VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

WEBOVÁ APLIKACE PRO TRÉNOVÁNÍ A VALIDACI MODELŮ STROJOVÉHO UČENÍ

WEB APPLICATION FOR TRAINING AND VALIDATION OF MACHINE LEARNING MODELS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE

AUTHOR

Bc. Jan Svoboda

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. Zoltán Galáž, Ph.D.

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Jan Svoboda ***ID:*** 200745

Ročník: 2 ***Akademický rok:*** 2021/22

NÁZEV TÉMATU:

Webová aplikace pro trénování a validaci modelů strojového učení

POKYNY PRO VYPRACOVÁNÍ:

Cílem diplomové práce je návrh a implementace webové aplikace pro trénování a validaci modelů strojového učení. V diplomové práci bude navržena a naprogramována webová aplikace s plnou podporou autentizace a autorizace uživatelů. Tato aplikace bude poskytovat možnost nahrání dat a spuštění trénování uživatelem zvoleného modelu strojového učení (trénování bude konfigurovatelné). Trénování bude probíhat jako asynchronní úloha a aplikace bude uživatele informovat o jejím stavu. Po skončení trénování bude uživateli poskytnuta možnost validace natrénovaného modelu a výstupem budou klasické predikční metriky a vizualizace. Finální aplikace bude kontejnerizována a nasazena na testovacím serveru, kde bude na reálných datech zdravých jedinců a pacientů trpících Parkinsonovou nemocí, ověřena její funkčnost, rychlost, bezpečnost a stabilita.

DOPORUČENÁ LITERATURA:

[1] BISHOP, Christopher M. Pattern recognition and machine learning. New York: Springer, c2006. Information science and statistics. ISBN 0-387-31073-8.

[2] PECINOVSKÝ, Rudolf. Python: kompletní příručka jazyka pro verzi 3.10. Praha: Grada Publishing, 2021. Knihovna programátora (Grada). ISBN 978-80-271-3442-7.

Termín zadání: 7.2.2022 ***Termín odevzdání:*** 24.5.2022

Vedoucí práce: Ing. Zoltán Galáž, Ph.D.

prof. Ing. Jiří Mišurec, CSc.
předseda rady studijního programu

ABSTRAKT

Delegování výpočetně náročných operací na stranu výpočetních serverů je převládajícím trendem. Kontejnerizované aplikace pak poskytují abstrakci nad těmito servery a umožňují dosáhnutí požadované škálovatelnosti. Architektura implementované aplikace sloužící jako platforma pro asynchronní spuštění úlohy trénování a validace modelů strojového učení je kontejnerizací silně ovlivněna. Jednotlivým uživatelům webové aplikace je přidělen separátní kontejner s připraveným prostředím pro běh modelu se zadanými parametry. Průběh procesu učení je webovou aplikací monitorován. Po obdržení výsledků je provedena analýza a v uživatelském prostředí jsou vykresleny komponenty poskytující vizualizaci.

KLÍČOVÁ SLOVA

Strojové učení, webová aplikace, vizualizace, kontejnerizace, Java, Python, React, TypeScript, JavaScript, Docker

ABSTRACT

The delegation of resource-intensive operations to the server-side computers is an ongoing trend. Containerized applications serve as an abstraction over the servers and provide needed scalability. The architecture of the implemented application utilized as a platform for an asynchronous execution of training and validation of machine learning models is heavily influenced by the containerization. Individual users of a web application are given a separate container with prepared environment for running a model with selected parameters. Status of the training process is monitored by the application. After the result is obtained the analysis takes place and components providing visualization are rendered in the user interface.

KEYWORDS

Machine Learning, Web Application, Visualization, Containerization, Java, Python, React, TypeScript, JavaScript, Docker

SVOBODA, Jan. *Webová aplikace pro trénování a validaci modelů strojového učení*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2022, 87 s. Diplomová práce. Vedoucí práce: Ing. Zoltán Galáž, Ph.D.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Jan Svoboda
VUT ID autora:	200745
Typ práce:	Diplomová práce
Akademický rok:	2021/22
Téma závěrečné práce:	Webová aplikace pro trénování a validaci modelů strojového učení

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského 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, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Tímto bych rád poděkoval vedoucímu panu Ing. Zoltánu Galážovi, Ph.D. za odborné vedení práce a vždy konstruktivní připomínky.

Obsah

Seznam obrázků

Seznam výpisů

Úvod

Webové aplikace poskytující uživatelům potřebné informace za dobu své existence prošly řadou rapidních změn [1]. Šíře použitých technologií s měnícími se požadavky na kvalitu a funkcionalitu aplikací činí z webového vývoje důležitou oblast softwarového inženýrství. Řada aplikací nyní díky tzv. *cloudovým* řešením umožňuje veškeré náročné výpočty delegovat na stranu výpočetních serverů. Klientská část aplikace se tak stará jen o vykreslení uživatelského prostředí, což snižuje nároky na uživatelovo zařízení. Tato řešení také umožňují přesunutí perzistentní vrstvy na stranu poskytovatele aplikace a činí tak službu velmi dostupnou. Obecně je dosahováno škálovatelnosti těchto aplikací pomocí miniaturizace jednotlivých služeb aplikace (*microservices*) a jejich duplikací při zvýšeném provozu. Naprosto určující technologií této oblasti je kontejnerizace. Zmíněnými vlastnostmi je inspirována architektura aplikace implementovaná v diplomové práci.

Náplní textu je popis tvorby webové aplikace umožňující asynchronní spuštění trénování a validace modelů strojového učení. Celý proces učení, respektive jeho jednotlivé části, je možné v rámci uživatelského rozhraní monitorovat. Po získání výsledků z procesu validace je nad těmito výsledky provedena analýza. Nedílnou součástí analýzy je vizuální složka, proto aplikace nabízí možnost vykreslení grafů poskytující potřebnou vizualizaci k usnadnění interpretace výsledků. Aplikace je sestavena z několika dílčích částí, přičemž každý stavební blok je umístěn do separátního kontejneru. Aplikace je tak plně kontejnerizována. Webový klient poskytující uživatelské prostředí je realizován pomocí knihovny *React*, která vývojáři umožňuje definovat jednotlivé komponenty, které je poté možné kombinovat, a vytvářet tak složitější prvky uživatelského prostředí. Webový server zprostředkovávající komunikaci mezi klientem a jemu přiděleném kontejneru s běhovým prostředím *Python*, které je nakonfigurováno ke spuštění již zmíněného procesu učení, je implementován za pomoci aplikačního rámce *Spring*.

V navazujících kapitolách je popsán softwarový vývoj na obecné úrovni. V textu se také nachází i popis řady přístupů a doporučených návyků k psaní čitelného kódu. Kapitoly zabývající se praktickým použitím zmíněných principů jsou doplněny o výpisy kódu ze samotné implementace aplikace či z vhodných příkladů, které ilustrují danou problematiku. Příloha A obsahuje grafické ukázky uživatelského prostředí. Autor dále přikládá odkaz na úložiště[1], do kterého byly během řešení diplomové práce pravidelně nahrávány ukázky z jednotlivých iterací vývoje aplikace.

[1] `https://drive.google.com/drive/folders/17BWR5fbwnzHdIt4Ofe438MTBGxZYUjsl`

1 Teoretická východiska

V rámci teoretické části budou popsány základní koncepty softwarového vývoje, zejména budou diskutovány tzv. *best practices*, tedy jakási doporučení pro psaní takového kódu, který se pak stává čitelným a srozumitelným. Pro větší softwarové projekty, na kterých pracuje více vývojářů, je dodržování těchto technik silně doporučováno. Je důležité si uvědomit, že kód je čten mnohonásobně častěji, než je psán samotným vývojářem. Dodržování zmíněných návyků pak ostatním vývojářům (a ostatně také autorovi) umožní snadnější orientaci v implementovaném projektu, a tím je maximalizován čas pro samotné pochopení významu a funkce kódu. Respektování technik také vede k vhodné úrovni abstrakce a nezávislosti dílčích modulů (*decoupling*), díky kterému se stává kód více modulární a je snadněji rozšiřitelný.

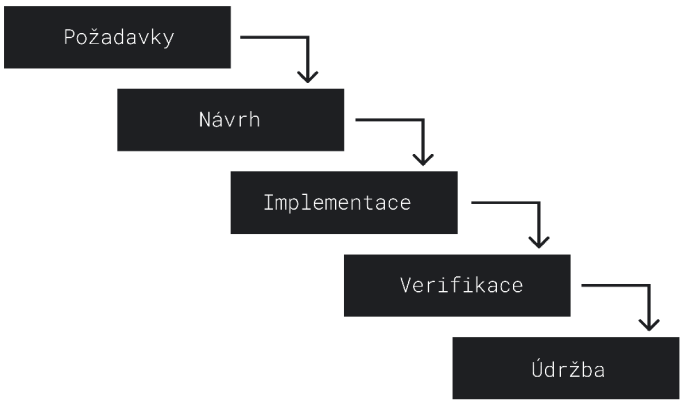
Dále bude pozornost věnována technologiím, které jsou nedílnou součástí webového vývoje. Jednou z nich je technologie *Docker*. Části aplikace je možno kontejnerizovat, tedy umístit do samostatného prostředí - kontejneru, a to právě díky technologii *Docker*. Kontejnerizaci je věnována značná část následujícího textu, jsou zde diskutovány výhody této technologie a popsány základní koncepty. Použití je ilustrováno na výpisech obsahující ukázky kódu.

V poslední kapitole teoretické části jsou rozebrány základní principy pojmu strojového učení. Autor se věnuje i popsání vybraných modelů strojového učení použitých v aplikaci. V kapitole je také uvedena ukázka kódu a grafické schéma, které popisuje vytváření příznakového vektoru (*feature vector*). Ten je předán jako parametr do metody poskytující spuštění trénování modelu, respektive predikční metody, která po exekuci volajícímu poskytne pomocí natrénovaného modelu patřičná predikovaná data.

1.1 Vývoj software

Vývoj software (*software development*) jako ostatní technologická odvětví prošel zajímavou historií, která byla nejen diktována neustálým zdokonalováním samotné technologie, ale také zdokonalováním procesů. Při vhodné kombinaci zmíněných pojmů je dosahováno maximální (a zejména udržitelné) efektivity. U veškerého vývoje je kladen velký důraz na časovou složku. Daný softwarový produkt je nutné dostat na trh v co možná nejkratším čase, a to nejlépe tak, aby konkurence nedokázala včasně zareagovat. Toto vytváří velký nátlak na samotné vývojáře, jelikož je požadována co možná nejrychlejší implementace projektu. Bohužel tento nátlak historicky vedl k poměrně neúspěšným softwarovým produktům, ať už z hlediska bezpečnosti, funkčnosti, optimalizace či následné údržby. Toto ostatně popisuje Robert C. Martin ve své knize *Clean Code* [2].

Pro vývoj projektů bylo typické oddělení jednotlivých fází procesu vývoje (vodopádový model [3]). Nejprve byly definovány požadavky, poté se provedla analýza, jež byla následována složitým návrhem. Teprve po těchto krocích se začalo se samotnou implementací. Realita vývoje však ne vždy odpovídala počátečnímu návrhu, protože například bylo zjištěno, že některé body funkcionality vyžadují jiný postup implementace než bylo prvotně určeno. V průběhu vývoje se navíc mohou požadavky měnit. Například konkurence uvede nový produkt či zákazník upřesní specifikaci, která již není v souladu s původním zadáním. Velkým problémem tohoto postupu je také fáze verifikace. Pokud k testování dochází až po celém procesu vývoje, chyby se odhalují až zpětně. Čím později se tyto chyby odhalí, tím náročnější a ekonomicky nákladnější je jejich oprava. Proto byla postupně vytvářena řada nových technik vývoje software, které se snaží na zmíněné výzvy reagovat a odstranit nedostatky předchozích postupů.



Obr. 1.1: Diagram vodopádového vývoje.

1.1.1 Agilní metody

Na agilní metody vývoje je možno nahlížet jako na snahu zavádět takové procesy, které upřednostňují malé časté změny a rychlou zpětnou vazbu. Je definována určitá kostra vývoje, nicméně je zde ponechán prostor pro změnu požadavků. Principem, který zastřešuje tyto metody, je co nejdříve přinášet hodnotu, to znamená, co nejdříve vytvořit implementaci projektu, která se blíží k hlavním požadavkům. Společným jmenovatelem je tedy častá iterace mezi jednotlivými fázemi vývoje, kdy se fáze vykonávají ihned za sebou, a to v krátkém časovém intervalu (typicky definovány termínem *sprint* trvajícím v řádu týdnů). Celý tým pak dostává jasnou a rychlou zpětnou vazbu, a to zejména ve fázi verifikační, kdy je produkční kód testován sadou testů. Dále vývojový tým také může sbírat zpětnou vazbu přímo od zákazníka, jelikož již má minimální funkční verzi softwarového produktu, kterou zákazníkovi lze představit. Tým vývojářů tak postupně zapracovává zákazníkovy připomínky.

Iterace tedy umožňují pružně reagovat na změnu požadavků, navíc je při tomto procesu vývojáři reálně vytvářen produkt [4]. Díky automatizovaným testům a refaktorizaci (přepsání kódu do formy, která splňuje *best practices*) je také možné ke stávajícímu kódu přidávat novou funkcionalitu, aniž by byla následná integrace zatížena chybami z minulé iterace. Postupem času se tedy směřuje k jakýmsi jednotkám práce či atomizaci procesů. Zmíněná rychlá zpětná vazba, které je vytvářená právě za pomoci automatizovaných testů (vývojář se téměř okamžitě dozví, že jeho implementace nesplňuje požadavky) přispívá ke stabilitě aplikace. Problematikou testování se podrobněji zaobírá kapitola 1.1.3.

V současné době je hojně využíváno nástrojů CI/CD (*continuous integration, continuous deployment* či *continuous delivery*), které umožňují vývojářskému týmu po zaintegrování změn automatické sestavení aplikace (*build*) a běh automatizovaných testů. Tento proces je schopný rychle odhalit chyby, navíc výstupem je sestavená aplikace, kterou je možné po přípravě nasadit na produkční servery. Jedním z hojně používaných CI/CD nástrojů je aplikace *TeamCity*[1] od společnosti *JetBrains*.

Pro praktický vývoj aplikace a spouštění automatických testů autor práce použil řešení poskytováno platformou *GitHub*[2]. Tato platforma nenabízí jen vytváření repozitářu pro verzování kódu, ale také definování tzv. *GitHub Actions*. Jednotlivé

[1] `https://www.jetbrains.com/teamcity`
[2] `https://github.com`

akce je možné konfigurovat pomocí skriptů. Je zde určeno, kdy bude akce spuštěna (například po nahrání nové revize kódu do repozitáře) a jaké jednotlivé kroky budou provedeny. *GitHub* pak nabízí historii běhů nadefinovaných akcí a vývojář je tak schopný zjistit, zda akce proběhla v pořádku (například zda se projekt úspěšně zkompiloval či zda testy proběhly úspěšně). Ukázka akce, které slouží právě k sestavení *front-end* části aplikace a spuštění sady testů, se nachází v poslední části kapitoly 3.1 zabývající se testováním implementované aplikace.

1.1.2 Čistý kód

Čistý kód neboli tzv. *clean code* vychází z premisi, že kód je mnohem častěji čten než-li psán samotným vývojářem. Proto je závaděno několik doporučených technik, které čtenářům (a zejména pak recenzentům) kódu umožní snadnější orientaci. Orientací je zde myšleno jednak zavádění jednotného formátování a stylizace kódu, které například definuje, jakým způsobem se pojmenovávají proměnné, metody v produkčním prostředí či v rámci testování. Dále pak jde o zavádění konceptů sémantických, významových, které autora kódu vedou k použití správné úrovní abstrakce. V následující části budou základní koncepty stručně popsány.

Principy SOLID

Tyto principy jsou jakýmsi základním nástrojem pro tvorbu čitelného a modulárního kódu [5]. Je ovšem nutné zmínit, že se jedná opravdu jen o nástroje, a velmi záleží na tom, jak s nimi vývojařský tým pracuje, a to podle typu projektu či fáze vývoje. Spojení SOLID referuje k počátečním písmenům jednotlivých principů:
***S**ingle-responsibility* princip,
***O**pen-closed* princip,
***L**iskov substitution* princip,
***I**nterface segregation* princip,
***D**ependency inversion* princip.

V následující kódové ukázce bude popsán význam alespoň části zmíněných principů.

Výpis 1.1: Ukázka principů SOLID.

```
1     @RequestMapping("/api/dataset")
2     public class DatasetController
3     {
4         private final FileService fileService;
5
6         public DatasetController(FileService fileService)
7         {
8             this.fileService = fileService;
```

```
9         }
10
11        @GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
12        public List<FileInformation> getAllFiles()
13        {
14            return fileService.getAllFiles();
15        }
16    }
17
18    public interface FileService
19    {
20        List<FileInformation> getAllFiles();
21    }
22
23    public class LocalFileService implements FileService
24    {
25        @Override
26        public List<FileInformation> getAllFiles()
27        {
28            //specifická implementace
29        }
30    }
31
32    public class ContainerFileService implements FileService
33    {
34        @Override
35        public List<FileInformation> getAllFiles()
36        {
37            //specifická implementace
38        }
39    }
```

Uvažujme tedy modelový příklad napsaný v jazyce *Java* za pomocí aplikačního rámce *Spring*, kdy vstupním bodem do serverové aplikace je metoda `getAllFiles()`, která je namapována na adresu `<doména>/api/dataset`. Volajícímu je tedy navrácena odpověď se seznamem jednotlivých souborů v datové množině (*dataset*). Můžeme si všimnout, že `DatasetController` vykonává opravdu jen jednu funkci (vystavění odpovědi v podobě JSON). Nenabízí volajícím funkce z jiné domény (například poskytnutí výsledků trénování modelu strojového učení na dané datové množině). Toto tedy znamená, že `DatasetContoller` splňuje *Single-responsibility* princip.

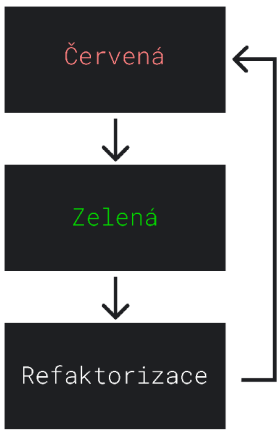
V konstruktoru se nachází parametr typu `FileService`. Tato třída je ve skutečnosti rozhraním (definováno na řádku 18). Díky tomuto rozhraní dochází k větší modularitě kódu, protože `DatasetController` není závislý na implementaci rozhraní `FileService`. Tedy při změně implementace `FileService` se nemusí třída `DatasetController` jakkoliv měnit, protože s rozhraním `FileService` má dohodnut jen kontrakt, a to takový, že metoda `getAllFiles()` bude vracet seznam souborů.

Výhodou modularity je snadná rozšířitelnost (zde se promítá *Open-closed* princip), použití tříd v jiném kontextu či aplikaci, a zároveň je ušetřen čas při rekompilaci, protože změna implementace rozhraní nespustí rekompilaci třídy, která toto rozhraní používá.

Rozhraní `FileService` je pak implementováno pomocí tříd `LocalFileService` a `ContainerFileService`. Prvotní implementace se používá pro lokální adresář, který se nachází na samotném serveru, druhá implementace je pak používána pro získání souborů na vzdáleném úložišti (například v *Docker* kontejneru, viz kapitola 1.2.3). Pokud by například bylo nutné soubory ukládat do databáze, tak je možné implementovat další třídu poskytující tuto funkcionalitu (opět *Open-closed* princip). Toto značí, že třída `DatasetController` není závislá na specifické implementaci, nýbrž na abstrakci, která je zprostředkována pomocí rozhraní `FileService` (princip *Dependency inversion*).

1.1.3 Vývoj řízený testy

Vývoj řízený testy (*test-driven development* či TDD) [6] je princip vývoje softwaru, kdy je v prvé řadě nadefinována sada testů, pomocí které se sestavují požadavky na produkční kód, jenž je touto sadou testů testován. Vývojář se při implementaci samotné aplikace řídí schématem 1.2.



Obr. 1.2: Schéma vývoje řízeného testy.

