ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA STROJNÍ

DIPLOMOVÁ PRÁCE

Interaktivní aplikace pro asistované editace YOLO datasetů

VYPRACOVAL: Bc. Gabriela Chládková

VEDOUCÍ: Ing. Matouš Cejnek Ph.D.

2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Chládková** Jméno: **Gabriela** Osobní číslo: **474836**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídící techniky**

Studijní program: **Automatizační a přístrojová technika**

Specializace: **Automatizace a průmyslová informatika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Interaktivní aplikace pro asistované editace YOLO datasetů

Název diplomové práce anglicky:

Interactive application for assisted editing of YOLO datasets

Pokyny pro vypracování:

- vytvořte aplikaci s uživatelským rozhraním, která umožní editovat YOLO dataset
- vytvořte pro aplikaci modul, který umožní spouštět na pozadí trénink a predikci YOLO modelů
- rozšiřte aplikaci tak, aby predikce získané z natrénovaného modelu navrhovaly anotace nových dat

Seznam doporučené literatury:

[1] Beazley, David, and Brian K. Jones. Python cookbook: Recipes for mastering Python 3. " O'Reilly Media, Inc.", 2013.
[2] Y. Rouizi, Mastering YOLO: Build an Automatic Number Plate Recognition System. Independently Published, 2023.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Matouš Cejnek, Ph.D. U12110.3

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **26.04.2024** Termín odevzdání diplomové práce: **31.05.2024**

Platnost zadání diplomové práce: _____

Ing. Matouš Cejnek, Ph.D. podpis vedoucí(ho) práce	prof. Ing. Tomáš Vyhlídal, Ph.D. podpis vedoucí(ho) ústavu/katedry	doc. Ing. Miroslav Španiel, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomantka bere na vědomí, že je povinna vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

24.4.2024
Datum převzetí zadání

Podpis studentky

Prohlášení:

Prohlašuji, že jsem diplomovou práci na téma „Interaktivní aplikace pro asistované editace YOLO datasetů“ vypracovala samostatně pod vedením Ing. Matouše Cejnka, Ph.D a že veškeré podklady, ze kterých jsem čerpala jsou uvedeny v seznamu použité literatury.

V Praze dne: 4. června 2023 Podpis:

Poděkování:

Chtěla bych poděkovat vedoucímu mé diplomové práce Ing. Matouši Cejnkovi Ph.D. za věcné připomínky, cenné rady a vstřícnost při konzultacích a věnovaný čas, čehož si moc vážím.

Abstrakt

Cílem této práce je vytvoření interaktivní aplikace pro asistované anotování YOLO datasetu, umožňující úpravu a tvorbu anotací. Aplikace bude navržena tak, aby podporovala různé typy anotací a poskytovala uživatelsky přívětivé rozhraní. Důraz bude kladen na možný budoucí rozvoj a využity budou existující nástroje např. pro správu modelů pro trénink a predikci. V rámci práce budou zohledněny i aspekty automatizace procesu anotace pomocí strojového učení, kdy aplikace umožní trénování modelu a predikovaní nových anotací, což by mělo zefektivnit a zjednodušit pracovní postup.

Klíčová slova: Anotace datasetu, Webová aplikace, JavaScript, React, YOLO, Predikce

Abstract

The goal of this thesis is to create an interactive application for assisted editing of YOLO datasets, allowing for the modification and creation of annotations. The application will be designed to support various types of annotations and provide a user-friendly interface. Emphasis will be placed on future development potential, and existing tools, such as those for model management for training and prediction, will be utilized. The project will also consider aspects of automating the annotation process through machine learning, where the application will enable model training and prediction of new annotations, aiming to streamline and simplify the workflow.

Key words: Labeling of dataset, Web aplication, JavaScript, React, YOLO, Prediction

Obsah

1. Úvod

V době, kdy množství dostupných dat exponenciálně roste [1], stává se jejich efektivní zpracování a analýza velmi podstatnou oblastí. K efektivnímu zpracování dat přispívá i obor počítačové vidění a strojového učení, který v posledních pár letech zaznamenal významný rozvoj. S tím spojená přesná a rychlá anotace dat je nezbytnou podmínkou pro trénování kvalitních modelů. Proces anotování datasetů je však časově náročný a trénovací datasety vyžadují často manuální lidskou práci, která nejen snižuje efektivitu, ale také zvyšuje riziko chyb.

Cílem této diplomové práce je vývoj aplikace pro asistované anotování datasetů, která usnadní a zrychlí tento proces pomocí intuitivního uživatelského rozhraní a integrace predikování anotací z již natrénovaných modelů. Aplikace umožní uživatelům nejen zobrazování datasetů a tvorbu či úpravu anotací, ale také nabídne možnost trénování modelů a predikci anotací pro vybrané, např. dosud neanotované obrázky. Tento přístup má potenciál snížit časovou náročnost anotování a zároveň i zvýšit jeho přesnost a konzistenci, jelikož uživatel bude moct zaměřit svou pozornost především na zpřesnění navrhovaných anotací.

V následujících kapitolách bude popsána nejprve technika anotování pro počítačové vidění, a popsán „You Only Look Once“ (YOLO) model [2], se kterým v současné chvíli aplikace pracuje, dále se pak budu věnovat obecně tvorbě webové aplikace v Reactu, který jsem si po zvážení předem vybrala a představím jeho dostupné knihovny pro grafické práce na webové stránce. Dále se v praktické části zaměřím na popis návrhu a architektury aplikace, její hlavní funkce a technologické řešení jednotlivých částí. Součástí práce bude také evaluace řešení a jejího přínosu, případně budoucího směrování.

2. Teoretická část

2.1. Umělá inteligence a anotace dat

Anotace dat představuje klíčový proces v oblasti strojového učení a umělé inteligence, tento postup zahrnuje přiřazování popisů, štítků nebo dalších metadat k jednotlivým částem obrázku, ať už se jedná o vizuální data, audio, nebo jiné typy, což umožňuje počítačovým systémům lépe porozumět obsahu a kontextu dat. Anotace je nezbytným krokem pro trénování AI modelů a bez kvalitních anotací by nebylo možné dosáhnout vysoké přesnosti a spolehlivosti v predikcích

modelů. Velmi rozšířenou oblastí je vizuální anotace, kterou se moje aplikace zabývá a v následující kapitole představím základní typy vizuálních anotací včetně těch které bude práce zahrnovat.

2.1.1. Typy vizuálních anotací

V závislosti na aplikaci a požadovaném výsledku existuje několik hlavních typů vizuálních anotací:

- **Klasifikace**

Klasifikace je základní forma vizuální anotace. Spočívá v přiřazení jednoho nebo více štítků („label“) celému obrázku na základě objektů, které obsahuje, případně může také zahrnovat určení míry jistoty, s jakou je klasifikace provedena. Klasifikace se používá tam, kde je důležité vědět, jaký typ objektu je na obrázku, ale není potřeba znát jeho polohu nebo další podrobnosti (viz. Obr. 1).

Obr. 1 – Příklad klasifikace obrázku s kočkou [3]

- **Detekce**

Kromě přiřazení štítku každému objektu zahrnuje tato technika také informace o umístění objektu v obraze. To se provádí nejčastěji pomocí ohraničujících rámečků (tzv. „bounding boxes“), které vymezují prostor, kde se objekt nachází. Zpravidla se jedná o obdélníky, ale nyní existují modely, které u detekce objektů zvládají využívat polygony a získávat tak více informací, příkladem takového modelu je YOLOv8[4]. Na Obr. 2 je ukázka takové anotace.

Obr. 2 – Příklad použití detekce objektu kočky [3]

- **Segmentace**

Segmentace obrazu představuje nejpodrobnější formu vizuální anotace. Přiřazuje každému pixelu v obrázku určitou třídu, což poskytuje velmi přesné informace o tvaru a poloze objektů. Na Obr. 3 je příklad segmentace. Existují dva hlavní typy segmentace:

- **Sémantická segmentace**: Nerozlišuje se mezi jednotlivými instancemi objektů, a tak více stejných objektů bude patřit do jedné anotace.

- **Instanční segmentace**: Podobně jako sémantická každému pixelu je přiřazena třída, ale navíc rozlišuje mezi různými instancemi téhož objektu.

Obr. 3 – Příklad použití segmentace pro obrázek s kočkou [3]

2.1.2. Techniky anotace obrázků

Proces anotace se může lišit svou složitostí a tím, kolik manuální práce vyžaduje. Ruční anotace teoreticky poskytuje vysokou kvalitu, ale velkou časovou náročnost a náklady, navíc takový postup nemusí mít konzistentní výsledek. To lze do určité míry automatizovat pomocí nástrojů, které anotace predikují a předběžně objekty označí – lidé pak pouze provádí kontrolu a případnou úpravu. Alternativně lze plně využít algoritmy strojového učení k automatickému generování anotací, což může výrazně urychlit proces a snížit náklady, přičemž kvalita závisí na přesnosti těchto algoritmů.

2.2. YOLO model

Pro práci byl jako výchozí model zvolen YOLO. Jedná se model pro detekci objektů v obrazcích a videích pomocí konvolučních neuronových sítí (CNN). Tento algoritmus disponuje schopností provádět detekci objektů v reálném čase (tzv. „real-time“) s vysokou přesností a efektivitou. Toho dosahuje tím, že je obrázek nebo video analyzováno pouze jednou – detekce a klasifikace objektů probíhá současně, což celý proces, v porovnání s některými dalšími neuronovými sítěmi jako je např. R-CNN, urychluje.

Pojmem real-time se v oblasti detekce objektů rozumí, že odezva modelu dosahuje přibližně 30 ms nebo méně, což pochopitelně závisí na HW parametrech, datasetu a dalším nastavením. Je to velmi důležitý atribut pro využití např. v autonomních vozidlech a podobných aplikacích, kde jiné sítě, ač třeba spolehlivější, takové rychlosti nedosahují.

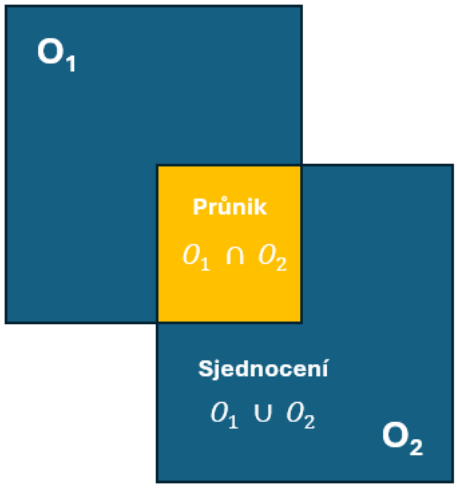
2.2.1. Princip fungování

Zpracování začíná tím, že vstupní obrázek je předán konvoluční neuronové síti (CNN), která z obrázku extrahuje charakteristické rysy. Děje se tak pomocí rozdělení obrázku na mřížku $m \times m$, kde každá buňka má určitý počet bounding boxes, pomocí kterých predikuje pravděpodobnost výskyt objektu dané třídy a jeho umístění. *Poznámka: Toto umístění může zasahovat i mimo hranici buňky.* Čím větší pravděpodobnost tím lépe. Poté jsou vyfiltrován predikce s malou pravděpodobností, než je stanovená hranice, a následuje filtrování pomocí postprocesingového algoritmu zvaného „non-max suppression“ [5], ten využívá tzv. vzorec

„Intersection over Union“ (v překladu průnik ku stejnocenní), znázorněný na Obr. 4, který evaluuje překryv dvou rámečků podle vzorce:

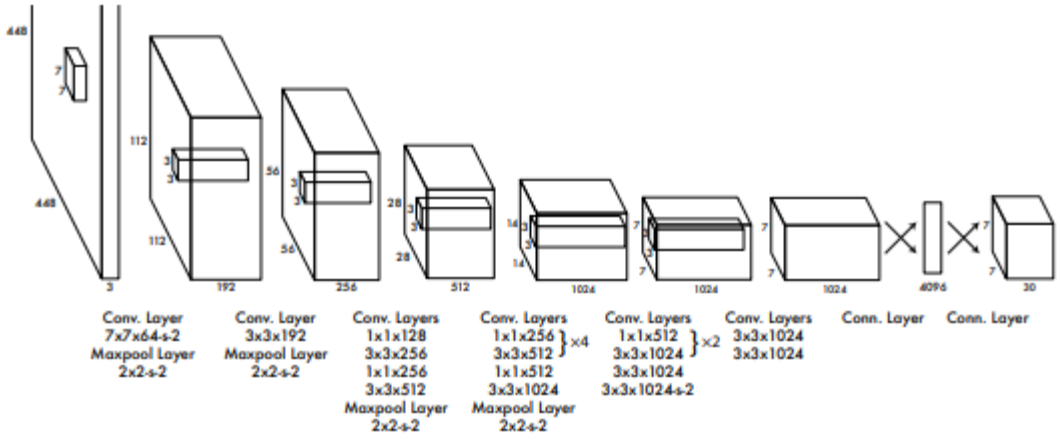
$$IoU = \frac{O_1 \cap O_2}{O_1 \cup O_2},$$

kde O_1, O_l jsou jednotlivé překrývající se oblasti. Pokud IoU překročí pro danou třídu určenou hranici, dojde k vyfiltrování menší pravděpodobnosti výskytu objektu. Účelem tohoto algoritmu je zjisti, které rámečky patří jedné instanci objektu a vybrat nejpřesnější předpověď. Výpočet probíhá jak na úrovni buněk, tak skrz celý obrázek.



Obr. 4 – Znázornění výpočtu IoU

Posledním stupněm je série plně propojených vrstev, které předpoví výstupní vektory jednotlivých tříd, reprezentující ohraničující rámečky a jejich pravděpodobnosti pro každý objekt na obrázku. Celé schéma architektury YOLO modelu je Na Obr. 5.

