. Teoreticky stačí normalizované vektory a kosinová podobnost, prakticky se řeší dvě věci: délka textu (dlouhé dokumenty se dělí na pasáže a agregují) a anizotropie (vektory mají tendenci „táhnout“ jedním směrem), kterou zmírní centrování a L2 normalizace [16]. Pro systémy, které generují odpovědi, je důležitá i myšlenka retrieval-augmented přístupu: nejprve vyhledat podpůrné pasáže a teprve poté tvořit výstup, čímž se snižuje riziko halucinací a zvyšuje ověřitelnost [2], [5]. Výsledkem je kombinace dvou komplementárních pohledů symbolického (jasná pole a pravidla) a sémantického (vektorový význam), které se navzájem posilují.

3.4 Spolehlivost, vyhodnocení a omezení

Teoretický rámec NLP stojí na jasném měření: u klasifikací se sleduje accuracy/precision/recall/F1, u extrakce kvalita detekce entit, u vyhledávání metriky typu

MRR/nDCG a u generování BLEU/ROUGE s vědomím, že tyto ukazatele hodnotí shodu s referencemi, nikoli pravdivost, proto se u generativních systémů stále častěji uplatňuje kombinace s retrievalem nebo kontrola fakticity [5], [13], [2]. Stejně podstatná je spolehlivost: jazykové modely mohou halucinovat, být zkreslené daty (bias) a vracet nevalidní struktury. Teoretická odpověď zahrnuje filtraci a kuraci tréninkových dat, konzervativní řízení dekódování, validaci proti schématu, jednoduché fallbacky a audit známých rizik [9], [2], [13]. Z hlediska efektivity hrají roli obecné principy strojového učení, kvantizace a distilace modelů, cachování a batchování požadavků, případně rozdělení úlohy na lehkou extrakci a těžší inference pouze tam, kde je to nutné. Smyslem těchto postupů není změnit podstatu modelů, ale dostat teoretické vlastnosti (porozumění, generování, sémantickou blízkost) do praxe při daných omezeních času a zdrojů [18], [9], [17].

PRAKTICKÁ ČÁST

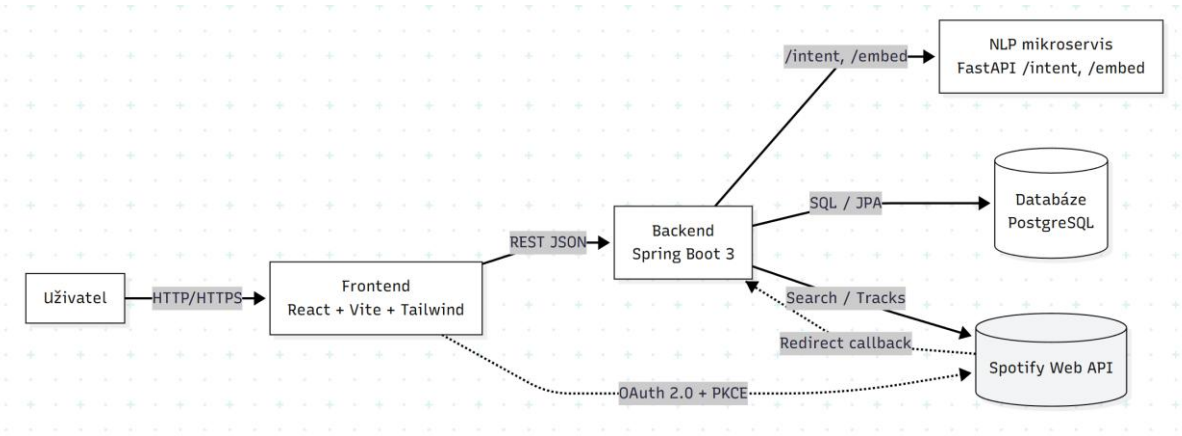
Cílem praktické části je navrhnout a implementovat aplikaci, která z uživatelského textového zadání v přirozeném jazyce automaticky sestaví doporučení skladeb na Spotify a umožní z nich vytvořit playlist. Důraz je kladen na převod neurčitého zadání do strukturovaného záměru (intent), sběr kandidátů přes Spotify Search API a jejich následné přeskórování (reranking) pomocí semantických reprezentací a měkkých omezení (mood, energia, tempo, jazyk apod.). Praktická část demonstruje, jak principy popsané v teoretickém úvodu převádíme do konkrétního, běžně nasaditelného řešení.

4. Architektura systému

Aplikace je rozdělena do tří samostatných částí:

- **Frontend** (React + Vite + Tailwind) poskytuje jednoduché a rychlé uživatelské rozhraní pro přihlášení přes Spotify a zadání promptu. Zobrazuje výsledné kandidáty a umožňuje playlist uložit do účtu uživatele.
- **Backend** (Spring Boot 3) slouží jako orchestrátor – provádí OAuth tok se Spotify (Authorization Code s PKCE), volá NLP mikroservis pro parsování záměru, dotazuje Spotify Search API pro sběr kandidátů, aplikuje řazení a diverzifikaci a vrací finální výběr.
- **NLP mikroservis** (FastAPI) poskytuje dvě klíčové funkce: robustní parsování záměru z přirozeného jazyka do striktní JSON schémy (včetně sanitizace a normalizace) s využitím lokálně běžícího LLM, a generování embedů pro semantické skórování.

Z hlediska datových toků uživatel komunikuje pouze s frontendem, který přes REST rozhraní volá backend. Backend následně komunikuje jednak s NLP mikroservisem (endpointy /intent a /embed), jednak se Spotify Web API (vyhledávání skladeb/interpretů a práce s playlisty). Tok autorizace Spotify probíhá v prohlížeči uživatele a po přesměrování je dokončen na backendu, který zajišťuje bezpečnou práci s přístupovými tokeny.



Obrázek 4: Architektura *systému a datové toky*

5. Funkční a nefunkční požadavky

Tato část formálně vymezuje, co má systém dělat (funkční požadavky) a jak dobře to má dělat (nefunkční požadavky). Požadavky vycházejí z hlavního cíle aplikace – převést volně formulovaný text uživatele do playlistu na Spotify – a z architektury popsané v předchozí kapitole (frontend, backend–orchestrátor, NLP mikroservis, databáze, integrace na Spotify Web API). Důraz klademe na měřitelnost, sledovatelnost k architektonickým komponentám a reálné provozní limity (OAuth 2.0 s PKCE, rate-limity Spotify, latence LLM/NLP).

5.1 Funkční požadavky

Tabulka 3: Funkční požadavky

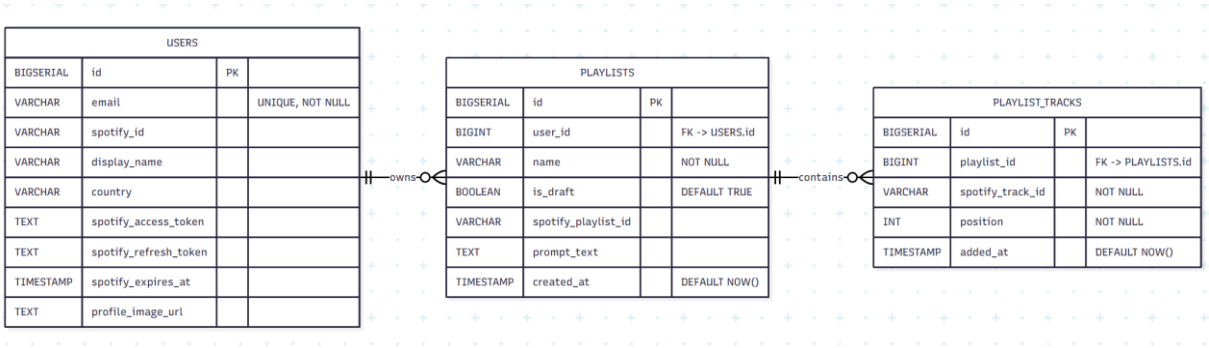
F1	Přihlášení přes Spotify	Aplikace musí zprostředkovat bezpečné přihlášení přes Spotify (OAuth 2.0 s PKCE) a získat access/refresh tokeny.
F2	Odhlášení	Aplikace musí umožnit odhlášení uživatele a bezpečné ukončení relace (zrušení session/tokenů).
F3	Zadání promptu	Aplikace musí umožnit zadat volně formulovaný textový prompt a odeslat jej na backend k dalšímu zpracování.
F4	Parsování záměru (NLP)	Aplikace musí pro každý prompt zavolat NLP endpoint /intent a přijmout validní JSON dle schématu.
F5	Respektování jazykových preferencí	Aplikace musí respektovat explicitně uvedený jazyk (ISO-639-1) a volbu „languages_strict“.
F6	Sběr kandidátů (Spotify Search)	Aplikace musí z dotazu vygenerovat 1–3 vyhledávací dotazy do Spotify a stáhnout relevantní kandidáty.
F7	Filtrování duplicit a „seen“	Aplikace musí odfiltrovat duplicitní skladby a zohlednit dříve „viděné“ položky uživatele.
F8	Reranking a diverzita	Aplikace musí přeskórovat kandidáty (semantika + shoda s targets + jazyk) a zajistit diverzitu (např. MMR).
F9	Zobrazení výsledků	Aplikace musí zobrazit 20–40 skladeb s názvem, interpretem a odkazem do Spotify.
F10	Uložení playlistu	Aplikace musí umožnit vytvořit playlist v účtu uživatele a přidat do něj vybrané skladby.
F11	Historie dotazů	Aplikace musí ukládat a zobrazovat historii promptů s možností opakování/úprav.
F12	Chybové stavy a retry	Aplikace musí srozumitelně informovat o chybách (NLP/Spotify) a bezpečně opakovat dočasně selhaná volání.

5.2 Nefunkční požadavky

Tabulka 4: Nefunkční požadavky

N1	Výkonnost end-to-end	Systém musí doručit výsledek pro běžný prompt s p95 latencí ≤ 3,5 s.
N2	Výkonnost NLP	Systém musí zpracovat volání /intent s p95 latencí ≤ 800 ms
N3	Výkonnost Spotify Search	Systém musí u každého vyhledávacího dotazu dosáhnout p95 latence ≤ 700 ms; max. 3 dotazy na prompt.
N4	Výkon rerankingu	Systém musí přeskórovat 40 kandidátů do ≤ 150 ms na instanci.
N5	Dostupnost	Systém musí dosahovat dostupnosti ≥ 99 % během hodnoticího období.
N6	Škálovatelnost	Systém musí být horizontálně škálovatelný (stateless backend/NLP, readiness/liveness).
N7	Bezpečnost tokenů	Systém musí bezpečně nakládat s tokeny (server-side storage, HTTPS, CORS whitelist).
N8	Observabilita	Systém musí poskytovat strukturované logy, korelační ID a základní metriky latencí a chybovosti.
N9	Kompatibilita a UX	Systém musí fungovat v aktuálních verzích hlavních prohlížečů a být responzivní (desktop/mobil).
N10	Soulad s limity Spotify	Systém musí respektovat rate-limity a používat exponenciální backoff při chybách 429/5xx.
N11	Správa konfigurace	Systém musí využívat konfigurační proměnné (.env/secret store) mimo verzovací systém.

6. Datový model a perzistence



Obrázek 5: Relační schéma databáze

Schéma je záměrně minimalistické a tvoří ho tři tabulky: **USERS**, **PLAYLISTS** a **PLAYLIST_TRACKS**. USERS eviduje identitu uživatele a přístup ke Spotify (OAuth), PLAYLISTS představuje hlavičku playlistu vzniklého z promptu (včetně případného propojení na playlist ve Spotify) a PLAYLIST_TRACKS tvoří pořadí konkrétních skladeb v rámci daného playlistu. Žádná lokální hesla neuchováváme, autentizace probíhá výhradně přes Spotify, čímž se zjednodušuje bezpečnostní model.

6.1 Vazby a integrita

Vztahy jsou jednoduše stromové: USERS 1–N PLAYLISTS a PLAYLISTS 1–N PLAYLIST_TRACKS. Vynucujeme referenční integritu s ON DELETE CASCADE, aby smazání uživatele automaticky odstranilo jeho playlisty i jejich položky. V rámci každého playlistu držíme jedinečnost skladeb (unikátní dvojice playlist_id, spotify_track_id), čímž bráníme duplicitám, a zároveň explicitně ukládáme pořadí skladeb. Schéma ponechává metadata skladeb (název, interpret, obrázky) na straně Spotify. V databázi uchováváme jen identifikátor, aby se při zobrazení vždy použila aktuální data z API.