Vývojář si nejprve nadefinuje požadovaná rozhraní, která představují základní kostru aplikace. Než začne se samotnou implementací těchto rozhraní, nejprve implementuje sadu testů. Je důležité, aby před psaním produkčního kódu byla sada testů spuštěna. Tím je ověřeno, že testy opravdu verifikují budoucí implementaci produkčního kódu a jsou neúspěšné (červená fáze). Nemůže pak nastat taková situace,

kdy je některý z testů úspěšný při ještě neimplementovaném testovaném subjektu. Po tomto kroku vývojář přistupuje k samotnému psaní produkčního kódu. V této fázi se nutně nesnaží o psaní úhledného a čistého kódu, jelikož cílem je dosáhnutí stavu, kdy všechny testy úspěšně prochází (zelený stav). Tímto krokem si vývojář udělá jasnější představu o řešení daného problému.

Následuje fáze refaktorizace, při které se produkční kód přepisuje do té podoby, aby splňoval všechny náležitosti a doporučení pro čistý kód. Při refaktorizaci tak nedochází ke změně chování vůči externím částem aplikace, ale dochází k optimalizaci vnitřní struktury testovaného subjektu. Při refaktorizace se může stát, že některé testy již nejsou úspěšné, vracíme se tak zpět do červené fáze a cyklus je znovu započat. Iteruje se tak dlouho, dokud všechny testy nejsou úspěšné a dokud není docíleno odpovídající kvality kódu.

Při testování aplikace je důležité psát takové testy, které testují chování subjektu vůči zbytku aplikace. Pokud jsou testy příliš závislé na vnitřní struktuře testovaného subjektu, je nutné po každé refaktorizaci také upravovat samotné testy. Toto způsobuje následné zpomalení celého procesu. Má být tedy testován kontrakt aplikačního rozhraní, nikoliv samotná implementace. Je také doporučováno, aby většina testů byla psána jako jednotkové testy (*unit tests*), které testují subjekt separovaný od okolí. Tím je docíleno rychlé exekuce testů, jelikož není nutné čekat na inicializaci závislostí. Integrační testy pak ověřují funkčnost kombinace jednotek, které spoluvytvářejí jednotku na vyšší úrovni abstrakce. Na vrcholu testovací pyramidy [7] se nachází *end-to-end* testy verifikující funkčnost aplikace jako celku.

Obr. 1.3: Testovací pyramida.

1.2 Webový vývoj

Vývoj webových aplikací je komplexní oblastí, která je utvářena značným počtem technologií. Vývojář se musí orientovat nejen v tzv. *Front-end* technologiích, pomocí kterých je vytvářeno uživatelské rozhraní aplikace, ale také v principech, na kterých stojí programování aplikačních serverů. V neposlední řadě pak musí rozumět základům perzistentní vrsty, databázovým systémům a dotazovacích jazyků, pomocí kterých je možné nad databází vytvářet dotazy a získávat tak potřebná data. Je nutné tuto komplexnost před uživateli skrýt, aby bylo docíleno jednoduchého a přehledného uživatelského prostředí. Je také požadována dynamičnost (změna obsahu webu na základě uživatelových vstupů a informacích o uživateli), dále rychlá odezva, díky které jsou webové aplikace schopné konkurovat plynulosti desktopových aplikací. Pro minimalizaci doby odezvy je nutné omezit dotazy na server, které mohou způsobit jisté zpoždění. O toto se snaží *Single-page* aplikace, při kterých se webová stránka načte jen na začátku, a data jsou podle potřeby dynamicky získávána ze serveru. Uživatelé prohlíží web pomocí různých zařízení, proto je nutné vykreslovat uživatelské prostředí uzpůsobené pro velikost obrazovky, a dosáhnout tak responzivního designu [8].

Základní schéma webové komunikace je možné definovat jako komunikaci mezi klientem a serverem. Provoz zaštiťuje protokol HTTP (*Hypertext Transfer Protocol*), respektive jeho šifrovaná varianta HTTPS.

Obr. 1.4: Komunikace klienta a serveru.

1.2.1 Řízení přístupu u webové aplikace

Řízením přístupu je obecně míněna kontrola entit, které požadují přístup k informacím. Systémy řízení přístupu tento přístup regulují podle toho, zda autorita entitě přístup k informacím povolila či nikoliv. Autentizace a autorizace jsou dva důležité termíny z oboru počítačové bezpečnosti, respektive ze zmíněné oblasti řízení přístupu. Pojmem autentizace je míněno prokázání identity entity, tedy že daná entita je opravdu tou entitou, za kterou ji protistrana považuje. Po autentizaci entity se přistupuje k aktu autorizace. Autorizací entity je míněna ta skutečnost, že daná entita má dostatečná práva či povolení k vykonání specifické operace. Při řízení přístupu

k určitým datům jsou tedy taktéž kontrolována přístupová práva entity. Pokud entita nemá dostatečná povolení, k požadovaným datům ji nebude přístup umožněn. V samotné aplikaci řešené v rámci diplomové práce je pro řízení přístupu využíváno technologie JWT tokenů.

1.2.2 *JSON Web Token*

Technologie *JSON Web Token* (JWT)[3] je standardem používaným zejména pro autorizaci uživatelů v rámci webových aplikací. Token je složen ze tří částí. První částí je *header* (hlavička), následuje *payload* (přenášená data) a poté *signature* (podpis). Všechny části jsou kódovány ve formátu Base64URL (v zápisu vyjádřeno funkcí `encode()`). Hlavička nese informace o typu tokenu a zejména o použitém algoritmu k vytvoření podpisu.

Výpis 1.2: Hlavička JWT.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Důležitá data jsou poté umístěna v části *payload*. Jsou zde typicky uváděny informace o vydavateli tokenu (*issuer*, klíč `iss`), dále termín vystavení (*issued at*, klíč `iat`), termín expirace (*expiration*, klíč `exp`) a údaje o uživateli (*subject*, klíč `sub`), které jsou použity k jeho identifikaci. Může se například jednat o jedinečný identifikátor přidělený serverem.

Výpis 1.3: Přenášena data v rámci JWT.

```
{
  "sub": "user@domain.com",
  "iat": 1638198040,
  "exp": 1638284440
}
```

Poslední částí je samotný podpis, který se používá k verifikaci integrity dat, tedy že data nebyla jakkoliv změněna. Pokud je jako `secret` použit privátní klíč entity, protistrana si pomocí veřejného klíče entity může ověřit, že se opravdu jedná o token vystavený danou entitou. Vytvoření *signature* při použití funkce RSA-SHA256 je zmíněno níže.

Výpis 1.4: Podpis JWT.

```
RSASHA256(encode(header) + "." + encode(payload), secret)
```

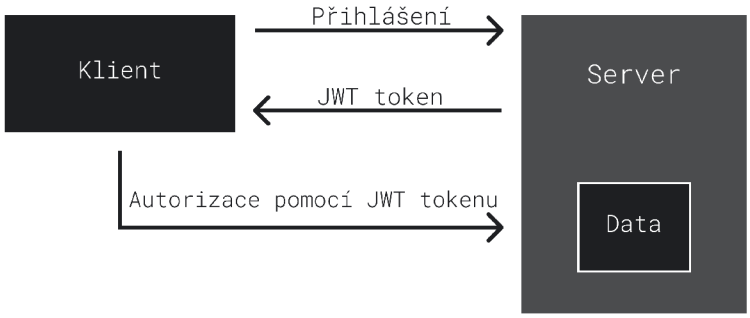
[3] `https://jwt.io/introduction`

Jak bylo zmíněno na začátku, jednotlivé části jsou kódovány do formátu Base64URL a odděleny tečkou. Tímto je vytvořena finální podoba JWT tokenu.

Výpis 1.5: Předpis a ukázka JWT.

```
encode(header) + "." + encode(payload) + "." + encode(signature)
```

Po definování struktury tokenu je nutné alespoň schématicky naznačit, jakým způsobem autorizace pomocí JWT tokenu funguje v rámci komunikace klienta se serverem. Podrobnější popis implementace autorizace je popsán v kapitole 3.3.



Obr. 1.5: Schéma autorizace pomocí JWT.

Server je autoritou pro vydávání tokenu. Do tokenu je přidáván zmíněný termín expirace. Po expiraci se uživatel musí znovu přihlásit či musí být implementován jiný způsob obnovy a vygenerování nového tokenu. Klient si token uschová, a v případě, že chce přistoupit k požadovaným datům, zašle serveru svůj token. Server provede ověření a na základě práv přidělených klientovi přístup k datům povolí či nepovolí. Klient zasílá token v rámci HTTP autorizační hlavičky.

Výpis 1.6: Umístění JWT tokenu do autorizační hlavičky.

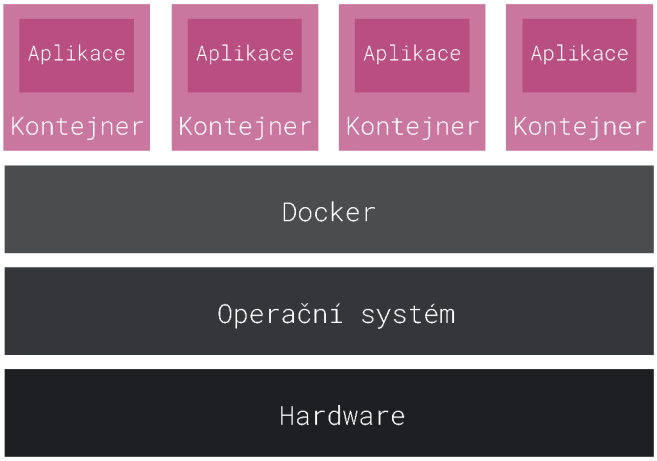
```
Authorization: Bearer <token>
```

1.2.3 Technologie *Docker*

Docker[4] je nástrojem pro virtualizaci běhového prostředí. Na rozdíl od jiných virtualizačních nástrojů *Docker* virtualizuje prostředí na úrovni operačního systému. Jedná se tedy o odlehčený způsob, jakým je možné oddělit virtualizované prostředí od hostujícího prostředí. Toto virtualizované prostředí je nazýváno kontejner. Značnou výhodou této technologie je již zmíněné oddělení, cílovou aplikaci je možné spustit v rámci kontejneru a provést tak bezpečnou separaci. Hostující prostředí je poté

[4] `https://www.docker.com`

chráněno od hrozeb, které mohou vzniknout za běhu aplikace (kdy například útočník využije vulnerability aplikace a získá přístup ke kontejneru). *Docker* je také využíván pro oddělení jednotlivých částí aplikace, kdy se serverová část, databáze či jiná složka aplikace umístí do separátního kontejneru. Toto umožňuje snadnější správu a následnou škálovatelnost celé aplikace.



Obr. 1.6: Jednotlivé vrstvy technologie *Docker*.

Zejména při vývoji větších projektů *Docker* zastává důležitou roli, protože umožňuje sjednocení vývojového prostředí pro všechny vývojáře. Na začátku jsou v kontejneru nainstalovány všechny potřebné závislosti, dále je pak toto prostředí rozdistribuováno jednotlivým vývojářům. Každý vývojář pracuje s lokální kopií daného prostředí, tímto je tedy zařízena jednotnost prostředí, ve kterém je aplikace vyvíjena. Je odstraněna zdlouhavá synchronizace jednotlivých závislostí, a navíc je umožněno provádět investigaci chybových stavů ve sjednoceném prostředí.

Nyní přejděme k praktické ukázce použití kontejnerů. Technologie *Docker* využívá dva hlavní termíny. První z nich je *image* (obraz), který slouží jako předloha pro vytvoření samotného běhového prostředí, tedy již zmíněného kontejneru. *Image* je možné definovat pomocí konfiguračního souboru nazývaného *Dockerfile*. Ukázka konfigurační souboru je na následujícím kódovém výpisu.

Výpis 1.7: Sestavení obrazu a spuštění kontejneru.

```
1 FROM python:3.8-slim-buster
2 WORKDIR /app
3 COPY ./requirements.txt ./requirements.txt
4 RUN pip3 install -r requirements.txt
5 COPY ./server.py ./server.py
6 CMD ["python3", "server.py"]
```

Konfigurace je poměrně intuitivní. Jako základ je použito prostředí s předinstalovanou verzí 3.8 jazyka *Python*. Dále je zvolen pracovní adresář `app`, který slouží jako výchozí adresář pro kopírování souborů z hostujícího prostředí. Kopírování souboru `requirements.txt` je realizováno pomocí příkazu `COPY`. Prvním parametrem je soubor nacházející se na lokálním disku, druhý parametr pak určuje lokalizaci a jméno nově vzniklého souboru v kontejneru. Příkazem `RUN` je pak možné spustit zvolený příkaz. V našem případě se jedná o instalaci balíčků, které jsou definovány v souboru `requirements.txt`. Tímto je prostředí připraveno pro běh souboru `server.py`. Po nakopírování souboru `server.py` bude pomocí příkazu `CMD` při startu kontejneru serverová aplikace spuštěna. U tohoto příkazu je důležité zmínit, že pokud bude skript `server.py` z jakéhokoliv důvodu ukončen, bude ukončen i samotný běh kontejneru. V terminálovém okně je poté možné sestavit *image* a spustit kontejner pomocí příkazů ve výpisu 1.8.

Výpis 1.8: Sestavení obrazu a spuštění kontejneru.

```
1 docker build -t python-server
2 docker run -dp 8888:9999 python-server
```

Přepínač `-t` umožňuje kontejneru přidělit *tag* (značka), kterým je poté *image* identifikován. Následující přepínače `-dp` vyjadřují příkaz spuštění kontejneru v tzv. *detached* módu (na pozadí), respektive namapování portů mezi hostujícím prostředím a samotným kontejnerem. Díky tomuto je umožněna síťová komunikace s aplikačním serverem.

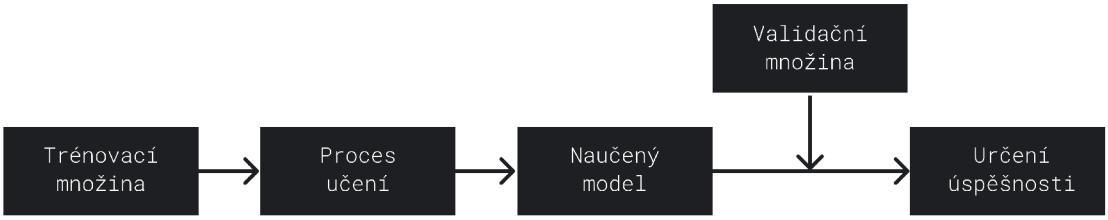
Aplikace může být tvořena více kontejnery, které je možné umístit do různých sítí. Je například možné skrýt kontejner s databází za kontejner se serverovou aplikací tak, aby bylo možné přistupovat do databáze právě jen přes webový aplikační server. Aplikace složené z více kontejnerů lze konfigurovat pomocí souboru `docker-compose.yml`.

Výpis 1.9: Příklad konfigurace aplikace složené z více kontejnerů.

```
1 services:
2   application:
3     image: python
4     ports:
5     - "7777:7777"
6   database:
7     image: mysql
8     ports:
9     - "8888:8888"
```

1.3 Strojové učení

Algoritmy v oblasti strojového učení jsou implementovány tak, aby byl výsledný program (tedy model) schopný se učit, respektive adaptovat okolním podmínkám. To, jak je model úspešný, je počítáno pomocí předem zvolené funkce, kterou se model buď snaží maximalizovat či minimalizovat. Proces učení je v mnoha případech časově a výpočetně náročný. Jedná se často o iterativní postup, kdy vývojář hledá optimální parametry a vlastnosti modelu k úspěšnému zvládnutí požadovaného úkolu. Ovšem nejde jen o hledání vhodných parametrů. Úspěšnost modelu a čas potřebný k natrénování modelu je velmi ovlivněn samotnými daty. Je nutné zajistit odpovídající kvalitu trénovacích dat a jejich značení [10]. V praxi je hojně využíván postup pojmenovaný jako učení s učitelem – správný výsledek je již předem určený. Vývojář vytvoří takovou množinu dat, kdy každá jednotlivá datová jednotka je pomocí značky přiřazena do určité třídy (viz kapitola 2.5.2). Při procesu učení (respektive trénování) model pomocí definovaných principů upravuje své vnitřní parametry tak, aby bylo postupně dosáhnuto stavu, kdy je každá datová jednotka přidělena do té třídy, do jaké ji původně přidělil vývojář (ideální případ). Na strojové učení může být nahlíženo jako na snahu o optimální reprezentaci dat v předem definovaném prostoru [11]. Optimální reprezentace vstupních dat (k tomu napomáhá již zvolení datové množiny s odpovídající kvalitou dat), je pak předpokladem pro snadnější dosáhnutí požadovaného výsledku, a to často i v kratším čase. Na následujícím diagramu 1.7 je znázorněn proces učení modelu strojového učení.



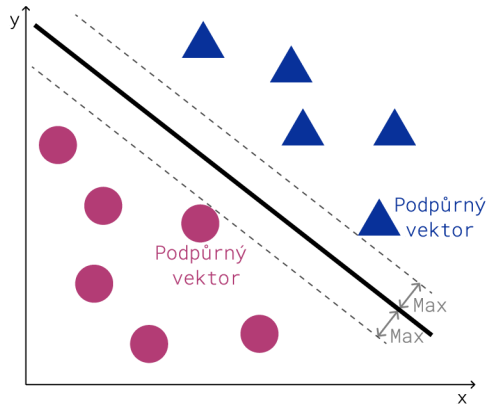
Obr. 1.7: Diagram průběhu strojového učení.

1.3.1 Modely strojového učení

Následující podkapitoly se věnují popisu modelů a algoritmů strojového učení, které budou využity aplikací implementované v rámci diplomové práce. Ačkoliv se práce nezaměřuje na samotnou implementaci modelů, je vhodné jednotlivým modelům porozumět alespoň na úrovni vnějšího kontraktu. Tedy vědět pro jakou situaci je vhodné daný model zvolit. To samé platí i pro optimalizační parametry, které by měl vnější volající znát, a zároveň by si měl být vědom, jak ovlivňují chování modelu.

Metoda podpůrných vektorů

Metoda podpůrných vektorů nesoucí anglický název *Support Vector Machines*, je určena k rozdělení vstupních dat do jednotlivých tříd (klasifikace). Obecně metoda rozděluje data do dvou tříd [12], pracuje tedy jako binární klasifikátor. Podstatou je rozdělení prostoru vstupních dat o n dimenzích za pomocí nadroviny (nabývající $n-1$ dimenzí) tak, aby vektory jednotlivých tříd byly nadrovině co nejvzdálenější. V případě, že je prostor rovinou, je nadrovina definována jako přímka. Podpůrný vektor reprezentovaný bodem v prostoru je ze všech vektorů dané třídy právě nejbližším vektorem k rozdělující nadrovině. Metoda podpůrných vektorů je snahou o nalezení nejširšího hraničního pásma, která odděluje vektory jednotlivých tříd. Problematiku nalezení takové pásma je možné vizualizovat pomocí schématu 1.8.



Obr. 1.8: Rozdělení roviny pomocí metody SVM.