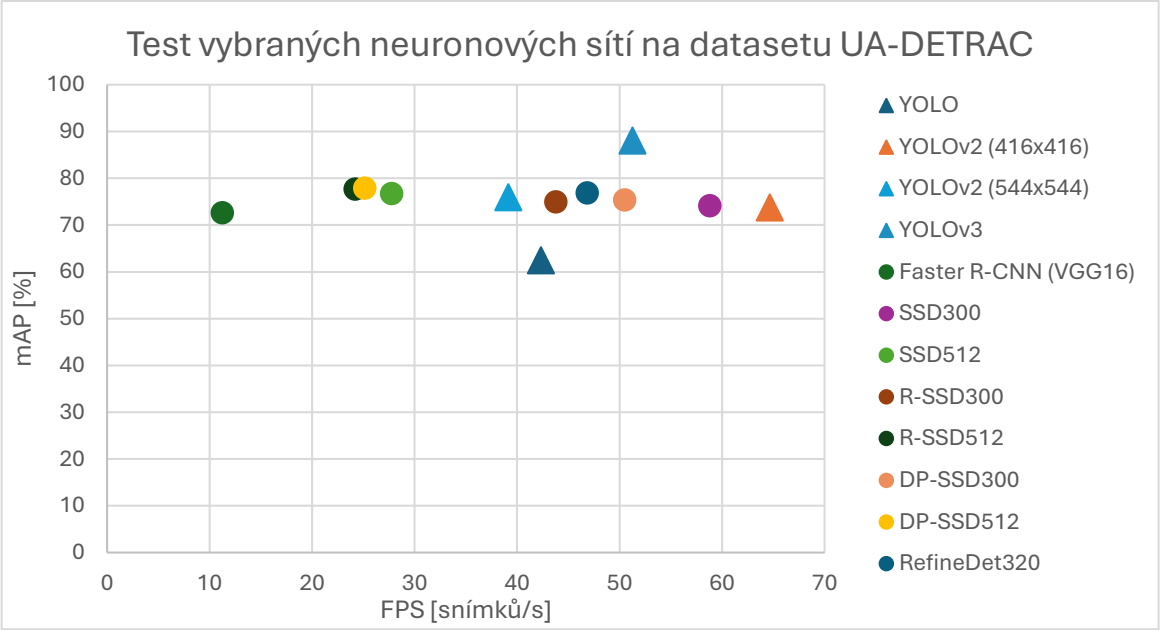


Obr. 5 – Schéma architektury YOLO (v původní verzi) [1]

2.2.2. Srovnání rychlosti a přesnosti

Pro ukázku efektivnosti byl převzat test rychlosti a přesnosti několika vybraných modelů [6], přesnost je v testu vyjádřena hodnotou „Mean Average Precision“ (Průměrná střední přesnost), zkráceně mAP, což je časté evaluační měřítko pro modely strojového učení a čím vyšší číslo tím přesnější a spolehlivější model je.

V testu jsou použité dnes již starší verze YOLO, jelikož je dostupný již model YOLOv10, ale už tyto modely nabízely velice dobré výsledky, jak je vidět na grafickém znázornění na Obr. 6. Test byl proveden na datasetu UA-DETRAC [7] na grafické kartě NVIDIA GeForce GTX 1080Ti GPU. Jak je vidět, YOLOv2 byl v testu nejrychlejším modelem s FPS 73,82 snímků/s a mAP 64,65 % a jako nejpřesnější vyšlo pozdější YOLOv3 s mAP 81,09 % a FPS 51.26 snímků/s. Zároveň je vidět, že dvoufázový model typu Faster R-CNN (tedy rychlejší verze R-CNN) má rychlost pod úrovní real-time.



Obr. 6 – Srovnání výkonnosti vybraných neuronových sítí na datasetu UA-DETRAC

2.2.3. Verze

V průběhu let vyšlo několik verzí navazujících na YOLO, snažící se model vylepšit. I na uvedeném testu na Obr. 6 je možné vidět, jak novější verze překonaly původní model, docíleno toho bylo například použitím jiné loss funkce, aplikací principu tzv. „anchor box“, který umožnují detekovat i víc objeků v jedné buňce zároveň, což původní verze neumožnovala,

a dalších metod [8]. V současné době je aktuální verze již YOLOv10, pro praktickou část aplikace byl vybrán model YOLOv8.

2.3. React: Struktura a rendrování

React je populární open-source JavaScriptová knihovna pro vytváření uživatelských rozhraní, kterou vyvinula společnost Facebook a v posledních letech patří k jedné z nejpoužívanější frameworku/knihovně [9]. Díky komponentovému přístupu umožňuje vývojářům aplikaci rozdělit na samostatné a znovupoužitelné části, které mohou být vyvíjeny a spravovány nezávisle a díky využití virtuálního DOM („Document Object Model“ – objektový model dokumentu), jehož princip je popsán níže, React umožňuje efektivní správu stavu aplikace a optimalizuje aktualizace uživatelského rozhraní.

JSX

Jedním z klíčových rysů Reactu je používání JSX, neboli syntaktické nadstavby, která umožňuje psát HTML přímo v JavaScriptu. Příklad takového kódu je v (Kód 1). Používání JSX není povinné, v Kód 2 je totožný validní kód bez využití JSX, ale již z takhle krátké ukázky je vidět jeho výhoda v usnadnění tvorby komponent a větší čitelnost kódu.

```
import React from 'react';

function App() {
  return (
    <div>
      <h1>Vítejte v Reactu</h1>
      <p>Toto je jednoduchý příklad aplikace.</p>
    </div>
  );
}

export default App;
```

Kód 1 – Ukázka React komponenty vracející JSX

```
import React from 'react';

function App() {
  return React.createElement(
    'div',
    null,
    React.createElement('h1', null, 'Vítejte v Reactu'),
    React.createElement('p', null, 'Toto je jednoduchý příklad aplikace.')
  );
}

export default App;
```

Kód 2 – Stejná komponenta bez použití JSX

Nejmenším stavebním blokem jsou tzv. elementy, což jsou to jednoduché JavaScriptové objekty, které popisují, co chceme vidět na obrazovce. Díky JSX se nemusíme starat o vytváření těchto objektů a volání funkce `React.createElement()` se děje na pozadí. Podobně jako ve funkci `React.createElement()` i v JSX můžeme definovat atributy: `type` (povinný), `props`, `ref`, `key` a případně `children`, jejichž funkce je následující [10]:

- `type` – Definuje typ React elementu
- `key` – Slouží jako jednoznačná identifikace, používá se především při mapování pole. Pokud chybí je null.
- `ref` – Je reference na skutečný DOM uzel. Umožňuje vám získat přímý přístup k DOM prvku nebo instanci komponenty. Pokud chybí je null.
- `props` – Objekt obsahující vlastnosti, které jsou předávány komponentě, nebo null hodnota.
- `children` – Jsou to, co chcete, aby bylo předáno do tohoto prvku. Při přidávání více potomků používáme pole.

Jednotlivé React komponenty strukturují elementy hierarchicky do stromů, kvůli využití virtuálního DOM. Samotné komponenty mohou být implementovány buď jako třídy nebo funkce, ale oba přístupy přijímají `props` jako vstupy a vrací strom elementů jako výstup. U funkční komponenty je výstupem je návratová hodnota funkce, v případě třídy je výstupem návratová hodnota metody render.

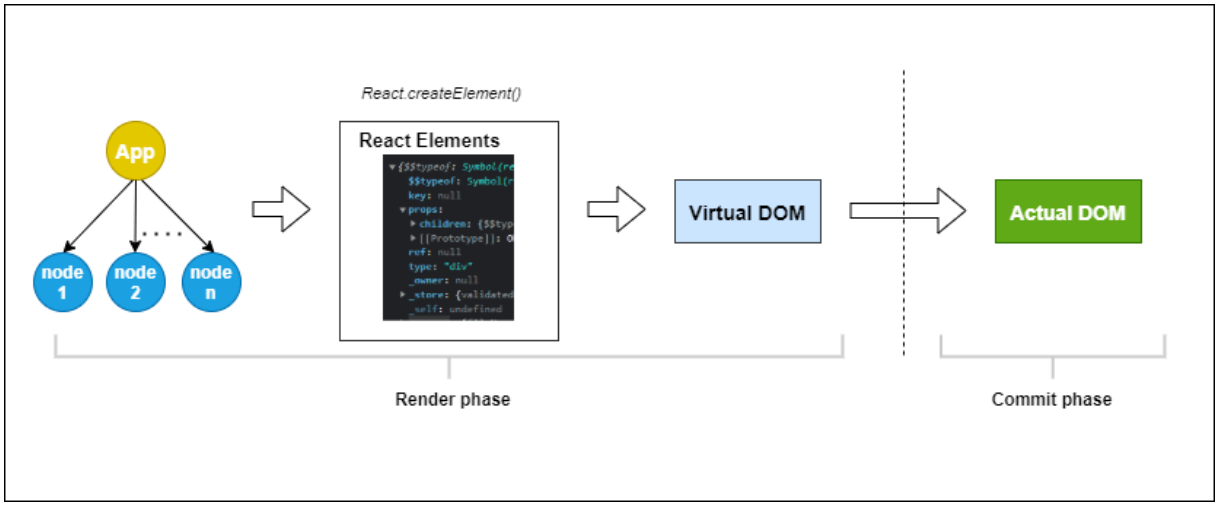
React sleduje komponenty vytvářením instancí pro ně. Každá instance má stav a životní cyklus. V třídních komponentách můžeme přistupovat ke stavu a životnímu cyklu pomocí předdefinovaných metod a klíčového slova `this`, zatímco ve funkčních komponentách používáme tvz. „React hooks“ (funkce umožňují správu vnitřního stavu komponenty a jejího životního cyklu). V současnosti je spíše preferovaný přístup využití funkčních komponent, kvůli stručnější syntaxi.

2.3.1.1. Vykreslovací proces v Reactu

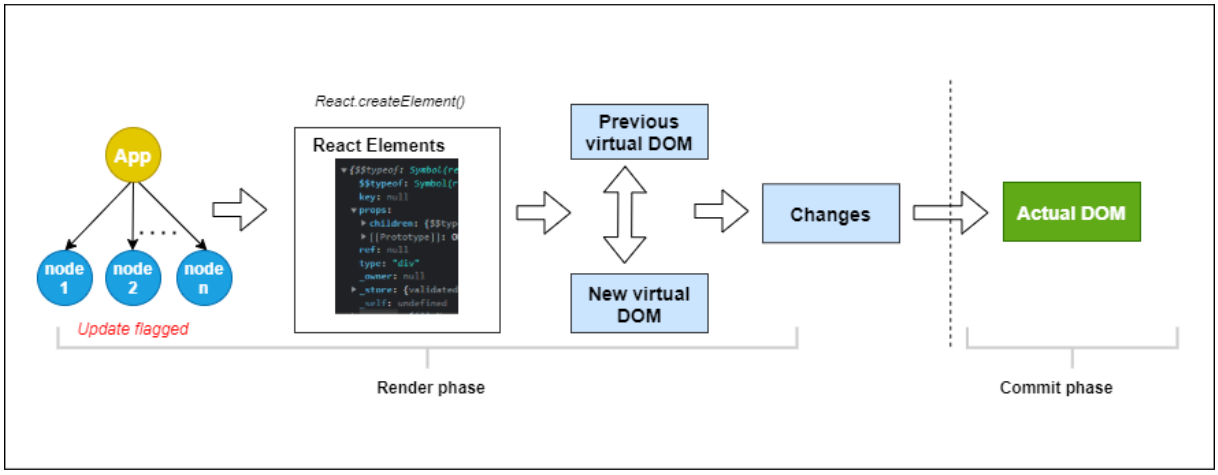
Vykreslovací proces neboli „render“ je v Reactu proces, který popisuje uživatelské rozhraní na základě aktuálního stavu aplikace a `props`. Abych mohla tento proces představit, je nejprve nutno rozlišit mezi virtuálním a skutečným DOM.

Skutečný DOM je struktura, která hierarchicky reprezentuje HTML dokument v prohlížeči, kde jednotlivé elementy odpovídají uživatelskému rozhraní (UI) stránky. Pokud na stránce dojde ke změně, je nutno provést aktualizaci DOM, což může vést k překreslení i elementy, kterých se změna vůbec netýká. Zápis do skutečného DOM tím pádem může být neefektivní a nákladný. React aplikuje koncept virtuálního DOM, který rovněž reprezentuje uspořádání elementů, avšak jedná se pouze o JavaScriptový objekt popisující instanci komponenty a jeho potomky, jeho změna je tedy pouze vytvoření nového objektu. Virtuální DOM je způsob, jak zajistit nejmenší možnou změnu toho skutečného a dále bude podrobněji popsán.

Při spuštění aplikace nastává počáteční render, který je znázorněn na Obr. 7. Při změně stavu dochází k překreslení (tzv. „rerender“), jehož účelem je zjistit které části UI je třeba aktualizovat. To je znázorněno na Obr. 8. Tento překreslovací proces lze rozdělit na fáze „Render“ a „Commit“ (Záznam změn).



Obr. 7 – Počáteční render [11]



Obr. 8 – Proces překreslení [11]

Fáze Render

V této fázi se provádí tvorba stromu virtuálního DOM. Začíná se od kořenové komponenty a postupuje směrem dolů a vytváří se obraz toho, jak by měl vypadat skutečný DOM. V případě rerendru následuje podobný přístup, ale s rozdílem, že se vytváří nový virtuální DOM dle aktuálního stavu komponent. React musí zajistit, že uživatelské rozhraní aplikace je vždy synchronizováno s React stavem. K dosažení tohoto cíle, kdykoli se změní stav komponenty, musí přerenderovat všechny potenciálně ovlivněné komponenty, tj. komponentu, která vlastní změněný stav a její potomky.

Dále probíhá proces rekonciliace neboli synchronizace virtuálního DOM se skutečným a to co nejefektivněji. K vyřešení tohoto problému se používá algoritmus známý jako „diffing“ (porovnání), jeho účelem je nalézt co nejmenší počet operací k transformaci starého stromu na nový. Složitost tohoto algoritmu je pouze O(n) [12] a jedná se tedy o vhodný způsob, jak aktualizovat uživatelské rozhraní.

Fáze Commit

V této fázi probíhá manipulace se skutečným DOM. Při počátečním renderování je přes API volána funkce `appendChild()` k umístění všech vytvořených DOM uzlů na obrazovku. Při přerenderování se přes API aplikují vypočtené nezbytné operace z fáze renderování a dojde ke změně UI. Knihovna React nekomunikuje se skutečným DOM, ale používá k tomu balíčky třetích stran jako např. React DOM (pro webové platformy) a React Native (pro mobilní platformy).

2.4. Kreslící nástroje v Reactu

Grafických nástrojů a knihoven v Reactu existuje celá řada, já se zde ale zaměřím na takové, které podporují geometrické tvary, jejich posouvání a další funkce, které mohou souviset s anotováním obrázků. Při rešerši takových nástrojů knihoven byly nalezeny tyto, které dále představím: SVG, Canvas, D3.js , react-konva.