6.2 Bezpečnost a správa tajemství

OAuth tokeny Spotify ukládáme pouze server-side (USERS) a pracujeme s nimi výhradně na backendu. Frontend k nim nepřistupuje. Přístupové údaje a klíče jsou předávány přes proměnné prostředí a mimo repozitář. Tímto návrhem minimalizujeme úniky citlivých údajů a zjednodušujeme audit.

6.3 Realizace v aplikaci

Relační databáze je jedinou komponentou spuštěnou v Dockeru. Databázový kontejner vychází z oficiálního obrazu PostgreSQL 15 a používá persistenční svazek (volume) pro uchování dat

mimo životní cyklus kontejneru. Inicializace schématu probíhá automaticky pomocí připraveného DDL skriptu “init.sql“. Toto uspořádání spojuje jednoduchost lokálního běhu aplikace s reprodukovatelným a stabilním databázovým prostředím, vhodným napříč vývojovými i provozními scénář

7. Backend Spring Boot

Spring Boot je moderní nadstavba rámce Spring, která zjednodušuje tvorbu produkčně připravených serverových aplikací. Opírá se o autokonfiguraci, princip convention over configuration a dependency injection, nabízí hotové startery pro web, validaci, bezpečnost a perzistenci a přirozeně podporuje Spring Web MVC pro stavbu REST rozhraní. Díky tomu lze rychle sestavit vrstvenou, testovatelnou a provozně stabilní aplikaci bez nadbytečné konfigurační režie.

V kontextu této práce plní backend roli rozhraní mezi uživatelem a okolním světem služeb: přijímá požadavky, uplatňuje pravidla systému (validace, bezpečnost, limity), koordinačně volá specializované subsystémy (jazykové zpracování, platformní API) a zajišťuje konzistenci dat v relační databázi. Je to vědomě centralizovaný orchestrátor synchronního toku: na jednom místě drží posloupnost kroků, sjednocuje chybové stavy a garantuje jednoznačné SLA pro celý průchod požadavku. Tím zároveň izoluje klienta od technických detailů integrací a chrání doménu před změnami externích rozhraní. Výsledkem je tenký, predikovatelný klient, specializované podpůrné služby a stabilní serverový střed, který udržuje bezpečnost, odolnost a sledovatelnost celého řešení.

7.1 Struktura backendové části

Obrázek 6: Struktura zdrojového kódu backendové aplikace

Backend je členěn do několika svébytných modulů, které dohromady skládají čistou, vrstvenou architekturu nad Spring Web MVC. Vstupní balík upce.kzrv.music_reco drží „spouštěcí bod“ aplikace (MusicRecoApplication) a definuje hranice, za nimiž se jednotlivé části setkávají formou jasných rozhraní. Uvnitř nenajdeme monolitický „god class“, ale mozaiku malých, srozumitelných stavebních kamenů.

config/ je tichým dirigentem celého orchestru. Schovává autokonfigurační beany, nastavení HTTP klientů (časové limity, politiku opakování bezpečných volání), bezpečnostní pravidla i typované „properties“. Díky tomu je zbytek kódu zbaven zbytečné nízkoúrovňové práce a může

se soustředit na samotnou doménu. Závislosti i omezení jsou přitom vyjádřeny jasně, přehledně a soustředěny na jednom místě.

common/ poskytuje průřezové nástroje a jednotnou práci s chybami. Chybová vrstva mapuje výjimky na konzistentní HTTP odpovědi tak, aby klient vždy dostal srozumitelnou zprávu a korelační identifikátor pro dohledání v logu. Bez této vrstvy by se diagnostika rozplynula do jednotlivých modulů.

dto/ tvoří čistou hranici mezi naším API a okolním světem. Zde žijí transportní objekty pro autentizaci, historii, NLP i doporučování a také modely odpovědí ze Spotify. Udržujeme je schválně „chudé“ a bez logiky. Umožňují nám měnit interní implementaci, aniž bychom porušili kontrakt s frontendem nebo se Spotify.

auth/ zapouzdřuje přihlášení přes Spotify. Kontroler vystavuje minimalistické konce pro login a callback, servis provádí výměnu autorizačního kódu za tokeny, stará se o jejich obnovu a bezpečné uložení. Frontend nikdy nepřichází do styku s citlivými údaji, vše zůstává na serveru a řídí se politikou definovanou v konfiguraci.

user/ poskytuje elementární operace nad profilem aktuálního uživatele. Vedle kontroleru „/me“ má jednoduchou perzistenci nad tabulkou users.

playlist/ je domovem aplikačních případů kolem tvorby a správy playlistů. PromptController přijímá textové zadání, PlaylistService orchestrace zápisů do DB a případně vytváření playlistů ve Spotify. Entitní a repozitářová podvrstva drží hlavičky playlistů i jejich položky s pořadím, vše transakčně, aby byl výsledek vždy konzistentní.

history/ odemyká uživateli zpětný pohled na práci systému. Zajišťuje načítání a formátování záznamů tak, aby šlo snadno navázat na minulé kroky nebo porovnat výstupy.

recommendation/ představuje srdce domény. RecoService funguje jako orchestrátor: přijme prompt, požádá NLP o striktní intent, sestaví dotazy do vyhledávání, stáhne kandidáty, aplikuje skórování a diverzifikaci, a nakonec výsledek uloží. QueryBuilder převádí „lidské“ nápovědy na bezpečné dotazy, Ranker skládá hybridní skóre a hlídá rozmanitost, RecoResult nese výsledek pro UI. NlpClient je jedinou cestou, jak backend mluví s NLP – drží kontrakt a technické detaily volání mimo doménu.

spotify/ je pečlivě navržená adaptérová vrstva k Web API Spotify. SpotifyClient obsluhuje vyhledávání i práci s playlisty, SpotifyAuthService a SpotifyTokenService zajišťují správu tokenů a jejich obnovu, SpotifyParamBuilder hlídá čistotu a bezpečnost parametrů dotazů. Jde

o učebnicový „anti-corruption layer“: vnitřní kód nikdy neřeší nuance cizího API, dostává již přeložená, stabilní data.

Výše uvedené členění vymezuje hranice zodpovědností v backendu – od konfigurace a průřezových služeb přes autentizaci, uživatele a playlistovou logiku až po jádro doporučování a integraci na Spotify. V následujících podkapitolách se k jednotlivým modulům vrátíme detailněji.

7.2 Konfigurační soubor

```yaml
spring:
  application:
    name: music_reco
  datasource:
    url: jdbc:postgresql://localhost:5432/mydb
    username: user
    password: password
  jpa:
    hibernate:
      ddl-auto: none
    show-sql: false
    properties:
      hibernate:
        format_sql: true
  codec:
    max-in-memory-size: 16MB
spotify:
  client-id: ****
  client-secret: ****
  redirect-uri: http://localhost:8080/api/auth/callback
jwt:
  secret: ****

app:
  nlp:
    base-url: http://localhost:8000
  spotify:
    api-base: https://api.spotify.com/v1
  reco:
    desired-cap: 40
    per-query: 50
    offsets: [ 0, 50, 100 ]
    max-candidates: 1200
    artist-cap: 2
    year-slack: 1
    ban-title-search-exclude: true
    ban-title-exact-penalty: 0.55
    ban-title-partial-penalty: 0.85
    ban-title-partial-title-max-words: 2
    weights:
      sem: 0.65
      genre: 0.20
      year: 0.10
      pop: 0.05
      namePenalty: 0.20
    name-bias:
      stopwords: [ "focus","study","melancholic","melancholy","relax","chill","sleep","workout","training","sad","happy","calm" ]
    spotify:
      playlistFetchConcurrency: 2
      requestSpacingMs: 150
      retryDefaultSeconds: 2
      retryMaxSeconds: 8
    mmr:
      enabled: true
      lambda: 0.75
      pool: 200
    personalization:
      seenPenalty: 0.85
```

Obrázek 7: Soubor application.yaml

Soubor „application.yaml“ centralizuje běhovou konfiguraci služby *music_reco*: název aplikace, připojení k PostgreSQL, chování JPA/Hibernate a limity dekodéru, dále přihlašovací

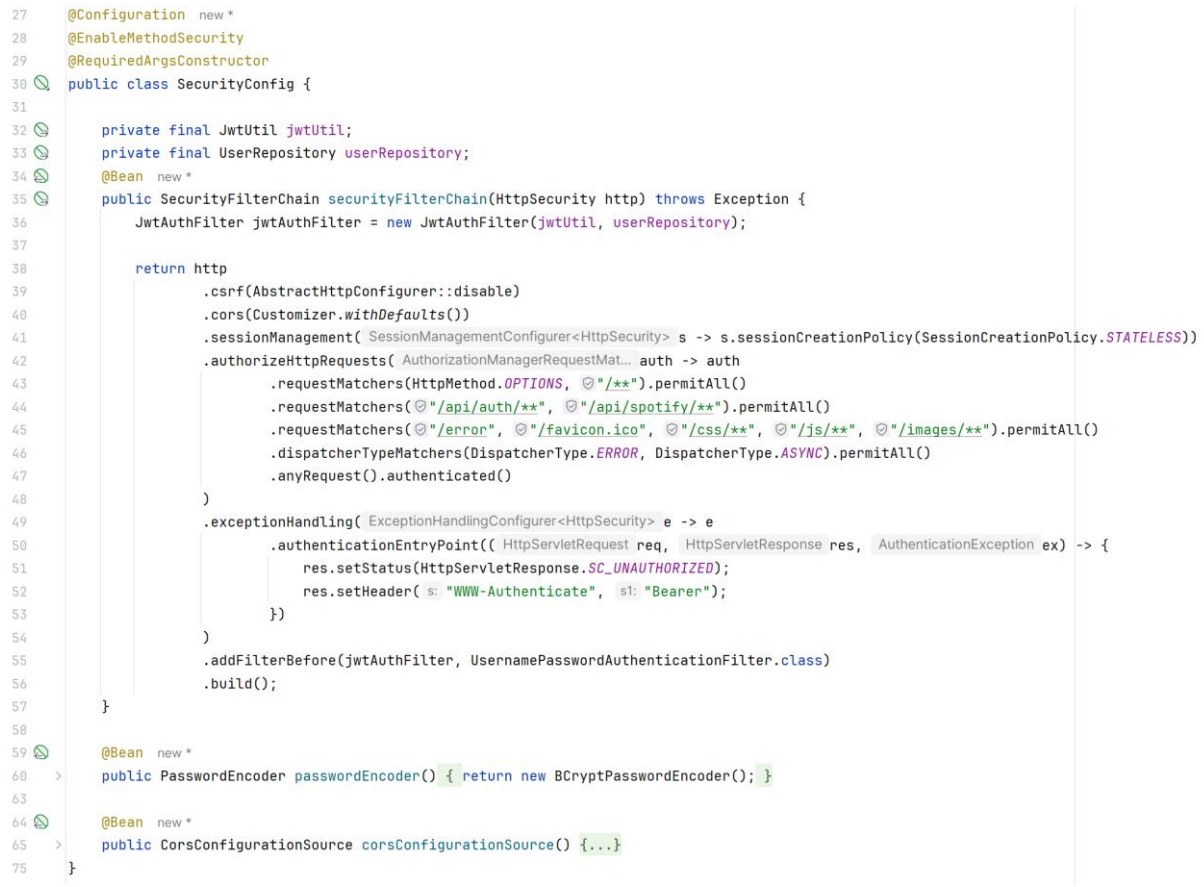
údaje pro Spotify OAuth a tajemství pro JWT. Doménová sekce app.* parametrizuje NLP a doporučovací logiku (adresy služeb, objemy a stránkování sběru kandidátů, váhy skórování, MMR-diverzifikaci a lehkou personalizaci).

7.3 Autentizace a autorizace: OAuth2 PKCE (Spotify) a interní JWT

V aplikaci uplatňujeme **dvoustupňový model identity**: nejprve proběhne externí ověření uživatele u poskytovatele (Spotify) pomocí standardu OAuth 2.0 – Authorization Code s PKCE, poté jsou všechna volání našeho API autorizována interním krátkodobým JWT. PKCE (Proof Key for Code Exchange) rozšiřuje klasický autorizační kód o dvojici code_verifier/code_challenge a spolu s parametrem state brání odcizení či podvržení toku při přesměrování. Server po návratu z OAuth vymění kód za access/refresh tokeny a z profilu Spotify jednoznačně určí uživatele. Pro frontend následně vystaví vlastní JWT, tj. kompaktní, digitálně podepsaný token s minimálními nároky. Díky tomu je volání našeho API stateless a rychlé (ověření podpisu probíhá lokálně), interní autorizace je oddělena od detailů OAuth/Spotify a klient nikdy nemanipuluje s přístupovými tokeny ke Spotify.