Z matematického hlediska je možné na problém klasifikace do jednotlivých tříd nahlížet jako na funkci [13]:

$$f : \mathbb{R}^{\mathrm{N}} \rightarrow (\pm 1), \tag{1.1}$$

která přiřazuje $N-$rozměrná data $\mathbf{x}_{\mathrm{i}}$ do odpovídající třídy y_{i}. Zároveň je korektně prováděna klasifikace pro novou dvojici $(\mathbf{x}, y)$, tedy $f(\mathbf{x}) = y$. Při uvažování $\mathbb{R}^2$ prostoru můžeme rozdělení vstupních dat (tedy vektorů) realizovat pomocí přímky definované jako:

$$\vec{w} \cdot \vec{x} + b = 0, \tag{1.2}$$

kde $\vec{w}$ reprezentuje normálu nadroviny, b vyjadřuje koeficient posunutí. Vzdálenost nadroviny od počátku souřadnic je možné vyjádřit zápisem $|b|/||\vec{w}||$. Pro vstupní vektory platí:

$$\vec{w} \cdot \vec{x_{\mathrm{i}}} + b \geq +1 \text{ pro } y_{\mathrm{i}} = +1, \tag{1.3}$$

$$\vec{w} \cdot \vec{x_{\mathrm{i}}} + b \leq -1 \text{ pro } y_{\mathrm{i}} = -1, \tag{1.4}$$

kde $\vec{x_i}$ představuje vstupní vektor, parametr y_i udává klasifikační třídu. Je tak vytvořena sada trénovacích dat $(\vec{x_{\mathrm{i}}}, y_{\mathrm{i}})$. Vztahy 1.3, 1.4 tedy vyjadřují rozdělení vstupních vektorů do příslušných tříd. V našem případě podpůrnými vektory dílčích tříd prochází přímka p_1, respektive p_2. Podpůrné vektory jsou tedy hraničním případem, který lze vyjádřit jako:

$$p_1 : \vec{w} \cdot \vec{x} + b = 1, \tag{1.5}$$

$$p_2 : \vec{w} \cdot \vec{x} + b = -1. \tag{1.6}$$

Přímky p_1, p_2 jsou rovnoběžné s rozdělující přímkou, respektive jsou kolmé na normálový vektor $\vec{w}$ rozdělující nadroviny, a vymezují tak hraniční pásmo. Vzdálenost přímek od počátku souřadnic je definována jako:

$$\frac{|1 - b|}{||\vec{w}||} \text{ pro } p_1, \tag{1.7}$$

$$\frac{|-1 - b|}{||\vec{w}||} \text{ pro } p_2. \tag{1.8}$$

Jejich vzájemná vzdálenost poté bude rovna rozdílu jejich vzdáleností od počátku souřadnic (rozdíl vztahů 1.7 a 1.8), tedy $2/||\vec{w}||$. V případě, že je soubor dat lineárně separabilní, nebude se v hraničním pásmu nacházet žádný vektor. Nejširšího pásma dosáhneme minimalizací $||\vec{w}||$ [14] při dodržení vztahů 1.3 a 1.4.

Doposud jsme uvažovali takové problémy, které jsou lineárně separabilní. Při řešení praktických problémů, které nejdou v původním prostoru klasifikovat pomocí lineárních vztahů, je využíváno jádrových transformací. Data z původního prostoru jsou převedena do vyšší dimenze, ve které již můžeme provést lineární separaci. Provádíme tedy zobrazení Φ:

$$\Phi : \mathbf{R}^{\mathrm{d}} \mapsto \mathbf{R}^{\mathrm{D}}, \tag{1.9}$$

kde $\mathbf{R}^{\mathrm{d}}$ je původní prostor a $\mathbf{R}^{\mathrm{D}}$ prostor vyšší dimenze. Transformačních jader existuje celá řada, mezi nejvíce používané patří jádrové transformace Gaussova typu, polynomiální či sigmoidální [12].

Metodu podpůrných vektorů je možné optimalizovat pomocí dvou parametrů. *Gamma* parametr určuje, kolik bodů bude zahrnuto do konečného výpočtu. Pokud budou

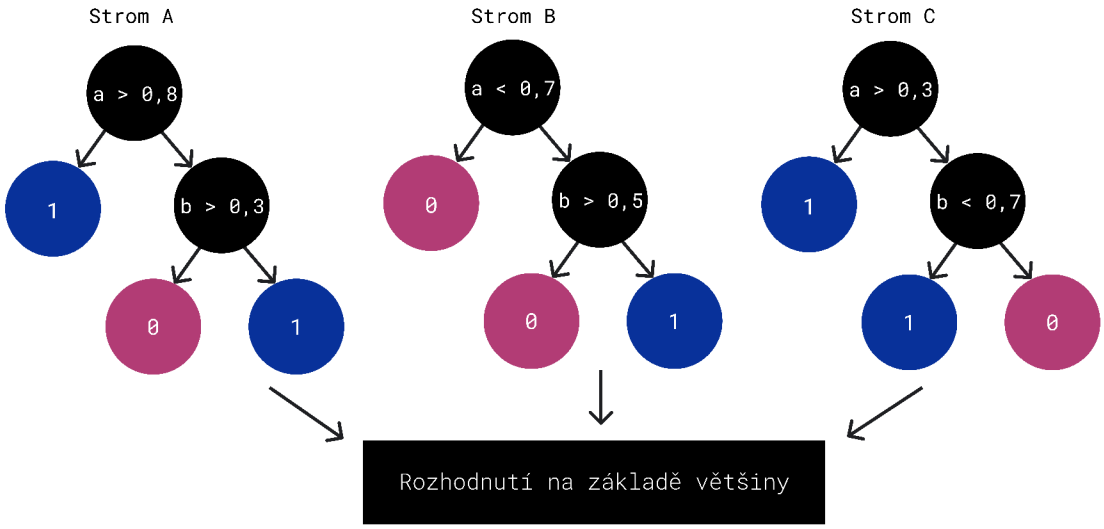
zvoleny nízké hodnoty parametru *gamma*, model bude počítat s větším počtem okolních vektorů, vysoké hodnoty pak zajistí, že se do výpočtu budou zahrnovat jen vektory nejbližší. Ve své podstatě je tento parametr převrácenou hodnotou poloměru kružnice, v jejíž vnitřní oblasti leží vektory ovlivňující výpočet. Parametr označený písmenem C udává, do jaké míry je tolerována nepřesná klasifikace vstupních dat. Funguje jako regulátor mezi chybou a komplexností celého výpočtu [15].

Náhodný les

Dalším algoritmem, který je možné použít ke klasifikaci, je náhodný les. Jak už název napovídá, model je tvořen více rozhodovacími stromy, které pak utváří les. Daný strom je tvořen z uzlů, které jsou reprezentovány určitým příznakem na základě kterého dochází k větvení. Nabízí se tak následující otázka. Jakým způsobem volit pořadí příznaků, aby vstupní data byla správně klasifikována? Jednotlivé algoritmy větvení poskytují různá řešení. Například algoritmus ID3 vybere pro větvení ten příznak, který nabídne největší informační zisk [16], respektive v porovnání s ostatními příznaky nejvíce sníží entropii na základě vztahu:

$$I(S, A) = H(S) - H(S|A), \tag{1.10}$$

kde I vyjadřuje očekávaný informační zisk, $H(S)$ značí entropii datové množiny S, $H(S|A)$ pak vyjadřuje váhovanou sumu entropií množin, které vzniknou je-li datová množina S rozdělena na základě příznaku a. Celkový počet takto vytvořených množin je roven mohutnosti A, jelikož A vyjadřuje všechny dostupné hodnoty, kterých nabývá příznak a. Stromy jsou sestavovány do lesa. Každý strom je natrénován



Obr. 1.9: Náhodný les.

pomocí náhodně zvolených dat ze vstupní datové množiny, což vede k sestrojení různých stromů. Výsledkem je zlepšení vlastností a menší závislost na vstupních datech [17]. Pomocí většinového hlasování je pak zvolena výsledná třída daného vzorku.

2 Architektura aplikace

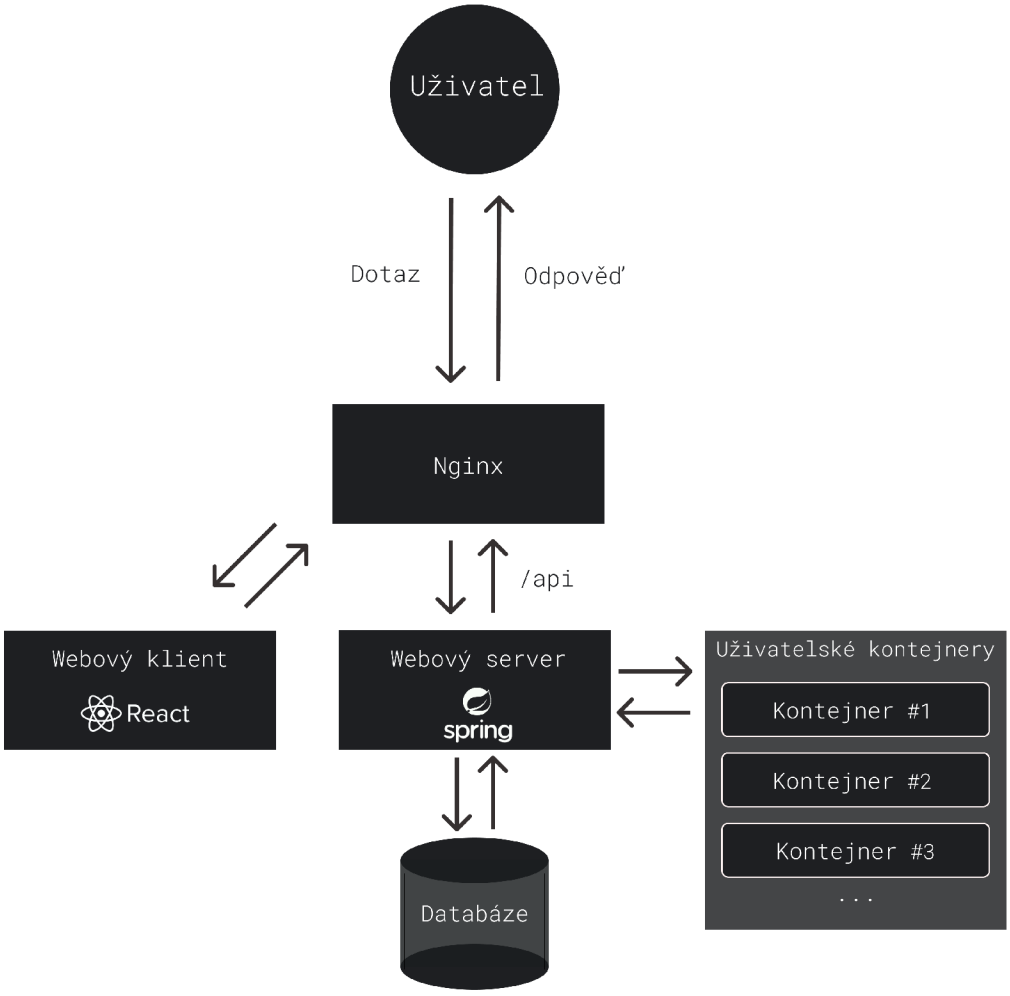
Následující část má zá úkol seznámit čtenáře se stavebními bloky celé aplikace, jež byla implementována v rámci diplomové práce. Aplikace má být platformou pro exekuci trénování a validaci modelů strojového učení. Tato hlavní funkcionalita se také odráží ve struktuře celé aplikace. Aplikace pro uživatele sestrojí separátní kontejner, ve kterém je možné spustit proces učení modelu strojového učení. Před samotnou exekucí učení je uživateli umožněno nahrát do kontejneru data, nad kterými se bude zvolený model učit, a jejichž část bude použita k validaci modelu. Bude následovat výčet jednotlivých částí aplikace společně s ukázkami použitých technologií v rámci každé části aplikace. Nebude ještě diskutována přesná implementace funkcionality aplikace, nýbrž bude naznačena práce s jednotlivými technologiemi, což později povede k lepší čtenářově orientaci v rámci samotné implementace aplikace.

2.1 Jednotlivé části aplikace

Aplikace je složena z pěti hlavních částí. Webový klient, který je naimplementován pomocí knihovny *React*, slouží jako uživatelské prostředí (*user interface* - UI), se kterým uživatel provádí interakci. Webový klient zasílá dotazy na webový server, který je jakýmsi prostředníkem mezi klientem, databází a kontejnery s *Python* běhovým prostředím. Dostáváme se tak k důležité části, a to k samotným kontejnerům, na kterých dochází k exekuci trénování a validaci modelů. Každému uživateli je při jeho prvotním přihlášení přidělen separátní kontejner, na kterém jsou instalovány všechny závislosti potřebné ke spuštění procesu učení. Do kontejneru se také ukládají všechny uživatelovy datové množiny, které nahraje pomocí webového klienta. Separace do jednotlivých kontejnerů je poměrně důležitá. Jedním důvodem je bezpečnost. Pokud se útočník díky chybě v kódu (například nahráním skriptu) zmocní kontejneru, nedostane se tak k citlivým datům ostatních uživatelů, jelikož právě jednotlivé kontejnery jsou od sebe odděleny.

Kontejnerizace umožní také lepší škálovatelnost, popřípadě v následujících iteracích aplikace může být vyvinuta funkcionalita, kdy bude skupině uživatelů přidělen jeden či více kontejnerů. Uživatelé budou moct mezi sebou sdílet datové množiny a zároveň kontrolovat výsledky validace modelů. *Docker* také umožňuje spravovat množství přidělené paměti, popřípadě vytíženost procesoru. Správce aplikace tak eventuálně může jednotlivým uživatelům nastavit (respektive jim přiděleným kontejnerům) přidělené výpočetní prostředky, a to například v závislosti na úrovni předplatného.

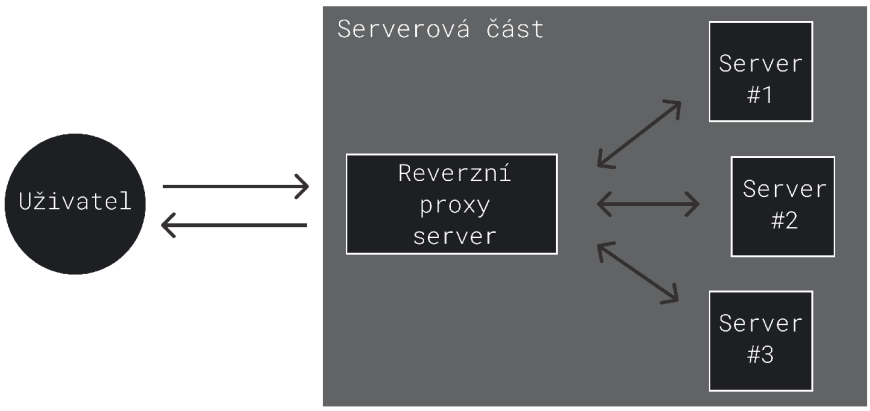
Všechny části aplikace jsou umístěny do kontejnerů. S kontejnery, na kterých běží trénování a validace modelů je možné komunikovat jen pomocí webového serveru. Uživatelským kontejnerům mezi sebou není umožněno komunikovat (již zmíněné důvody bezpečnosti). Pro potřeby databázového úložiště bylo v produkčním prostředí zvoleno řešení *MySql*. Ve vývojovém (testovacím) prostředí je použita databáze *H2*, která neukládá soubory na disk, ale pracuje jako tzv. *in-memory* databáze (soubory jsou ukládány do operační paměti). Databáze je znovu vytvořena při každém spuštění aplikace. V rámci testovacího prostředí je značnou výhodou právě konzistentní stav této databáze při každém novém spuštění. Vývojář tak nemusí složitě mazat datové položky, které vznikly během vývoje, a které už jsou například nejsou validní.



Obr. 2.1: Architektura aplikace.

2.2 Server *Nginx*

Vstupem do aplikace je server *Nginx*, který plní roli reverzní proxy. Jeho hlavní funkcí je přesměrovávat komunikaci na aplikační servery. V rámci řešení diplomové práce server *Nginx* přesměrovává dotazy, jejichž adresa začíná `\api`, na webový server *Spring*. Veškerá další komunikace je přesměrována na server poskytující aplikaci webového klienta (implementována za pomoci knihovny *React*). Výhod použití reverzní proxy je hned několik. Zaprvé umožnuje zajistit šifrovanou HTTPS komunikaci. Certifikát TLS je nutné nahrát jen na tento reverzní proxy server. Není potřeba nijak konfigurovat aplikační servery pro podporu šifrované komunikace. V rámci serverové části může komunikace obecně probíhat v nezašifrovaném režimu. Proxy server může být dále využit jako tvz. *load balancer*, kdy je zátěž v podobě klientských dotazů optimálně rozložena mezi jednotlivé aplikační servery. V neposlední řadě může být server využit také k mitigaci DoS útoků (*denial of service*, odmítnutí či nedostupnost služeb). V konfiguraci *Nginx* je možné definovat omezení pro počet připojení (celkově či pro jednu IP adresu), počet dotazů za časový interval, vypršení časovače a další omezení.



Obr. 2.2: Reverzní proxy server.

2.3 Webový klient

Webový klient zprostředkovává grafické uživatelské rozhraní, se kterým samotný uživatel interaguje. Aby aplikace nabízela nejen statický obsah (rozdílný obsah pro rozdílné uživatele, zobrazení vybraných informací na základě stavu aplikace) je nutné s HTML dokumentem manipulovat pomocí rozhraní DOM (*Document Object Model*). Toto umožňuje programovací jazyk *JavaScript*, který je v oblasti webových stranek naprosto dominantním jazykem. Jelikož se jedná o jazyk, který nevyžaduje

explicitní definici typů (což způsobuje značná úskalí při vývoji aplikací), jsou v posledních letech aplikace migrovány na typovaný jazyk *TypeScript*[1]. Soubory psané v jazyku *TypeScript* jsou před spuštěním překládány do jazyku *JavaScript*, a jsou tak zpětně kompatibilní. Výhodou jazyka *TypeScript* jsou právě zmíněné typované objekty a primitiva, které minimalizují riziko chyb. Také jsou pomocí typů jasně definována rozhraní či parametry funkce. Vývojáři je tak při psaní kódu ve vývojovém prostředí nabídnuto, jaké parametry (a s jakým typem) funkce nabízí, a zároveň jsou nabídnuty dostupné proměnné nad objektem, se kterým vývojář pracuje. Nevýhodou je pomalejší psaní kódu z důvodu větší náročnosti kladené na vývojáře. Mnozí však argumentují, že bez použití *TypeScriptu* je vývoj stabilní aplikace ještě pomalejší, jelikož je nutné započítat i čas potřebný k odhalování a opravení chyb způsobených dynamickým typováním. Pro implementaci webového klienta byl tedy primárně zvolen jazyk *TypeScript*.

2.3.1 Knihovna *React*

V oblasti webového vývoje je dostupná celá řada nástrojů usnadňující implementaci. Společným jmenovatelem je tedy použití nástrojů poskytující vhodnou úroveň abstrakce, kdy se vývojář nemusí zabývat vývojem nízkoúrovňových služeb, ale rovnou tyto služby díky nástrojům využívá, a je mu tak umožněno se plně soustředit na vývoj požadované funkcionality. Jedním z nástrojů je knihovna *React*, která je založena na principu tvorby komponent.

2.3.2 *React* a komponenty

Jednotlivé komponenty je možné sestavovat a vytvářet tak komponenty pokročilější. Komponenty můžeme přirovnat k objektům z objektově orientovaného paradigmatu. V ideálním případě by každá komponenta měla zprostředkovávat jednu jasně definovanou funkcionalitu (viz kapitola 1.1.2 zabývající se principy SOLID. Poté je docíleno minimální duplikace kódu. Navíc pokud je každá komponenta prvkem sama o sobě, je možné snadno otestovat její chování (diskutováno v kapitole 1.1.3).

V následujícím výpisu 2.1 se nachází komponenta `ProjectList` poskytující funkcionalitu zobrazení seznamu komponent typu `IndividualProject`. Stěžejní metodou je metoda `return()`, která definuje způsob zobrazení komponenty (tedy vykreslení na uživatelově obrazovce). V rámci knihovny *React* je použito rozšíření JSX[2], které umožňuje použití HTML značek a zároveň pomocí primitiv nabízí vložení a generování dynamického obsahu. Jedná se například o zobrazení textu na základě

[1] `https://www.jetbrains.com/lp/devecosystem-2021/`
[2] `https://reactjs.org/docs/introducing-jsx.html`

obsahu proměnné (řádek 36) či průchod seznamem prvků se zobrazením jednotlivých elementů s využitím metody `map()`. Jednotlivým elementům je možné předávat informaci o CSS stylování (*Cascading Style Sheets* - kaskádové styly) v podobě atributu `className`.

Knihovna *React* také nabízí sadu metod (tzv. *Hooks*), které se mimo jiné starají o zprostředkování a manipulaci stavu komponenty. Hojně používanou je metoda `useState()`, pomocí které je možné pozměnit stav parametru. Změna jakéhokoliv parametru pak vyvolá překreslení celé komponenty a uživateli je tak zobrazen obsah konzistentní s aktuálním stavem aplikace. Metoda `useState()` také umožňuje nastavit počáteční hodnotu parametru. Například na řádku 10 je pomocí této metody definováno pole prvků `projects` s elementy typu `Project` (zde vidíme i explicitní použití jazyku *TypeScript*). Pole `projects` je na počátku prázdné.