SVG

SVG (Scalable Vector Graphics) je široce používaný formát pro tvorbu vektorové grafiky. Umožnuje snadné kreslení tvarů a manipulaci s nimi, v základu nabízí tvary obdélníku, úsečky, kruhu, elipsy, polygonu a cesty bodů. Dále je dobře podporovaný v Reactu, což umožňuje vytvářet vizualizace přímo v komponentách. Na ukázce Kód 3 je příklad použití.

```
const Rectangle = () => {
  return (
    <div>
      <svg width={200} height={100}>
        <rect width={200} height={100} fill="blue" />
      </svg>
    </div>
  );
};
```

Kód 3 – Příklad implementace obdélníku s využitím SVG

Canvas

Canvas je HTML element určený pro kreslení převážně 2D grafiky. Na rozdíl od SVG je bitmapově orientovaný. Tento přístup je ideální pro složité a dynamické vizualizace, nicméně, bitmapová grafika může ztrácet kvalitu při změně velikosti. Také na rozdíl od SVG nemá předem připravených tolik tvarů a poskytuje pouze kresbu obdélníku nebo cest bodů, tudíž pokud chceme nějaký složitější tvar je na nás zařídit správnou implementaci. Práce s Canvasem v Reactu může být složitější, protože využívá imperativní přístup ke kreslení, na rozdíl od deklarativní styl Reactu. Na ukázce Kód 4 je příklad použití.

```
const Rectangle = () => {
  const canvasRef = useRef();
  let ctx = null;

  // initialize the canvas context
  useEffect(() => {
    const canvas = canvasRef.current;
    canvas.width = canvas.clientWidth;
    canvas.height = canvas.clientHeight;

    ctx = canvas.getContext("2d");
  }, []);

  return (
    <div>
      <canvas ref={canvasRef}></canvas>
    </div>
  );
}
```

Kód 4 – Příklad implementace obdélníku s využitím Canvas

D3.js

D3.js [13] je velmi silná knihovna pro tvorbu datových vizualizací ale i kreslení geometrických tvarů. Knihovna je připravena na široké využití funkcionalit jako posouvání tvarů, zoom apod. Nicméně opět se jedná o imperativní styl programování a práce s D3.js v Reactu má svá úskalí. Při tvorbě s D3.js se počítá s manipulací přímo se skutečným DOM, což může vést ke konfliktům s virtuálním DOM. Na ukázce Kód 5 je příklad použití.

```
const RectangleD3 = () => {
  const svgRef = useRef(null);

  useEffect(() => {
    const svg = d3.select(svgRef.current);
    svg.append('rect')
    .attr('width', 200)
    .attr('height', 100)
    .attr('fill', 'blue');
  }, []);

  return (
    <div>
      <svg ref={svgRef} width={200} height={100} />
    </div>
  );
};
```

Kód 5 – Příklad implementace obdélníku s využitím D3.js

React-konva

React-konva [14] je knihovna pro práci s 2D grafikou, která poskytuje deklarativní prostředí pro práci s Canvas v Reactu. Obdobně jako D3.js podporuje zoom a další pokročilé funkcionality, a to s pomocí `props`, jako je například v ukázce Kód 6 u elementu `Rect` atributy `x`, `y`, dají se zde ale využívat i tzv. „eventy“ jako `onDragEnd` apod. sloužící k jednoduché správě chování prvku.

React-konva poskytuje předpřipravené tvary obdélníku, kruhu, úsečky, elipsy, pravidelného mnohoúhelníku, a kromě geometrických tvarů podporuje i obrázek a text. Patří do frameworku Konva, kde originální JavaScriptová knihovna Konva.js obsahuje podobné funkcionality, ale čistě imperativní styl a disponuje daleko lepší dokumentací. Kvůli tomu je častá nutnost dohledávat požadované chování zde.

Dle oficiálního githubu [15] je react-konva v ideálním případě pomalejší oproti obyčejnému používání Canvas z důvodu abstrakce navíc, to však kompenzuje redukcí komplexnosti a dobrou kompatibilitou s Reactem.

```
const RectangleKonva = () => {
  return (
    <Stage width={200} height={100}>
      <Layer>
        <Rect
          x={0}
          y={0}
          width={200}
          height={100}
          fill="blue"
        />
      </Layer>
    </Stage>
  );
};
```

Kód 6 – Příklad implementace obdélníku s využitím React-konva

2.4.1. Přehled

V Tab. 1 je zobrazeno porovnání všech zmíněných nástrojů a knihoven. Pro praktickou část byla zvolena knihovna react-konva, kvůli své dobré komptabilitě s Reactem, rychlosti a podpoře velkého množství elementů a velké možnost funkcionalit.

	SVG	**Canvas**	**D3.js**	**React-konva**
Grafická reprezentace	Vektorová	Bitmapová	Vektorová, bitmapová	Bitmapová, (vektorová – velmi okrajové použití)
Deklarativní /imperativní typ	Deklarativní	Imperativní	Imperativní	Deklarativní
Dostupné tvary	Obdélník, úsečka, kruh, elipsa, polygon, cesta bodů	Obdélník, cesta bodů	Obdélník, úsečka, kruh, elipsa, polygon, cesta bodů	Obdélníku, úsečka, kruh, elipsa, pravidelný mnohoúhelník
Obrázek	Ano	Ano	Ano	Ano

Text	Ano	Ano	Ano	Ano
Podpora komplexních tvarů	Nízká	Střední	Střední	Vysoká
Podpora vrstvení	Omezená	Základní	Pokročilá	Pokročilá
Podpora interaktivity	Dobrá	Dobrá	Výborná	Výborná
Určení	Spíše statické prvky v menším počtu	Dynamické prvky, větší množství prvků	Statické a dynamické prvky množství prvků	Dynamické prvky, větší množství prvků

Tab. 1 – Srovnání grafických nástrojů a knihoven pro React

3. Praktická část

Cílem praktické části je navrhnout samotnou aplikaci, která bude sloužit k asistovanému anotování datasetů. Aplikace bude sloužit nejen k úpravě stávajících anotací, ale i k tvorbě nových a také jejich predikci na základě již natrénovaného modelu. To by mělo zefektivnit celý proces tvorby anotovaného datasetu.

I přesto, že na trhu existují v této oblasti už podobné aplikace a v rozsahu této diplomové práce není možnost jejich rozsáhlým funkcionalitám konkurovat, i přesto tato práce přináší přínos především ve svém vzniku na míru našim potřebám (viz. kapitola 3.1.) a možnosti dalšího rozvoje díky dokumentaci zdrojového kódu.

V následujících kapitolách bude nejprve popsán návrh celé struktury aplikace a jejího případu použití a dále pak bude podrobněji rozepsán samotný vývoj jednotlivých celků a rozhodnutí, které její tvorbu provázely.

3.1. Funkční požadavky

V této kapitole se zaměříme na podrobné specifikace funkčních požadavků naší aplikace, které budou dávat jasnou představu o tom, co má být implementováno a jaké funkce musí aplikace poskytovat uživatelům. Níže jsou v Tab. 2 jednotlivé požadavky přehledně rozepsány, včetně jejích zařazení. Zároveň je při vývoji kladen důraz na to, aby v budoucnu byla aplikace dále jednoduše rozšiřitelná.

Dataloader	Načítání datasetu z lokálních konfigurovatelných složek a nahrávání do paměti nebo databáze.
Api	Podpora tvorby, úpravy a mazání anotací.
	Možnost různých typů anotací: keypoint, box, polygon.
Model	Verzování YOLO modelu a možnost rozšíření na další typy.
	Trénink a predikce vybraných obrázků, možnost spouštění z UI.
	Paralelizování práce modelu pro optimalizaci uživatelského dojmu.
Grafické rozhraní	Webová nebo desktopová aplikace.
	Autentifikace uživatele.
	Zobrazení menu datasetů (projektů) pro daného uživatele.
	Zobrazení vybraného datasetu, jeho položek a dalších informací.
	Interaktivní okno s vybraným obrázkem, kde proběhne načtení stávajících anotací a umožní nám jejich uživatelskou úpravu, tvorbu nové, nebo odstranění už nepotřebné.
	V okně s vybraným obrázkem možnost listování v datasetu.
	Rozlišené třídy anotací.
	Tag typu obrázku (trénovací/validační/…)
	Možnost zoomu a posunutí obrázku pro přesnější práci.

Tab. 2 – Rozepsané funkční požadavky

3.2. Existující řešení

Jak bylo zmíněno v úvodu, aplikace není ve svém řešení úplně inovativní a na trhu jsou dostupné různé robustní anotační nástroje nejen pro anotaci obrázků jako je např. LabelStudio [16] a Roboflow [17], Computer Vision Annotation Tool (CVAT) [18], kterými se tato práce inspiruje a dále na míru rozvijí požadavky zmíněné v bodu 3.1.

3.2.1. LabelStudio

LabelStudio je open-source nástroj pro anotování různých typů dat, včetně textu, obrázků, videí a zvuku. Díky své flexibilitě a široké škále podporovaných formátů mohou uživatelé snadno konfigurovat rozhraní pro anotace podle svých potřeb a využívat pokročilé funkce, jako je podpora více uživatelů, automatické anotace a integrace s externími nástroji pro strojové učení.

3.2.2. Roboflow

Roboflow je platforma zaměřená na správu a přípravu datasetů pro projekty počítačového vidění. Nabízí širokou škálu nástrojů pro import, organizaci a anotování obrázků, verzování datasetu i automatizací části procesu, avšak může být složitější na prvotní naučení. Zároveň se jedná se o komerčnější řešení, přestože nabízí i plán zdarma, ten je ale omezený množstvím uživatelů, počtem trénování a predikci apod.

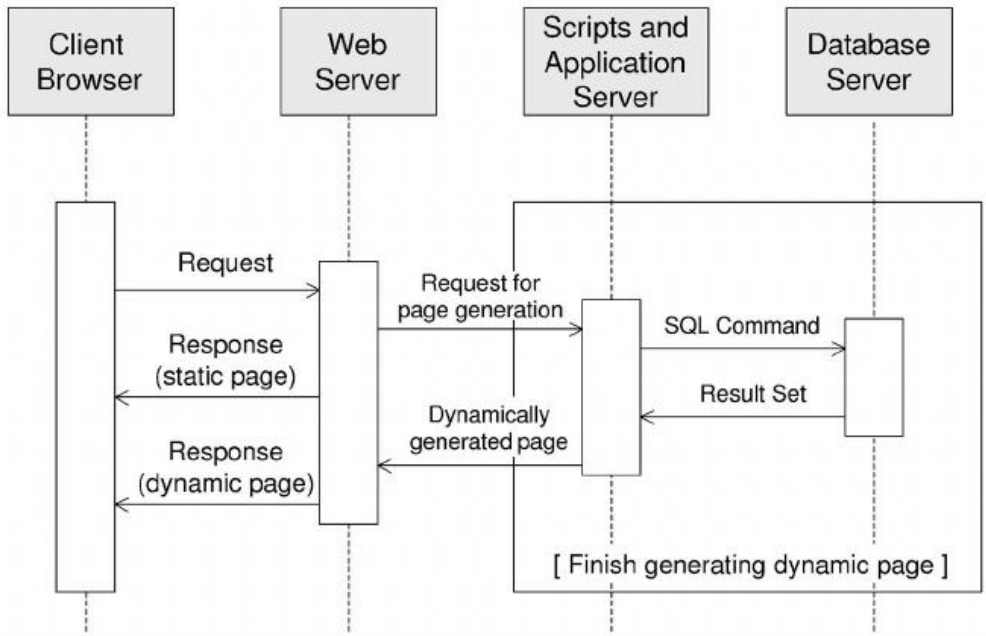
3.2.3. CVAT

CVAT je výkonný open-source nástroj pro anotaci dat v rámci oblasti počítačového vidění. Obsahuje dva typy plánů – hostovaný na jejich cloudu CVAT.AI anebo tzv. „self-hosted“, kdy je správa serveru na uživateli. Cloudová verze má velmi omezené možnosti pro služby zdarma například co do počtu uživatelů, projektů apod. a především nenabízí automatizaci anotace. Self-hosted verze tyto parametry na počty uživatelů apod. nemá, avšak ani ona automatické anotování nenabízí a jedinou možností je tedy placená verze.

3.3. Návrh aplikace

Návrh aplikace vycházel z funkčních požadavků z kapitoly 3.2., byla využit běžná struktura webové aplikace, rozdělená na tzv. „backendovou“ (to co není uživateli vidět) část, se

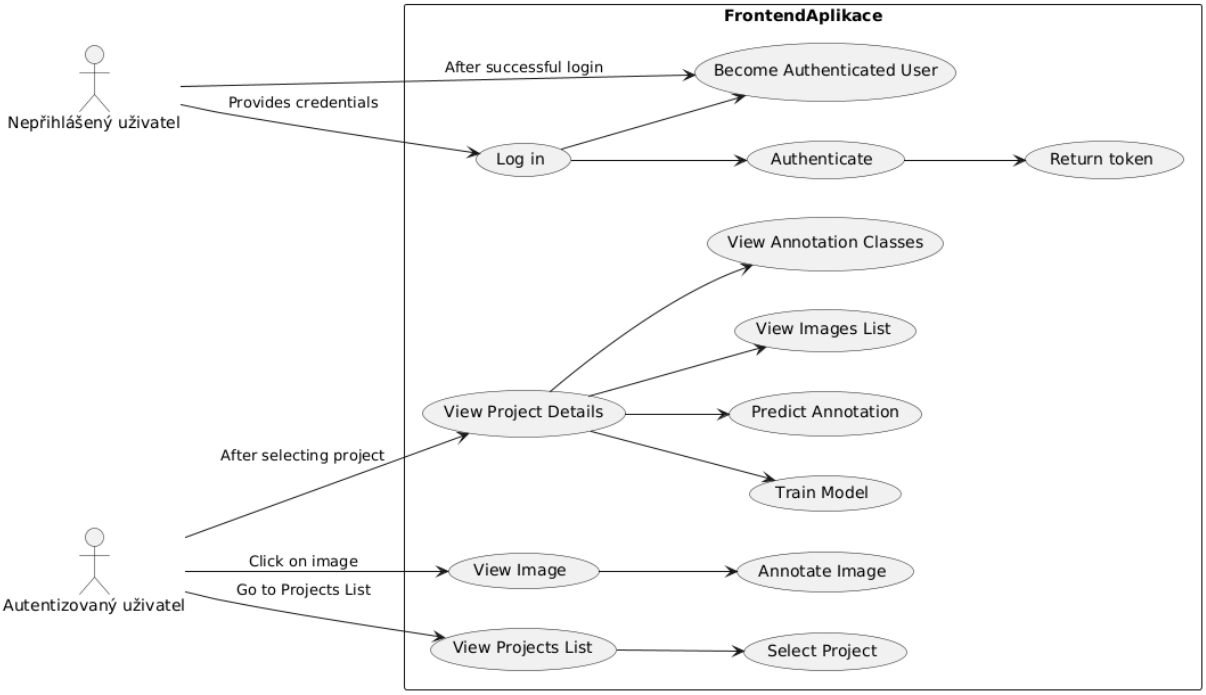
serverem a databázovým úložištěm, a tzv. „frontendovou“ část (uživatelské rozhraní), která se skládá z webového serveru a klientského prohlížeče viz Obr. 9, kde je vidět i průběh komunikace mezi těmito částmi. Kromě samotného Api na práci s datasetem a komunikaci s uživatelským rozhraním bylo třeba ještě navrhnout i prostředek na správu a trénování modelů detekce objektů. Pro tento účel byla zvolena knihovna MLflow, která obsahuje i intuitivní uživatelské rozhraní.