Tok přihlášení v kostce:

- Klient zavolá náš login endpoint. Server vygeneruje code_verifier, připraví state a přesměruje prohlížeč na autorizační URL Spotify (se zvolenými scopes).
- Spotify po úspěchu vrátí prohlížeč na náš callback s kódem a state. Server state ověří a vymění kód za access/refresh tokeny.
- Server načte profil uživatele (Spotify ID, display name, country), vytvoří/aktualizuje záznam v DB a vydá interní JWT.
- Další požadavky klienta nesou Authorization: Bearer <jwt>, náš filtr JWT token ověří a propustí na chráněné endpointy.



```java
@Configuration  new *
@EnableMethodSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    private final JwtUtil jwtUtil;
    private final UserRepository userRepository;
    @Bean  new *
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        JwtAuthFilter jwtAuthFilter = new JwtAuthFilter(jwtUtil, userRepository);

        return http
                .csrf(AbstractHttpConfigurer::disable)
                .cors(Customizer.withDefaults())
                .sessionManagement( SessionManagementConfigurer<HttpSecurity> s -> s.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> auth
                        .requestMatchers(HttpMethod.OPTIONS, "/**").permitAll()
                        .requestMatchers("/api/auth/**", "/api/spotify/**").permitAll()
                        .requestMatchers("/error", "/favicon.ico", "/css/**", "/js/**", "/images/**").permitAll()
                        .dispatcherTypeMatchers(DispatcherType.ERROR, DispatcherType.ASYNC).permitAll()
                        .anyRequest().authenticated()
                )
                .exceptionHandling( ExceptionHandlingConfigurer<HttpSecurity> e -> e
                        .authenticationEntryPoint(( HttpServletRequest req,  HttpServletResponse res,  AuthenticationException ex) -> {
                            res.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
                            res.setHeader( s: "WWW-Authenticate",  s1: "Bearer");
                        })
                )
                .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)
                .build();
    }

    @Bean  new *
    public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }

    @Bean  new *
    public CorsConfigurationSource corsConfigurationSource() {...}
}
```

Obrázek 8: SecurityConfig.java

SecurityConfig – konfiguruje Spring Security pro stateless provoz. Definuje veřejné cesty (např. /api/auth/login, /api/auth/callback) a chráněné oblasti API, zapíná CORS podle prostředí a registruje JwtAuthFilter do filtračního řetězce. Chování neautorizovaných/zakázaných požadavků sjednocuje na jasné HTTP kódy (401/403) a vypíná serverovou session.

```java
@Slf4j  2 usages  new *
@RequiredArgsConstructor
public class JwtAuthFilter extends OncePerRequestFilter {

    private final JwtUtil jwtUtil;
    private final UserRepository userRepository;

    @Override  no usages  new *
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain) throws IOException, ServletException {...}

    private void unauthorized(HttpServletResponse res, String msg) throws IOException {...}

    @Override  no usages  new *
    protected boolean shouldNotFilterErrorDispatch() { return true; }
}
```

Obrázek 9: JwtAuthFilter.java

JwtAuthFilter – zachytává požadavky na chráněné endpointy, z hlavičky Authorization čte Bearer token, pomocí JwtUtil ověřuje podpis (tajemství z konfigurace), časovou platnost a povinné nároky. Při neplatném tokenu vrací standardizovanou odpověď a do logu zapisuje korelační ID pro dohledání.

```
14    @Component  6 usages  new *
15    @RequiredArgsConstructor
16    public class JwtUtil {
17
18        @Value("YW55IHNlY3JldCBrZXkgd2lsbCB3b3JrIGZvciBleGFtcGxlIHN5c3RlbQ==")
19        private String secret;
20
21        private static final long EXPIRATION_MS = 24L * 60 * 60 * 1000;  1 usage
22
23        public String generateToken(String email) {...}
33        public String extractEmail(String token) { return getClaims(token).getBody().getSubject(); }
36
37        public boolean validateToken(String token, User expectedUser) {...}
51
52        private Jws<Claims> getClaims(String token) {...}
58
59        private Key getSigningKey() {...}
63    }
```

Obrázek 10: JwtUtil.java

JwtUtil – zapouzdřuje generování a verifikaci interních JWT. Při vydání nastavuje krátkou expiraci a minimální identitu (typicky userId), volitelně doplňuje nenutné „claims“. Při ověřování provádí bezpečné parsování a rozlišuje typy chyb (podpis, expirace, struktura).

```
14    @RestController  new *
15    @RequestMapping("/api")
16    @RequiredArgsConstructor
17    public class AuthController {
18
19        private final AuthService authService;
20        private final SpotifyAuthService spotifyAuthService;
21        private final JwtUtil jwtUtil;
22
23        @GetMapping("/auth/login")  new *
24        public void redirectToSpotify(HttpServletResponse response) throws IOException {
25            String url = spotifyAuthService.buildAuthorizationUri();
26            response.sendRedirect(url);
27        }
28
29        @GetMapping("/auth/callback")  new *
30        public void handleSpotifyCallback(@RequestParam String code, HttpServletResponse response) throws IOException {
31            SpotifyTokenResponse token = spotifyAuthService.exchangeCodeForToken(code);
32            SpotifyProfile profile = spotifyAuthService.getSpotifyProfile(token.getAccessToken());
33
34            User user = authService.createOrUpdateUser(profile, token);
35            String jwt = jwtUtil.generateToken(user.getEmail());
36
37            response.sendRedirect( s: "http://localhost:5173/?token=" + jwt);
38        }
39    }
```

Obrázek 11: AuthController.java

AuthController – vstupní bod přihlášení a návratu z OAuth. Inicializuje PKCE tok (generuje code_verifier/challenge, state, skládá autorizační URL), obsluhuje callback, deleguje výměnu tokenů a po úspěchu předává klientovi interní JWT.

Po vstupu do AuthController přebírá řízení AuthService, který kompaktně zorchestrovává celý OAuth tok: přes SpotifyAuthService připraví PKCE a autorizační URL, po návratu z OAuth využije SpotifyTokenService k výměně kódu za access/refresh tokeny a přes SpotifyClient získá profil uživatele. Identita se následně **upsertuje** v databázi pomocí UserRepository (včetně uložení refresh tokenu a expirace), načež JwtUtil vydá krátkodobé interní JWT se subjektem uživatele.

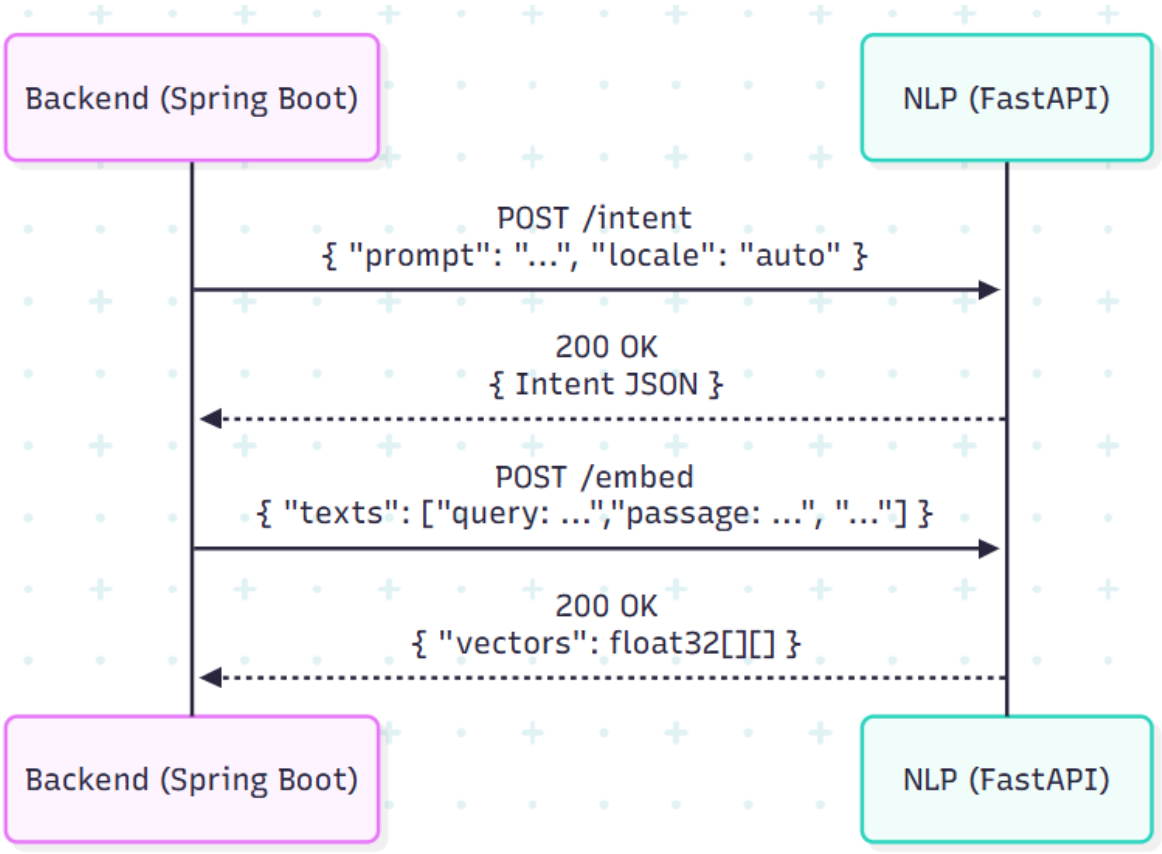
Další volání API se autentizují přes Authorization: Bearer <jwt> a JwtAuthFilter je ověřuje ještě před vstupem do kontrolerů. Při expiraci přístupového tokenu vůči Spotify probíhá tichý refresh přes SpotifyTokenService, který SpotifyClient transparentně využije.

8. NLP mikroservis

NLP vrstva je oddělená, protože naším cílem není „magie textu“, ale spolehlivý převod neurčitého zadání do přesných řídicích signálů pro doporučovací jádro. Volíme k tomu lokálně provozovaný jazykový model: chráníme tím citlivá data uživatelů, máme pod kontrolou verzi i chování modelu a eliminujeme závislost na externí infrastruktuře, latenci i proměnlivé náklady za volání. Lokální nasazení navíc umožňuje auditovat celý, opakovat experimenty beze změn podmínek a konzistentně měřit dopady úprav.

Jako běhové prostředí volíme Ollama, protože nabízí jednoduché a stabilní rozhraní pro lokální LLM (jednoduchý HTTP endpoint, správa modelů, snadná výměna verzí/kvantizací) bez nutnosti stavět vlastní serving stack. Tato volba je pragmatická: dovoluje nám rychle iterovat nad promptem a pravidly, držet latenci v mezích interaktivní aplikace a přitom zachovat determinismus na úrovni celého systému – co do něj vstoupí, to jsme schopni přesně popsat, validovat a v případě potřeby zreprodukovat.

Python zde není volbou „zvyku“, ale nástrojem: jeho ekosystém přirozeně spojuje práci s modely (sentence-transformers), robustní validaci (Pydantic) a lehké webové rozhraní (FastAPI). NLP služba se tím stává překládací vrstvou mezi uživatelovým jazykem a technickým světem doporučování: z textu vzniká striktní Intent se sjednocenými významy a z kontextu písní vektory pro semantické srovnání.



Obrázek 12: Komunikace mezi Spring Boot a NLP service

8.1 Intent – účel, pravidla a zpracování

Intent převádí volné uživatelské zadání do jednoznačného, strojově čitelného záměru (Intent JSON), se kterým backend dokáže deterministicky pracovat při sběru kandidátů a následném skórování. Cílem je odstranit neurčitost přirozeného jazyka a sjednotit významy (jazyk, rok, nálada, cílové hudební rysy) do předem daného kontraktu, aby žádný další krok pipeline nemusel dohánět „domněnky“ nebo řešit okrajové případy.