Pro naplnění pole `projects` je možné využít metodu `useEffect()`, která je volána při prvotním sestavení komponenty. V jejím těle se proto nejčastěji vyskytuje volání aplikačního rozhraní serveru (v našem případě je použita knihovna *axios*), které zprostředkuje klientské aplikaci data k zobrazení. Je důležité si povšimnout, že toto volání je asynchronní, kdy vykonávání dalšího kódu není voláním blokováno. Asynchronní operace jsou řešeny pomocí konstruktu *Promise* (není již specifický pro knihovnu *React*). V okamžiku, kdy jsou data klientem úspěšně získána, dojde ke změně stavu komponenty a komponenta je překreslena. Na toto čekání je samozřejmě možné dynamicky reagovat, a to například zobrazením animace načítání či vhodným textem (řádek 33). Je také nabídnuta možnost ošetření chybového stavu v případě selhání výměny dat (řádek 25). Uživateli je pak skrze parametr `errorMessage` zobrazen obsah chybové zprávy.

Externí volající komponentě může předat parametry, se kterými komponenta vnitřně pracuje jako s tzv. *properties* (často je uváděno jen *props*). Díky jazyku *TypeScript* je možné typ *properties* jasně definovat a externí volající je tak obeznámen se seznamem parametrů, které komponenta vyžaduje (v našem případě se jedná o typ `ProjectListProps`).

Výpis 2.1: Ukázka implementace *React* komponenty.

```
1 interface Project
2 {
3   id: number,
4   name: string
5 }
6
```

```
7  interface ProjectListProps
8  {
9    apiUrl: string
10 }
11
12 function ProjectList(props: ProjectListProps)
13 {
14   const [isLoaded, setLoaded] = useState(false);
15   const [projects, setProjects] = useState<Project[]>([]);
16   const [errorMessage, setErrorMessage] = useState("");
17
18   useEffect(() => {
19     axios.get<any>(props.apiUrl)
20       .then((response) => {
21           setLoaded(true);
22           setProjects(response.data.projects);
23         },
24         (error) => {
25           setLoaded(true);
26           setErrorMessage(error.message);
27         }
28       )
29   }, [])
30
31   if (!isLoaded)
32   {
33     return(<p className="text-loading">Loading...</p>);
34   }
35   else if (errorMessage !== "")
36   {
37     return(<p className="text-error">{errorMessage}</p>);
38   }
39   else if (projects.length === 0)
40   {
41     return(<p className="text-information">No projects created.</p>);
42   }
43   else
44   {
45     return (
46       <ul>
47         {projects.map(project => (
48           <li key={project.id}>
49             <IndividualProject id={project.id} name={project.name} />
50           </li>
51         ))}
52       </ul>
53     );
54   }
55 }
```

2.4 Webový aplikační server

Stěžejní částí celé aplikace je webový aplikační server propojující další části aplikace. Jelikož server zprostředkovává veškerou interní komunikaci, bylo nutné zvolit značně robustní řešení. Jedním z nabízených řešení, se kterým je autor prakticky seznámen, je ekosystém jazyku *Java*. V posledních letech v oblasti vývoje webových aplikací mu jsou silnou konkurencí jazyky jako *Python* (řešení typu *Flask*) či *Javascript/TypeScript* (pomocí řešení *Node* je možno implementovat aplikační servery). Ale právě pomocí spojení statického typování jazyka *Java* s aplikačním rámcem *Spring*, je možné dosáhnout robustní implementace. *Spring* poskytuje nástroje redukující množství napsaného kódu, a to díky použitému návrhovému vzoru dekorátor (prakticky pak díky anotacím). Dále nabízí architekturu oddělující jednotlivé domény (viz dále) nebo objektově relační mapování, které řeší problém odstínění od specifik databázového systému.

2.4.1 Aplikační rámec *Spring*

Aplikační rámec *Spring* je souhrným termínem pro kolekci modulů poskytující příslušnou funkcionalitu (webové rozhraní, zajištění bezpečnosti a další). V rámci diplomové práce byl konkrétně zvolen aplikační rámec *Spring Boot*. Důvodem je odstínění vývojáře od nastavení aplikace. *Spring Boot* na pozadí nastaví aplikační server. Vývojář se tak tímto krokem nemusí zdržovat a může začít se samotnout deklarací a následnou implementaci rozhraní webového serveru. *Spring* nabízí velmi intuitivní výběr jednotlivých modulů a generování projektu s vybraným nástrojem pro sestavení aplikace (tzv. *build tool*) na následující webové adrese[3].

Jako nástroj pro sestavení aplikace byl zvolen *Gradle*, který umožňuje spravovat *Java* kód na úrovni modulů, definovat závislosti celého projektu či spustit požadované akce. Je s ním možné interagovat pomocí příkazové řádky. Často je používán příkaz `gradlew`, který na pozadí nakonfiguruje požadovanou verzi *Gradle*. Například pomocí příkazu `./gradlew run -x test` je možné spustit celou aplikaci, aniž by byly před spuštěním aplikace exekuovány testy. Všechny potřebné závislosti je pak nutné přidat do souboru `build.gradle`. Na závislosti aplikace implementované v rámci diplomové práce je možné nahlédnout v repozitáři na této adrese[4].

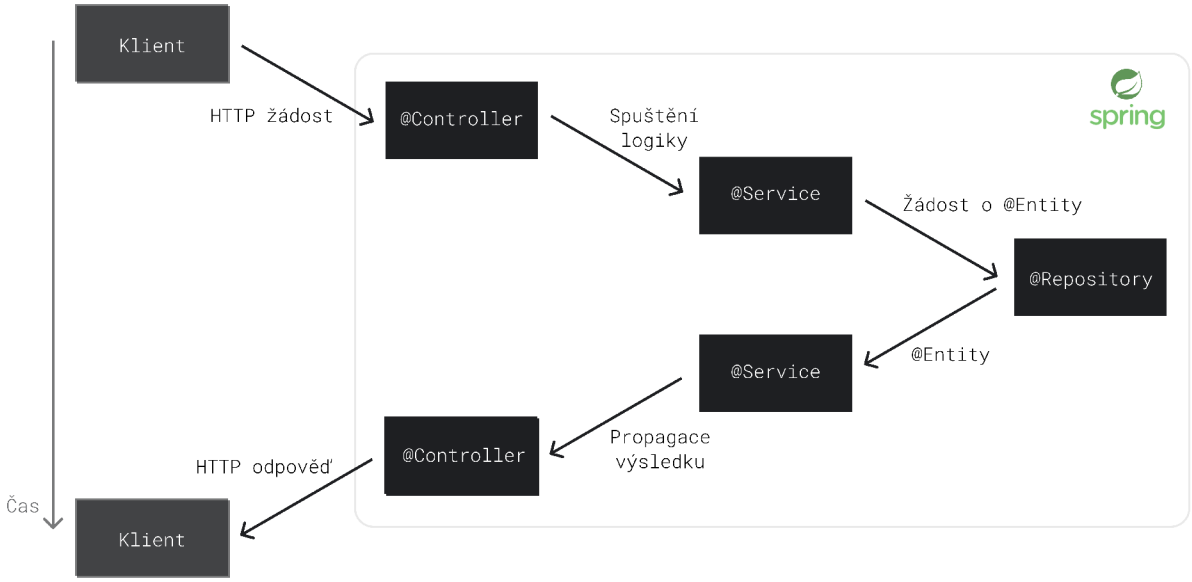
[3] `https://start.spring.io/`

[4] `https://github.com/JanSvoboda6/ml_runner/blob/main/java/web/build.gradle`

2.4.2 Struktura *Spring* aplikace

Jak bylo naznačeno, aplikační rámec *Spring* definuje řadu anotací. Základní anotace, které se používají na úrovni třídy, a které reflektují rozdělení aplikace do jednotlivých logických úrovní budou v krátkosti popsány. Společným rodičem v dalším odstavci zmíněných anotací (`@Controller`, `@Service`, `@Repository`) je anotace `@Component`. Takto anotovanou třídu *Spring* framework detekuje a v případě potřeby se ji pokusí instanciovat se všemi závislostmi. Konstruktor, který má být při instancializaci použit, je označen anotací `@Autowired`.

Na úrovni interakce klienta s aplikačním serverem je nutné definovat rozhraní. Jelikož se v diplomové práci autor snaží implementovat aplikaci webového serveru je vnějšímu volajícímu umožněno se serverem navázat komunikaci na základě protokolu HTTP. Třídy anotované pomocí `@Controller` poté provádí mapování HTTP dotazů na volání metod. Metody, které aplikační rozhraní webového serveru vystavuje vnějšímu volajícímu, jsou označeny anotacemi jako `@GetMapping` či `@PostMapping`). Třídy doplněné o anotaci `@Controller` jsou tedy vstupním bodem do aplikace. Vstupní data jsou dále zpracována samotnou logikou aplikace. Hlavní logiku tedy tvoří třídy označené pomocí anotace `@Service`. Po zpracování je obvykle nutné data uložit. O to se pak stará perzistentní vrstva, se kterou je možné interagovat pomocí tříd anotovaných jako `@Repository`. Schéma 2.3 podává zjednodušující pohled na vzájemnou interakci tříd. Můžeme tedy pozorovat, že stavebním kamenem aplikace jsou třídy označené již zmíněnými anotacemi. Třída s označením `@Entity` reprezentuje datovou strukturu, která je získána z databáze, a kterou je posléze možné uložit do databáze s pozměněnými hodnotami.



Obr. 2.3: Interakce tříd v kontextu aplikačního rámce *Spring*.

Výhodou celého systému je separovatelnost jednotlivých částí od sebe. Vývojáři se tak snadněji mentálně oddělují jednotlivé kategorie tříd, a tím nedochází k situacím, kdy jedna třída zprostředkovává příliš mnoho rozdílných funkcionalit, ale opravdu má jen jedinou odpovědnost. Opět zde vystupují principy SOLID zmíněné v kapitole 1.1.2.

Model aplikace je pak primitivní. Třída označená jako `@Controller` se stará o zprostředkování vstupních dat, třída doplněna o `@Service` provádí operace nad těmito daty, třída s anotací `Repository` pak zpracovaná data ukládá. Separace je užitečná i při testování. Je například možné použít redukovanou třídu označenou `@Repository`, která poskytuje metody, jež ve své vnitřní implementaci vlastně vůbec nekomunikují s databází, ale jen vrací předem ustanovenou datovou strukturu (hlavní princip bude vysvětlen v kapitole 3.1). Tato separace pak vede k možnosti psaní testů s menším počtem běhových závislostí (například spuštění a příprava databáze), což vede ke snížení běhového času testu, a tak k urychlení procesu zpětné vazby, které testy primárně poskytují.

2.4.3 Konfigurace *Spring* aplikace

Pro získání alepoň částečného přehledu o praktickém fungování aplikačního rámce *Spring* je nutné zmínit ještě možnosti konfigurace, které jsou vývojařům nabízeny. Centrálním bodem je soubor `application.properties`, který umožňuje přidání nastavení již definovaných parametrů, které *Spring* poskytuje, či vlastních parametrů, které vývojář v rámci aplikace používá. Typicky se v tomto souboru definuje přístup k databázi či volba portu, na kterém webový server naslouchá. V rámci diplomové práce je také hojně využívána možnost odkázání se na systémové proměnné (*environment variables*), pomocí kterých lze nastavovat citlivé údaje až při spuštění aplikace na zvoleném produkčním serveru (viz kapitola 3.4). Tyto údaje tak nejsou veřejně dostupné v repozitáři (řádky 3 a 4 ve výpisu 2.2).

Výpis 2.2: Ukázka konfiguračního souboru `application.properties`.

```
1 spring.datasource.url=jdbc:h2:mem:test_database
2 server.port=8080
3 spring.mail.username=${EMAIL}
4 spring.mail.password=${EMAIL_PASSWORD}
```

S hodnotami parametrů lze poté pracovat i v rámci implementace kódu. Aplikační rámec *Spring* skrze anotaci `@Value` zpřístupňuje konfigurační parametry pro použití v *Java* souborech.

Výpis 2.3: Anotace `@Value`.

```
public class RandomClass
{
  @Value("${server.port}")
  private int PORT;

  //Další parametry

  //Metody třídy
}
```

Dalším použitým primitivem je anotace `@Scheduled` označující metody, které mají být opakovaně volány v předem definovaném intervalu. Jedná se o obdobu příkazu `cron` pod platformou *Linux*. Pravidelně spouštěné metody byly například využity pro mazání uživatelských účtů, které nebyly aktivovány.

Výpis 2.4: Anotace `@Scheduled`.

```
@Service
public class NonVerifiedUserPurger
{
    //Inicializace závislostí

    @Scheduled(fixedRate = 10, timeUnit = TimeUnit.SECONDS)
    public void purge()
    {
        //Tělo metody
    }
}
```

2.5 Uživatelský kontejner

Kontejner sloužící jako běhové prostředí pro trénování a validaci modelů pro modely strojového učení využívá knihovnu *scikit-learn*. Tato knihovna je postavena na platformě *Python*. Jelikož je kontejner sestavován pro každého uživatele, je nutné docílit minimální velikosti kontejneru. Z tohoto důvodu byl pro komunikaci kontejneru s aplikačním serverem použit aplikační rámec *Flask*, který je rovněž napsán v jazyku *Python*. V rámci souboru *Dockerfile*, který mimo jiné definuje závislosti aplikace spuštěné uvnitř kontejneru, je možné proto použít základní vrstvu definovanou jako `python:3.8-slim-buster`, která poskytuje minimální prostředí pro běh *Python* skriptů. Není tedy potřeba instalace žádných dalších programovacích jazyků, platforem či jejich závislostí.

2.5.1 Aplikační rámec *Flask*

Aplikační rámec *Flask* je jednoduchým řešením pro vytvoření webového serveru. Ve své základní verzi na rozdíl od aplikačního rámce *Spring* nenabízí veškerou pokročilou funkcionalitu (validace příchozích dotazů, abstrakce perzistentní vrstvy a další). Jedná se tedy o vhodného kandidáta právě pro využití v uživatelském kontejneru, kde je kladen důraz na minimální velikost výsledného kontejneru. V oficiální dokumentaci[5] je ostatně zmíněno, že framework *Flask* nachází své využití zejména v mikroslužbách (*microservices*). Kód základní webové aplikace využívající tohoto aplikačního rámce je zobrazen ve výpisu 2.5. Ukázková aplikace poslouchá na portu 80 a vytváří odpovědi ve formátu JSON. Pro objekty jiné než slovníky je možné importovat metodu `jsonify()` a použít ji pro sestavení odpovědi ve zmíněném formátu.

Výpis 2.5: Jednoduchá aplikace využívající aplikační rámec *Flask*.

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5
6  @app.route("/hello")
7  def get_greeting():
8    return {"greeting": "Hello World!"}
9
10 if __name__ == "__main__":
11   app.run(host="0.0.0.0", port=80)
```

Můžeme pozorovat, že definování webové aplikace pomocí *Flask* je velmi jednoduché. Samotné použití uživatelského kontejneru a aplikačního rámce *Flask* v diplomové práci bude podrobněji popsáno v kapitole 3.2.

2.5.2 Knihovna *scikit-learn*

Knihovna *scikit-learn* je volně dostupným nástrojem nabízejícím řadu modelů strojového učení. Tato knihovna je vhodná i pro méně zkušené uživatele, jelikož poskytuje obsáhlou dokumentaci s přiloženými příklady. Zároveň poskytuje velké množství typů modelů strojového učení a nabízí možnost analýzy výsledků. Následující výpis kódu 2.6 (klasifikace obrazů s číslicemi 0 a 1) ilustruje použití knihovny.

[5] `https://flask.palletsprojects.com/en/2.1.x/foreword/`

Výpis 2.6: Ukázka trénování modelu pomocí knihovny *scikit-learn*.

```
1  zeros = []
2  ones = []
3  NUMBER_OF_TESTING_IMAGES = 10
4
5  for filename in glob('/zeros/*.jpg'):
6      image = Image.open(filename)
7      zeros.append(image.getdata())
8  for filename in glob('/ones/*.jpg'):
9      image = Image.open(filename)
10     ones.append(image.getdata())
11
12 zeros = np.array(zeros)
13 ones = np.array(ones)
14
15 number_of_zeros = len(zeros)
16 number_of_ones = len(ones)
17
18 zeros_training = zeros[:number_of_zeros - NUMBER_OF_TESTING_IMAGES]
19 zeros_testing = zeros[-NUMBER_OF_TESTING_IMAGES:]
20
21 ones_training = ones[:number_of_ones - NUMBER_OF_TESTING_IMAGES]
22 ones_testing = ones[-NUMBER_OF_TESTING_IMAGES:]
23
24 labels_testing_zeros = np.zeros(len(zeros_testing))
25 labels_testing_ones = np.ones(len(ones_testing))
26
27 features = np.concatenate((zeros_training, ones_training), axis=0)
28 training_labels = np.concatenate((labels_training_zeros, labels_training_ones))
29
30 classifier = svm.SVC()
31 classifier.fit(features, training_labels)
32
33 accuracy_zeros = accuracy_score(labels_testing_zeros,
34                                 classifier.predict(zeros_testing))
35 accuracy_ones = accuracy_score(labels_testing_ones,
36                                classifier.predict(ones_testing))
```

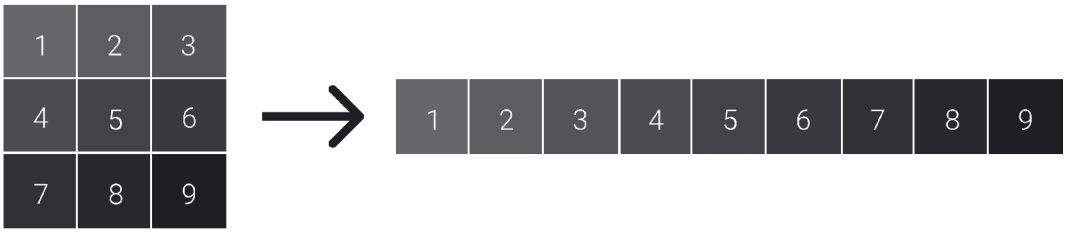
Na začátku dojde k načtení dat z příslušných adresářů. Dále se provede rozdělení datové množiny na trénovací, respektive testovací část. Přiřazení do jednotlivé třídy (tedy zda se jedná o obraz číslice 0 či 1) lze provést příkazy nacházejícími se na řádcích 25 a 26. Dále je nutné vytvořit trénovací vektor. Tohoto jde dosáhnout konkatenací jednotlivých vektorů s příslušnými trénovacími daty. Tento krok se provede i pro vektor přiřazující data do jednotlivých tříd. Pomocí příkazu `svm.SVC()` dojde k vytvoření modelu *Support Vector Machines*. Pomocí metody `fit()` dojde ke spuštění trénování modelu. Nad natrénovaným modelem je poté možnost zavolat metodu `predict()`, která provede predikci nad testovacími daty.

Důležitým bodem je sestavení vstupních dat o správném formátu, kdy jsou příznaky pro jednotlivý vstup uspořádány do vektoru. Tyto jednotlivé vektory jsou poté seskládány do výsledného příznakového vektoru (reálně se však jedná o 2D pole). Klasifikační značky pak vytváří samostatný vektor, jehož délka musí být rovna počtu řádků příznakového vektoru.

Výpis 2.7: Vektor příznaků a vektor klasifikačních značek.