Obr. 9 – Architektura aplikace [19]

Use Case diagram

K návrhu uživatelského rozhraní aplikace byl využit diagram případů užití (Use Case diagram), tento diagram zachycuje scénáře, které mohou být v aplikaci aktérem (např. uživatelem) spuštěny. Jedná se o jednoduchou srozumitelnou reprezentaci s vysokou mírou abstrakce. Diagram na Obr. 10 popisuje frontendovou část aplikace a doplňuje tak funkční požadavky z bodu 3.2, což napomáhá lepší orientaci v požadavcích a usnadňuje dále implementaci potřebných funkcionalit pro obě části aplikace.



Obr. 10 – Diagram případů užití uživatelského rozhraní aplikace

Aktéři vyskytující se v systému

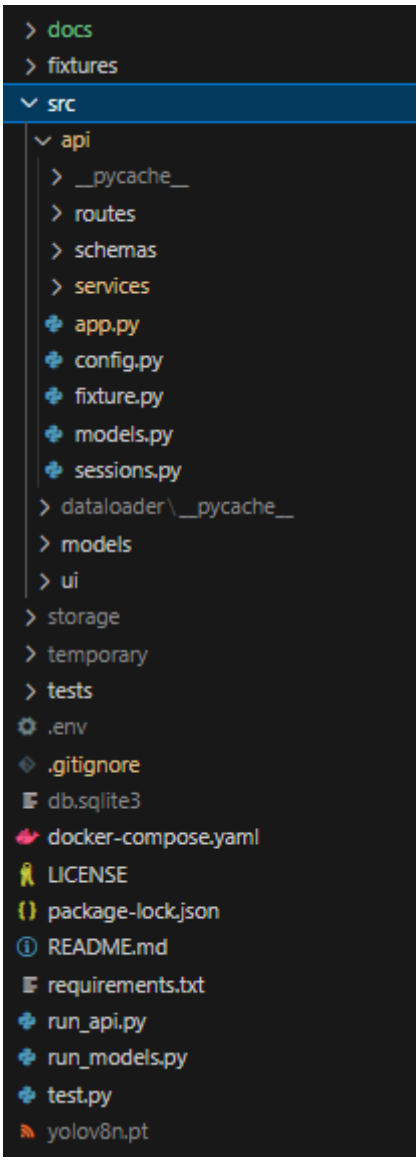
- Nepřihlášený uživatel – Jde o uživatele, který se právě vstoupí na stránku, je ve stavu, kde nemá možnost využívat většinu jejích funkcionalit.
- Přihlášený (autentizovaný) uživatel – Má možnost aplikaci plně využívat.

Vybrané skupiny Use Case diagramu

- Přihlášení uživatele. Aktérem je neregistrovaný uživatel (u ostatních je to už vždy registrovaný uživatel).
- Zobrazení seznamu projektů (datasetů) a základní informace o nich
- Zobrazení a správa vybraného datasetu, zobrazení anotačních tříd
    - Změna typu obrázku – tzv. „split_type“ (trénovací/validační/prázdné)
    - Trénování modelu
    - Dostupné natrénované modely
    - Predikce anotací
- Zobrazení obrázku
    - Úprava stávajících anotací, případně jejich odstranění
    - Tvorba nové anotace
    - Změna anotační třídy

3.4. Vypracování backendu

Byly implementovány celky Api a MLFlow (podle stejnojmenné knihovny) [20]. Api zajišťuje načítání datasetů ze složek do databáze a poskytuje frontendové aplikaci přístup k těmto datům pro další práci. Umožňuje také frontendové části aplikace komunikaci s celkem MLflow, kdy zajišťuje spouštění trénování modelů a predikci anotací. MLflow poskytuje funkcionalitu pro sledování experimentů, jejich metrik, uchovávání modelů, a jejich správu. Je implementován jako samostatný server s uživatelským rozhraním. Na Obr. 11 je vidět adresářová struktura projektu s rozepsanou backendovou částí, kromě složky „ui“, která obsahuje celou frontendovou část.



Obr. 11 – Struktura projektu

3.4.1. Api

Jak bylo zmíněno v kapitole 3.3 backendová část aplikace se skládá z datové struktury a serverové části, ta byla dále rozdělena na tyto vrstvy[22]:

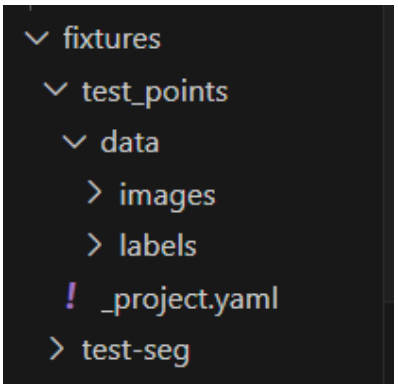
- Prezentační vrstva – Tvoří ji tzv. „endpointy“ (koncové body) na dotazování, využito FastAPI [21]
- Schématická mezivrstva – Úzce spjatá s prezentační vrstvou, tvoří ji datový model
- Aplikační vrstva – Logika aplikace, nepracuje přímo s databází ale přes datovou vrstvu
- Datová – Namapována databáze 1:1, využívá knihovnu SQLAlchemy

3.4.1.1. Dataloader

Dataloader má za funkci zajistit načtení datasetů ze složek na disku do databáze. Vzniknul k tomu soubor `fixtures.py`, který nejprve vytvoří uživatele pomocí volání funkce `create_user` a poté pomocí funkcí `load_projects` a `load_project` prochází jednotlivé projekty ve složce fixtures, kde nejprve načte projektový soubor `_project.yaml`, kde jsou kromě názvu projektu, anotačních typů a seznamu anotačních tříd, uvedeny i cesty k obrázkům („`/images`“) a anotacím („`/labels`“). Příklad takového souboru je Kód 7 a na Obr. 12 je k němu i ukázka odpovídající struktury adresáře.

```yaml
images_path: data/images
labels_path: data/labels

project_name: Test points

annotation_type: box
classes:
  - name: Red
    color: FF0000
  - name: Blue
    color: 0000FF
  - name: Green
    color: 00FF00
```

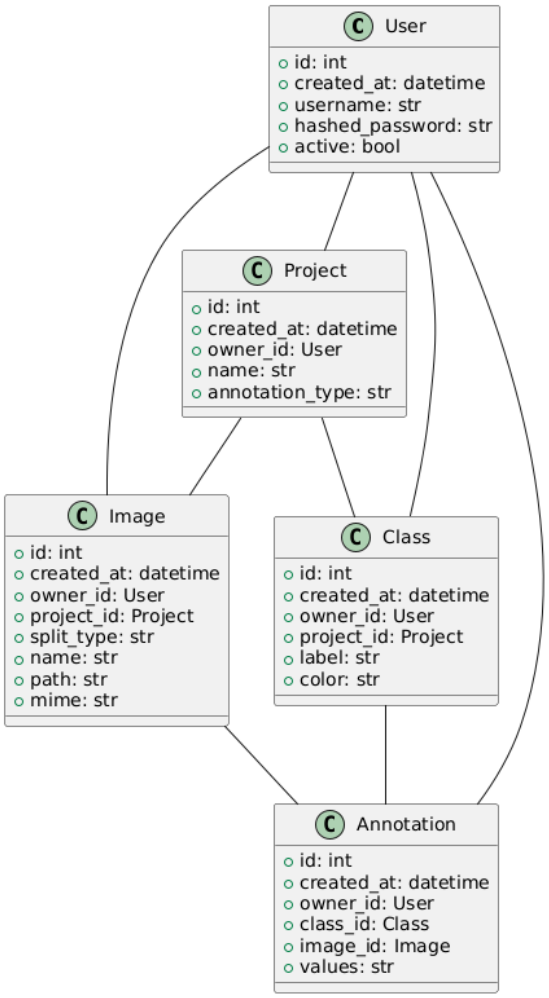
Kód 7 – příklad projektového YAML souboru s metadaty projektu



Obr. 12 – Struktura datasetu na lokálním disku

3.4.1.2. Datová vrstva

Knihovna SQLAlchemy [23] byla využita pro svoje jednoduché a intuitivní použití v prostředí Pythonu, konkrétně byla použita verze SQLAlchemy ORM, která využívá tzv. princip „Object Relational Mapper“ pro práci s databázemi, tento princip umožnuje napamovat třídy v Pythonu na databázové tabulky a reprezentovat data v nich pomocí instancí těchto tříd. Použila jsem tzv. přístup „Code first“, neboli přístup, kdy nejprve vzniká datová vrstva a až poté se tvoří obraz databáze. Na Obr. 13 je vidět návrh databáze s tabulkami User, Project, Image, Class, Annotation a danými atributy a na Kód 8 je příklad takového objektu pro reprezentaci tabulky Project.



Obr. 13 – Diagram tříd znázorňující datový model aplikace

```
class Project(Base):
    __tablename__ = "projects"

    id = Column(Integer, primary_key=True)
    created_at = Column(TIMESTAMP, default=datetime.datetime.utcnow)
    owner_id: Mapped[int] = mapped_column(ForeignKey("users.id"))
    name = Column(String(255))
    annotation_type = Column(String(255))
```

Kód 8 – Příklad třídy Project reprezentující projekt (dataset)

3.4.1.3. Datová struktura

Jak bylo v předchozí kapitole zmíněno použita byla knihovna SQLAlchemy. Ta nabízí několik možných systému pro správu relačních databází (RDBMS) a vybrané z nich jsou přehledně porovnány v Tab. 3. SQL Lite nabízí nejjednodušší alternativu, ale to může být vykoupeno méně pokročilými funkcemi. Na druhé straně je velmi robustní, avšak objemný MSSQL, který například podporuje i pokročilé procedury apod. V současné době se ale všechny pokročilejší

RDBMS zdáli pro tuto použití zbytečné, a proto byl zvolen SQLLite. V budoucnosti však díky využití SQLAlchemy nic nebrání poměrně snadné změně.

	RDBMS		
	SQL Lite	**MSSQL**	**PostgreSQL**
Typ	Embeddable, souborová databáze, nevyžaduje server	Komplexní, vyžaduje server	Komplexní, vyžaduje server
Licencování	Public Domain	Komerční, s možností volné verze Express	Open Source (PostgreSQL License)
Zálohování	Zálohy pomocí souboru DB	Různé možnosti včetně SQL Server Management Studio	Zálohy pomocí pg_dump, pg_basebackup, atd.
Transakce	ACID compliant	ACID compliant	ACID compliant
Podpora SQL standardu	Základní podpora SQL standardu	Rozsáhlá podpora SQL standardu, vlastní rozšíření	Velmi dobrá podpora SQL standardu, rozsáhlé funkce
Škálovatelnost	Vhodné pro malé až střední aplikace	Vysoká škálovatelnost, vhodné pro komerční řešení	Velmi dobrá škálovatelnost, vhodné pro komerční použití
Indexování	Podporuje základní typy indexů	Pokročilé možnosti indexování, Full-Text Search	Pokročilé možnosti indexování, Full-Text Search
Jazyk procedur	Ne, nebo velmi omezený	T-SQL (Transact-SQL)	PL/pgSQL, PL/Python, PL/Perl, atd.

Tab. 3 – Srovnání databázových RDBMS

3.4.1.4. Prezentační vrstva

Prezentační vrstva je zodpovědná za zpracování HTTP požadavků a odpovědí a pro její implementaci byl vybrán frameworku FastAPI. Tato vrstva se skládá z několika klíčových endpointů, které umožňují interakci s aplikací a poskytují uživatelům přístup k různým funkcím. Ideálně by vrstva neměla obsahovat žádnou business logiku.

Tato vrstva také zahrnuje validaci vstupních dat a autentizaci uživatelů, aby byla zajištěna správnost a bezpečnost zpracovaných požadavků.

3.4.1.4.1. Schématická vrstva

Jako podvrstva prezentační vrstvy (a zároveň prostředník mezi prezentační a aplikační vrstvou) byl implementován datový model (DTO) pomocí knihovny Pydantic [24][24ten blog]. Mezi její hlavní výhody patří snadná validace a serializace dat, jejíž součástí je i možnost automatické konverze mezi typy (respektive její vypnutí). Knihovna je také snadno integrovatelná s FastAPI, které používáme.

Datové modely Api jsou rozděleny do tří celků: Auth, Project a Images (pro autorizaci, projekty a obrázky s anotacemi). Každý celek má své specifické modely, které jsou přizpůsobeny konkrétním potřebám dané části aplikace.

Auth:

Tento celek zahrnuje modely související s autorizací uživatelů, jako jsou uživatelské účty, přihlašovací přístupy apod.

- **Token**

Reprezentuje strukturu odpovědi na vytvoření JWT tokenu s atributy samotného stringu tokenu a jeho typu (typicky "bearer").

```
class Token(BaseModel):
    access_token: str
    token_type: str
```

Kód 9 – Pydantic model pro JWT token

- **TokenData**

Reprezentuje strukturu, kdy podle tokenu mapujeme uživatele, který je zde reprezentován pomocí již zmíněného unikátního `username`, defaultně `None`.

```
class TokenData(BaseModel):
    username: str | None = None
```

Kód 10 – Pydantic model pro namapování uživatele z tokenu

- **UserBase**

Model představuje základní informace o uživateli s atributy unikátního identifikátoru (ID), unikátního username a zda je uživatel aktivní, v budoucnu je zde prostor i pro přidána rolí, k ovládání oprávnění jednotlivých uživatelů.

```
class UserBase(BaseModel):
    id: int
    username: str
    active: bool
```

Kód 11 – Pydantic model pro uživatele

Images:

Tento celek zahrnuje modely pro správu obrázků a anotací, které umožňují ukládání, načítání a zpracování obrazových dat a jejich anotací.

- **ImageBase**

Model obrázku se základními informacemi jako unikátním ID, názvem, cestou a typem v datasetu.

```
class ImageBase(BaseModel):
    id: int
    name: str
    path: str
    split_type: str
```

Kód 12 – Pydantic model pro základní informace o obrázku

- **ImageContent**

Reprezentuje konkrétní obrázek. Dědí všechny atributy z ImageBase a přidává navíc obsah obrázku zakódovaný ve formátu Base64 společně s velikostí souboru.

```
class ImageContent(ImageBase):
    src: str
    size: int
```

Kód 13 – Pydantic model pro obsah obrázku

- **AnnotationBase**

Reprezentuje anotace s atributy `id`, `class_id`, `image_id` a `values`. Atribut `values` představuje `string` čísel oddělených mezerou, jeho podoba závisí na typu anotace – pro polygony zahrnuje řetězec x,y souřadnic jednotlivých bodů, pro bounding boxy obsahuje x,y souřadnice levého horního vrcholu, šířku a výšku.

```python
class AnnotationBase(BaseModel):
    id: int
    class_id: int
    image_id: int
    values: str
```

Kód 14 – Pydantic model pro anotaci

Projects:

Celek obsahuje modely, které reprezentují projekt s metadaty a anotační třídy.