Backend volá POST /intent se strukturou { prompt, locale }. Služba vytvoří vstup pro LLM, získá návrh Intentu, upravený výstup sanitizuje a validuje, případně doplní nápovědy, a vrací striktní JSON dle schématu.

```python
@app.post( path: "/intent", response_model=Intent)
async def route_intent(req: IntentReq):
    raw = await llm_intent(req.prompt, CATALOG.seeds)
    raw = sanitize(raw)

    try:
        intent = Intent.model_validate(raw)
    except ValidationError as e:
        raise HTTPException( status_code: 422,  detail: f"Bad intent JSON from LLM: {e}")

    fixed_seeds, fixed_freeform = normalize_seed_genres(intent.seed_genres, intent.genres_freeform)
    intent.seed_genres = fixed_seeds
    intent.genres_freeform = fixed_freeform

    if not intent.query_hints:
        hints = []
        if intent.mood: hints.append(intent.mood)
        if intent.activity and intent.activity != "none": hints.append(intent.activity)
        hints.extend(intent.genres_freeform[:3])
        intent.query_hints = [h for h in dict.fromkeys(hints) if h]

    if not intent.artist_hints:
        intent.artist_hints = []

    return intent
```

Obrázek 13: NLP Service - /intent

8.1.1 System-prompt a způsob interpretace

```
SYSTEM_INSTRUCTIONS = """
You are a music intent parser. Return a STRICT JSON object in English with lowercase keys, and NO extra text.

Schema (use exactly these keys):
{
  "mood": "melancholic|happy|dark|chill|energetic|romantic|nostalgic|focus|other",
  "seed_genres": ["..."],            // english, normalized; if unsure, leave empty
  "genres_freeform": ["..."],        // any user words/tags (any language)
  "language": ["<iso-639-1>"|"any", ...],
  "year_range": null | { "from": 1980, "to": 1989 },
  "activity": "workout|study|relax|party|drive|sleep|none",
  "instrumental_ratio": 0.0,         // 0..1 (soft quota)
  "targets": {
    "valence": [0,1],
    "energy": [0,1],
    "danceability": [0,1],
    "tempo": [40,220],
    "acousticness": [0,1],
    "instrumentalness": [0,1],
    "speechiness": [0,1]
  },
  "constraints": {
    "languages_strict": false,
    "must_be_in_decade": false
  },
  "notes": "",
  "query_hints": [],                 // short search phrases
  "artist_hints": []                 // only if user explicitly named artists
  "playlist_name": ""                // short human title, 2-5 words, no quotes/emojis
}

Rules:
- Respond ONLY with valid JSON matching the schema. No commentary.
- Use english lowercase for technical fields. For user-like phrases in query_hints keep original phrasing/language if helpful.
- Language:
  - Do NOT infer the target language from the language of the user's prompt.
  - Set "language" ONLY when the user explicitly refers to a target language (in any wording or language), e.g. "in english", "english songs", "на английском", "по-английски", "en español", etc.
  - Use ISO-639-1 codes only (e.g., "en", "ru", "es"). If uncertain, use ["any"].
  - If the user explicitly requests a specific language (e.g., "in english", "только на английском", "english only"), set language=[<iso>] and constraints.languages_strict=true.
  - If the user mentions multiple target languages (e.g., "english or spanish"), set both (e.g., ["en","es"]) and constraints.languages_strict=false.
  - Do NOT set "language" from genre or demonym words alone (e.g., "french house", "latin pop" are genres; keep language=["any"] unless a song language is explicitly requested).
- Years:
  - NEVER set year_range unless the user EXPLICITLY mentioned concrete years or a decade (e.g., "1990-1999", "90s").
  - Words like "nostalgic", "vintage", "old", "study", "focus" alone are NOT sufficient to set year_range.
- Seed genres:
  - Include only normalized, standard english genre tags you are confident about. Otherwise leave empty and prefer genres_freeform/query_hints.
  - Do not derive language from genre names (e.g., "french house" does not imply "fr").
- query_hints:
  - Provide 5-10 concise phrases helpful for search.
  - Avoid single generic words as the only hints (e.g., just "focus"). Prefer descriptive combinations (e.g., "deep focus", "ambient study", "post-punk classics").
  - Include reasonable variants (e.g., hyphenated and spaced forms: "post-punk", "post punk").
  - Do not include plain language words as hints (e.g., "english") unless they are part of a meaningful phrase.
- targets:
  - Prefer ranges (min..max), consistent with the mood/activity. Do not produce contradictory ranges.
- artist_hints:
  - Only include if the user explicitly named or clearly implied specific artists; otherwise keep empty.
- playlist_name:
  - 2-5 words, descriptive (e.g., "deep focus ambient", "melancholic indie"),
  - do NOT include years unless user explicitly requested a decade/years,
  - avoid bland single words like "focus" alone,
  - prefer the user's requested target language if any; otherwise the user's prompt language.
"""
```

Obrázek 14: NLP service – prompt

Model neprosíme o „kreativní nápady“, ale nutíme jej předat jediný, formálně správný objekt přesně podle našeho schématu. Proto je prompt formulován normativně: stanovuje, že výstupem musí být čistý JSON (bez textu okolo) s pevnými klíči a očekávanými tvary hodnot. Do uživatelské zprávy zároveň injektujeme kurátorovaný seznam povolených žánrů, čímž výrazně zmenšujeme prostor, ve kterém se model může mýlit nebo si „vymýšlet“ tagy. Nízká teplota vzorkování a lokální běh přes Ollama zajišťují stabilitu, totéž zadání vede k témuž druhu odpovědi a latence zůstává v mezích interaktivního použití.

Klíčové je, že některé informace se nesmí domýšlet. Například jazyk skladeb nastavujeme pouze tehdy, když si o něj uživatel výslovně řekne („anglicky“, „en español“, „český“). Pokud to neudělá, výsledek zůstává u neutrální hodnoty [„any”]. Tím se vyhneme časté chybě, kdy by se z výrazu „french house“ mylně vyvozovala francouzština, v našem pojetí je to žánr, nikoli jazykový požadavek. Stejná opatrnost platí pro časové omezení: pole year_range se vyplní

pouze tehdy, když se v promptu objeví konkrétní roky nebo jasně odkazovaná dekáda („1995–2005“, „90s“). Věty typu „nostalgic indie“ nebo „old school hip-hop“ nejsou samy o sobě dostatečné k tomu, abychom zamkli výběr na 90. léta. Nostalgie se raději projeví v jiných signálech (nižší tempo, jemnější energie, příslušné query_hints), než abychom tvrdě omezili období a zbytečně ochudili kandidáty.

Zároveň rozlišujeme kanonické žánry a volný slovník. To, co model s vysokou jistotou přiřadí do našeho katalogu (např. „indie-pop“, „drum-and-bass“), skončí v seed_genres a stává se přesným filtrem. Neurčité nebo uživatelské výrazy zůstávají ve genres_freeform a slouží jako vodítka pro vyhledávání. Praktický efekt je dvojí: jednak máme spolehlivé, strojově kontrolovatelné tagy, jednak neztrácíme nuance z přirozeného jazyka. Tato dvojkolejnost je důležitá – dotaz „melancholic indie for study“ přinese vedle seedů i „study / melancholic“ jako nápovědy, což u Spotify Search funguje lépe než snaha „vše nacpat do jednoho pole“.

Dalším prvkem discipliny jsou intervaly cílových rysů. Místo aby model řekl „energy = 0.3“, dáváme mu rámec „energy v rozmezí 0.2–0.5“. Ranker pak neporovnává křehká čísla, ale blízkost pásmu – skladba těsně za hranicí se neodsoudí k nule, jen dostane menší bonus. U tempa používáme rozmezí 40–220 BPM, u dalších rysů škálu 0..1, to odráží realitu hudebních dat a opět snižuje riziko extrémů. Když uživatel řekne „party bangers“, model typicky zvedne energy a danceability do vyšších intervalů, naopak „deep focus ambient“ srazí tempo a valence. Tím se „význam“ přirozených slov převádí do parametrů, se kterými lze nadále počítat.

Specifickou roli hrají query_hints: krátké, vyhledávání-přátelské fráze (5–10 položek), často i ve dvou variantách („post-punk“ / „post punk“). Jejich úkolem není „být hezké“, ale fungovat ve vyhledávání, proto model vedeme k tomu, aby se vyhnul obecným slovům bez kontextu a naopak kombinoval užitečné výrazy. Pokud jsou vstupy příliš strohé, backend má připravený rozumný fallback (doplní nápovědy z nálady/aktivity a volných žánrů), ale díky promptu se k němu uchylujeme méně často.

Nakonec, prompt pamatuje i na okrajové formulace. Když uživatel napíše „english or spanish“, model předá language: ["en","es"], ale současně nevynutí jazykovou striktnost – ponechá prostor, aby se doporučení mohla rozkročit podle relevance. Pokud někdo žádá „english only“, explicitně se nastaví languages_strict=true. Tato jemná politika je důležitá pro spokojenost uživatele: respektujeme jeho přání, ale nenecháme se zavléct k příliš úzké filtraci, když to výslovně nepožádal.

Přes všechna pravidla zůstává jazykový model pravděpodobnostním systémem: někdy vrátí JSON v kódovém bloku, jindy „uhne“ na hraně intervalu nebo zvolí opatrnější interpretaci. Proto prompt není jediná obrana – jeho výstup vždy prochází sanitizací a validací. Díky tomu se nečekané drobnosti vyhladí dřív, než data dorazí do jádra doporučování, a my si udržíme to, co je nejdůležitější: předvídatelný, opakovatelný a srozumitelný vstup do další fáze pipeline.

8.1.2 Kontrola, sanitizace a návrat výsledku

Nejprve z odpovědi LLM spolehlivě vydolujeme čistý JSON: odstraníme případné Markdown bloky a, když je to nutné, vytáhneme první platný objekt regulárním výrazem. Pokud JSON nelze bezpečně získat, vrací služba 500 s krátkým popisem problému, tím končíme hned u zdroje a nedovolíme, aby „rozbitý“ výstup pokračoval do doporučování. Následuje sanitizace, která z návrhu udělá datově ukázněný objekt: jazyk normalizujeme na ISO-639-1 (nebo ["any"] bez míchání s jinými kódy), intervaly všech rysů ořízneme do povolených mezí a zajistíme pořadí min ≤ max, číselné poměry (např. instrumental_ratio) zůstanou v rozsahu 0..1, seznamy očistíme a deduplikujeme a pro mood/activity nastavíme bezpečné defaulty. Žánry poté projdou synonymní mapou a přísným fuzzy-matchingem na kurátorovaný katalog. Jen jednoznačně rozpoznané tagy zůstanou v seed_genres, ostatní ponecháme ve genres_freeform, aby se neztratila uživatelská nuance. Pokud model neposlal použitelné query_hints, doplníme je šetrným fallbackem z nálady, aktivity a prvních volných žánrů.

```python
def sanitize(intent: Dict[str, Any]) -> Dict[str, Any]:  1 usage
    # year_range
    yr = intent.get("year_range", None)
    if isinstance(yr, dict):
        f = yr.get("from")
        t = yr.get("to")
        if not (isinstance(f, int) and isinstance(t, int) and f > 0 and t > 0):
            intent["year_range"] = None
        else:
            if t < f:
                f, t = t, f
            intent["year_range"] = {"from": f, "to": t}
    else:
        intent["year_range"] = None

    # language
    langs = intent.get("language") or []
    norm_langs = []
    for x in langs:
        if isinstance(x, str):
            s = _normalize(x)
            if s == "any":
                norm_langs = ["any"]; break
            if _is_iso639_1(s):
                norm_langs.append(s)
    if not norm_langs:
        norm_langs = ["any"]
    if "any" in norm_langs and len(norm_langs) > 1:
        norm_langs = ["any"]
    intent["language"] = list(dict.fromkeys(norm_langs))
```

Obrázek 15: NLP service - sanitize function

Teprve takto „srovnaný“ objekt validujeme Pydanticem vůči kontraktu Intent. Neodpovídá-li struktura či typy, vracíme 422 s jasnou chybovou zprávou, jinak odesíláme striktní Intent JSON zpět backendu. Výsledkem je, že každý požadavek končí buď srozumitelným selháním v okamžiku, kdy ještě nic neohrozil, nebo naopak deterministickým, normalizovaným **výstupem**, který lze rovnou použít pro sestavení dotazů, filtry i reranking – bez dodatečných ad-hoc oprav.

8.2 Embeddingy /embed – účel a použití

Embed převádí texty na numerické reprezentace pro semantické skórování v jádru doporučování. Backend posílá dotaz (uživatelský prompt) a pasáže (stručné textové kontexty skladeb), my je zpracujeme modelem intfloat/multilingual-e5-small a vrátíme unit-norm vektory připravené pro kosinovou podobnost. Volba E5 je záměrná: model je vícejazyčný, optimalizovaný pro vyhledávání (instrukční ladění) a běží rozumně rychle i na CPU, což odpovídá našemu provoznímu profilu.

Backend volá POST /embed s polem texts. První prvek považujeme za dotaz (prefixujeme query:), všechny další za pasáže (passage:). Pořadí je zachováno a odpověď obsahuje stejný počet vektorů jako vstupů.