```
features = [[x1, x2, x3], [y1, y2, y3], [z1, z2, z3], ...]
labels   = [    0,          0,          1,     ...]
```

Pokud jsou vstupní data tvořena obrazy, je nutné 2D pole pixelů transformovat do jednorozměrného pole, které pak vytváří příznakový vektor. Tuto operaci naznačuje schéma 2.4.



Obr. 2.4: Sestavení příznakového vektoru.

3 Implementace aplikace

Nyní se dostáváme k popisu samotné implementace aplikace. Po vzoru TDD (*test-driven development*) bude nejprve rozebrán způsob implementace testů a bude prakticky osvětlen princip automatického spouštění testů na platformě *GitHub*. Kapitola 3.2 pak podává podrobnější popis hlavní funkcionality aplikace, a to popis zejména migrace dat v rámci aplikace a následného trénování a validace modelů strojového učení. Bude zde také popsán způsob asynchronního spuštění běhu modelu a následná vizualizace výsledků. Implementací autorizace uživatelů pomocí JWT tokenů (viz 1.2.2) se pak zabývá kapitola 3.3. Text v kapitole 3.4 popisuje nasazení aplikace do produkčního prostředí s pomocí kontejnerizace. Poslední kapitola 3.6 v rámci diplomové práce je pak meditací nad sérií možných rozšíření aplikace.

3.1 Testování aplikace

Pro každou funkcionalitu softwarového projektu by měla být definována sada testů. Je zároveň žádoucí, aby aplikační testy byly plně automatizované. Při vzrůstající komplexitě programu je časově velmi náročné manuálně ověřit funkčnost programu při každé provedené změně. Odhalování chyb v aplikacích je poté tedy zejména doménou automatizovaných testů. Jelikož jsou v práci použity vyspělé aplikační rámce, je pro ně nabízena celá řada knihoven pro psaní testů a ověřování správné funkce aplikačního kódu.

3.1.1 Testování klientské části aplikace

Pro vysvětlení způsobu testování a způsobu použití testovacích knihoven použijeme vzorový test, jehož implementace se nachází v následujícím výpisu 3.1.

Výpis 3.1: Ukázka implementace testu pro klientskou část aplikace.

```
1  describe('Rendering a login form', () => {
2    test('When an exception occurs then an error message is shown', async() => {
3      const message = 'Server side error occurred!';
4      const error = {response: {data: {message: message}}};
5      jest.spyOn(LoginService, 'login').mockRejectedValue(error);
6
7      const history = createMemoryHistory();
8      history.replace = jest.fn();
9
10     await act(async () => {
11         render(<Router history={history}><LoginPage/></Router>);
12     });
13
14     const email = screen.getByPlaceholderText(/email/i);
```

```
15      const password = screen.getByPlaceholderText(/password/i);
16      const loginButton = screen.getByText(/login/i);
17
18      await act(async () => {
19          fireEvent.change(email, {target: {value: 'user@user.com'}});
20          fireEvent.change(password, {target: {value: 'Thisisarandompassword_999'}});
21          fireEvent.click(loginButton);
22      });
23
24      const errorMessage = screen.getByText(message);
25      expect(errorMessage).toBeInTheDocument();
26    });
27  });
```

Sadu testů, které validují podobnou funkcionalitu, je možné sdružit pomocí příkazu `describe()`. Samotný test je popsán deklarací ve formátu *„při provedení akce A nastane výsledek B"*. Jednou z knihoven umožňující nahrazení reálných implementací metod je knihovna *Jest*. Toto nahrazení implementace metody či objektu je v anglické literatuře souhrně nazváno jako tzv. *mocking*. Existuje také řada odchylek, kdy objekt není nahrazen prázdnou implementací, ale je zabalen do jiného testovacího objektu, který umožní sledovat akce provedené nad tímto objektem a v testech tak kontrolovat, zda vybrané akce byly provedeny. Pro účely diplomové práce však budeme uvažovat jen souhrný název *mocking*.

Pomocí knihovny *Jest* tedy na řádku 5 ve výpisu 3.1 nahradíme zvolené metody. Tento konstrukt je zejména vhodný pro náhradu metod, které uvnitř zasílají dotazy k reálným webovým serverům, a jež vrací tzv. *Promise* (reprezentuje operaci, jež nebyla ještě dokončena, typické u asynchronního volání). Pokud tedy potřebujeme navodit situaci, kdy webový server volaný v rámci metody vrací odpověď (ať už JSON objekt s hodnotami či chybovým hlášením), aniž by byl webový server reálně dostupný, je možné použít metody `mockResolvedValue()` nebo `mockRejectedValue()` s předáním zvolených hodnot.

Jelikož je v rámci komponenty `LoginPage` použito přesměrování na jinou komponentu, je nutné zabalit `LoginPage` do komponenty `Router`, která spravuje přístup k jednotlivým komponentám. Metoda `act()` pak slouží k vykonání specifické akce či série akcí. Na řádku 11 je v jejím těle voláno vykreslení komponenty `LoginPage`. Počínaje řádkem 19 je simulována interakce uživatele s aplikací. Nejprve je vyplněno pole pro emailovou adresu, dále pole pro heslo a konečným krokem je stisknutí tlačítka pro přihlašení uživatele. Je důležité si povšimnout, že se vždy jedná o asynchronní kód, jelikož je nutné v rámci exekuce testu počkat na změnu stavu `LoginPage` komponenty a na její překreslení, které reflektuje současný stav. Instance

`screen` nabízená knihovnou *React Testing Library* reprezentuje vykreslenou obrazovku v podobě HTML značek. Nad touto instancí je pak možné volat řadu metod pro získání specifické značky. Toho je mimo jiné využito na řádku 25, kdy je získána značka obsahující chybové hlášení a poté je pomocí metody `expect()`, respektive metody `toBeInTheDocument()`, zjištěno, zda bylo chybové hlášení zobrazeno uživateli.

3.1.2 Testování funkcionality webového serveru

Samotný framework *Spring* nabízí sadu nástrojů pro testování aplikace. V kódu nacházejícím se ve výpisu 3.2 je použita anotace `@SpringBootTest`, díky které se *Spring* postará o sestrojení závislosti. Testovací třídy označené touto anotací jsou tedy vhodné pro integrační testování. Anotace `@AutoConfigureMockMvc` pak umožní spustit testy, aniž by musel být nastartován server, a je tak možné dosáhnout rychlejšího běhu testů. Zároveň však je simulováno prostředí shodné s produkčním prostředím[1]. Pomocí `mockMvc` je pak možné validovat funkci API rozhraní (například status HTTP odpovědi či její obsah).

Výpis 3.2: Testování API rozhraní.

```
1  @SpringBootTest
2  @AutoConfigureMockMvc
3  public class AuthenticationControllerValidationTest
4  {
5    public static final String INVALID_EMAIL = "user@@@email.com";
6    public static final String PASSWORD = "TheStrongPassword_999";
7
8    @Autowired
9    private MockMvc mockMvc;
10
11   @Test
12   public void whenInvalidEmailIsPassed_thenUserWillNotBeRegistered()
13   {
14       JSONObject json = new JSONObject();
15       json.put("username", INVALID_EMAIL);
16       json.put("password", PASSWORD);
17
18       mockMvc.perform(post("/api/auth/register")
19                       .contentType(MediaType.APPLICATION_JSON)
20                       .content(json.toString()))
21               .andExpect(status().is4xxClientError())
22               .andExpect(content().string("Email is not in a valid format!"));
23   }
24 }
```

V rámci výpisu 3.3 je ukázan příklad jednotkového testu. Pro oddělené testování jednotky je možné využít knihovnu *Mockito*. Jak již z názvu vyplývá, tato knihovna

[1] `https://spring.io/guides/gs/testing-web/`

umožňuje realizovat zmíněný *mocking*. Pomocí konstruktu `when().thenReturn()` je při zavolání nahrazované metody vrácena zvolená hodnota. Všimněme si, že *mocking* nám do jisté míry pomáhá se psaním takového kódu, kdy je dodržován princip *Dependency inversion*. Je nutné předat do konstruktoru třídy či při volání metody všechny její závislosti. Pokud je nějaká závislost v rámci třídy interně inicializována, je pak složité danou třídu otestovat, jelikož závislost není možné v testech nahradit námi zvolenou implementací pomocí *mocking* principu. Metoda `verify()` slouží ke kontrole, že zvolená metoda nad předanou instancí byla během exekuce testu skutečně zavolána.

Výpis 3.3: Ukázka jednotkového testu.

```
1  public class AuthorizationTokenFilterTest
2  {
3    public static final String RANDOM_JWT = "randomJWT";
4
5    private JsonWebTokenUtility jsonWebTokenUtility;
6    private AuthorizationTokenFilter authorizationTokenFilter;
7
8    @BeforeEach
9    public void before()
10   {
11       jsonWebTokenUtility = Mockito.mock(JsonWebTokenUtility.class);
12       authorizationTokenFilter = new AuthorizationTokenFilter(jsonWebTokenUtility);
13   }
14
15   @Test
16   public void whenJWTTokenIsPresentInHeader_thenTokenIsValidated()
17   {
18       HttpServletRequest request = Mockito.mock(HttpServletRequest.class);
19       HttpServletResponse response = Mockito.mock(HttpServletResponse.class);
20       FilterChain filterChain = Mockito.mock(FilterChain.class);
21       Mockito.when(jsonWebTokenUtility.parseJwt(request)).thenReturn(RANDOM_JWT);
22
23       authorizationTokenFilter.doFilterInternal(request, response, filterChain);
24
25       Mockito.verify(jsonWebTokenUtility).validateJwtToken(RANDOM_JWT);
26   }
27 }
```

3.1.3 Automatické spuštění testů

Pro průběžné testování aplikace byly využity *Github* akce zmíněné v kapitole 1.1.1. Definice akce pro spuštění testů pro klientskou webovou aplikaci je zobrazena ve výpisu 3.4 Další akce používané při vývoji aplikace je možné najít v repozitáři na následujícím odkazu[2].

[2] `https://github.com/JanSvoboda6/ml_runner/actions`

Výpis 3.4: *GitHub* akce pro spuštění sady testů.

```
name: React tests

on:
  push:
    branches: [ main ]

jobs:
  test:
    name: react-tests
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Node
        uses: actions/setup-node@v2
        with:
          cache: 'npm'
          cache-dependency-path: react/ml-runner/package-lock.json
      - name: Tests
        env:
          CI: false
        run: |
          cd react/ml-runner
          npm ci
          npm run build --if-present
          npm test
```

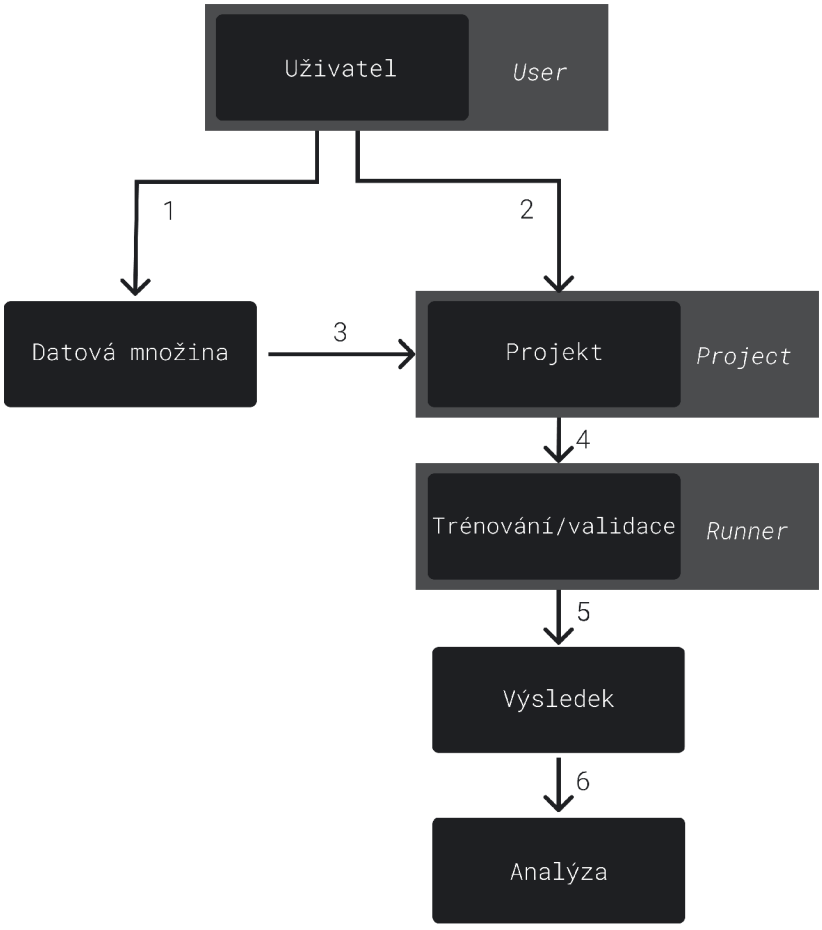
Definice akce nepotřebuje hlubší komentář. Nejprve je akce pojmenována, dále je zvoleno, kdy má být akce spuštěna (tedy při nahrání změn do repozitáře nad hlavní větví) a pak jsou definovány jednotlivé kroky exekuce. Ke spuštění akce je využito linuxové distribuce *Ubuntu* s připraveným prostředím *Node*. Součástí příkazu `run` je změna adresáře (je nutné zvolit adresář v rámci repozitáře, ve kterém se nachází kód klientské aplikace), instalace závislostí pomocí `npm ci`, sestavení aplikace a konečné spuštění testů realizované příkazem `npm test`.

3.2 Hlavní funkcionalita

V této kapitole bude věnována pozornost samotnému spuštění procesu učení. Po přihlášení (viz kapitola 3.3) je uživateli přidělen kontejner s běhovým prostředím *Python* a nainstalovanou knihovnou *scikit-learn*. Uživateli je poté umožněno vytvořit nový projekt. Avšak ještě předtím je nutné nahrát samotnou datovou množinu. Data z datové množiny se pak ukládají přímo na přidělený kontejner. Data jsou z uživatelského prostředí namapována na adresář, který se nachází v kontejneru. Uživatel může s daty manipulovat na adrese `/datasets`, na které se nachází jednoduchý průzkumník souborů, který je synchronizován s kontejnerovým adresářem.

Nyní je možné postoupit k vytvoření projektu. Projekt zaštiťuje zejména výběr modelu strojového učení a příslušných adresářů, ve kterých se nacházejí data jednotlivých tříd. Po vytvoření projektu je uživatel přesměrován na hlavní stránku, kde je zobrazen seznam všech již vytvořených projektů. Uživateli je nyní umožněno spustit proces učení u zvoleného projektu. Po otevření formuláře je nutné zadat parametry modelu pro příslušný běh. Po zaslání formuláře je asynchronně spuštěno trénování a validace zvoleného modelu se zadanými parametry, a to v kontejneru, který je přidělen danému uživateli.

Aplikace nyní monitoruje samotný běh. Po doběhnutí jsou uživateli zobrazeny výsledky. Aplikace také poskytuje analýzu výsledků jednotlivých běhů a současně provádí jejich agregaci. Snímky obrazovky z jednotlivých kroků se nachází v příloze A. Je také poskytnut odkaz na již zmíněné webové úložiště[3], na kterém se nachází nahrávky se současnou podobou uživatelského prostředí. Diagram 3.1 nabízí vizualizaci jednotlivých procesů v aplikaci.



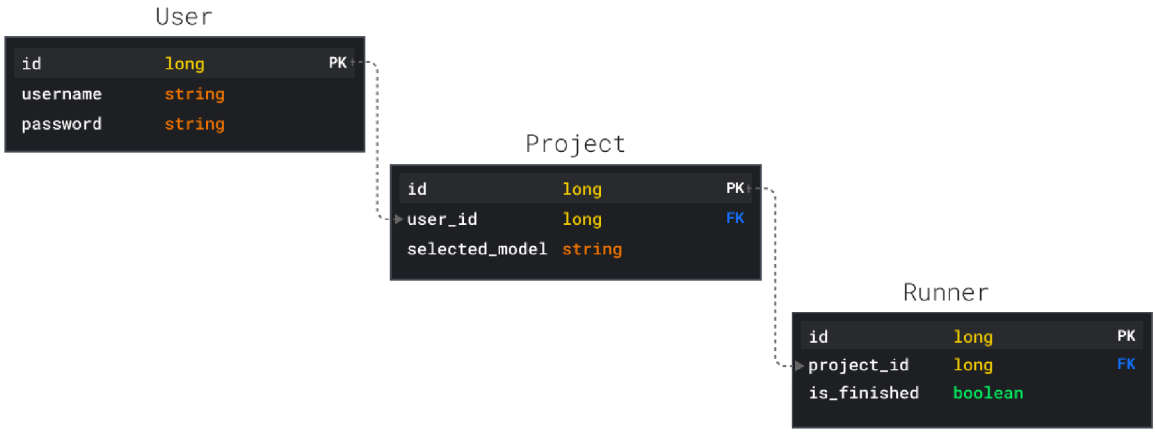
Obr. 3.1: Diagram interakce s aplikací.

[3] https://drive.google.com/drive/folders/17BWR5fbwnzHdIt4Ofe438MTBGxZYUjsl

V šedých obdélnících je kurzívou uveden název tříd, které v aplikaci vyjadřují pojmy uvedené v rámci černých obdélníků. Diagram je tvořen následujícími kroky:

1. nahrání souborů datové množiny,
2. přechod na formulář vytvářející projekt (`Project`),
3. výběr příslušných adresářů tříd datové množiny,
4. spuštění procesu učení (`Runner`),
5. získání a zobrazení výsledků,
6. následná analýza.

Jednotlivé objekty v rámci databáze mezi sebou vytvářejí vztah, který je uveden na zjednodušeném schématu 3.2. Uživateli (objekt `User`) je tedy umožněno vytvořit více projektů (objekt `Project`). Pod každým projektem se pak nachází množství jednotlivých běhů, které představují jednu exekuci procesu učení se zvolenými parametry (objekt `Runner`). Zkratky PK – *primary key*, respektive FK – *foreign key*, vyjadřují pojmy primárního, respektive cizího klíče.



Obr. 3.2: Databázové schéma.

3.2.1 Řízení asynchronního spuštění trénování a validace

Pojďme se nyní podrobněji podívat na to, jakým způsobem se projekt (respektive `Runner`) v rámci aplikace spouští, a jak jsou následně získávány výsledky. Po zaslání formuláře (z klientské strany) se zvolenými parametry daného běhu je na webovém serveru exekuováno metoda `runProject()` v rámci třídy `RunnerController`, která je označena anotací `@RestController`. Dochází tady tedy k mapování obsahu HTTP dotazu (jenž je naplněn hodnotami ze zmíněného formuláře) na parametry metody `runProject()`.

Výpis 3.5: Třída `RunnerController`.

```
@RestController
@RequestMapping("/api/project/runner")
public class RunnerController
{
  //Deklarace a inicializace závislostí v konstruktoru

  @PostMapping("/run")
  public ResponseEntity<?> runProject(
    @RequestHeader(name = "Authorization") String token,
    @Valid @RequestBody RunRequest request)
  {
    runnerService.runProject(
      request.getHyperParameters(),
      validator.validateProject(request.getProjectId()),
      validator.validateContainerEntity(request.getContainerEntity())
    );

    return ResponseEntity.ok().body("Project is running!");
  }
}
```

Obsah dotazu je také nutné zvalidovat. *Spring* k tomuto účelu nabízí anotaci `@Valid`. Interní parametry validovaných tříd (v našem případě `RunRequest`) je pak možné označit například anotací `@NotBlank`, která *Spring* instruuje, že daná proměnná nemůže být inicializována prázdnou hodnotou. Pro pokročilejší validaci, kdy je například nutná interakce s databází, musí již vývojář přijít s vlastním konceptem. V naší situaci je použita instance `validator` k ověření, že projekt, jehož identifikátor je předán v rámci HTTP dotazu, skutečně existuje.

Třída `RunnerServiceImpl` se pak stará o vytvoření objektu `Runner` symbolizujícího jedno spuštění projektu se zvolenými parametry. Můžeme si povšimnout, že byla implementována i tzv. *scheduling* funkcionalita. Pokud uživatel již nějaký běh spustil a tento běh je stále ve stavu aktivní exekuce, tak je další běh umístěn do fronty (`RunnerQueue`) a po dokončení prvního běhu je započato jeho reálné spuštění. Toto plánování běhů pak vede k menší zatíženosti uživatelského kontejneru. Při spuštění několika běhu současně by se snadno mohly vyčerpat zdroje přidělené kontejneru.

Výpis 3.6: Metoda `runProject()`.