- **ProjectBase**

Reprezentuje projekt a jeho atributy.

```python
class ProjectBase(BaseModel):
    id: int
    name: str
    annotation_type: str
```

Kód 15 – Pydantic model pro projekt

- **ClassBase**

Reprezentuje anotační třídu v projektu.

```python
class ClassBase(BaseModel):
    id: int
    project_id: int
    label: str
    color: str
```

Kód 16 – Pydantic model pro anotační třídu

3.4.1.4.2. Endpoints

Jednotlivé endpointy byly rozděleny do funkčních celků Authentication, Image, Annotation, Project, Classes (pro autorizaci, správu obrázeků, anotací, projektů a anotačních tříd) viz. Tab. 4. V následujících podkapitolách je vždy rozepsána jejich úloha, požadované a volitelné parametry, implementační detaily a očekávaná odpověď, kterou chceme posílat na uživatelské rozhraní.

Celek	Typ Endpoitu	Název endpoitu
Authentication	POST	/auth/token
	GET	/auth/user/me

Image	GET	/image/project
	GET	/image/content
Project	GET	/project/all
	POST	/project/create
	POST	/project/train_model
	POST	/project/get_models
	POST	/project/predict
Annotation	GET	/annotation/image
	PUT	/annotation/update
	DELETE	/annotation/delete
	POST	/annotation/create
Classes	GET	/class/project
	POST	/class/create

Tab. 4 – Seznam endpointů

Auth Endopoints:

- **/auth/token**

Endpoint slouží pro autentifikaci uživatele a získání přístupového tokenu (access token). Parametrem requestu jsou přihlašovací jméno a heslo, které jsou předány prostřednictvím formuláře OAuth2PasswordRequestForm. Vrací JWT token v objektu typu Token, sloužící k identifikaci uživatele při ostatních požadavcích. Dotazuje se na servicu create_access_token vygeneruje JWT token sloužící k autentifikaci uživatele.

Jedná se o jediný přístupový endpoint, který nevyžaduje v hlavičce požadavku platný přístupový token, všechny další endpointy ho požadují, jinak vrací 401, neautorizováno.

- **/auth/user/me**

Slouží pro získání informací o aktuálně autorizovaném uživateli. Endpoint vrací objekt typu User, který obsahuje detaily o uživateli (viz Kap. 3.4.1.4.1). Dotazuje se na funkci get_current_user, která z přiloženého tokenu extrahuje a ověřuje identitu uživatele.

`Funce user_me(Annotated[UserBase, Depends(get_current_active_user)])` využívá dependency injection, kde current user se získá voláním `get_current_active_user`.

```python
@auth_router.get("/user/me/", response_model=UserBase, tags=["Authentication"])
async def user_me(current_user: Annotated[UserBase, Depends(get_current_active_user)],):
    return current_user
```

Kód 17 – Implementace endpointu /user/me

Project Endpoints:

- **/project/all**

Tento endpoint slouží k získání všech projektů, které patří aktuálně autorizovaném uživatel, a tedy vrací seznam objektů typu `ProjectBase`, kde každý objekt obsahuje informace o projektu (viz Kap. 3.4.1.4.1). Dotazuje se na servisu `get_projects`, která na základě přiloženého tokenu extrahuje a ověřuje identitu uživatele.

- **/project/create**

Slouží k vytvoření nového projektu pro aktuálně autorizovaném uživatele. Parametry requestu jsou `name` a `annotation_type`, které definují název projektu a typ anotace. Endpoint vrací jako odpověd objekt typu `ProjectBase`, který obsahuje detaily o nově vytvořeném projektu.

- **/project/train_model**

Tento endpoint slouží k zahájení tréninku modelu pro specifikovaný projekt, který patří aktuálně autentifikovanému uživateli. Parametr požadavku je `project_id`, který identifikuje projekt. Endpoint volá službu `train_model`, která zahájí proces tréninku.

- **/project/get_models**

Tento endpoint slouží k získání seznamu modelů, které byly natrénovány pro specifikovaný projekt, který patří aktuálně autentifikovanému uživateli. Parametr požadavku je `project_id`, který identifikuje projekt. Endpoint vrací seznam modelů.

- **/project/predict**

Tento endpoint slouží k provedení predikce pomocí natrénovaného modelu na specifikovaném projektu, který patří aktuálně autentifikovanému uživateli. Parametr requestu je `project_id`, který identifikuje projekt. Jak bylo zmíněno v servise, vrací pouze prázdný list.

Classes:

- **/class/project**

Tento endpoint slouží k získání všech anotačních tříd pro daný projekt, který patří aktuálně autentifikovanému uživateli. Tyto anotační třídy jsou definované ve složce projektu v souboru _project.yaml a je možno je odtud měnit (viz Kap. 3.4.1.1 a Kap. 3.7.3. Endpoint vrací seznam objektů typu `ClassBase`, kde každý objekt obsahuje informace o třídě (viz Kap. 3.4.1.4.1). Dotazuje se na servisu `get_classes`, která na základě přiloženého tokenu extrahuje a ověřuje identitu uživatele a identifikaci projektu.

- **/class/create**

Tento endpoint slouží k vytvoření nové třídy (class) v rámci projektu pro aktuálně autentifikovaného uživatele. Parametry requestu jsou `project_id`, `label` a `color`, které definují projekt, název a barvu třídy. Endpoint vrací 201 objekt typu `ClassBase`, který obsahuje detaily o nově vytvořené třídě.

Image Endpoints:

- **/image/project**

Vypsání listu všech obrázků v daném projektu s názvem, cestou a typem obrázku v datasetu (např. trénovací, evaluační). Implementačně volá servicu `get_images(current_user, project_id)` s aktuálně přihlášeným uživatelem a projektovým ID. Vrací jako odpověď `List[ImageBase]`

```
[
    {
        "id": 1,
        "name": "Image1",
        "path": "/images/image1.jpg",
        "split_type": "train"
    },
    {
        "id": 2,
        "name": "Image2",
        "path": "/images/image2.jpg",
        "split_type": "val"
    }
]
```

Kód 18 – Příklad odpovědi z endpointu /image/project

- **/image/content**

Endpoint slouží pro získání obsahu konkrétního obrázku. Implementačně volá funkci `get_image_content(current_user, image_id)`. Funkcionalita zahrnuje načtení obrázku z databáze, přečtení jeho obsahu, zakódování v base64 a vrácení spolu s metadaty obrázku. Vrací objekt typu `ImageContent`.

```
{
    "id": 1,
    "name": "Image1",
    "path": "/images/image1.jpg",
    "split_type": "train",
    "src": "Base64EncodedImageContent",
    "size": 1024
}
```

Kód 19 – Příklad odpovědi z endpointu /image/content

Annotation Endpoints:

- **/annotation/image**

Endpoint slouží pro získání seznamu anotací spojených s konkrétním obrázkem. Tento endpoint vyžaduje autorizování uživatele, jehož identita je zajištěna prostřednictvím „dependency injection“. Parametrem požadavku je `image_id`, což je ID obrázku, pro který se mají získat anotace. Implementace dotazuje službu `get_image_annotations(current_user, image_id)`, která provede dotaz na databázi a vrátí seznam anotací spojených s daným obrázkem. Odpověď je vrácena jako seznam objektů typu `AnnotationBase`, kde každý objekt obsahuje ID anotace, ID třídy, ID obrázku a hodnoty anotace.

```
[
    {
        "id": 1,
        "class_id": 1,
        "image_id": 1,
        "values": "some values"
    },
    {
        "id": 2,
        "class_id": 2,
        "image_id": 1,
        "values": "other values"
    }
]
```


Kód 20 – Příklad odpovědi z endpointu /annotation/image

- **/annotation/update**

Endpoint slouží pro aktualizaci existující anotace s novými hodnotami. Parametrem požadavku je `annotation_id`, což je ID anotace, kterou je třeba aktualizovat. Možné volitelné parametry zahrnují `image_id` pro nové ID obrázku, `class_id` pro nové ID třídy, a `values` pro nové hodnoty anotace. Implementace volá službu `update_annotation(current_user, annotation_id, image_id, class_id, values)`, která aktualizuje specifikované pole v databázi a validuje nové hodnoty, pokud jsou poskytnuty. Odpověď je vrácena jako objekt typu `AnnotationBase`, který obsahuje ID anotace, ID třídy, ID obrázku a aktualizované hodnoty anotace.

- **/annotation/create**

Endpoint slouží pro vytvoření nové anotace pro konkrétní obrázek. Vyžaduje autorizovaného uživatele, jehož identita je zajištěna prostřednictvím dependency injection. Parametry zahrnují `image_id`, což je ID obrázku, pro který se vytváří nová anotace, `class_id` pro ID třídy anotace, a `values` pro hodnoty anotace. Implementace volá službu `create_annotation`, která vytvoří novou anotaci v databázi a propojí ji s daným ID obrázku a třídy. Odpověď je vrácena jako objekt typu `AnnotationBase`, který obsahuje ID nové anotace, ID třídy, ID obrázku a hodnoty anotace.


```
{
    "id": 1,
    "class_id": 1,
    "image_id": 1,
    "values": "0.2 0.5 0.05 0.05"
}
```


Kód 21 – Příklad odpovědi z endpointu /annotation/create

- **/annotation/delete**

Endpoint slouží pro odstranění existující anotace. Tento endpoint vyžaduje autorizovaného uživatele, jehož identita je zajištěna prostřednictvím dependency injection. Parametrem požadavku je `annotation_id`, což je ID anotace, kterou je třeba smazat. Implementace volá službu `delete_annotation(current_user, annotation_id)`, která provede odstranění specifikované anotace z databáze.

Vrací potvrzení o úspěšném odstranění anotace, zpravidla jako jednoduchý status (např. "success" nebo prázdný JSON objekt `{}`).

3.4.1.5. Aplikační vrstva

Pro aplikaci byly navrženy následující funkce zajišťující veškorou logiku mezi prezentační vrstvou a databází. Rozděleny jsou podle účelu na celky Auth, Project, Training a Image (pro autorizační funkce, správu projektů, funkce asistovaného anotování a správu obrázků).

Auth:

- **Vytvoření uživatele**

Funkce `create_user(username, password, active)` vytváří nového uživatele s uvedeným uživatelským jménem, heslem a údajem, za jde účet aktivní.

Využívá pomocnou funkci `get_password_hash(password)`, která převádí zadané heslo na hashovaný formát pomocí BCRYPT algoritmu [25bcrypt]. Tento hash je poté uložen v databázi místo samotného hesla a hash vrátí, aby byl spolu s uživatelem přidán do databáze. Pokud uživatelské jméno již existuje, vrátí chybu, jinak vrátí uživatele typu User z Pydantic.

- **Získání uživatele**

K účelu získání uživatele bylo implementováno více funkcí.

Funkce `get_user(username: str)` hledá v databázi podle uživatelského jména a vrací uživatelský objekt. Pokud uživatel neexistuje, vrátí `None`.

Funkce `get_current_user(token: Annotated[str, Depends(oauth2_scheme)]` extrahuje uživatele z JWT tokenu. Token je injektován pomocí dependency injection, kde FastApi používá `Depends(oauth2_scheme)`. Ověřuje platnost tokenu a získává uživatele ze systému na základě dat obsažených v tokenu. Pokud token není platný nebo uživatel neexistuje, vrací chybu.

Funkce `get_current_active_user(current_user: Annotated[UserBase, Depends(get_current_user)])` ověřuje, zda je aktuálně autentifikovaný uživatel aktivní. Pokud uživatel není aktivní, vrací chybu. Používá se k zajištění, že uživatelé, kteří se snaží přistupovat k chráněným zdrojům, mají aktivní účty.

- **Autentizace uživatele**

Funkce `authenticate_user(username: str, password: str)` ověřuje, zda existuje uživatel s daným uživatelským jménem a zda heslo odpovídá hashovanému heslu uloženému v databázi. Vrátí uživatelský objekt, pokud jsou přihlašovací údaje správné nebo `False`. Využívá pomocou funcki `verify_password(plain_password, hashed_password)`, která ověřuje, zda zadané heslo odpovídá zašifrovanému heslu v databázi. Používá na to knihovnu `passlib` a její kontext pro ověření hesla, což zajišťuje bezpečné porovnání hesla s jeho hashovanou verzí.

- **Vytvoření přístupového tokenu**

Funkce `create_access_token(data: dict, expires_delta: timedelta | None = None)` pro vytvoření JWT pro autentifikaci uživatele. Token obsahuje údaje o uživateli a expirační čas, který je předáván z prezentační vrstvy, případně pokud není specifikován tak se automaticky nastaví 15 minut. Vrací JWT token

Project:

- **Vytvoření anotační třídy**

Tato funkce `create_class(current_user, project_id, label, color)` vytváří a vrací novou anotační třídu v rámci specifikovaného projektu. Funkce přijímá parametry pro ID projektu, popis třídy a barvu. Nová třída je přidána do databáze a uložena. Funkce vrací vytvořenou třídu.

- **Vytvoření projektu**

Tato funkce `create_project(current_user, name, annotation_type)` vytváří nový projekt (typ `Project`). Přijímá název projektu a typ anotace. Nový projekt je přidán do databáze a uložen. Funkce vrací vytvořený projekt.

- **Získání projektů**

Tato funkce `get_projects(current_user)` vrací seznam všech projektů v databázi. Funkce je implementována s parametrem `current_user` pro budoucí vývoj a omezení přístupu a rozdělení oprávnění např. jen pro určité projekty.

- **Získání anotačních tříd v projektu**

Tato funkce `get_classes(current_user, project_id)` vrací seznam všech tříd (`Class`) asociovaných s konkrétním projektem. Funkce filtruje třídy podle ID projektu a vrací výsledky z databáze. Tato funkce umožňuje získat všechny třídy, které jsou součástí daného projektu.

Training:

- **Trénování modelu**

Následující funkce funkce `train_model(current_user, project_id)` spouští trénování modelu jako samostatný proces, pokud není jiný aktuálně běžící trénink. Volá nejprve funkci `export_data(project_id)` pro export datasetu do specifické struktury kompatibilní s YOLO. Ta se liší především rozdělením dat do validačních a trénovacích složek („val“ a „train“) a strukturou konfiguračního projektového souboru data.yaml, který tato funkce také generuje.