8.2.1 Limity a očekávání kvality

Embeddingy nejsou všemocné: velmi krátké texty („ok“, „fast“) mají nízkou informační hodnotu; špatně připravené pasáže (bez žánru/roku/BPM) hůře zachytí hudební kontext; jazykové nesoulady (dotaz v jednom jazyce, pasáže v jiném) mohou snižovat shodu. Model je trénovaný obecně – u vysoce specifických výrazů (subžánry, slang) může být shoda méně přesná. Proto embeddingy používáme hybridně: semantika je jen jednou složkou celkového skóre vedle dalších signálů (žánr, popularita, diverzita).

8.2.2 Zpracování, kontrola a návrat výsledku

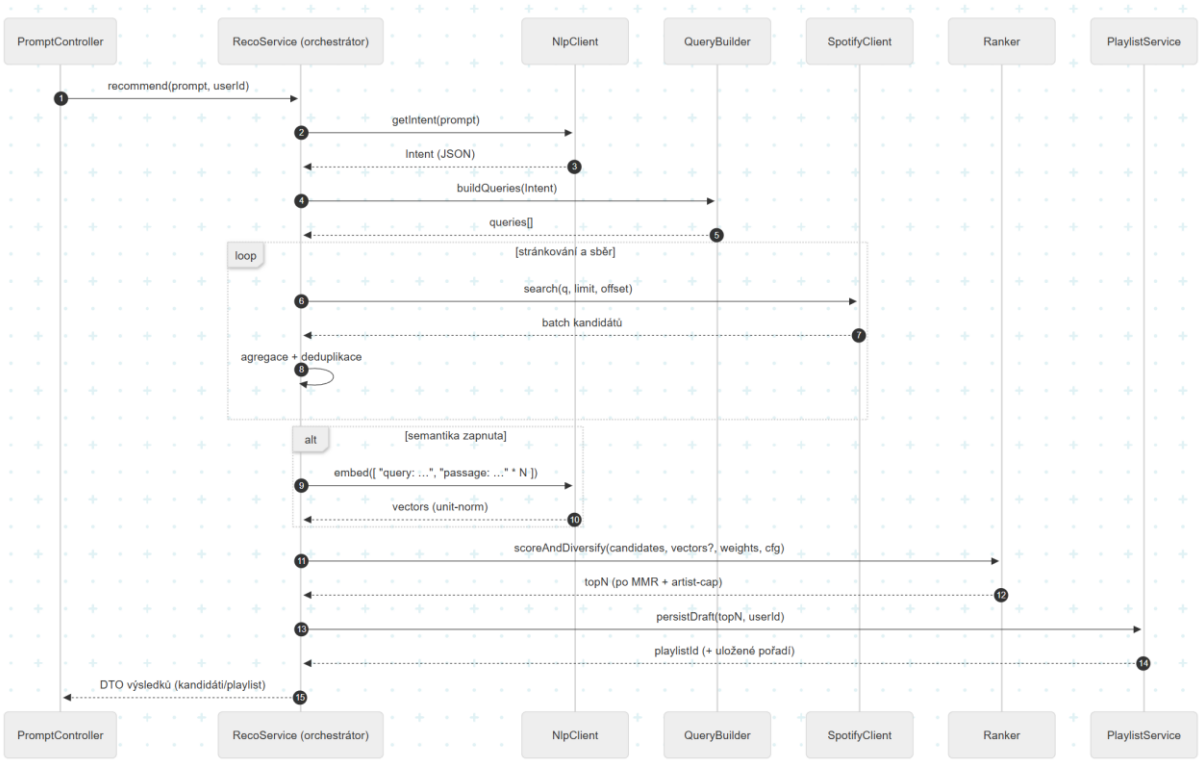
Po přijetí požadavku zkonstruujeme interní pole textů s prefixy a předáme je funkci pro dávkové zpracování. Vektory generujeme v dávkách (batch size 128) a necháváme je normalizovat už v modelu, poté převádíme na float32 a vracíme ve struktuře { "vectors": [...] }. Pokud pole texts dorazí prázdné, vracíme prázdný výsledek (deterministické chování bez chybových stavů). Tím garantujeme, že backend vždy obdrží pořadově zarovnané, jednotkově škálované vektory připravené ke srovnání.

```
198  EMB_MODEL = SentenceTransformer("intfloat/multilingual-e5-small")
199
200  def embed_texts(texts: List[str]) -> List[List[float]]:  1 usage
201      batch_size = 128
202      out = []
203      for i in range(0, len(texts), batch_size):
204          chunk = texts[i:i+batch_size]
205          vecs = EMB_MODEL.encode(chunk, normalize_embeddings=True)
206          out.extend(vecs)
207      return np.asarray(out, dtype=np.float32).tolist()
```

Obrázek 16: NLP service - dávkové zpracování vstupních vět

9. Recommendation systém

Doporučovací modul převádí volné uživatelské zadání a katalog Spotify do uskutečnitelného výběru skladeb: z textu vznikne Intent (NLP), z něj se sestaví dotazy do vyhledávání, nasbíraní kandidáti se pročistí, ohodnotí a rozumně rozprostřou, až vznikne finální pořadí pro playlist. Cílem je determinismus a čitelnost – každý krok má jasný vstup a výstup, dá se ladit přes konfigurační váhy, a při chybě se umíme degradovat (bez semantiky, s menší sadou dotazů) namísto pádu. Jádrem je orchestrace v RecoService, která propojuje API vrstvu s NLP, QueryBuilderem, klientem Spotify a Rankerem; výsledky se následně uloží (draft) a volitelně promítnou do uživatelova účtu na Spotify. Skórování je hybridní – kombinuje sémantiku (je-li zapnuta), žánrové stopy, popularitu a měkké limity (jazyk, roky) – a doplňuje je diverzifikace (MMR, limit na interpreta), aby výběr nebyl jednostranný. Celá konstrukce je transparentní a reprodukovatelná: od prvního HTTP požadavku až po uložený playlist je zřejmé, proč se skladba ocitla právě na daném místě, a kde případně upravit váhy či limity, aby systém odpovídal zamýšlenému uživatelskému zážitku.



Obrázek 17: Architektura doporučovacího systému

9.1 API vrstva: PromptController

```
26   @RestController  new *
27   @RequestMapping("/api/prompt")
28   @RequiredArgsConstructor
29   @Slf4j
30   public class PromptController {
31
32       private final RecoService recoService;
33       private final PlaylistService playlistService;
34       private final SpotifyTokenService tokenService;
35
36       @PostMapping  new *
37       public Mono<ResponseEntity<PromptResponse>> generate(@RequestBody PromptRequest promptRequest,
38                                                            Authentication authentication) {...}
107
108      @PostMapping("/confirm")  new *
109      public Mono<ResponseEntity<ConfirmResponse>> confirm(@RequestBody ConfirmRequest req,
110                                                           Authentication authentication) {...}
136
137      /unchecked/  1 usage  new *
138      private static String pickAlbumImage(Map<String, Object> album) {...}
161  }
```

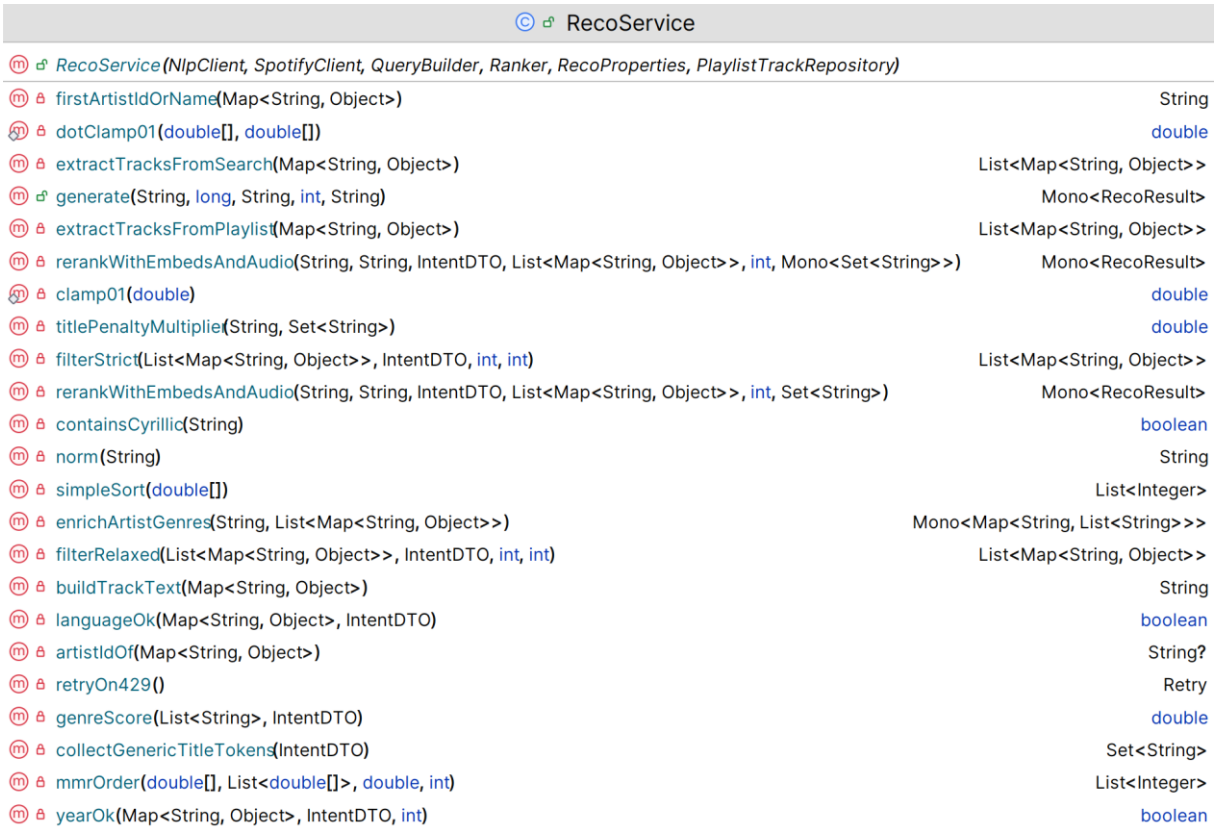
Obrázek 18: PromptController.java

PromptController je štíhlá vstupní brána do doporučování: převezme uživatelský prompt, ověří kontext volajícího (JWT) a provede základní validaci vstupu, ale veškerou doménovou práci okamžitě deleguje do RecoService. Neřeší skládání dotazů, skórování ani diverzifikaci, stará se o stabilní API kontrakt, serializaci/deserializeci a srozumitelné mapování chyb na HTTP kódy (400 pro nevalidní vstup, 401/403 pro problém s autentizací, 5xx pro selhání downstream služeb). Náhledové volání je bezpečně opakovatelné (bez trvalého zápisu), finální volání vrací identifikátory uloženého výsledku. V případě degradace (např. nedostupnost embeddingů) kontroler nic nepřerušuje – orchestrace pod ním automaticky přepne na pravidlové skórování a vrátí korektní, byť méně „chytrý“ výsledek. Díky této separaci zůstává API předvídatelné a udržitelné: kontroler zajišťuje čistý vstup i výstup, zatímco „co a proč doporučit“ se odehrává v RecoService a navazujících třídách.

Veřejné rozhraní tvoří tři metody se srozumitelně rozdělenými rolemi. generate(...) spustí doporučování nad dodaným textem a vrátí seřazené kandidáty jako náhled, aniž by cokoli měnilo stav v DB či na Spotify. confirm(...) převede vybraný seznam do našeho „draft“ playlistu a je-li to požadováno, tak rovnou vytvoří playlist v uživatelově účtu na Spotify včetně pořadí skladeb. draft(...) poskytne aktuální rozpracovaný návrh pro opětovné zobrazení nebo následné potvrzení. Důležitá provozní jistota: uvnitř generate i confirm před startem byznys logiky vždy ověřujeme platnost Spotify access-tokenu a v případě expirace jej automaticky obnovujeme

pomocí uloženého refresh tokenu; pro klienta je tím celé volání transparentní a odolné vůči časovým limitům a chybám „401“ ze strany Spotify.

9.2 Orchestrace: RecoService (hlavní tok)



Obrázek 19: RecoService.java

RecoService je centrální „*řídicí pult*“ doporučování: v jedné třídě spojuje práci všech pomocných komponent – klienta NLP, stavitele dotazů na Spotify, sběračů kandidátů, semantiky a skórovací/diverzifikační logiky – a převádí volný text na seřazený, odůvodnitelný výběr skladeb. Je to čistá service layer bez znalosti HTTP nebo UI. Dovnitř přichází jen vstupy potřebné pro doporučení (text, uživatel, limity) a ven odchází doménový výsledek, který kontroler pouze serializuje. Závislosti jsou injektované (*NlpClient, QueryBuilder, SpotifyClient, Ranker, PlaylistService*) a většina chování je řízena konfigurací *app.reco.**, aby bylo možné systém ladit bez zásahů do kódu.