```
@Service
public class RunnerServiceImpl implements RunnerService
{
  //Deklarace a inicializace závislostí v konstruktoru

  @Override
  public void runProject(
```

```
8       List<HyperParameter> hyperParameters,
9       Project project,
10      ContainerEntity containerEntity
11    )
12    {
13        Runner runner = createRunner(project, hyperParameters);
14        if (isAnyRunnerRunning(containerEntity.getUser()))
15        {
16            runner.setStatus(RunnerStatus.SCHEDULED);
17            runnerRepository.save(runner);
18            runnerQueueRepository.save(
19                new RunnerQueueEntity(containerEntity.getUser(), runner));
20            return;
21        } else
22        {
23            runner.setStatus(RunnerStatus.INITIAL);
24        }
25        projectRunner.run(runnerRepository.save(runner), containerEntity);
26    }
27  }
```

O odstranění z fronty `RunnerQueue` se stará třída `RunnerScheduler`. Její implementace je uvedena ve výpisu 3.7. Je iterováno přes všechny uživatele aplikace. Pokud je v současné chvíli pro daného uživatele již spuštěn jakýkoliv běh, tak nebude uskutečněno spuštění jiného běhu. Pokud však v současnosti není exekuován žádný další běh, tak je nahlédnuto do fronty `RunnerQueue` pomocí repozitáře `queueRepository`. Pokud fronta není prázdná, tak se z ní odebere nejdéle čekající položka (`Runner`) a dojde k jejímu spuštění.

Výpis 3.7: Třída `RunnerScheduler`.

```
1   @Service
2   public class RunnerScheduler
3   {
4     @Scheduled(fixedRate = 10, timeUnit = TimeUnit.SECONDS)
5     @Transactional
6     public void scheduleRunning()
7     {
8       List<User> users = userRepository.findAll();
9       for(User user: users)
10      {
11        if (runnerService.isAnyRunnerRunning(user))
12        {
13          break;
14        }
15
16        List<Runner> runnersToBeStarted = new ArrayList<>();
17        List<RunnerQueueEntity> runnerQueueEntities = queueRepository.getAll(user);
18        runnerQueueEntities.forEach(
19          runnerQueueEntity -> runnersToBeStarted.add(runnerQueueEntity.getRunner())
20        );
21
```

```
22        if(!runnersToBeStarted.isEmpty())
23        {
24          runnersToBeStarted.sort(Comparator.comparing(Runner::getTimestamp));
25          Runner runner = runnersToBeStarted.get(0);
26          runner.setStatus(RunnerStatus.INITIAL);
27          queueRepository.deleteByRunnerId(runner.getId());
28          projectRunner.run(runner, getContainerEntityOfUser(user));
29        }
30      }
31    }
32  }
```

Na straně webového serveru *Spring* je tedy ještě zavolána metoda `run()`, která je poskytnuta instancí `projectRunner` třídy `ContainerProjectRunner`.

Výpis 3.8: Třída `ConainerProjectRunner`.

```
1  @Service
2  public class ContainerProjectRunner implements ProjectRunner
3  {
4    @Override
5    public void run(Runner runner, ContainerEntity container)
6    {
7      HttpEntity<String> httpEntity = mapRunnerToHttpEntity(runner);
8
9      requestMaker.makePostRequest(
10         container.getConnectionString(),
11         RequestMethod.RUN_PROJECT, httpEntity
12     );
13   }
14 }
```

Zde můžeme jasně pozorovat, že se webový server chová jako prostředník mezi webovým klientem a samotným kontejnerem, kde bude spuštěno učení modelu. Na řádku 7 ve výpisu 3.8 jsou nejprve namapovány parametry objektu `Runner` na tělo HTTP dotazu. Dále je získána URL adresa, na které je možné kontaktovat kontejner, respektive server *Flask* (viz dále). Takto jsou kontejneru předány všechny potřebné parametry a kontejner je zároveň požádán o to, aby došlo ke spuštění proces učení.

3.2.2 Spuštění skriptu v rámci uživatelského kontejneru

Nyní se dostáváme na stranu samotného uživatelského kontejneru. Pro připomenutí – v kontejneru běží webový server realizovaný pomocí aplikačního rámce *Flask*. HTTP dotaz od serveru *Spring* je zachycen aplikačním rozhraním v podobě metody `run_project()` nacházející se ve výpisu 3.9.

Výpis 3.9: Aplikační rozhraní serveru *Flask* s metodou `run_project()`.

```
@app.route('/runproject', methods=['POST'])
def run_project():
  runner = request.get_json()
  id = str(runner['runnerId'])

  Path('runners_info/' + id).mkdir(parents=True, exist_ok=True)

  with open('runners_info/' + id) + '/configuration.json', 'w') as json_file:
      json.dump(runner, json_file)

  with open('runners_info/' + id) + '/status.txt', 'w') as status_file:
      status_file.write('INITIAL' + '\n')

  log_file = open('runners_info/' + id) + '/log.txt', 'w')

  if runner['selectedModel'] == 'Support Vector Machines':
      subprocess.Popen(['nohup', 'python3', 'models/svm.py', id],
                       stdout=log_file,
                       stderr=log_file,
                       preexec_fn=os.setpgrp)
  #Odpověd webovému serveru
```

Po přečtení předávaných dat z dotazu (instance `runner`) je vytvořen adresář identifikovaný pomocí jedinečného identifikátoru běhu. Tento adresář slouží k ukládání souborů, se kterými *Python* skript spuštěný na pozadí pracuje. Do souboru s názvem `configuration.json` jsou nahrány konfigurační parametry z instance `runner` ve formátu JSON, se kterými je pak nakonfigurován model strojového učení. Zároveň jsou zde vypsány i další údaje. Jedná se například o cesty k adresářům, kde se nachází data jednotlivých tříd.

Výpis 3.10: Zkrácená verze konfiguračního souboru `configuration.json`.

```
"runnerId": 1,
"selectedModel": "Support Vector Machines",
"classificationLabels": [
    {
        "folderPath": "first/",
        "labelName": "First"
    },
    {
        "folderPath": "second/",
        "labelName": "Second"
    }],
"hyperParameters": [
    {
        "name": "gamma",
        "value": "10"
    },
    {
        "name": "c",
        "value": "1"
    }]
```

Do souboru `status.txt` je pak při změně stavu běhu přidán aktuální stav. Pomocí tohoto souboru je možné průběžně kontrolovat stav běhu a zjistit, zda je běh úspěšně ukončen či zda došlo při běhu k chybovému stavu. Posledním použitým souborem je soubor `log.txt`, který slouží k výpisu logů z následně spuštěného skriptu, jenž se již stará o exekuci procesu trénování a validace modelu strojového učení.

Nyní může dojít k spuštění samotného skriptu, a to podle zvoleného typu modelu (řádek 16 ve výpisu 3.9). Skript je spuštěn pomocí příkazu `nohup`, tedy tak, aby běžel na pozadí. Jakékoliv výstupy skriptu jsou přesměrovány a zapsány do zmíněného logovacího souboru `log.txt`.

3.2.3 Skript realizující proces trénování a validace modelu

Skript zodpovídá za provedení několika úkonů. Nejprve musí lokalizovat konfigurační soubor `configuration.json`. To vykoná za pomoci znalosti adresářové struktury a z předaného identifikátoru běhu (proměnná `id` na řádku 17 v předcházejícím výpisu 3.9). Obsah konfiguračního souboru načte do paměti a zjistí tak, kde může vyhledat soubory z datové množiny. Dále nahrané vzorky rozdělí do trénovací a verifikační množiny. Spustí proces trénování a následně proces validace. Do logovací souboru `log.txt`, který si pak uživatel může zobrazit (viz dále), se zapíše veškerý obsah, který je vytisknut pomocí příkazu `print()`. Tímto způsobem je tak možné propagovat výsledky či další informace. Skript také průběžně zapisuje změnu stavu běhu do souboru `status.txt` pomocí metody `inform_on_status_change()`. Ve výpisu není zobrazena přesná implementace skriptu z důvodu velkého rozsahu souboru, je zde naznačen ideový příklad pro trénování a validaci modelu SVM.

Výpis 3.11: Skript určený pro spuštění procesu učení.

```
1  def run(runner_id):
2    with open('runners_info/' + runner_id + '/configuration.json', 'r') as json_file:
3        configuration = json.load(json_file)
4
5    gamma = float(configuration['hyperParameters']['gamma'])
6    c = float(configuration['hyperParameters']['c'])
7    classification_labels = configuration['classificationLabels']
8
9    inform_on_status_change(runner_id, Status.LOADING_DATA)
10
11   samples = []
12   labels = []
13   for idx, label in enumerate(classification_labels):
14       for file in glob(label['folderPath'] + '*.npy'):
15           sample = np.load(file)
16           samples.append(sample)
17           labels.append(idx)
18
19   samples = np.array(samples)
20   labels = np.array(labels)
```

```
21    training_samples,testing_samples,training_labels,testing_labels = split(samples,labels)
22
23    inform_on_status_change(runner_id, Status.TRAINING)
24    classifier = svm.SVC(verbose=0, gamma=gamma, C=c)
25    classifier.fit(training_samples, training_labels)
26
27    inform_on_status_change(runner_id, Status.PREDICTING)
28    predicted_labels = classifier.predict(testing_samples)
29    accuracy = accuracy_score(testing_labels, predicted_labels)
30
31    print('ACCURACY: ' + str(accuracy))
32    inform_on_status_change(runner_id, Status.FINISHED)
33
34
35  if __name__ == "__main__":
36    try:
37      runner_id = str(sys.argv[1])
38      run(runner_id)
39    except:
40      traceback.print_exc()
41      inform_on_status_change(runner_id, Status.FAILED)
```

3.2.4 Rekurentní dotazování ze strany klientské aplikace

Na webovém klientu je po odstartování učení modelu spuštěno rekurentní dotazování na to, zda model proces trénování, respektive validace, již dokončil. Je-li tomu tak, je odeslán dotaz na výsledek procesu validace. Situace je popsána na následujícím výpisu kódu 3.12. Pro zjednodušení zde není uvažováno ošetření chybového stavu, kdy se běh nachází ve stavu `'FAILED'`.

Výpis 3.12: Rekurentní dotazování na výsledek učení.

```
1   function Runner(props: RunnerProps)
2   {
3     const [status, setStatus] = useState("INITIAL");
4     const [accuracy, setAccuracy] = useState(0);
5
6     useEffect(() =>
7     {
8       RunnerService.getStatus(props.runnerId)
9         .then((response) =>
10          {
11            setStatus(response.data.status);
12            response.data.status === "FINISHED"? getResult() : startRecurrentRequests();
13          });
14    }, [])
15
16    const getResult = () =>
17    {
18      RunnerService.getResult(props.runnerId)
19        .then(response =>
20          {
21            setAccuracy(response.data.accuracy);
22          })
23    }
24
25    const startRecurrentRequests = () =>
```

```
26    {
27      intervalId = setInterval(isRunnerInEndState, FIVE_SECONDS);
28    }
29
30    const isRunnerInEndState = () =>
31    {
32      RunnerService.getStatus(props.runnerId)
33        .then((response) =>
34          {
35            setStatus(response.data.status);
36            if(response.data.status == "FINISHED")
37            {
38              getResult();
39              clearInterval(intervalId);
40            }
41          }
42        );
43    }
44    //Následuje vykreslení výsledků...
45  }
```

3.2.5 Ukončení běhu a propagace výsledku

Webový server opět předává dotaz na stav běhu až ke kontejneru s prostředím *Python*. Propagování informace o ukončeném běhu je řešeno poměrně primitivně. Skript, který spouští proces učení zvoleného modelu, zapíše po dokončené fázi verifikace do souboru `status.txt` řetězec `'FINISHED'`. Metoda `get_status()` pak slouží k tomu, aby podala informaci o jednotlivých chronologicky seřazených stavech běhu v podobě JSON objektu s položkou `chronologicalStatuses`. Pokud je zde řetězec `'FINISHED'` přítomen, znamená to, že proces učení je dokončen, a je možné posléze předat výsledky.

Výpis 3.13: Metoda `get_status()`.

```
1  @app.route('/project/runner/status', methods=['POST'])
2  def get_status():
3    runner_id = request.get_json()['runnerId']
4    statuses = []
5    with open('runners_info/' + str(runner_id) + '/status.txt', 'r') as status_file:
6      for line in status_file:
7        statuses.append(line.replace('\n', ''))
8    return jsonify({'chronologicalStatuses': statuses})
```

3.2.6 Analýza a vizualizace výsledků

Analýza výsledků probíhá na dvou úrovních. Prvotně je možné zobrazit výsledky pro jednotlivý běh. Po úspěšném doběhnutí je uživateli na hlavní tabuli (viz příloha A) zobrazen číselný údaj vyjadřující úspěšnost validace. Číselný údaj je zároveň odkazem, který po kliknutí uživatele přesměruje na stránku s výpisem logů (již zmíněný soubor `log.txt`), ve kterém se nachází základní metriky. Je zde uvedena úspěšnost,

dále také tzv. *classification report* (zpráva o klasifikaci), jež je generována pomocí knihovny *scikit-learn*. Výpis dále poskytuje matici záměn v textové podobě. Při rozšíření aplikace by bylo možné do výpisu přidat libovolné metriky, které jsou v textové podobě snadno interpretovatelné a není potřeba je vizualizovat.

Druhá úroveň pak zprostředkovává agregaci výsledků nad jednotlivými běhy v rámci projektu. Po kliknutí na „*Analysis*“ v pravém rohu na hlavní tabuli je uživatel přesměrován na stránku poskytující vizualizaci. Všechny vizualizační komponenty jsou zobrazeny v příloze A. Pro potřeby vizualizace byla použita knihovna *visx* nabízející různé grafické elementy, pomocí kterých je sestavován výsledný graf či jiná vizualizační komponenta. Hlavní komponentou na stránce „*Analysis*“ je teplotní mapa (*heat map*) vykreslující hodnotu úspěšnosti v závislosti na uživatelem vybraných parametrech modelu. Mapa je plně dynamická, při zvolení nové kombinace parametrů je automaticky překreslena. Uživatel může pomocí teplotní mapy analyzovat úspěšnost běhu na základě kombinace parametrů modelu a hledat mezi parametry korelaci. Pro plné pochopení funkcionality teplotní mapy autor opět doporučuje přehrát příslušné videoukázky dostupné na následujícím odkazu[4].

Uživateli je zobrazen také graf vykreslující průběh úspěšnosti v čase. Na horizontální ose jsou znázorněny identifikátory jednotlivých běhů. Na vertikální ose je vynesena hodnota úspěšnosti. Jelikož implementace teplotní mapy je poměrně rozsáhlá, bude v textu diplomové práce stručně popsán jen příklad značně zjednodušené implementace grafu (výpis 3.14). Mimo další parametry je komponentě `XYChart` nutné předat údaje o rozměrech. Vnitřní komponenta `AnimatedGrid` pak tyto rozměry přebírá taktéž. Předpona `Animated` knihovnu *visx* instruuje, aby při počátečním zobrazení grafu jeho komponenty vykreslila pomocí animace přechodu. Parametry `colums` a `rows` určují, zda mají být v rámci mřížky vykresleny sloupce respektive řádky. Komponenta `AnimatedAxis` slouží k vykreslení osy, kterou je zároveň možné pojmenovat pomocí parametru `label`. O vykreslení zvolených dat se stará komponenta `AreaSeries`, která zároveň barevně vyplní plochu pod křivkou. Při přejetí kurzoru myši nad grafem je uživateli zobrazen tzv. *tooltip*, tedy okno podávající informace o právě zvoleném bodu na grafu.

Výpis 3.14: Komponenta zprostředkovávající dvourozměrný graf.

```
1 <XYChart height={500} width={1000}>
2   <AnimatedGrid columns={true} rows={true}}/>
3   <AnimatedAxis orientation="bottom" label={"Id [-]"}/>
4   <AnimatedAxis orientation="left" label={"Accuracy [%]"}/>
5   <AreaSeries dataKey="Accuracy" data={props.data}/>
6   <Tooltip/>
7 </XYChart>
```

[4] `https://drive.google.com/drive/folders/17BWR5fbwnzHdIt4Ofe438MTBGxZYUjsl`

3.3 Autorizace uživatelů

V této kapitole bude popsán průběh autorizace uživatelů, a zejména pak použití JWT tokenu při jakékoliv interakci webového klienta s webovým serverem. Po registraci je uživateli vytvořen účet, pomocí kterého se může přihlásit. Při tomto přihlášení je zaslán dotaz na webový server a dojde k exekuci metody `authenticateUser()`. Po provedení potřebného ověření zadaného uživatelského jména a hesla, je vygenerován jedinečný JWT token, který obsahuje informace o uživateli, na jejichž základě lze uživatele identifikovat (instance `authentication`). Metoda je ukončena zasláním JWT tokenu uživateli.

Výpis 3.15: Generování JWT tokenu při přihlášení uživatele.

```
@PostMapping("/login")
public ResponseEntity<?> authenticateUser(@RequestBody LoginRequest request)
{
  Authentication authentication;
  try
  {
    authentication = authenticationManager
                      .authenticate(request.getUsername(), request.getPassword());
  } catch (AuthenticationException exception)
  {
    //Ošetření chybové stavu při zadání neplatného jména či hesla
  }
  String jwtToken = jsonWebTokenUtility.generateJwtToken(authentication);
  return ResponseEntity.ok(new JwtResponse(jwtToken);
}
```

Uživatel, respektive klientská aplikace, si tento vygenerovaný token uschová do paměti v podobě `localStorage`. Pokud klient bude požadovat přístup k jakýmkoliv datům na webovém serveru, musí při dotazu zaslat také svůj JWT token, kterým server pak provede autorizaci. Zaslání dotazu s JWT tokenem se nachází na výpisu 3.16.

Výpis 3.16: Klientův dotaz se zasláním JWT tokenu v rámci hlavičky.

```
const getResource = () =>
{
  const user: User = JSON.parse(localStorage.getItem('user') || '{}');
  if (user && user.accessToken)
  {
    return axios.get(API_URL, '
                { headers: { Authorization: 'Bearer ' + user.accessToken }});
  }
};
```

Server je při zpracování dotazu schopný z JWT tokenu získat potřebné identifikační informace o uživateli (například uživatelské jméno), které je potom vyhledáno v databázi, zároveň jsou vyhledány uživatelovy role, na jejichž základě může být udělen přístup k datům. Na úrovni aplikace implementované v rámci diplomové práce nebyly role explicitně použity pro řízení přístupu. Každopádně koncept rolí by bylo

možné využít pro pokročilé udělování přístupu či pro odemykání pokročilé funkcionality. Alespoň částečně je toto téma diskutováno v kapitole 3.6. Řízení přístupu k datům by šlo realizovat naivním způsobem, což je naznačeno ve výpisu 3.17.

Výpis 3.17: Autorizace uživatele na základě JWT tokenu.

```
@GetMapping
public Resource getResource(@RequestHeader(name="Authorization") String token)
{
    String username = jsonWebTokenUtility.getUsernameFromJwtToken(token);
    Optional<User> user = userRepository.findByUsername(username);
    if(isUserEligibleToAccessResourse(user))
    {
        return getResource();
    }
}
```

Reálně je však možné využít modulu *Spring Security*. Tento modul definuje sadu filtrů, přes které je příchozí HTTP dotaz filtrován. Je možné rozšířit abstraktní třídu `OncePerRequestFilter` a implementovat metodu `doFilterInternal()`. Ve výpisu 3.18 je pak naznačena implementace třídy `AuthorizationTokenFilter`, která validuje JWT token příchozího HTTP dotazu.

Výpis 3.18: Validace JWT tokenu.