Poté je využita pomocná funkce `train(project_id, config_path, annotation_type)`, která na základě typu projektu inicializuje YOLO model a předá cestu k projektovému souboru data.yaml. Poznámka.: Nastavení trénování je v současnosti možné pouze u funce train viz. Kód 22.

Původní fuknce `train_model` vrací status a zprávu, že proces začal, nebo že aktuálně běží jiný trénink.

```python
def train(project_name, config_path, annotation_type):
    if annotation_type == "poly":
        model = YOLO("yolov8n-seg.pt")
    else:
        model = YOLO("yolov8n.pt")
    config_abs_path = os.path.abspath(config_path)
    try:
        os.mkdir("temporary")
    except:
        pass
    os.chdir("temporary")
    model.train(data=config_abs_path, project=str(project_name), epochs=4, batch=4, imgsz=640)
```

Kód 22 – Funkce train a nastavení tréninku modelu (poslední řádek)

- **Vrácení modelů**

Tato funkce `get_models(current_user, project_id)` vyhledá dostupné modely z MLflow uložené v systému pro specifikovaný projekt. Vrací seznam cest k modelům (`{"models": [model_paths]}`).

- **Predikce anotací**

Tato funkce `get_prediction(current_user, project_id)` spouští samostatný proces pro provedení predikce pomocí pomocné funkce `predict(current_user, project_id)` a vrací prázdný slovník. Predikci provádí na základě aktuálně nejnovějšího dostupného modelu, ten získá pomocí `get_models(current_user, project_id)`.

Načítá aktuální dataset a jeho anotace a následně u obrázků, které nemají určený typ (split type) jako trénovací nebo anotační, pomocí volání API provede smazání starých anotací a vytvoří nové anotace na základě výsledků predikce ve formátu, který odpovídá Pydantic modelu AnnotationBase.

Image:

- **Vytvoření obrázku**

Funkce `create_image` vytváří záznam obrázku a ukládá soubor obrázku na disk. Přijímá následující parametry: aktuálního uživatele (current_user), ID projektu (project_id), binární obsah obrázku (content), název souboru (name), volitelný MIME typ (mime) a typ datového rozdělení (split_type). Pokud MIME typ není specifikován, odhadne se na základě názvu souboru. Funkce vytvoří složku pro ukládání obrázků, pokud neexistuje, uloží obsah obrázku

do souboru a vytvoří záznam v databázi s informacemi o obrázku. Nakonec vrátí objekt obrázku. Pokud se vytvoření obrázku nezdaří, vyvolá HTTPException.

- **Získání přehledu obrázků z datasetu**

Funkce `get_images(current_user, project_id)` získává všechny obrázky pro daný projekt. Přijímá následující parametry: aktuálního uživatele (current_user) a ID projektu (project_id). Vrací dotazovací objekt pro záznamy obrázků patřící k danému projektu. Pokud projekt nebo obrázky nelze nalézt, vyvolá HTTPException.

Funkce `get_image_content(current_user, image_id)` získává obsah konkrétního obrázku. Přijímá následující parametry: aktuálního uživatele (current_user) a ID obrázku (image_id). Funkce načte obrázek z databáze, a pokud obrázek neexistuje, vyvolá HTTPException s kódem 404. Poté otevře soubor obrázku, načte jeho binární obsah, zakóduje jej do formátu base64 a přidá tyto informace do objektu obrázku spolu s velikostí obsahu. Funkce vrátí aktualizovaný objekt obrázku.

- **Získání anotací dle obrázku**

Funkce `get_image_annotations(current_user, image_id)` získává všechny anotace pro konkrétní obrázek. Přijímá následující parametry: aktuálního uživatele (current_user) a ID obrázku (image_id). Vrací dotazovací objekt pro záznamy anotací spojené s daným obrázkem. Pokud obrázek nebo anotace nelze nalézt, vyvolá HTTPException.

- **Vytváření anotací**

Funkce `create_annotation` vytváří novou anotaci pro konkrétní obrázek. Přijímá následující parametry: aktuálního uživatele (current_user), ID obrázku (image_id), ID třídy (class_id) a hodnoty anotace (values). Pokud obrázek nebo třída neexistuje, vyvolá HTTPException s kódem 404. Funkce vytvoří nový záznam anotace v databázi a přiřadí jej k uživateli, který ji vytvořil. Nakonec vrátí objekt anotace.

- **Změna anotace**

Funkce `update_annotation (current_user, annotation_id, image_id, class_id, values)` aktualizuje existující anotaci. Přijímá následující parametry: aktuálního uživatele (current_user), ID anotace (annotation_id), volitelné ID obrázku (image_id), volitelné ID třídy

(class_id) a hodnoty anotace (values). Pokud anotace neexistuje, vyvolá HTTPException s kódem 404. Pokud je poskytováno nové ID obrázku nebo třídy, funkce ověří jejich existenci a aktualizuje příslušné pole anotace. Nakonec uloží změny do databáze a vrátí aktualizovaný objekt anotace.

- **Odstranění anotace**

Funkce `delete_annotation` smaže existující anotaci. Přijímá následující parametry: aktuálního uživatele (current_user) a ID anotace (annotation_id). Pokud anotace neexistuje, vyvolá HTTPException s kódem 404. Funkce odstraní záznam anotace z databáze a uloží změny. Nakonec vrátí smazaný objekt anotace.

3.5. MLflow

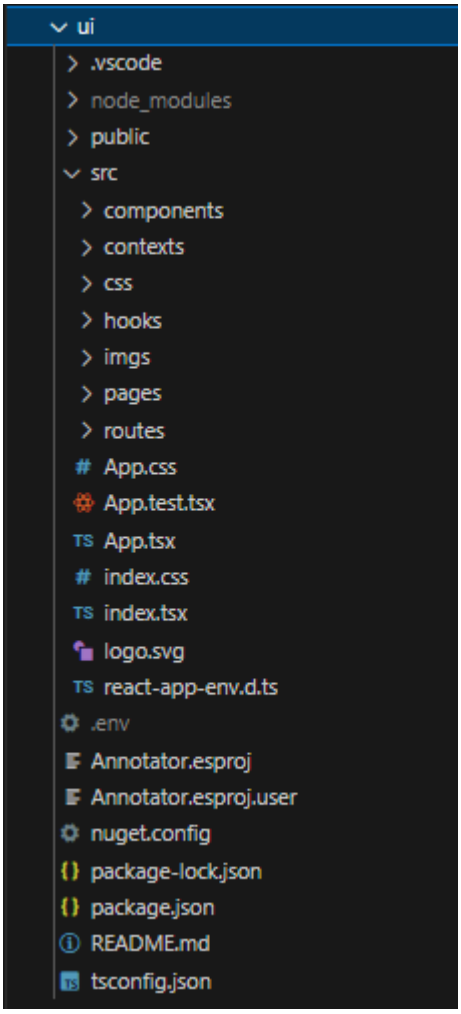
K správné funkci trénování modelů a predikcí anotací je potřeba spustit prostředí MLflow, a to s pomocí souboru `run_models.py`, kde jsou definované všechny parametry serveru. Tyto parametry jsou přebírány z konfiguračního souboru `.env`, jehož plné nastavení je popsáno v Kap 3.7.3 Prostředí slouží ke správě modelů a tréninků, případně i pro zobrazení metrik apod.

3.6. Vypracování uživatelského rozhraní

Uživatelské rozhraní bylo implementováno s využitím JavaScriptové knihovny React, která mi přišla výhodná pro svůj komponentový přístup, práci se stavem aplikace a daty, dostupné knihovny a možnost použití TypeScriptu. Typescript je nadstavba JavaScriptu, která jej rozširuje o statické typování, možnost používání rozhraní a další. Je kompilován do JavaScriptu, ale díky typové kontrole má potenciálně méně chyb za běhu, což jsem vnímala jako velkou výhodu. V následujících kapitolách jsou podrobněji popsány jednotlivé aspekty návrhu a implementace, včetně technických detailů a konkrétních řešení použitých při vývoji.

Adresářová struktura frontedové části aplikace je vidět na Obr. 14, celkem je aplikace rozdělena na celky pages (stránky), components (komponenty), routes (směrování), hooks (vlastní funkce umožňující opětovné použití logiky mezi různými komponentami nebo s různými parametry)

a css, neboli vzhled aplikace. Celkem byly vytvořeny 4 stránky – Login, Dashboard, Image a NotFound, které budou dále popsány.



Obr. 14 – Adresářová struktura uživatelského rozhraní

3.6.1. Směrování

Výchozím bodem aplikace je soubor App.tsx viz. Kód 23. Je zde vidět již uvedená struktura aplikace, kde pro jednotlivé komponenty jsou definovány URL cesty sloužící k navigaci aplikací.

```
// imports

function App() {
    return (
        <BrowserRouter>
            <ProjectProvider>
                <AuthProvider>
                    <div className="App">
                        <header className="header">
                            <Navbar/>
                        </header>
                        <main className="main-content">
                            <Routes>
                                <Route path="/login" element={<Login />} />
                                <Route index element={<Dashboard />} />
                                <Route path="/image/:id" element={<Image />} />
                                <Route path="*" element={<NotFound />} />
                            </Routes>
                        </main>
                    </div>
                </AuthProvider>
            </ProjectProvider >
        </BrowserRouter>
    );
}

export default App;
```

Kód 23 – Výchozí soubor aplikace

3.6.2. Použité React Hooks

Ve většině komponent byly použity React hook `useState` a `useEffect`, `useContext` a dále jsem využila `useRef`, `useNavigation` a `useLocation` (poslední dva jsou součástí součástí knihovny react-router-dom, ne přímo základní React Hooks) a zde uvádím jejich funkci v kontextu s mou aplikací.

useState

Zásadní pro spravování vnitřního stavu ve funkcionálních komponentách. Při změně hodnoty stavové proměnné je vyvolán rerender. Hodnota stavové proměnné se vždy projevuje až v následujícím renderu. V Kód 24 je příklad takového použití ve funkci `SignIn` pro přihlášení.

```
const [error, setError] = useState<number | null>(null); // inicializace stavu „value”

const SignIn = (params:any) => { // zvýšení „value“ o 1 při kliknutí na tlačítko
  setError(null)

  /* Kód pro Api call -> uložení odpovědi do 'const response' */

  if(!response.json().ok) {
    setError(“Error signing in”) // nastavení nové hodnoty
  }
```

Kód 24 – Příklad použití hooku useState

useEffect

Je to hook v Reactu, který umožňuje provádění tzv „vedlejších efektů“ v komponentách funkcionálního typu. Tento hook je důležitý pro správu operací, které přesahují synchronní vykreslování, jako jsou například asynchronní volání API, manipulace s DOM, přidání a odebrání event listenerů apod. Funkce hooku `useEffect` přijímá dva parametry: Funkci kterou má zavolat a závislé pole hodnot, kdy má být funkce zavolána, v případě, že má být volána pouze jednou při zavedení, je pole prázdné. Na Kód 25 je příklad použití hooku `useEffect` pro načítání obrázku z API při změně id.

```
const [id, setId] = useState<number|null>(null)
useEffect(() => {
        handleLoadingImage();
    }, [id]);
```

Kód 25 – Příklad použití hooku useEffect

useContext:

Tento hook slouží ke sdílení globálních dat z kontextu, který byl vytvořen pomocí `React.createContext()` a jeho funkci a implementaci v aplikaci podrobněji popíšu v Kap. 26. Umožňuje komponentám číst hodnoty kontextu bez nutnosti předávat vše z komponenty na komponentu („prop-drilling“).

useRef:

Hook vytváří měnitelnou referenci, která přežívá mezi rendrovacími cykly komponenty, často používané jako reference na DOM element. Na ukázce Kód 26 je příklad takového použití.

```
import { useRef } from 'react';

function MyComponent() {
  const inputRef = useRef(null);
  return <input ref={inputRef} />;
}
```

Kód 26 – Použití hooku useRef

useNavigation a **useLocation**:

Tyto hooky pochází z knihovny React Router. Hook `useNavigation` v aplikaci používám k přesměrování na požadovanou URL a `useLocation` vrací aktuální URL aplikace, včetně cesty, dotazových parametrů a hash. Tyto hooky usnadňují práci s navigací a sledováním změn v URL.

3.6.3. Kontext

Pro udržování globálního stavu projektu a autentizace uživatele jsem využila tzv. React Context. Projektový slouží k uchování ID projektu a typu projektu napříč aplikací, jeho definice je v Kód 27. Inicializace a nastavení probíhá pomocí ProjectProvideru, který obaluje celou aplikace v App komponentě (Kód 23). Implementace ProjectProvideru je vidět na ukázce Kód 28, při každé změně ID projektu nebo jeho typu ho znovu rendruji. Použití tohoto kontextu v libovolné komponentě projektu je vidět na poslední ukázce KOD 29.

Obdobně jsem si vytvořila autentizační kontext, jehož obsah je vidět v Kód 30 a stejným principem je definována jeho inicializace a nastavení v AuthProvideru, kde jsou především definované `SingIn` a `SingOut` funkce, které vyžívám při přihlašování a odhlašování komponentě. Navíc u autorizačního kontextu používám navíc řádku `const useSession = (): AuthContextData => useContext(AuthContext)` a poté používám v ostatních komponentách zkrácené `useSession`.

```
import { createContext, useContext } from 'react';

export type ProjectContextData = {
    projectId: number | null;
    setProjectId: (p: number | null) => void
    projectType: string | null;
    setProjectType: (pt: string | null) => void
}

// initial value
export const ProjectContext = createContext<ProjectContextData>({
    projectId: null,
    setProjectId: (_value: number | null) => { },
    projectType: null,
    setProjectType: (pt: string | null) => { }
})

const useProject = (): ProjectContextData => useContext(ProjectContext);

export default useProject;
```

Kód 27 – Definice projektového kontextu

```
import { ReactNode, useEffect, useState } from 'react'
import { ProjectContext } from './ProjectContext'


const ProjectProvider: React.FC<{ children: ReactNode }> = ({ children }) => {;
    const [projectId, setProjectId] = useState<number | null>(() => {
        const storedProjectId = localStorage.getItem('projectType');
        return storedProjectId !== null ? Number(storedProjectId) : null; // Number(null) is 0
    });
    const        [projectType,        setProjectType]        =        useState<string        |
null>(localStorage.getItem('projectType'));

    useEffect(() => {
        if (projectId !== null) {
            localStorage.setItem('projectId', projectId.toString());
        } else {
            localStorage.removeItem('projectId');
        }
    }, [projectId]);

    useEffect(() => {
        if (projectType !== null) {
            localStorage.setItem('projectType', projectType);
        } else {
            localStorage.removeItem('projectType');
        }
    }, [projectType]);


    return (
        <ProjectContext.Provider value={{ projectId, setProjectId, projectType, setProjectType }}>
            {children}
        </ProjectContext.Provider>
    );
};

export default ProjectProvider;
```

Kód 28 – Podoba ProjectProvideru

```
const { projectId, projectType } = useContext(ProjectContext);
```

Kód 29 – Použití kontextu

```
export type User = {
    id: number;
    username: string;
    active: boolean;
    // roles: string[]; TODO:implementace BE
}

export type SignInCredentials = {
    username: string;
    password: string;
}

export type AuthContextData = {
    user?: User;
    token?: string | null;
    isAuthenticated: boolean;
    loadingUserData: boolean;
    signIn: (credentials: SignInCredentials) => Promise<void> | Error;
    signOut: () => void;
    getUserData: () => Promise<void>;
}
```

Kód 30 – Definice autorizačního kontextu

Vznikají tím takto vlastní hooky, které jsou využívány v komponentách, které vždy daná data potřebují. Dále již jejich použití nebudu v jednotlivých komponentách uvádět. Obecně se jedná zejména o využití využití při volání API, nebo například když z API obdržím odpověď, 401 – „Unauthorized“ (neautorizováno) a pomocí autorizačního kontextu volám funkci `SingOut` pro odhlášení a odstranění zbytku dat a přesměrování na stránku Login.