Metoda *generate(...)* je vstupní bod celé orchestrace: v reaktivním toku (Reactor Mono) nejprve načte „seen“ tracky uživatele, vyžádá si z NLP Intent, z něj přes *QueryBuilder* sestaví dotazy na skladby i playlisty a paralelně je provede vůči Spotify se stránkováním (konfigurované perQuery a offsets), pacingem a retry. Výsledky se sjednotí do jednoho poolu, provede se merge + deduplikace podle trackId s pevnými capy (maxCandidates), poté následuje filtrace (nejdřív

strict, při malém poolu automaticky relax s vyšším yearSlack). Finální sada kandidátů míří do rerankingu (volitelně s /embed), kde se spočítá hybridní skóre a aplikují pravidla diverzifikace; metoda vrací *Mono<RecoResult>* s top-N výběrem připraveným pro náhled či uložení.

9.2.1 QueryBuilder: z Intentu na dotazy

Než začneme dotazy skládat, máme v ruce Intent z NLP: kanonické žánry v *seed_genres*, volnou slovní zásobu a nápovědy v *genres_freeform* a *query_hints*, případně *artist_hints* s explicitně zmíněnými interprety, dále language s přepínačem *languages_strict* a volitelný *year_range*. Tento objekt je už sanitizovaný a validovaný, takže QueryBuilder řeší čistě překlad významu do podoby vyhledávacích termů pro Spotify Search, nikoli dohady o formátu.

Cíl QueryBuilderu je dvojí: postavit malou, ale pokrývající sadu dotazů na skladby a totéž pro playlisty, aby se katalog „otevřel“ a přes komunitní obsah. V obou větvích platí stejná disciplína: dotazy musí být krátké, sémanticky nosné a bez šumu. Záměrně neprosazujeme jazyk ani rok: *languages_strict* a *year_range* uplatníme až ve filtru a skórování, v dotazech by to často znamenalo zbytečné zúžení zásahu.

Track queries (fromIntentForTracks)

Z *seed_genres* bereme kanonické, strojově kontrolovatelné štítky („indie“, „indie-pop“), které dávají dotazům jasný směr. K nim přimícháváme krátké, vyhledávání-užitečné fráze z *query_hints/genres_freeform* („melancholic indie“, „deep focus“, „post-punk / post punk“). Pokud uživatel zmínil interprety, *artist_hints* vytváří cílené dotazy na daná jména. Víceslovné termíny uvozujeme, aby se držely pohromadě („"deep focus"“), a generujeme rozumné varianty (spojovník vs. mezera). Tím vznikne 5–15 kompaktních dotazů, které se navzájem doplňují (žánrový pohled, náladový pohled, interpretační pohled) a minimalizují redundanci.

Playlist queries (fromIntentForPlaylists)

Stejnou logiku aplikujeme pro typ playlist: z *seed_genres* a *query_hints* skládáme dotazy, které míří na komunitní seznamy („melancholic indie“, „study ambient“, „post-punk classics“). Smyslem je expanze kandidátů o tracky, které přímé vyhledávání skladeb nevrátí (např. méně populární, ale žánrově přesné skladby). Tuto větev držíme úmyslně krátkou (např. top 10 playlistů na dotaz), protože na ní navazuje rozbalení playlistů do tracků.

Aby dotazy neprodukovaly triviální shody, uplatňujeme dvě jednoduché, ale účinné zásady z konfigurace:

- **name-bias.stopwords:** z dotazů vyhazujeme banální termíny („focus“, „study“, „sleep“, …), které často jen kopírují slova z promptu a v názvech skladeb dělají šum.
- **ban-title:** do dotazu přidáme negativní klauzuli typu NOT track:"focus", pokud je „focus“ krátký motiv z promptu. Tím snížíme návrat skladeb, které se sice „textově trefí“, ale obsahově nesplňují záměr.

Výsledkem jsou dva seznamy: trackQueries a playlistQueries. Dotazy jsou deduplikované, krátké, konzistentně uvozené a seřazené tak, aby se nejdřív vyčerpaly žánrové/náladové pohledy a teprve poté interpretační.

9.2.2 Sběr kandidátů: Search Tracks a Search Playlists

Jakmile máme z QueryBuilderu dva seznamy dotazů: jeden cílený na skladby a druhý na playlisty, spouštíme sběr kandidátů ve dvou paralelních větvích, ale s totožnými zásadami: krátké dotazy, pevné stránkování, řízené tempo volání a průběžná normalizace dat do jednotného tvaru.

Vyhledání skladeb (Search Tracks). Pro každý „track“ dotaz voláme Spotify Search opakovaně s konfigurovanými offsety (např. 0/50/100) a limitem perQuery. Volání běží souběžně (reaktivní Mono/Flux), aby latence zůstala v mezích. Mezi dávkami držíme rozumné rozestupy dle requestSpacingMs, čímž snižujeme riziko rate-limitu. Každou vrácenou dávku ihned převádíme pomocnou funkcí (v kódu *extractTracksFromSearch*) na minimální jednotnou reprezentaci: trackId, název, umělci, rok/albové info, dostupné žánry, popularita, případně BPM/preview. Smyslem je nevozít v paměti zbytečná pole a připravit data pro následné sloučení.

Vyhledání playlistů a jejich rozbalení (Search Playlists → Tracks). Druhá větev začíná analogicky, nad „playlist“ dotazy získáme několik nejrelevantnějších playlistů (typicky top 10). Z jejich ID pak postupně stahujeme položky (tracks). Právě tady přichází ke slovu provozní robustnost: mezi dotazy vkládáme časové rozestupy (delayElements), na odpověď s 429(rate limit) aplikujeme retry s horním stropem a v případě přetrvávajícího limitu nebo jiné chyby playlist tiše vynecháme (v kódu onErrorResume → prázdný seznam). Jednotlivé položky playlistů převádíme funkcí *extractTracksFromPlaylist* do stejného tvaru jako u přímého hledání skladeb. Tato větev tak přináší kurátorovanou diverzitu i méně populární, ale žánrově přesné kusy, které by čistě textové vyhledávání skladeb minulo.

Obě větve používají stejné zásady řízení provozu: respekt k Retry-After, konzervativní requestSpacingMs, souběžnost jen tam, kde dává smysl, a „thin payload“ (vytažení jen

potřebných polí). Z hlediska relevance je důležitá i země (country) v dotazu, aby Spotify vracelo regionálně odpovídající výsledky. Výstupem tohoto kroku nejsou hotové doporučení, ale surové dávky kandidátů, které jsou už teď ve sjednoceném tvaru a připravené k následujícímu sloučení. Díky tomu můžeme v dalším kroku provést merge + deduplikaci napříč oběma zdroji deterministicky (zachovat první výskyt, zbytek zahodit) a zároveň držet systém pod kontrolou přes capy, jakmile velikost poolu dosáhne maxCandidates, dál nesbíráme. Tato kombinace přímého „Search Tracks“ a expanze přes „Search Playlists“ je v praxi velmi účinná: první přináší ostré zásahy podle dotazu, druhá rozšiřuje kontext o hudbu vybranou lidmi/kurátory. Výsledek je bohatší a méně náchylný k „popularitnímu tunelu“, přičemž zůstává provozně odolný: selhání jednotlivého playlistu nebo dočasný rate-limit nezastaví celý sběr.

9.2.3 Sloučení a deduplikace (Merge + De-dup)

Obě větve (tracks i playlists) spojíme dohromady a z výsledných seznamů vytvoříme pořadovou mapu podle trackId (typicky LinkedHashMap). První výskyt každého tracku zachováváme, duplicity zahazujeme – tím stabilizujeme pořadí a současně eliminujeme opakování napříč dotazy i stránkami. Sloučení také respektuje kapacitu poolu (maxCandidates): jakmile máme „dost“, do další stránky už nechodíme. Tento krok je klíčový pro latenci i paměť – pool roste řízeně a deterministicky.

9.2.4 Filtrace

Než kandidáty pošleme dál, srovnám surový pool do smysluplného výběru. Filtrace je dvoufázová a stabilní (nezmění pořadí, jen odebírá položky), aby se chovala předvídatelně a nezkreslovala pozdější skórování.

Strict fáze – pokud Intent obsahuje *year_range*, uplatníme tvrdé časové síto s tolerancí yearSlack: povolíme jen skladby, jejichž rok vydání leží v intervalu [*from - yearSlack, to + yearSlack*]. Tím držíme zadání „v období“, ale necháváme prostor pro okrajové roky (reedice, přelomy dekád). Jazyk zpracujeme tvrdě pouze tehdy, když je v Intentu explicitně vyžádán a máme spolehlivý signál (např. jednoznačná shoda podle metadat/zdrojů dostupných v našem toku). Ve všech ostatních případech jazyk nevyřazujeme už ve filtru, aby nevznikaly falešné negativy; jeho vliv uplatní později skórování jako měkkou složku. Po aplikaci filtrů ještě respektujeme kapacitu, dál zpracováváme maximálně *maxCandidates* položek a zachováme původní pořadí z mergované mapy.

Relax fáze – pokud je po striktní fázi kandidátů málo (heuristika: méně než polovina cílové velikosti, alespoň pod hranici 10 kusů), přepneme do „*záchranného*“ módu: rozšíříme časové

okno (zvýšíme *yearSlack*, min. na 3 roky) a zmírníme podmínky tak, aby vznikl rozumný základ pro další kroky. Cílem není změnit povahu výběru, ale zabránit prázdnému výsledku u úzkých zadání (např. velmi specifická nálada + dekáda). Po relaxu opět dodržíme kapacitu a stabilitu pořadí.

Tvrdé síto chrání relevanci (neutečeme z dekády jen proto, že ji katalog reprezentuje nerovnoměrně), relax zajišťuje robustnost (i složitý prompt vrátí něco použitelného). Filtr vědomě neřeší duplicity interpretů ani „seen“ kusy – to patří do dalších kroků: diverzita a penalizace opaků jsou rozhodnutí skórování, ne binární vyřazení. Výsledkem této fáze je deterministický, očesaný pool kandidátů s garantovanou velikostí a zachovaným pořadím, připravený na další zpracování.

9.2.5 Reranking a diverzifikace (Ranker)

Účelem rerankingu je převést „očištěný“ pool kandidátů na jednoznačné pořadí. Každému tracku vypočítáme hybridní skóre jako váženou kombinaci několika normalizovaných složek a teprve poté výběr rozprostřeme pomocí diverzifikace (MMR) a omezíme přelití jednoho interpreta. Výsledek je stabilní, reprodukovatelný a dobře laditelný přes konfiguraci.

```
reco:
  desired-cap: 40
  per-query: 50
  offsets: [ 0, 50, 100 ]
  max-candidates: 1200
  artist-cap: 2
  year-slack: 1
  ban-title-search-exclude: true
  ban-title-exact-penalty: 0.55
  ban-title-partial-penalty: 0.85
  ban-title-partial-title-max-words: 2
  weights:
    sem: 0.65
    genre: 0.20
    year: 0.10
    pop: 0.05
    namePenalty: 0.20
  name-bias:
    stopwords: [ "focus","study","melancholic","melancholy","relax","chill","sleep","workout","training","sad","happy","calm" ]
  spotify:
    playlistFetchConcurrency: 2
    requestSpacingMs: 150
    retryDefaultSeconds: 2
    retryMaxSeconds: 8
  mmr:
    enabled: true
    lambda: 0.75
    pool: 200
  personalization:
    seenPenalty: 0.85
```

Obrázek 20: Recommendation properties

Celý proces rerankingu:

1. **Semantická složka.** Pokud je zapnuto /*embed*, využijeme kosinovou podobnost mezi vektorem dotazu a vektorem pasáže každého kandidáta (vektory jsou unit-norm, takže jde o skalární součin v rozsahu 0–1). Tato hodnota vystihuje „jak moc významově sedí“; když je embed dočasně nedostupný, semantiku prostě vynecháme a zbytek skóre pracuje dál.
2. **Žánrová shoda.** Zjišťujeme, nakolik kandidát odpovídá kanonickým seed-žánrům a volným tagům z *genres_freeform/query_hints*. Prakticky jde o krátkou míru překryvu/ shody (pouze z dostupných metadat) převedenou na 0–1. Smysl je prostý: neztratit to, co už Intent přesně pojmenoval.
3. **Časová blízkost.** Pokud v Intentu *year_range*, hodnotíme vzdálenost roku skladby od požadovaného intervalu s tolerancí year-slack. Čím blíž pásmu, tím vyšší příspěvek (0–1), za hranou interval plynule klesá, nikoli „tvrdě padá“ na nulu. Tím se vyhneme křehkosti prahů a zbytečným false negatives.
4. **Popularita a penalizace názvu.** Spotify „popularity“ normalizujeme na 0–1 a přidáme jako slabý signál pro robustnější pořadí. Naopak u krátkých, generických termínů z promptu (např. „focus“, „study“) penalizujeme kandidáty, jejichž název se s termínem **přesně** kryje nebo jej **částečně** obsahuje. Míra je řízená volbami *ban-title-exact-penalty*, *ban-title-partial-penalty* a *ban-title-partial-title-max-words*. Účelem je bránit triviálním „textovým shodám“, které nemusejí zachycovat intenci. Stejná idea stojí i za *ban-title-search-exclude* v QueryBuilderu (NOT track:"…") – tam bráníme šumu už ve vyhledávání, tady ho tlumíme při finálním řazení.
5. **Personalizace („seen“).** Aby výsledek nepůsobil jako „déjà vu“, snižujeme skóre uživatelem nedávno viděných skladeb koeficientem *personalization.seenPenalty*. Je to měkký zásah: oblíbený track může i tak zůstat vysoko, jen dostane menší preferenci.
6. **Celkové skóre.** Všechny složky sčítáme do vážené sumy podle *app.reco.weights*. počítáme něco ve tvaru:

$$score = w\_sem \cdot SEM + w\_genre \cdot GEN + w\_year \cdot YEAR + w\_pop \cdot POP - w\_namePenalty \cdot PENALTY$$

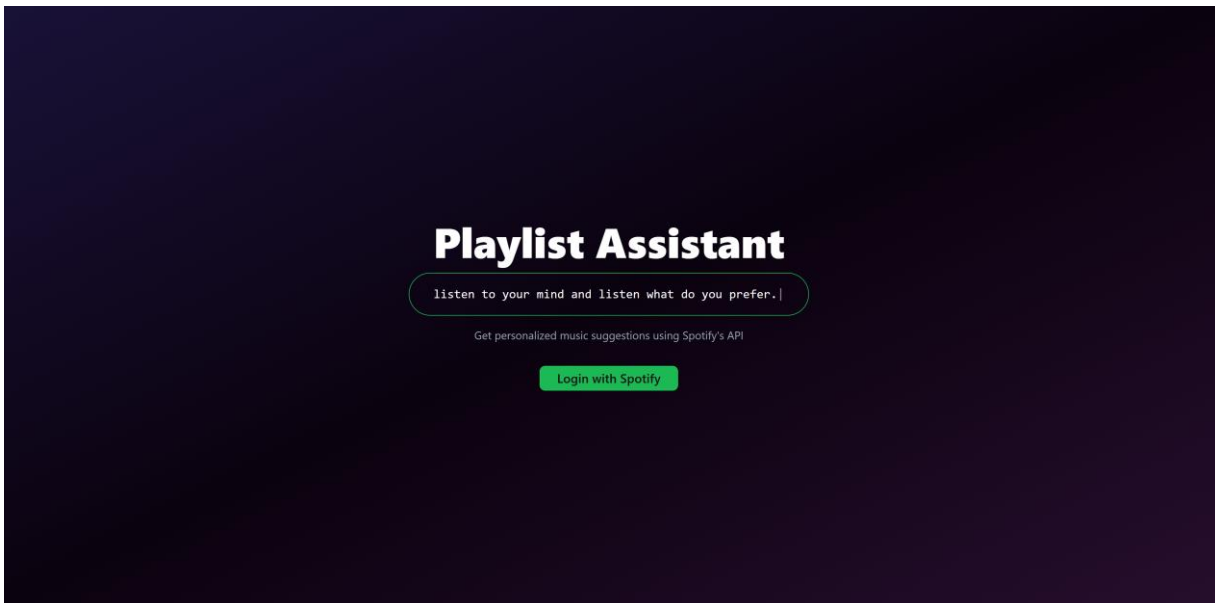
A poté aplikujeme multiplikativní srážku *seenPenalty*. Vše je škálováno na [0,1] a pořadí je deterministické, při stejných vstupech a konfiguraci dostaneme stejné výsledky. (Pozn.: intervalové „targets“ typu valence/energy jsou v kontraktu připravené, ale aktuálně je nepoužíváme kvůli limitům Spotify API. Ponecháváme je pro budoucí rozšíření bez zásahu do Rankeru.)

7. **Diverzifikace (MMR) a artist-cap.** Po výpočtu relevančního skóre aplikujeme ***Maximal Marginal Relevance***: vybíráme postupně položky, které maximalizují kombinaci *relevance* a *odlišnosti* od již vybraných. Poměr řídí mmr.lambda a velikost kandidátního zásobníku mmr.pool. Díky MMR se seznam „nerozpadne“ do monotónní řady podobných skladeb. Bezprostředně poté vynutíme artist-cap (např. 2 kusy na interpreta), aby jeden autor neovládl celý výsledek. Na závěr se výběr ořízne na desired-cap a vrátí se stabilní top-N.

Výsledkem rerankingu je transparentně odůvodnitelné pořadí, které spojuje významovou shodu s explicitní intencí, kurátorskou rozmanitostí a lehkou personalizací, přesně to, co od moderního doporučovače očekáváme.

10. Frontend

Frontend slouží jako tenký klient doporučovacího systému. Jeho úkolem je zprostředkovat uživatelský vstup (textový dotaz), přehledně zobrazit náhled navržených skladeb a umožnit vytvoření playlistu. Veškerá doménová logika, práce s API Spotify, NLP a reranking probíhají na serverové straně. Frontend je proto koncipován minimalisticky: důraz je kladen na srozumitelný UI tok, konzistentní vizuální prezentaci a rychlou odezvu.



Obrázek 21: Titulní stránka

10.1 Použité technologie a struktura

Aplikace je implementována v *React* + *TypeScript* s bundlerem *Vite* a stylováním pomocí *Tailwind CSS*. Zvolený stack umožňuje rychlý vývoj, typovou kontrolu rozhraní vůči backendu a jednoduché nasazení jako statickou SPA. Zásady návrhu: funkční komponenty a *React Hooks*, konzervativní práce se stavem (globálně pouze autentizace a stav náhledu), žádné přímé manipulace s DOM a důsledná práce s chybovými stavy sítě.

Zobrazená struktura reflektuje „tenký“ charakter klienta: adresář components sdružuje UI prvky podle úrovně znovupoužitelnosti, hooks obsahují stavové/efektové abstrakce a kořenové soubory (*App.tsx, main.tsx*) řeší bootstrap a kompozici aplikace. *Assets* slouží pro statická data (ikony, obrázky), *index.css* pro základní globální styly a *vite-env.d.ts* doplňuje typové definice pro Vite/TypeScript. Adresář *lib* je vyhrazen pro sdílené pomocné funkce a drobné utilitky.

V rámci *components* je vyčleněna složka *HomePage/* pro vstupní obrazovku aplikace (kompozitní kontejner). Podadresář *components/pages/* obsahuje hlavní obrazovky a „větší“ části UI toku: *LandingPage* (úvod a přihlášení), *AnimatedPage* (přechody/animace), *PromptPanel* (zadání dotazu), *TrackPreviewList* (náhled kandidátů), *HistoryModal* (historie generací), *HeaderUser* (uživatelský stav) a *Notifications* (globální oznámení). Adresář *components/ui/* shromažďuje atomické, čistě prezentační prvky (*button*, *dialog*, *input*), které se znovu používají napříč obrazovkami. V *hooks/useAuth.ts* je centralizována práce s autentizačním stavem (přihlášení/odhlášení, expirace), aby zůstala doménová logika mimo komponenty. Tato organizace minimalizuje vazby mezi částmi UI, usnadňuje orientaci v kódu a podporuje rychlé iterace bez zásahů do serverové logiky.

10.2 Autentizace a bezpečnost

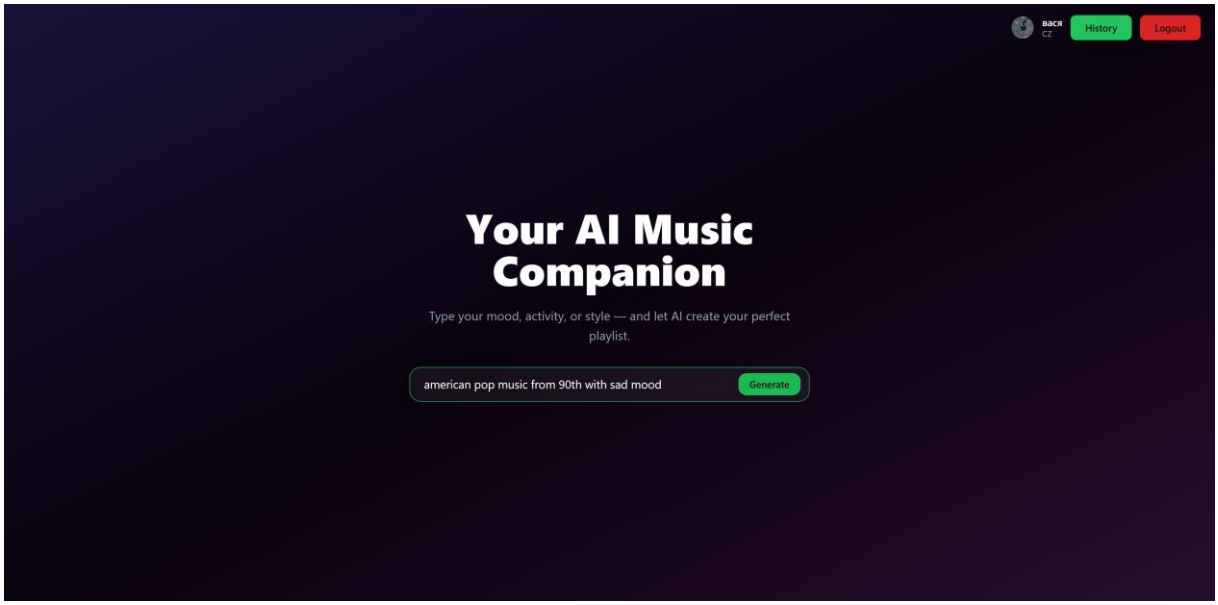
Přihlášení probíhá přes backend (*OAuth2 PKCE* vůči Spotify), aby se žádné Spotify tokeny neobjevily v prohlížeči. Frontend komunikuje výhradně s vlastním backendem přes *HTTPS* a používá pouze krátkodobý aplikační JWT pro autorizaci volání. Jsou ošetřeny běžné stavy: vypršení JWT (automatické odhlášení s oznámením), 401/403 (zobrazení výzvy k opětovnému

přihlášení), 429 (rate limit – doporučení opakovat akci později). CORS je nastaven restriktivně, tajné klíče se v klientovi nenacházejí.

10.3 Uživatelské toky

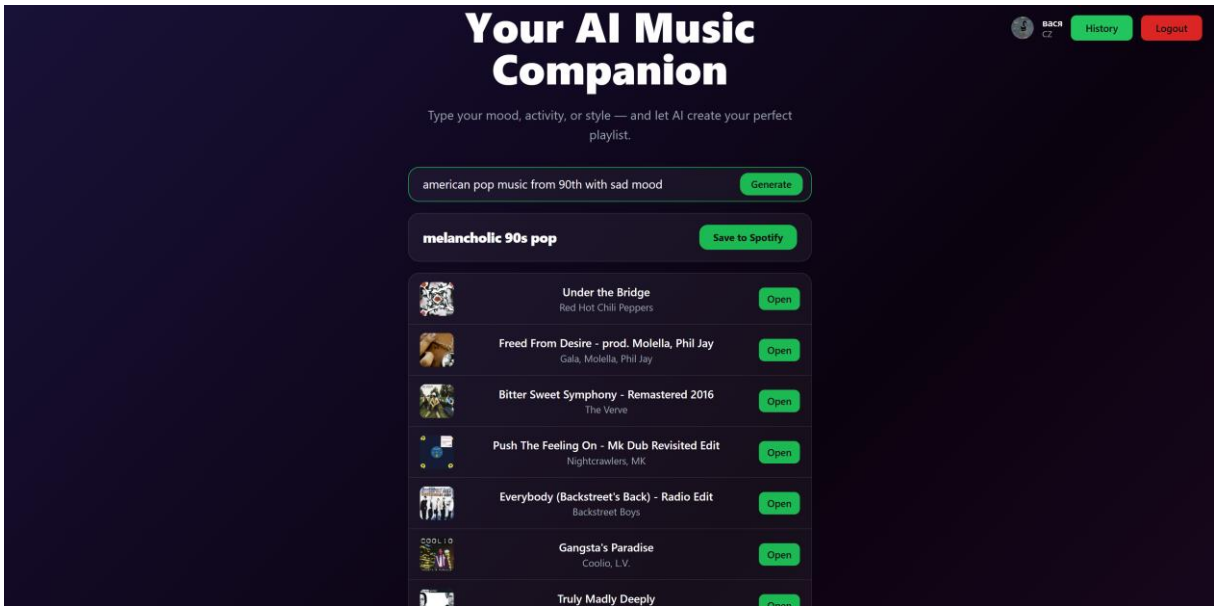
Přihlášení: Uživatel stiskne „Continue with Spotify“, frontend vyvolá */api/auth/login* (redirect na poskytovatele, PKCE). Po úspěšné autorizaci je uživatel vrácen na *callback* obrazovku, kde backend směrem k frontendu doručí krátkodobý aplikační JWT. Frontend stabilizuje stav (zobrazí jméno/avatara, zpřístupní chráněné akce) a ošetří hraniční situace: zamítnutí souhlasu, vypršení relace, 401/403, nabídka opětovného přihlášení.