```
public class AuthorizationTokenFilter extends OncePerRequestFilter
{
  //Deklarace a inicializace závislostí v konstruktoru

  @Override
  protected void doFilterInternal(HttpServletRequest request,
                                  HttpServletResponse response,
                                  FilterChain filterChain)
        throws ServletException, IOException, ValidationException
  {
    String jwt = parseJwt(request);
    if(jwt != null)
    {
      if(!validateJwtToken(jwt))
      {
        throw new ValidationException("Invalid JWT token supplied!");
      }
      //Zde pak následuje nastavení bezpečnostní kontextu
    }
    filterChain.doFilter(request, response);
  }

  private String parseJwt(HttpServletRequest request)
  {

    String headerAuth = request.getHeader("Authorization");
    final String BEARER_ = "Bearer ";
    if(StringUtils.hasText(headerAuth))
    {
```

```
30        if(!headerAuth.startsWith(BEARER_))
31        {
32          throw new ValidationException("No Bearer HTTP header part with JWT!");
33        }
34        return headerAuth.substring(BEARER_.length());
35      }
36      return null;
37    }
38  }
```

Zda má být validace JWT tokenu provedena je možné na základě adresy určit v metodě `configure()`, kterou nabízí abstraktní třída `WebSecurityConfigurerAdapter`. Na výpisu 3.19 je naznačena konfigurace autorizace, kdy příchozí HTTP dotazy na jakoukoliv adresu začínající `/api/auth/` nejsou autorizovány, jelikož na těchto adresách je zprostředkována registrace (`/api/auth/register`) a následné přihlášení (`/api/auth/login`), kdy se pro uživatele JWT token teprve generuje. Zmíněné adresy jsou tedy veřejné.

Pro všechny ostatní adresy je však autorizace nutná. *Spring* kontroluje zda se v HTTP dotazu nachází hlavička *Authorization*. Pokud hlavička přítomná není, *Spring* automaticky odpoví klientovi, že se jedná o neautorizovaný přístup. Pokud je přítomna, je vyvolána zmíněná implementace kontroly JWT tokenu v podobě třídy `AuthorizationTokenFilter` (výpis 3.18).

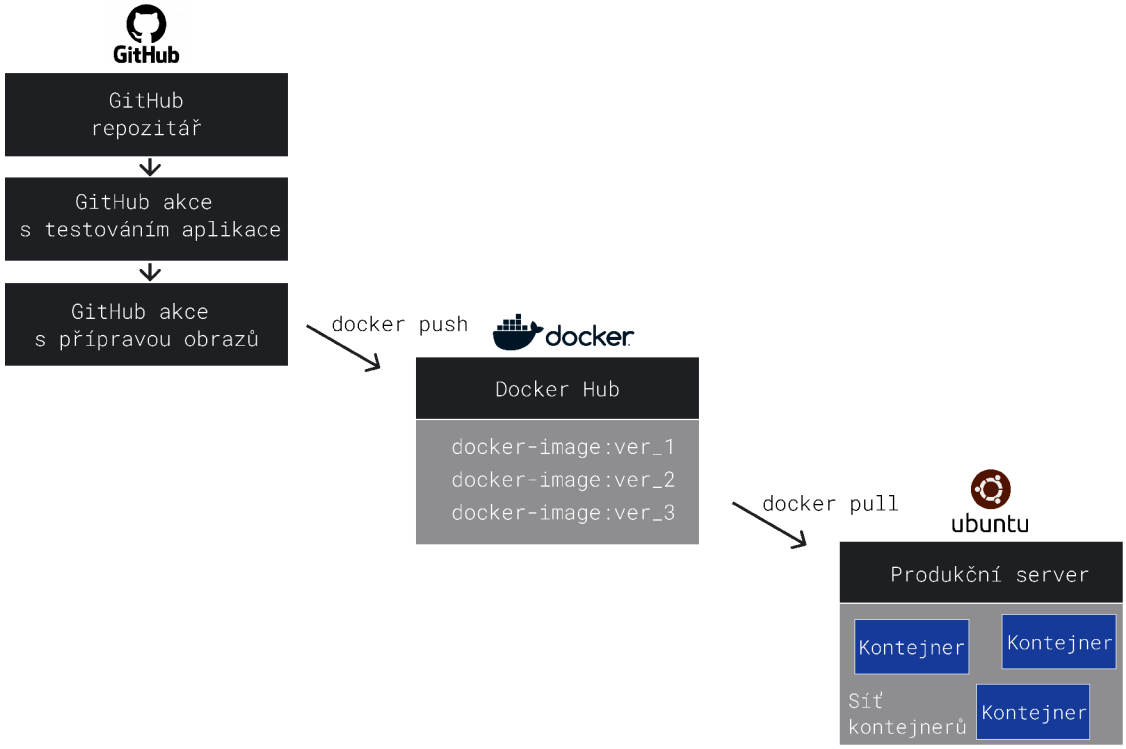
Výpis 3.19: Konfigurace autorizace na základě adresy.

```
1  @Override
2  protected void configure(HttpSecurity http) throws Exception
3  {
4      http.authorizeRequests().antMatchers("/api/auth/**").permitAll()
5              .anyRequest().authenticated();
6
7      http.addFilterBefore(new AuthorizationTokenFilter(),
8                UsernamePasswordAuthenticationFilter.class);
9  }
```

3.4 Nasazení aplikace

Produkční prostředí je možné simulovat virtuálním strojem s operačním systémem *Linux*, jelikož společnosti (například *Amazon* s řešením *AWS*), které poskytují možnost vytvoření serverové instance, standardně nabízí virtuální stroje s distribucí *Linux*. V průběhu řešení diplomové práce bylo pracováno právě s *Amazon AWS EC2* instancemi, kde bylo provedeno prvotní nasazení aplikace. Jelikož ale primárním cílem práce není docílit plného produkčního nasazení aplikace, což by obnášelo nastavení statické IP adresy, domény, generování TLS certifikátu, a s tím spojenými

poplatky, nebude toto řešení dále diskutováno. Nicméně produkční konfigurace aplikace nasazená do simulovaného produkčního prostředí v podobě lokálního virtuálního serveru by potřebovala jen nepatrné množství změn pro plné reálné nasazení.



Obr. 3.3: Nasazení aplikace.

Před nasazením aplikace do produkčního prostředí je neprve nutné zajistit, že aplikace bezchybně plní svou funkci. To znamená, že je neprve nutné kód otestovat sadou testů. Je využito již zmíněných *GitHub* akcí. Definice jednotlivých akcí je možné najít v repozitáři na následující adrese[5]. Pokud jednotlivé akce s testy doběhnou v pořádku, je spuštěna akce „*Publish images to Docker Hub*“, která se stará o nahrání *Docker* obrazů (*images*) na platformu *Docker Hub*. Tato platforma je jakýmsi repozitářem pro sestavené *Docker* obrazy. Aplikace je tedy plně kontejnerizována, kde je každá aplikační část reprezentována jedním *Docker* obrazem (respektive kontejnerem při následném spuštění). Z platformy *Docker Hub* je zároveň možné obrazy stáhnout do lokálního prostředí a distribuovat je tak na produkční server. Realizace je možná na základě souboru `docker-compose.yml`, ve kterém je umožněno uvést referenci na *Docker* obraz umístěný v repozitáři na platformě *Docker Hub*. Pokud je na serveru spuštěn příkaz `docker-compose -f docker-compose.yml -p web up`,

[5] `https://github.com/JanSvoboda6/ml_runner/actions`

jsou na pozadí stáhnuty všechny potřebné *Docker* obrazy a je sestavena celá aplikace z jednotlivých částí (*Nginx*, klientská aplikace *React*, webový aplikační server *Spring*, databáze *MySql*). Uživatelský kontejner s prostředím *Python* je spuštěn až při přihlášení uživatele. *Docker Hub* repozitář využívaný v diplomové práci je možné najít na této adrese[6].

Výpis 3.20 zobrazuje hlavní části konfiguračního souboru `docker-compose.yml`. Služba *Nginx* je namapována na port 80 hostovacího prostředí. *Nginx* pak komunikaci distribuuje na kontejner s webovým serverem (kontejner s názvem `backend`) či na kontejner poskytující webového klienta (kontejner `frontend`). Kontejnery jsou přirazeny do sítí. Je jim tak umožněna komunikace, kdy se kontejnery na sebe odkazují právě podle přiděleného jména (atribut `container_name`). Pomocí atributu `image` je předána reference na specifický *Docker Hub* repozitář, ze kterého jsou pak obrazy stáhnuty. Kontejneru s názvem `backend` je nasdílen *Docker* socket, jelikož při prvotním přihlášení uživatele vytváří pro uživatele nový kontejner a musí tak mít přístup k aplikaci *Docker* běžící v hostovacím prostředí. Databázový kontejner s hostem sdílí adresář `/var/lib/mysql`, kde jsou ukládány databázové položky. Pokud by kontejner jakkoliv selhal, data nebudou ztracena, protože jsou uložena na hostovacím počítači. Konfigurační soubor plného rozsahu je možné najít v *GitHub* repozitáři[7].

Výpis 3.20: Konfigurační soubor `docker-compose.yml`.

```
services:
  nginx:
    container_name: nginx
    image: <autorův_repozitář>:nginx
    ports:
      - "80:80"
    networks:
      - frontend_network
      - backend_network

  backend:
    container_name: backend
    image: <autorův_repozitář>:java-backend
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    networks:
      - frontend_network
      - backend_network
      - database_network

  frontend:
    container_name: frontend
    image: <autorův_repozitář>:react-frontend
    networks:
      - frontend_network

  database:
    container_name: database
    image: mysql
    volumes:
      - ~/database:/var/lib/mysql
    networks:
      - database_network
```

[6] `https://hub.docker.com/repository/docker/jansvoboda596/ml-runner`

[7] `https://github.com/JanSvoboda6/ml_runner/blob/main/docker/docker-compose.yml`

3.5 Poskytnutá testovací data

Pro manuální otestování aplikace byla autorovi poskytnuta testovací datová množina. Jelikož data mají sloužit jen k ověření funkčnosti, není věnována přílišná pozornost úspěšnosti natrénovaného modelu. Konkrétně se jedná o datovou množinu tvořenou různými metrikami ze záznamů cvičení pacientů, kteří na psací podložce vykonávají podle předem ustanoveného protokolu daná cvičení. Podle charakteristik pohybu pera na podložce je možné se domnívat, že daný subjekt trpí Parkinsonovou nemocí. V pohybech se například projeví ztuhlost či nadměrný třes v rukou [18].

Záznamy pacientů jsou tedy rozděleny do dvou tříd. Záznamy zdravých pacientů (31 záznamů) jsou zařazeny do třídy s označením „HC“ (*Healthy control*). Do třídy „PD“ (*Parkinson's disease*) jsou pak přiřazeny záznamy pacientů trpících Parkinsonovou nemocí (24 záznamů). V datové množině se nachází velké množství příznaků (kinematické, dynamické a další), které jsou pojmenovány podle klíče uvedeného ve výpisu 3.21.

Výpis 3.21: Formátování názvů sloupců s příznaky.

```
<pracoviště>.<číslo_cvičení>_<pořadí_záznamu>_<název_příznaku>
```

Jelikož jsou poskytnutá data uložena ve formátu `.xlsx` a aplikace v současnosti podporuje formát `.npy`, kde každý jednotlivý soubor obsahuje jeden vektor příznaků (neboli jeden řádek z datové množiny), bylo nutné poskytnutá data upravit. Z původního souboru `feature_matrix.xlsx` byl vybrán jen omezený počet příznaků pro snížení komplexnosti výpočtu (soubor `features.xlsx`). Následovalo exportování souboru `features.xlsx` do formátu `.csv`. Ve finálním souboru `features.csv` byly nahrazeny veškeré čárky v rámci desetinných čísel tečkami, aby bylo možné data dále zpracovat pomocí *Python* skriptu, který vyžaduje tento formát.

Výpis 3.22: Transformace dat pomocí *Python* skriptu.