3.6.4. Volání API

Většina funkcionalit aplikace je těsně navázaná na backendovou část, a proto důležitou částí bylo implementace API volání správného endpointu s danými parametry. Využita k tomu byla funkce fetch, která bere jako parametry URL a případně volitelně objekt obsahující možnosti pro konfiguraci požadavku. Pro dotazování na endopointy bylo použito relativní adresování pomocí proměnné `BASE_URL`, která je nastavitelná v .env souboru.

V ukázce Kód 31 je příklad takového volání použitého pro získání anotačních tříd. Jako URL je definován náš endpoint `/class/project` a v dalších parametrech byl definován typ metody (`method: 'GET'`), v nastavení hlaviček připojen autorizační token (`Authorization: Bearer ${token}`) a specifikaci typu obsahu (`Content-Type: application/json`).

Po volání asynchronní funkce fetch se čeká na odpověď, která se následně zkontroluje. Pokud odpověď není úspěšná, kód ošetří různé stavové kódy: 400 (chybějící ID projektu), 401 (neautorizovaný přístup, který přesměruje na přihlašovací stránku) nebo jiné chyby, které nastaví obecnou chybovou zprávu.

Následuje parsování JSON odpovědi. Pokud je odpověď ve správném formátu (pole), data se uloží do stavové proměnné, kde jsou připravené pro další práci na frontendové části. Pokud formát odpovědi není očekávaný, nastaví se chyba.

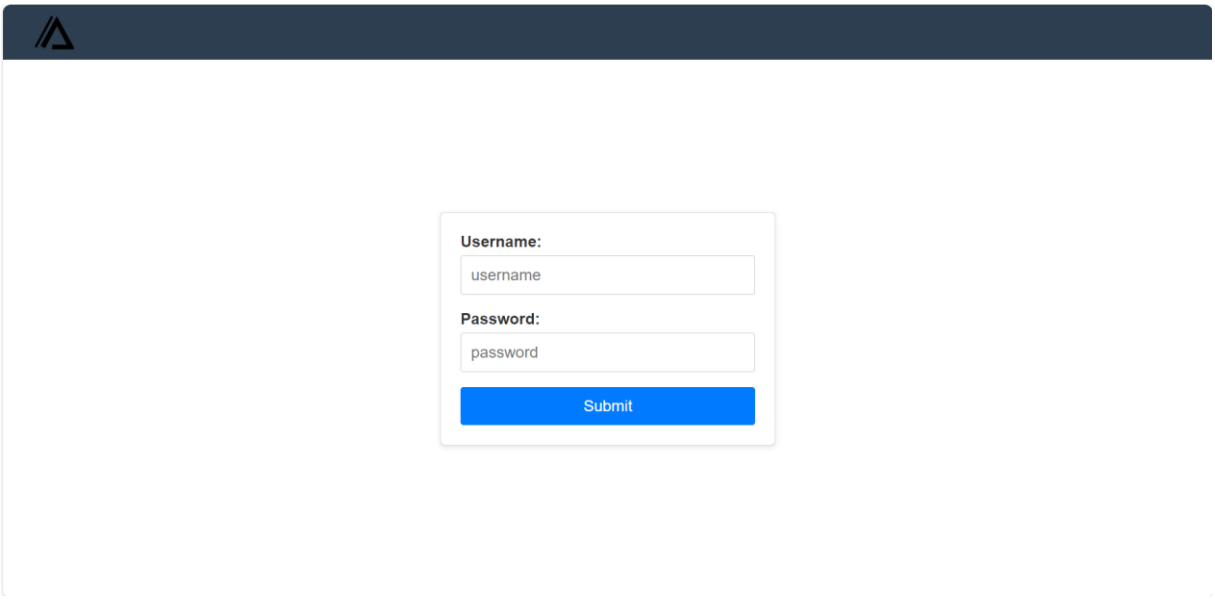
V případě chyby během volání API, `catch` blok ošetří chybu a nastaví odpovídající chybovou zprávu. Celý proces je zakončen nastavením stavu načítání na `false`, což signalizuje, že načítání skončilo.

```
const [responseDataClasses, setResponseDataClasses] = useState <{
        id: number;
        project_id: number;
        label: string;
        color: string
    }[]>([]);

 const handleLoadingClasses = async () => {
        setError(null);
        setLoading(true);

        try {
            const token = localStorage.getItem("token");
            const endpoint = `/class/project?project_id=${projectId}`;
            const url = `${BASE_URL}${endpoint}`;

            const response = await fetch(url, {
                method: 'GET',
                credentials: 'include',
                headers: {
                    'Authorization': `Bearer ${token}`,
                    'Content-Type': 'application/json'
                }
            });

            if (!response.ok) {
                if (response.status === 400) {
                    setError('Missing Project ID');
                } else if (response.status === 401) {
                    setError('Unauthorized');
                    navigate(loginPath);
                } else {
                    setError('Failed to load images');
                }
                throw new Error('Failed to load images');
            }

            const res = await response.json();
            if (Array.isArray(res)) {
                setResponseDataClasses(res);
            } else {
                setError("Response data is not in the expected format");
            }
        } catch (error) {
            if (error instanceof Error) {
                setError(error.message || 'An unknown error occurred');
            } else {
                setError('An unknown error occurred');
            }
        } finally {
            setLoading(false);
        }
    }
```

Kód 31 – Volání API endpointu pro získání anotačních tříd projektu

3.6.5. Login

Probíhá zde autorizace uživatele a v případě úspěchu nastavení autorizačního kontextu. Podoba stránky je vidět na Obr. 15.



Obr. 15 – Stránka přihlášení Login

3.6.5.1. Ověření uživatele

Z diagramu použití je názorně vidět, že pro zobrazení jakéhokoliv obsahu je nutná podmínka přihlášení uživatele. Autentizace uživatele probíhá pomocí JWT tokenu, který je nám na přihlašovací požadavek vygenerován na backendu a poslán v odpovědi. Token jsem se i po zvážením rizika útoků „Cross-Site Scripting“ (XXS) rozhodla schraňovat v `localStorage`, jelikož bude aplikace v současnosti používána spíše lokálně. Tento token je dále použit při každém požadavku na Api, jinak Api vrací 401 – „Unauthorized“ a uživatel je přesměrován na Login stránku.

Po úspěšném přihlášení proběhne ve výše zmíněném autorizačním kontextu aplikace nastavení daného uživatele a dále v komponentě proběhne přesměrování na dashboard s datasety.

3.6.6. Dashboard

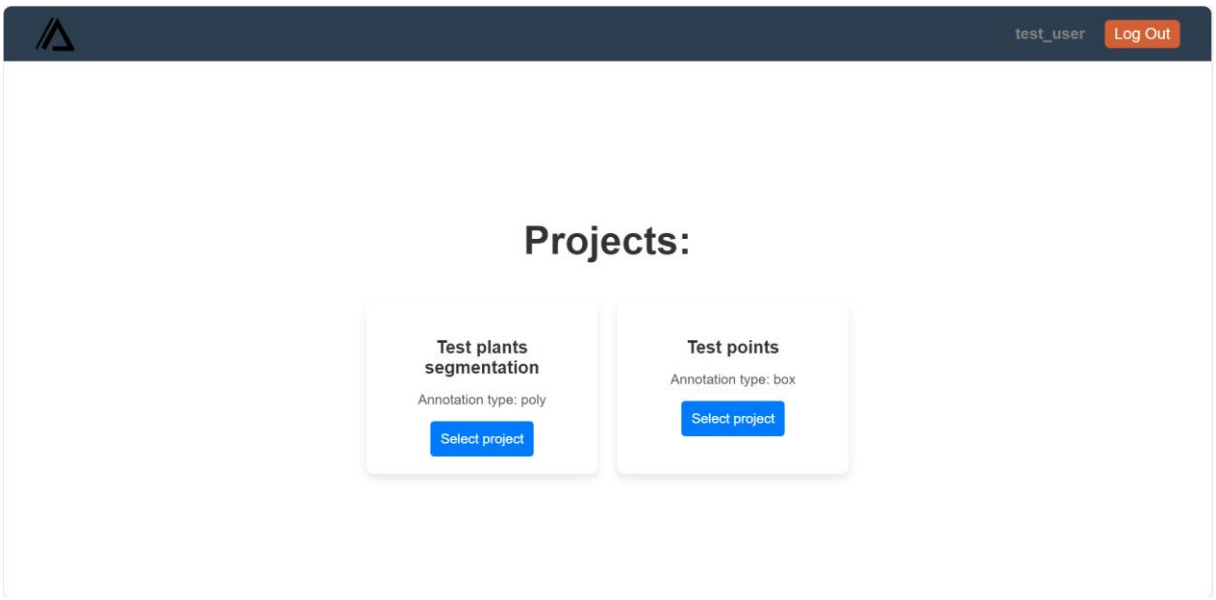
Slouží jako stránka aplikace na zobrazování buď dostupných datasetů pomocí komponenty `DatasetsList`, nebo seznamu obrázků ve zvoleném datasetu pomocí `SelectedDataset`. Výběr správné komponenty je prováděn pomocí atributu `ProjectId`, které je při vstupu na stránku Dashboardu `null`, proto je nejprve zobrazen seznam s datasety a po rozkliknutí vybraného, dojde ke změně `ProjectId` na konkrétní hodnotu a zobrazí se konkrétní dataset.

Také je zde implementována funkce vyskakovací okna `Modal` s funkcemi `openModal` a `closeModal`, se stavovou proměnnou `modalContent`, jejíž setter je předáván potomkům komponenty o úroveň níže, kde je vyskakovací okno naplněno potřebnými daty.

DatasetsList

Komponenta zobrazuje seznam dostupných projektů a jejich typ získaný voláním API endpointu `/project/all`. To má na starosti funkce `handleLoadingProject()`, která je spouštěna pomocí `useEffect` při načtení komponenty (tzv. i při přesměrování zpátky pomocí `useNavigate()`). Obdrženou odpověď z API ukládá do stavové proměnné `responseProject`, nebo pokud se API volání nepovede zajišťuje funkce místo toho nastavení chyby.

Data z `responseProject` jsou podmíněně rendrována a přehledně zobrazeny do dlaždic viz. Obr. 16. Pokud klikneme na projekt proběhne volání funkce `handleProjectSelect`, která aktualizuje ID projektu, podle výběru a dojde k přepnutí na `SelectedDataset`.



Obr. 16 – Obsah komponenty Dashboard

3.6.6.1. Vybraný dataset

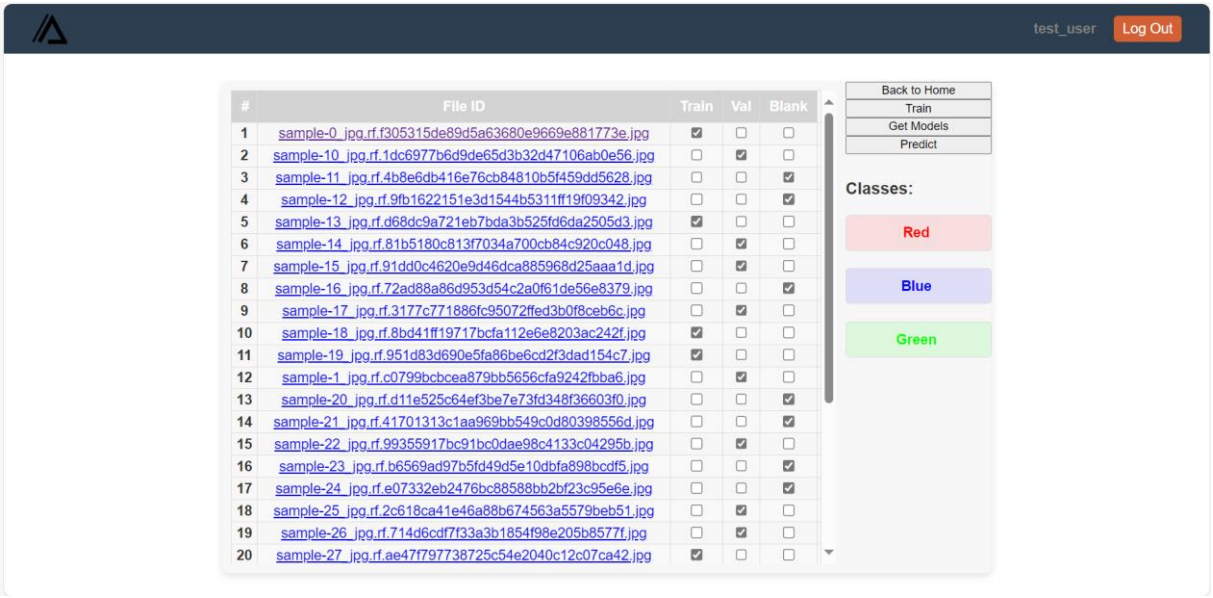
Komponenta slouží jako výchozí bod pro další práci s vybraným datasetem a to především:

- zobrazuje seznam s obrázky datasetu
  - změna určení obrázku

- rozkliknutí vybraného obrázku

- obsahuje kontrolní panel interakce s MLflow pro:
  - trénování modelu,
  - získání aktuálních natrénovaných modelů
  - predikci anotací
- zobrazuje dostupné anotační třídy v datasetu