Zadání dotazu (Prompt): Na vstupní stránce je textové pole s lehkou validací (min. délka, zákaz prázdného vstupu, indikace chyb). Tlačítko „Generate“ je aktivní jen pro validní vstup; po odeslání požadavku na backend se zobrazí loader/skeleton. Chyby sítě či validace se hlásí nenásilně (notifikace nad formulářem), uživateli zůstává původní vstup pro rychlou úpravu. V případě opakovaného odeslání se používá jednoduchý „debounce“ a blokace dvojkliku.



Obrázek 22: Hlavní stránka

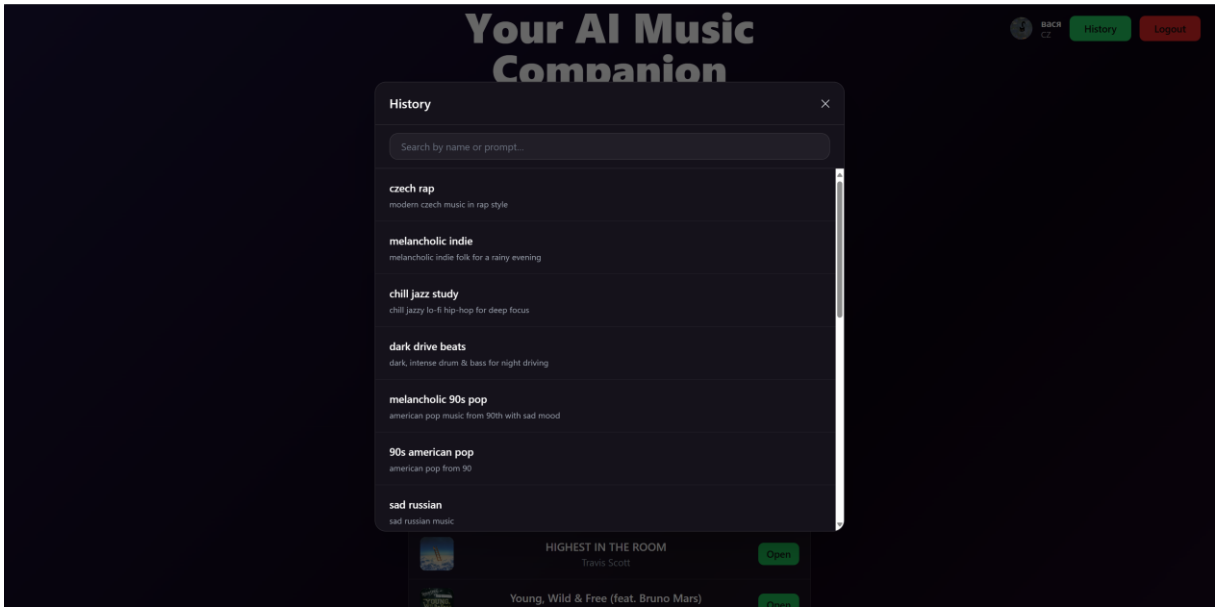
Náhled kandidátů (Preview): Po zpracování dotazu frontend zobrazí přehled doporučených skladeb. U každé položky je vidět obálka alba, název a interpret. Součástí je tlačítko „Open“, kterým lze skladbu otevřít ve Spotify a poslechnout si ji samostatně (otevření v spotify). Pokud uživateli aktuální návrh nevyhovuje, může zopakovat generování), nebo upravit/doplnit prompt a spustit generování znovu pro získání odlišného výsledku.



Obrázek 23: Seznam kandidátů

Vytvoření playlistu: Po potvrzení výběru dojde k volání endpointu pro vytvoření playlistu. Po úspěchu UI ukáže potvrzení jako zelená pop-up zprava.

Historie: Přehled nedávných dotazů/generací. Možnost návratu k předchozímu náhledu. Uživatel se může vrátit k předchozímu náhledu, podívat se na písnický a změnit/doplnit prompt a vygenerovat to znovu.



Obrázek 24: Historie dotazu

Vizuální styl je inspirován moderním minimalismem a estetikou Spotify: srozumitelné kontrasty, čitelné písmo, střídmá barevnost a konzistentní stavy ovládacích prvků. UI je

responzivní (mobil/tablet/desktop), využívá skeletony/indikátory načítání a „empty states“ s návodem k další akci.

ZÁVĚR

Tato práce ukázala, že propojení NLP a hudebních služeb může být praktickým nástrojem pro tvorbu playlistů z volně formulovaného textu. Cílem bylo navrhnout a implementovat řešení, které rozumí uživatelským popisům, převede je do srozumitelných parametrů a z dostupného katalogu poskládá poslouchatelný a personalizovaný seznam skladeb. Vznikla modulární webová aplikace s jasně oddělenou vrstvou pro porozumění textu, pro vyhledání kandidátů a pro jejich řazení. Na uživatelské straně nabízí jednoduché rozhraní „napiš, co chceš – dostaneš hudbu“.

Hlavní přínosy lze shrnout následovně. Zaprvé, byla navržena architektura s čistým kontraktem mezi částmi systému: NLP vrstva vrací pevně definovaný JSON a větný embedding, což zjednodušuje validaci i budoucí rozšiřování. Zadruhé, byla sestavena retrieval-a-ranking pipeline, která kombinuje symbolické filtry (parametry, dostupnost, verze) se sémantickou podobností a následnou úpravou pořadí pro plynulost. Výsledek nepůsobí monotónně díky jemné diverzifikaci. Zatřetí, práce stanovila základ vyhodnocení – metriky relevance (nDCG@k), rozmanitosti (ILD/coverage) a jednoduchý ukazatel plynulosti (průměrný skok tempa/energie) – a z nich odvodila praktická doporučení pro ladění vah a pravidel. Současně byly nastíněny principy personalizace: práce s dostupnými signály (např. uložené skladby či interakce), jemné posuny vah v řazení a limity na opakování interpretů, aby se výsledek přiblížil vkusu konkrétního uživatele bez ztráty tématu. V neposlední řadě je výstup systémově „čistý“: důsledná normalizace a kontrola struktury udržují chování předvídatelné i u vícejazyčných či vágních popisů.

Během vývoje se ukázalo několik poučení. Schémem řízená extrakce je v praxi důležitější než volba konkrétního modelu – jasná struktura a validace omezují chyby a zrychlují integraci. Dále platí, že „dobře se poslouchá“ není tentýž cíl jako „je to nejbližší k dotazu“: k dosažení pocitově správného výsledku je třeba kombinovat několik pohledů (shoda se záměrem, plynulost, střídmost opakování) a vyhnout se přehnané diverzifikaci, která by rozbila téma. Personalizace přitom funguje jako jemné doladění nad touto kostrou – posouvá pořadí podle uživatelových preferencí, ale hlídá soudržnost a nepřetěžuje playlist jedním interpretem nebo jedním stylem.

Možné směry další práce jsou zřejmé. Za prvé, hlubší personalizace (implicitní preference, učení-to-rank z interakcí, párové preference uživatelů). Za druhé, doplnění audio-rysu (tempo/energie z audia, tonální klíč, segmentace), které mohou vylepšit plynulost a přechody. Za třetí, robustnější NLP (bohatší pokrytí vícejazyčnosti a kódových směsí, přísnější

gramaticky řízené dekódování). Za čtvrté, provozní optimalizace – kvantizace a distilace modelů, caching/batching a efektivnější práce s kandidáty. A konečně, systematičtější uživatelské testování (A/B) a sběr zpětné vazby s vysvětlitelností „proč tato skladba“ přímo v UI.

Z praktického hlediska práce naplnila vytyčený cíl: ukazuje, že z přirozeného uživatelského popisu lze spolehlivě vytvořit tematicky věrný, poslechově soudržný a osobním preferencím přizpůsobený playlist. Zároveň nabízí pevný základ pro další experimenty, jak v oblasti doporučovacích metod, tak v uživatelském zážitku. Modularita návrhu dává prostoru růstu jasná pravidla: vyměnit model, přidat signál nebo změnit kritéria hodnocení lze bez přepsání celého systému. V tomto smyslu je výsledná aplikace nejen demonstrací funkčního přístupu, ale i platformou pro pokračující výzkum a zlepšování.

POUŽITÁ LITERATURA

[1] CHOLLET, François. Deep learning with Python. Second edition. Shelter Island: Manning, [2021]. ISBN 978-1-61729-686-4.

[2] TUNSTALL, Lewis; WERRA, Leandro von a WOLF, Thomas. Natural language processing with transformers: building language applications with Hugging Face. Revised color edition. Sebastopol, CA: O'Reilly, 2022. ISBN 978-1-0981-3679-6.

[3] AGGARWAL, Charu C. Recommender systems: the textbook. Cham: Springer, [2016]. ISBN 978-3-319-29657-9.

[4] RICCI, Francesco; ROKACH, Lior a SHAPIRA, Bracha. Recommender Systems Handbook. Springer Nature, 2022. ISBN 9781071621974.

[5] CHRISTOPHER D. MANNING; RAGHAVAN, Prabhakar a SCHÜTZE, Hinrich. Introduction to Information Retrieval. Cambridge University Press, 2008. ISBN 9781139472104.

[6] CELMA, Òscar. Music recommendation and discovery. New York: Springer, 2010. ISBN 3642132863.

[7] MÜLLER, Meinard. Fundamentals of Music Processing: Audio, Analysis, Algorithms, Applications. Springer, 2015. ISBN 9783319219455.

[8] JANNACH, Dietmar; ZANKER, Markus; FELFERNIG, Alexander a FRIEDRICH, Gerhard.Recommender Systems: An Introduction. Cambridge University Press, 2010. ISBN 9781139492591.

[9] GOODFELLOW, Ian; BENGIO, Yoshua a COURVILLE, Aaron. *Deep Learning*. MIT Press, 2016. ISBN 9780262035613.

[10] FALK, Kim. *Practical Recommender Systems*. Simon and Schuster, 2019. ISBN 9781638353980.

[11] KNEES, Peter a SCHEDL, Markus. *Music Similarity and Retrieval: An Introduction to Audio- and Web-based Strategies*. Springer, 2016. ISBN 9783662497227.

[12] ALICJA A. WIECZORKOWSKA. *Special Issue on Music Information Retrieval*. 2003.

[13] JURAFSKY, Dan a JAMES H. MARTIN. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, 2009. ISBN 9780131873216.

[14] GOLDBERG, Yoav. *Neural Network Methods for Natural Language Processing*. Springer Nature, 2022. ISBN 9783031021657.

[15] BAEZA-YATES, Ricardo a RIBEIRO-NETO, Berthier. *Modern Information Retrieval*. 2004. ISBN 8129702746.

[16] LESKOVEC, Jurij; RAJARAMAN, Anand a ULLMAN, Jeffrey D. *Mining of massive datasets*. Third edition. Cambridge: Cambridge University Press, 2020. ISBN 978-1-108-47634-8.

[17] VAJJALA, Sowmya; MAJUMDER, Bodhisattwa; GUPTA, Anuj a SURANA, Harshit. *Practical Natural Language Processing: A Comprehensive Guide to Building Real-World NLP Systems*. "O'Reilly Media, Inc.", 2020. ISBN 9781492054009.

[18] HUYEN, Chip. *Designing machine learning systems: an iterative process for production-ready applications*. Beijing: O'Reilly, 2022. ISBN 978-1-098-10796-3.

SEZNAM PŘÍLOH

PŘÍLOHA A*:* Zdrojový kód aplikace

Archiv MusicRecommendationSystem.zip obsahuje zdrojový kód celé aplikace.