```
1  data_frame = pandas.read_csv('features.csv', delimiter=';')
2  Path('healthy_control').mkdir(exist_ok=True)
3  Path('parkinson_disease').mkdir(exist_ok=True)
4  data_frame.iloc[:, 3:] = MinMaxScaler().fit_transform(data_frame.iloc[:, 3:])
5  for i in range(0, len(data_frame.index)):
6    row = data_frame.iloc[i]
7    is_healthy_control = row['disease'] == 'HC'
8    if is_healthy_control:
9      np.save('healthy_control/id_' + str(i + 1) + '_healthy', row[3:].to_numpy())
10   else:
11     np.save('parkinson_disease/id_' + str(i + 1) + '_disease', row[3:].to_numpy())
```

Po přečtení souboru skript vytvoří dva adresáře, kde se ukládají příznakové vektory jednotlivých tříd. Pomocí `MinMaxScaler()` jsou data v rámci jednoho příznaku (sloupce) transformována do rozsahu 0 až 1. Zejména model *Support Vector Machines* pak optimálně klasifikuje, jsou-li data normalizována či standardizována.

Pro transformaci jsou vybrány všechny sloupce počínaje čtvrtým sloupcem (indexování sloupců začíná od nuly). V prvních třech sloupcích se nachází informace o daném subjektu (název, identifikátor, třída). Transformace těchto sloupců by nedávala smysl. Poté je iterováno přes všechny subjekty (řádky) a je vždy vytvořen soubor s příznakovým vektorem ve formátu `.npy` v odpovídajícím adresáři.

Takto vytvořené soubory je možné nahrát do aplikace na adrese `/datasets`. Po vytvoření projektu a spuštění několika běhů je postupně nalezena optimální kombinace parametrů modelu. K tomu také napomáhá vizualizace v podobě teplotní mapy, kdy jsou vyšší hodnoty úspěšnosti validace modelu zobrazeny světlejší barvou. Uživatel pak může spustit model s podobnou kombinací parametrů, jejichž hodnoty jsou blízké hodnotám parametrů doposud nejúspěšnější kombinace, a postupně tak prohledat prostor parametrů.

3.6 Možná rozšíření aplikace

V poslední kapitole diplomové práce bude alespoň částečně naznačeno, jak by mohla být implementovaná aplikace dále rozšířena.

Dvoufaktorové ověření uživatele a správa účtu

Jelikož aplikace podporuje verifikaci emailových adres uživatelů, tak se jako rozšíření nabízí implementace dvoufaktorového ověření. Je možné využít řadu knihoven ke generování tajemství (tzv. *secret*), které je pak v podobě QR kódu zasláno uživateli. Uživatel si tento kód nahraje do mobilní aplikace (například *Google Authenticator*). Na základě inicializačního tajemství je pak v určitých okamžicích generován kód, který uživatel zadá při přihlášení do webové aplikace. Popřípadě by bylo možné uživatele přihlašovat přímo pomocí účtů *Google* s využitím *OAuth2* a delegovat tak správu účtu na jinou službu.

Použití technologie *Kubernetes* či *Docker Swarm*

Pro správu kontejnerů je standardně využíváno řešení v podobě technologie *Kubernetes*. Tato technologie umožňuje pokročilou konfiguraci kontejnerizovaných aplikací, zejména pak škálování, monitorování a znovuspuštění kontejnerů, dojde-li k výpadku. *Docker Swarm* je pak podobným řešením, které je vhodnější pro menší aplikace. Je zároveň součástí instalace *Docker*, to znamená, že není nutná žádná další pokročilá instalace. Nenabízí plnou funkcionalitu jako *Kubernetes*, nicméně pro nasazení aplikace by měla být plně dostačující.

Podpora uživatelem definovaných skriptů

Na poli hlavní funkcionality by šlo aplikaci rozšířit o nahrání skriptů definovaných samotným uživatelem. Platforma by tak nemusela nabízet jen skripty pro spuštění podporovaných modelů strojového učení, ale uživateli by bylo umožněno nahrát vlastní *Python* skripty s požadovanými knihovnami. Uživateli by tak musel být pravděpodobně umožněn přístup k příkazové řádce, aby mohl nainstalovat závislosti. Ve skriptech by se uživatel mohl odkazovat na nahrané soubory a zároveň by mu byl po spuštění skriptu zobrazen výpis v podobě logovacího souboru, což aplikace v současnosti již podporuje. Aplikace by se tak více přiblížela službám jako je například *Google Colab* či *Gradient* od společnosti *Paperspace*.

Bezpečnost a stabilita

Aplikace je připravena pro nasazení do reálného produkčního prostředí. Pro podporu šifrované komunikace za použití protokolu HTTPS je ovšem nutné přidat do konfigurace *Nginx* referenci na certifikát. Je potřebné také připravit infrastrukturu produkčního serveru se statickou IP adresou (nejlépe pak využít cloudové řešení), zajistit vhodnou doménu a vygenerovat již zmíněný TLS certifikát. Pro zvýšenou stabilitu aplikace je také možné limitovat přidělené zdroje uživatelským kontejnerům a zamezit tak zahlcení infrastruktury. Omezení přidělených zdrojů je přímo diskutováno v oficiální dokumentaci platformy *Docker*[8]. Obecně lze také doporučit ověření bezpečnosti aplikace pomocí penetračního testování.

[8] `https://docs.docker.com/config/containers/resource_constraints/`

Závěr

V rámci diplomové práce byla implementována hlavní funkcionalita webové aplikace poskytující asynchronní spuštění trénování a validace modelů strojového učení. Aplikace zároveň nabízí službu autentizace a autorizace uživatelů. Autor při vývoji použil řadu technologií, aplikačních rámců a knihoven, které jsou implementovány v různých programovacích jazycích. S pomocí technologie *Docker* byla aplikace kontejnerizována. Umístění jednotlivých bloků aplikace do separátních kontejnerů vede k vyšší modularitě, bezpečnosti a škálovatelnosti. V neposlední řadě kontejner poskytuje sjednocené běhové prostředí pro danou část aplikace. Pomocí repozitáře poskytovaného službou *Docker Hub* je možné distribuovat jednotlivé *Docker* obrazy na produkční server. Produkční prostředí je v práci simulováno virtuálním strojem s operačním systémem *Linux*.

Autor se při implementaci aplikace snažil dodržovat doporučení diskutována v teoretické části. Zejména bylo využito průběžného testování aplikace za pomocí *GitHub* akcí, v rámci kterých je spuštěna sada automatizovaných testů, a to vždy při nahrání změn do *GitHub* repozitáře. Aplikace byla vyvíjena iterativním přístupem při dodržování malých a častých změn. Při implementaci ucelených částí aplikace bylo praktikováno TDD, kdy byla pozornost věnována fázi refaktorizace. Aplikaci by tak mělo být poměrně jednoduché rozšířit, jelikož jsou stěžejní části současné podoby aplikace otestovány. Pokud budoucí integrace změní chování aplikace vůči externímu volajícímu, sada testů by na tuto změnu měla upozornit. Po zpracování a normalizaci byla datová množina pacientů trpících Parkinsonovou nemocí a zdravých subjektů použita pro manuální otestování aplikace.

Značné usílí bylo věnováno i tvorbě uživatelského prostředí. Výsledky procesu učení je možné nejen analyzovat pomocí základních metrik v textovém formátu, ale uživateli webové aplikace je nabídnuta také možnost vizualizace výsledků v podobě dynamicky vykreslené teplotní mapy či dvourozměrného grafu. V neposlední řadě byla diskutována možná rozšíření aplikace. Mezi stěžejní rozšíření může být zahrnuto zejména doplnění pokročilých metrik a vizualizací, spuštění uživatelem definovaných skriptů či pokročilá správa uživatelských účtů a přidělených kontejnerů. Implementovaná aplikace, která je ve své podstatě platformou pro exekuci asynchronních operací v oblasti strojového učení, by tak pomocí své škálovatelnosti a vhodných rozšíření mohla být silným nástrojem pro pokročilou analýzu a vizualizaci výsledků.

Literatura

[1] FORD, Rob. *Web design: The evolution of the digital world 1990-today.* Taschen, 2019. ISBN 978-3-8365-7267-5.

[2] MARTIN, Robert C. *Clean code: a handbook of agile software craftmanship.* Upper Saddle River, NJ: Prentice Hall, c2009. Robert C. Martin series. ISBN 9780132350884.

[3] BELL, Thomas E. a T. A. THAYER. *Software requirements: Are they really a problem?* [online]. ICSE '76, 1976. [cit.2021-11-23]. Dostupné na URL: <`https://static.aminer.org/pdf/PDF/000/361/405/software_requirements_are_they_really_a_problem.pdf`>

[4] MARTIN, Robert C. a Robert KOSS. *Agile software development.* New Jersey: Prentice-Hall, 2003. ISBN 0135974445.

[5] MCLEAN HALL, Gary. *Adaptive code: agile coding with design patterns and SOLID principles. Second edition.* Redmond: Microsoft Press, 2017. ISBN 1509302581.

[6] BECK, Kent. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., USA, 2002. ISBN 9780321146533.

[7] COHN, Mike. *Succeeding with Agile: Software Development Using Scrum (1st. ed.).* Addison-Wesley Professional, 2009. ISBN 978-0321579362.

[8] ESPOSITO, Dino. *Modern Web Development: Understanding Domains, Technologies, and User Experience* PHI LEARNING, 2017. ISBN 978-1-5093-0060-0.

[9] BURDA, Karel. *Úvod do kryptografie.* Brno: Akademické nakladatelství CERM, 2015. ISBN 978-80-7204-925-7.

[10] MIKULKA, J.; CHALUPA, D.; SVOBODA, J.; FILIPOVIČ, M.; REPKO, M.; MAXOVÁ, M. *Multimodal and Multiparametric Spatial Segmentation of Spine.* In Proceedings of the 2020 19th International Conference on Mechatronics – Mechatronika (ME). Praha: Czech Technical University in Prague, Faculty of Electrical Engineering, 2020. s. 89-93. ISBN: 978-1-7281-5600-2.

[11] CHOLLET, François. *Deep learning v jazyku Python: knihovny Keras, Tensorflow.* Praha: Grada, 2019. Knihovna programátora (Grada). ISBN 978-80-247-3100-1.

[12] CORTES, Corinna a Vladimir VAPNIK. *Support-vector networks. Machine Learning* [online]. 1995, 20(3), 273-297 [cit. 2021-11-25]. DOI: 10.1007/BF00994018. ISSN 0885-6125. Dostupné na URL: <http://image.diku.dk/imagecanon/material/cortes_vapnik95.pdf>

[13] HEARST, Marti, Susan DUMAIS, Edgar OSUNA. *Support vector machines. IEEE Intelligent Systems and their Applications* [online]. 1998, 13(4), 18-28 [cit. 2021-11-25]. DOI: 10.1109/5254.708428. ISSN 1094-7167. Dostupné na URL: <http://ieeexplore.ieee.org/document/708428/>

[14] BURGES, Christopher. *A Tutorial on Support Vector Machines for Pattern Recognition* [online]. In Data Mining and Knowledge Discovery, 1998. s. 121–167. [cit.2021-11-27]. Dostupné na URL: <http://research.microsoft.com/pubs/67119/svmtutorial.pdf>

[15] EVGENIOU, Theodoros a Pontil MASSIMILIANO. *Support Vector Machines: Theory and Applications* [online]. Springer Berlin Heidelberg, 2001. s. 249-257 [cit. 2021-11-27]. ISBN 978-3-540-42490-1. Dostupné na URL: <https://www.researchgate.net/publication/221621494_Support_Vector_Machines_Theory_and_Applications>

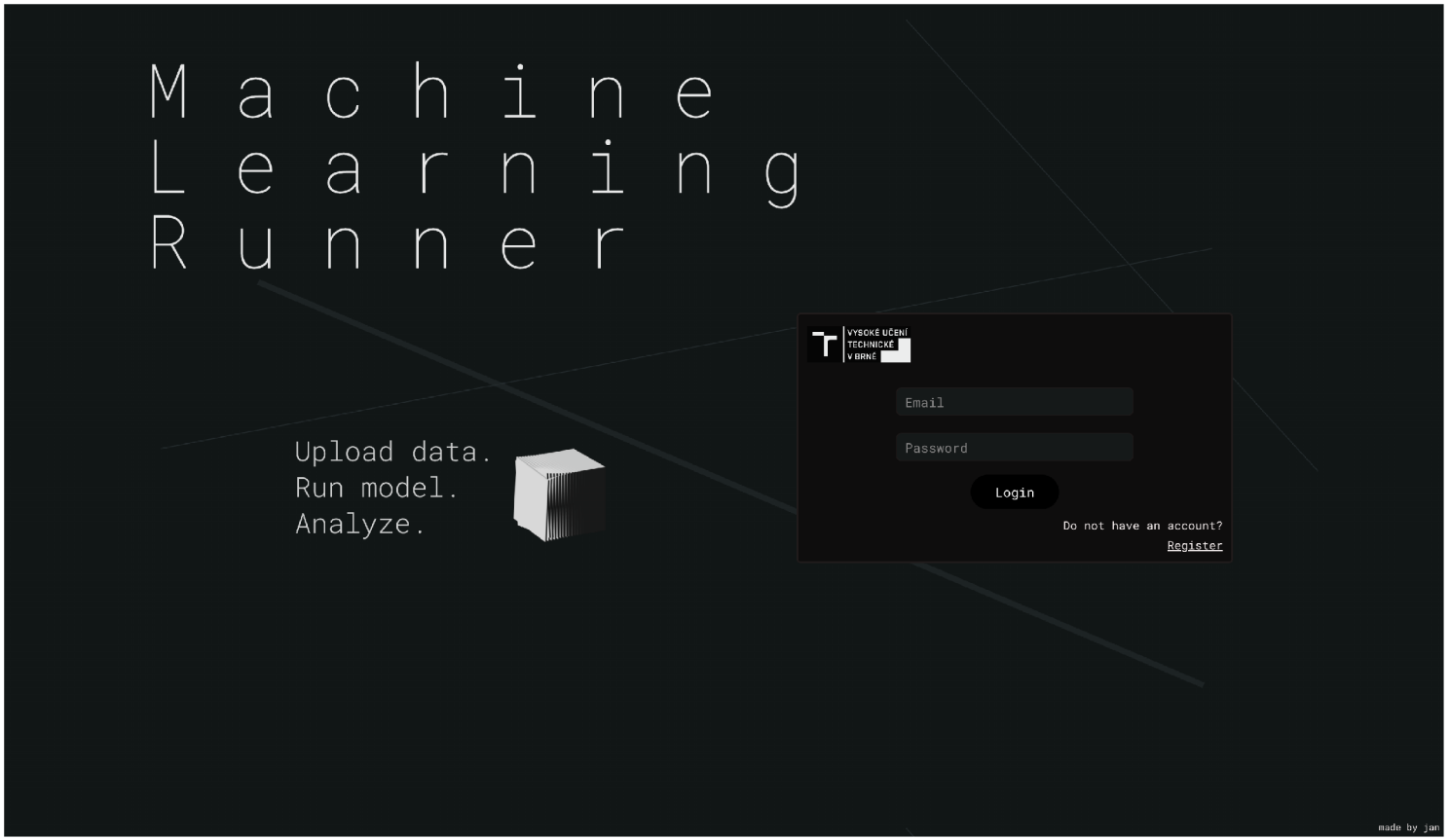
[16] QUINLAN, John. *Induction of Decision Trees* [online]. Machine Learning 1, 1986. s.81–106. https://doi.org/10.1023/A:1022643204877. [cit. 2022-05-13]. Dostupné na URL: <https://link.springer.com/content/pdf/10.1023/A:1022643204877.pdf>

[17] HO, Tin Kam. *Random decision forests* [online]. Proceedings of 3rd International Conference on Document Analysis and Recognition, 1995. s.278-282. [cit. 2022-05-13]. ISBN 0818671289. Dostupné na URL: <https://web.archive.org/web/20160417030218/http://ect.bell-labs.com/who/tkh/publications/papers/odt.pdf>

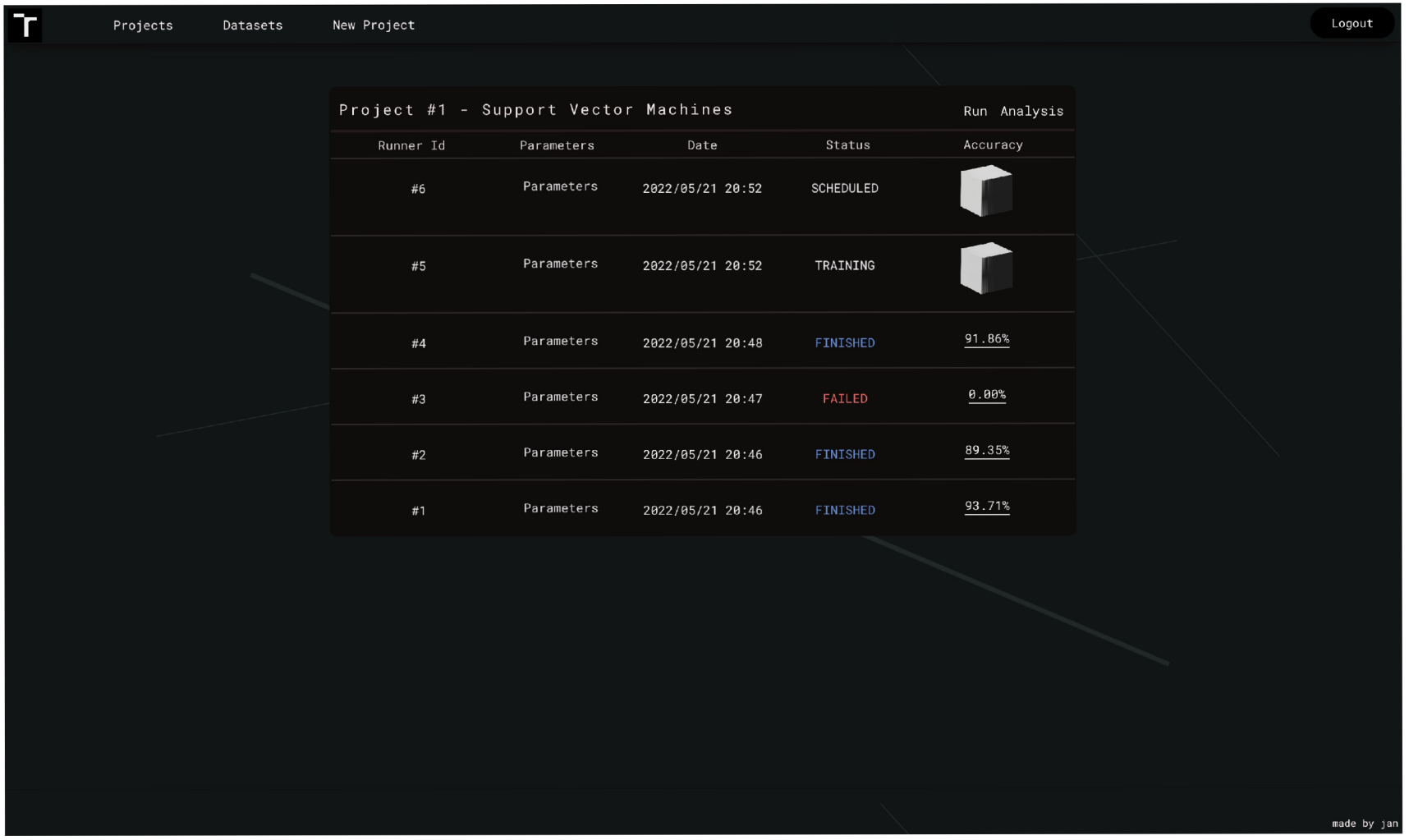
[18] CASCARANO, Giacomo Donato, Claudio LOCONSOLE a Antonio BRUNETTI. *Biometric handwriting analysis to support Parkinson's Disease assessment and grading* [online]. BMC Med Inform Decis Mak 19, 2019. https://doi.org/10.1186/s12911-019-0989-3. [cit. 2022-05-17]. Dostupné na URL: <https://bmcmedinformdecismak.biomedcentral.com/articles/10.1186/s12911-019-0989-3>

Seznam symbolů a zkratek

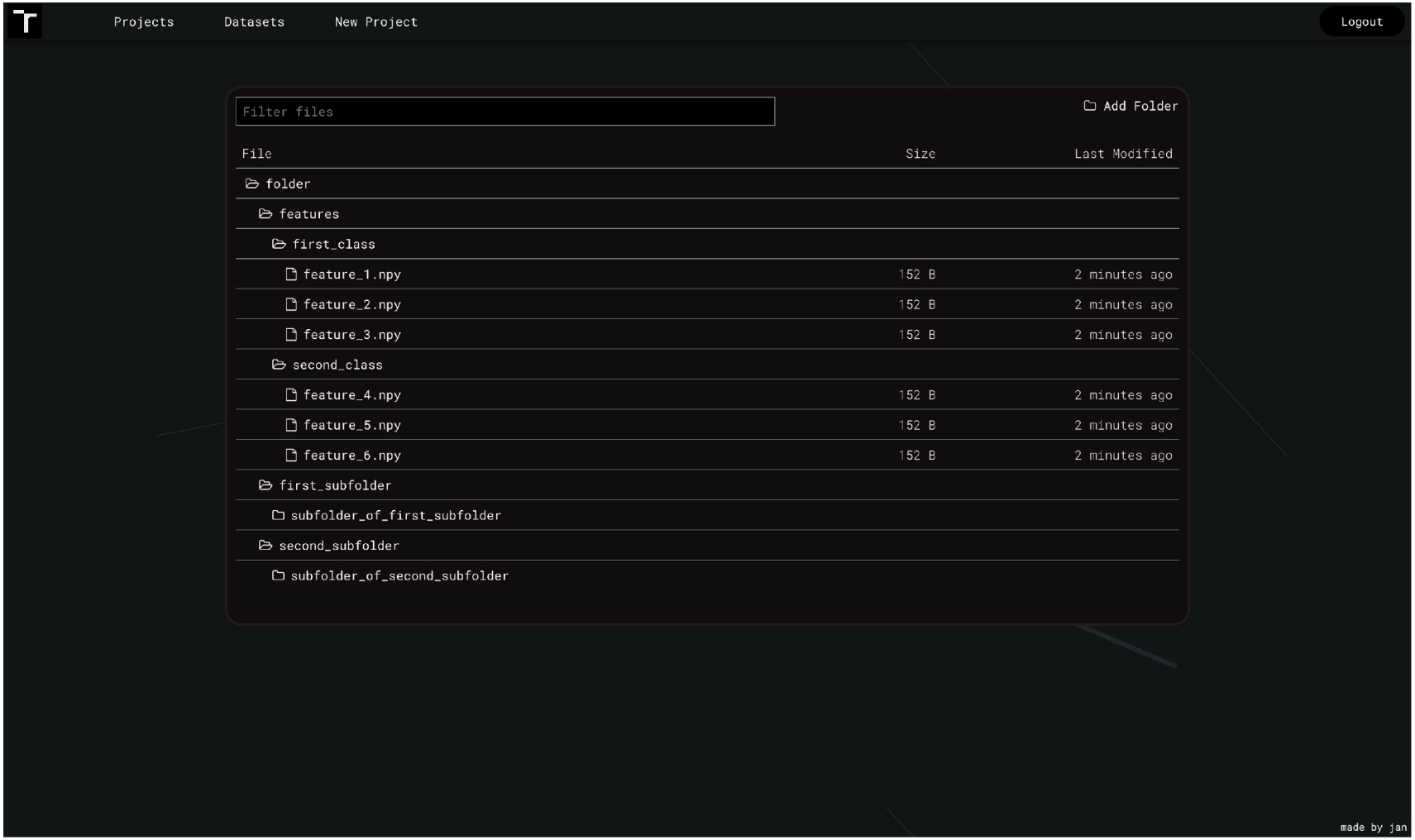
Best practices	Doporučené praktiky při implementaci kódu
Decoupling	Nezávislost dílčích částí
***Front-end* technologie**	Soubor technologií tvořící klientskou část aplikace
***Back-end* technologie**	Soubor technologií spuštěných na straně serveru
Feature vector	Příznakový vektor
Software development	Vývoj software
Sprint	Časový interval, na který se dělí vývoj
CI/CD	Nástroje pro integraci a iterativní sestavení aplikace
Build	Sestavení aplikace
Clean code	Čistý kód
SOLID	Doporučené principy pro psaní kódu
Framework	Aplikační rámec
JWT	*JSON WEB Token*
Dataset	Soubor dat
JSON	*JavaScript Object Notation*, formát zápisu dat
Unit tests	Jednotkové testy
End-to-end tests	Testy verifikující funkčnost aplikace jako celku
***Single-page* aplikace**	Aplikace minimalizující dobu načítání webu
HTTP	Hypertext Transfer Protocol
Support Vector Machines	Metoda podpůrných vektorů
User interface	Uživatelské prostředí
PK	*Primary key*, primární klíč
FK	*Foreign key*, cizí klíč



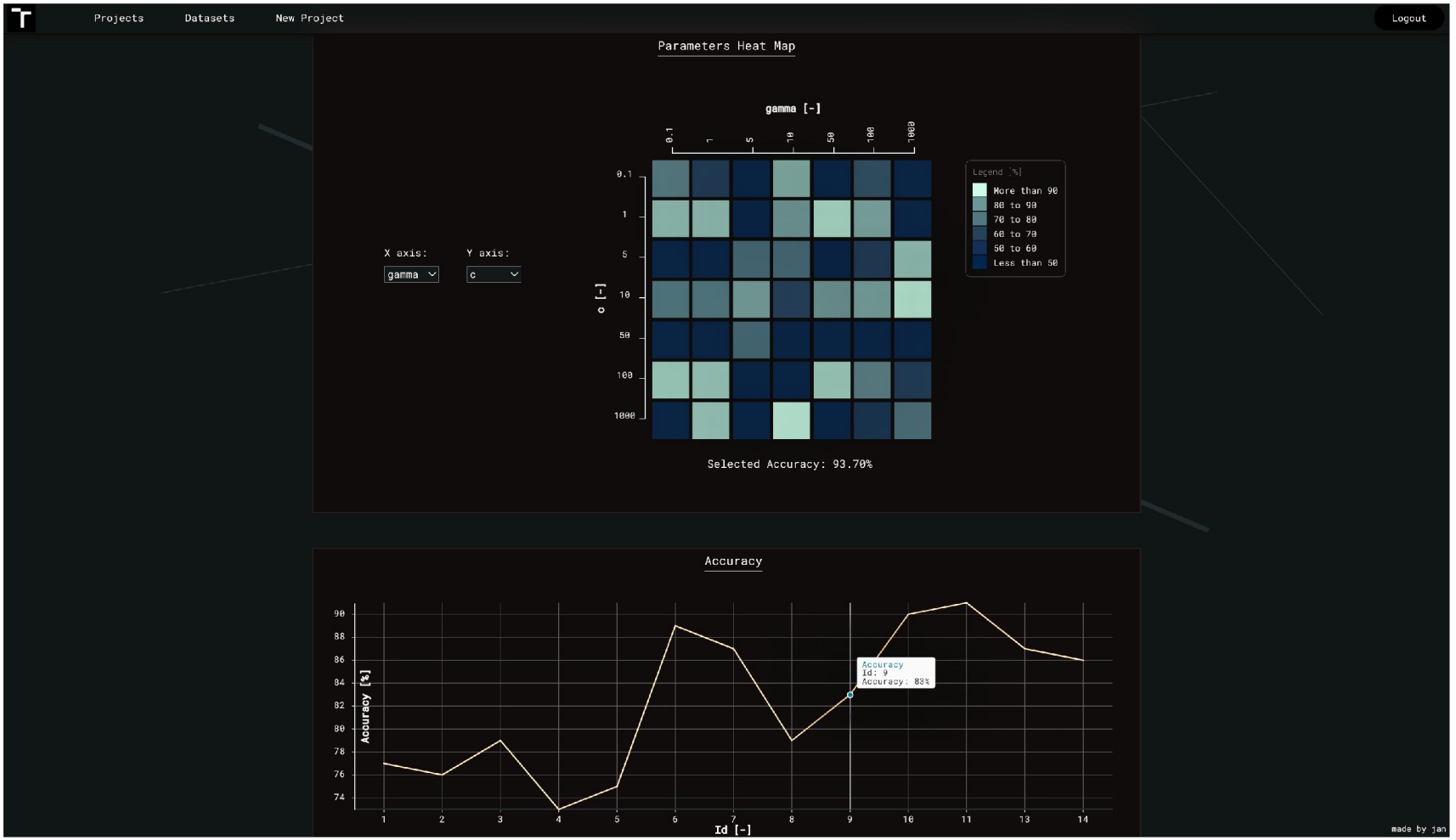
Obr. A.1: Úvodní stránka s přihlášením.



Obr. A.2: Tabule s vytvořenými projekty.



Obr. A.3: Prohlížeč souborů.



Obr. A.4: Vizualizace výsledků (naplněno náhodnými daty).