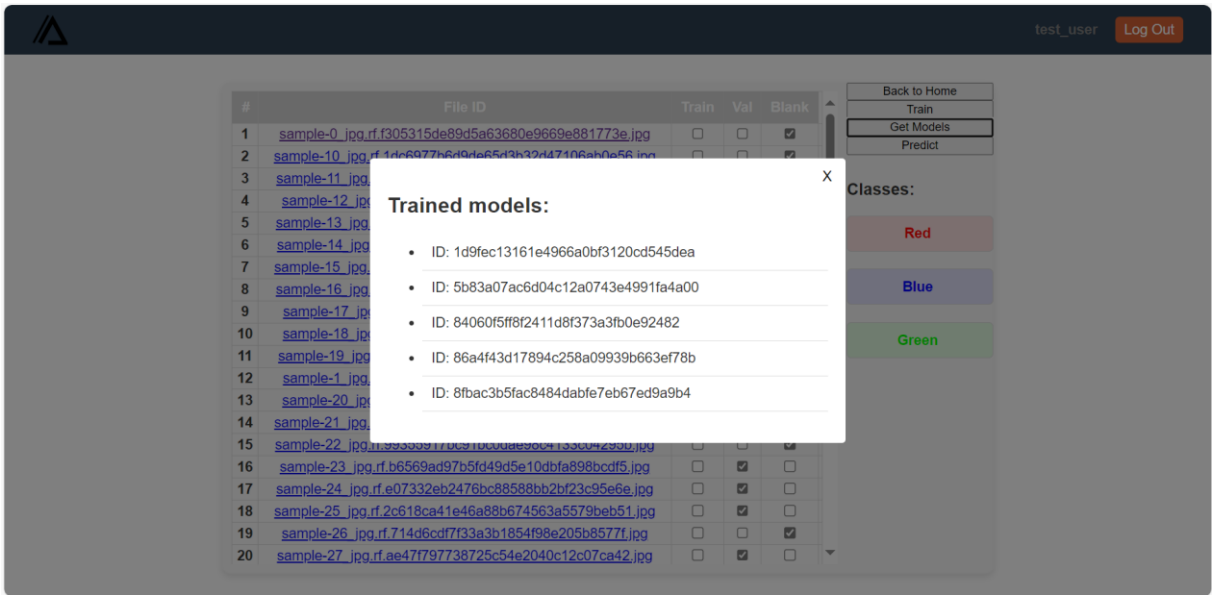
Všechny tyto funkcionality obsluhuje voláním API z endpointů: `/image/project`, `/project/get_models`, `/project/predict`, `/project/train_model`, `/class/project` a níže jsou jednotlivé body podrobněji popsány. Na Obr. 17 je podoba komponenty.



Obr. 17 – Zobrazení komponenty SelectedDataset

Predikce, Trénink, Modely

Funkce predikování, trénování a získání modelů jsou spouštěny vždy pomocí příslušného tlačítka. U tlačítka modelů bylo vytvořeno již zmíněné vyskakovací okno `Modal`, které zobrazuje odpověď z API, tedy obdržené modely, nebo případný error.



Obr. 18 – Zobrazení vyskakovacího okna natrénovaných modelů

Změna určení obrázku a výběr obrázku

Jak si můžeme povšimnout, tak u obrázků je zaškrtnutý tzv. `split_type` (určení obrázku), který můžeme libovolně měnit. Tato funkce je důležitá pro predikci, kdy predikce probíhá u neoznačených obrázků (viz. Kap 3.4.1.5). Pokud chceme zaškrtnou jiný typ, změnu na pozadí vykonává funkce `handleCheckboxChange` s parametry `index` a `newSplitType`, která dosazuje novou hodnotu typu do rendrovací funkce.

Zároveň je možné vybrat konkrétní obrázek z datasetu a kliknutím dojde k přesměrování na stránku `/image` a komponentu Image.

3.6.7. Image

Jedná se o stránku anotování obrázku. Obsahuje tlačítko zpět na seznam projektů (komponenta `DatasetsList`) a komponentu `ShowData`, která zde bude popsána.

Ve třídě `ShowData` a jejích potomcích byla implementována téměř veškerá logika samotné frontendové části aplikace. Komponentu tvoří celky: tabulka anotací `list-annotations`, kontejner s anotovaným obrázkem `imageAnnotator-container`, ovládací panel `image-style-buttons`, které budou níže rozebrány. Navíc bude i popsáno, jak probíhá komunikace s backendem. Z důvodu požadavku na listování mezi obrázky je v aplikaci udržován pojem o

aktuálním indexu obrázku a všech dostupných ID z datasetu. Komponenta momentálně podporuje dva typy anotací – polygony a bounding boxy.

3.6.7.1. Práce s daty

Získávání dat

Byly vytvořeny vlastní hooky `useFetchData` a `useFetchAnnotData` na získávání obrázků a anotací. Tyto funkce volají přijímají jako parametr ID obrázku, volají endpointy `/image/content` a `/annotation/image`, kde z backendu přichází odpověď typu `ImageContent` a `AnnotationBase` a tyto data vrací ve stavových proměnných `responseData` a `responseAnnotData`. Na ukázce Kód 32 je vidět částečná implementace této funkce.

```
/* Custom hook for fetching image data */
const useFetchData = (id: number) => {
    const [errorImg, setErrorImg] = useState<string | null>(null);
    const [loadingImg, setLoadingImg] = useState(false);
    const [responseData, setResponseData] = useState<ResponseData | null>(null);

    const handleLoadingImage = async () => {
        setErrorImg(null);
        setLoadingImg(true);

        try {
            const token = localStorage.getItem("token");
            const endpoint = `/image/content?image_id=${id}`;
            const url = `${BASE_URL}${endpoint}`;

            const response = await fetch(url, {
                method: 'GET',
                credentials: 'include',
                headers: {
                    'Authorization': `Bearer ${token}`,
                    'Content-Type': 'application/json'
                }
            })

            if (!response.ok) {
                /* Omitted code for setting error and loading state varible */
            }

            const res: ResponseData = await response.json();
            if (res) {
                setLoadingImg(false);
                setResponseData(res);
            } else {
                setLoadingImg(false);
                setErrorImg("Response data is not in the expected format");
            }
        }
        catch (err:any) {
            setLoadingImg(false);
            setErrorImg(err.message || "Something went wrong. Please try again later.");
        }
    }

    useEffect(() => {
        handleLoadingImage();
```

```
    }, [id]);

    return { responseData, loadingImg, errorImg };
}
```

Kód 32 – Zkrácená verze funkce useFetchData

Jejich následné použití ve vybrané komponentě je vidět na ukázce Kód 33. Jelikož je `values` datového typu `string` (viz Kap. 3.4.1.4.1 modely) je potřeba data dále v komponentě ShowData parsovat k získání odpovídajících souřadnic, tento proces je popsán níže.

```
    const { responseData, loadingImg, errorImg } = useFetchData(numericId);
    const     {     responseAnnotationData,     loadingAnnotation,     errorAnnotation     }     =
useFetchAnnotData(numericId);
```

Kód 33 – Použití funkcí useFetchData a useFetchAnnotData v komponentě ShowData

Parsování dat

Podle typu projektu (a tím pádem formátu dat) je zvolen způsob parsování a pomocí `useEffect` a funkce `parseBoxAnnotations`, nebo `parsePolyAnnotations` je naplněna proměnná `annotationCoordninatesBox`, nebo `annotationCoordninatesPoly` viz Kód 34. Tyto stavové proměnné slouží jako aktuální stav anotací v uživatelském rozhraní aplikace a pomocí nich jsou vykreslovány obrázky a anotace.

```
const [annotationCoordinatesBox, setAnnotationCoordinatesBox] = useState<{
        id: number;
        //image_id: number;
        class_id: number;
        values: {xRel:number, yRel:number, widthRel:number, heightRel:number}[];
    }[]>([]);
    const [annotationCoordinatesPoly, setAnnotationCoordinatesPoly] = useState<{
        id: number;
        //image_id: number;
        class_id: number;
        values: { xRel: number, yRel: number }[];
    }[]>([]);
```

Kód 34 – Stavová proměná pro aktuální stav anotace (různá pro různý typ projektu)

3.6.7.2. Tlačítka v kontrolním panelu

„Back To Project“

Jedná se o tlačítko zpět, sloužící k návratu na přehled všech dostupných datasetů, které uživatel může zobrazit a upravovat v komponentě DatasetsList.

Přídání nové anotace

Soubor tlačítek nesoucí název vždy podle anotačních tříd datasetu. Přepíná na režim tvorby nové anotace pomocí proměnné `isDrawingMode` dle vybrané anotační třídy. Jakmile je nějaká aktivní je u ní místo názvu nápis „Exit“, kdy zmáčknutí opět vypíná režim anotace.

„Save“

Tlačítko umožňuje uživatelům uložit všechny provedené změny v anotacích. To zahrnuje přidání nových anotací, úpravu existujících anotací, nebo změny v jejich typu. Po zmáčknutí je při zapnutého módu kreslení `isDrawingMode`, který je `true`, pokud zrovna vytváříme novou volána API funkce `fetchCreateAnnotation` pro vytvoření anotace na backendu nebo v opačném případě je postupně odeslána na backend pomocí `handleUpdateAnnotation` stav všech stávajících a tím pádem se všechny změny uloží.

„Delete“

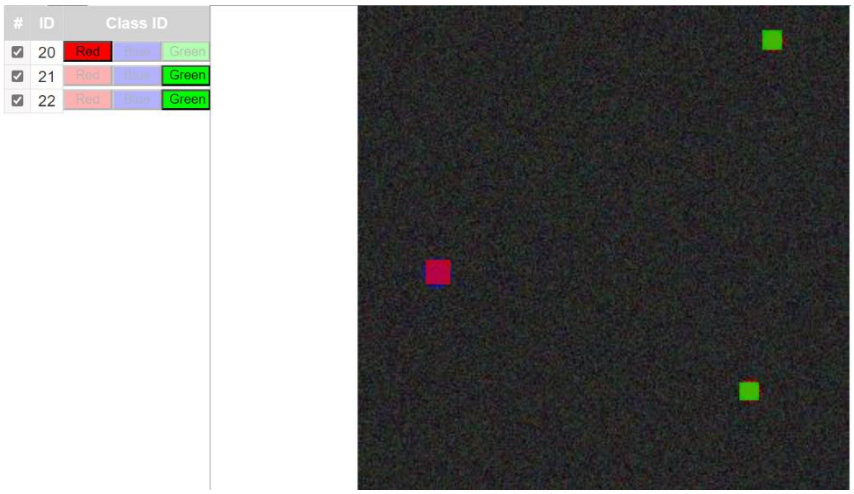
Tlačítko umožňuje smazání anotace a to jak v aktuální stavové proměnné, tak zároveň volá funkci `handleDeleteAnnotation` na odeslání požadavku na backend s konkrétním ID anotace.

„Revert“

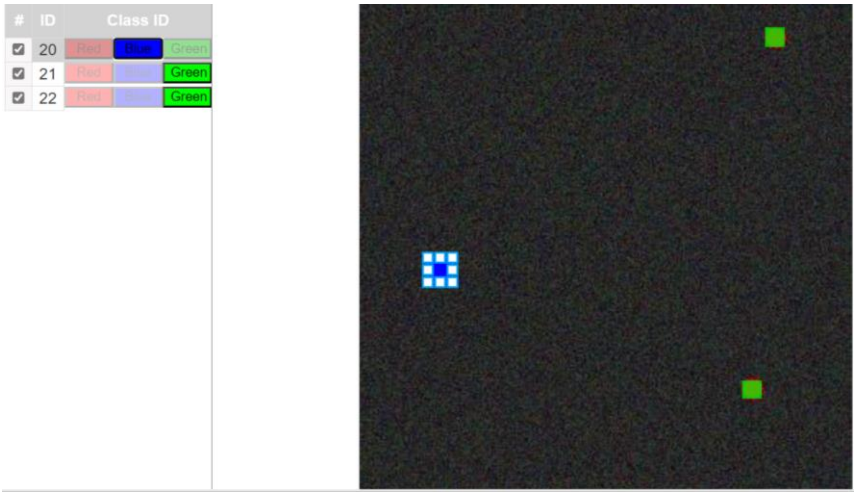
Tlačítko vrátí poslední změny v anotacích na stav posledního API volání.

3.6.7.3. Seznam anotací

Tato část zobrazuje aktuální anotace viditelné na obrazovce, zároveň lze libovolnou z nich vybrat a tím označit. Další funkcionalitou je možnost změny typu anotace. Pro uložení postupu je potřeba poté zmáčknout tlačítko „Save“. Na Obr. 19 a Obr. 20 je tato funkce ukázána. Implementačně používá komponentu ClassesButtons, kde je definované zobrazování anotačních tříd a přejímá z rodiče funkci `onToggle` v případě zmáčknutí jednoho z tlačítek.



Obr. 17 – Změna typu vybrané anotace před změnou



Obr. 18 – Změna typu vybrané anotace po změně

3.6.7.4. Kontejner anotace

Pro grafické vykreslování obrázku a anotací jsem použila knihovnu react-konva (viz. Kap. 2.4). Implementaci anotací jsem rozdělila do dvou celků – stávající anotace a nové anotace, které se po skončení tvorby uloží a odešlou přes API na backend. Stávající anotace udržuje stavová proměnná `annotationCoordinatesBox` nebo `annotationCoordinatesPoly`, nové anotace `newPoints`, která je univerzální pro oba typy.

Zahrnuje to funkce upravovat, mazat a vytvářen nové anotace, podle vybrané třídy. A všechny tyto změny je možno uložit na BE.

Typy anotací

Komponenta podporuje používání anotací s bounding box nebo s polygony, z důvodu rozdílných atributů (viz. Kap. 3.4.1.4.1) a požadavků na chování výsledných tvarů, byla implementace rozdělena a ve většině funkcí je podmínka na zjištění typu projektu, podle čehož se dále řídí konkrétní chování funkce.

U polygonů bylo rozhodnuto tvar reprezentovat pomocí cesty bodů, jak je vidět v Kód 35, kde definuji, které body úsečka propojí. K tomu byly na spojnicích těchto úseček použity kružnice `React.Circle` z knihovny react-konva pro přehlednější uživatelský dojem.

Bounding box oproti tomu reprezentuji pomocí obdélníku `Konva.Rect`, jehož příklad je v Kód 36.

```
const flattenedPoints = newPoints.concat(isFinished ? [] : curMousePos).reduce((a, b) =>
a.concat(b), []);

<Line points={flattenedPoints} fill="#5DD27A40" stroke="#5DD27A" strokeWidth={3}
closed={isFinished} />
{
    newPoints.map((point, index) => {
        const width = 6;
        const x = point[0];
        const y = point[1];
        const startPointAttr =
            index === 0
                ? {
                    hitStrokeWidth: 12,
                    onMouseOver: handleMouseOverStartPoint,
                    onMouseOut: handleMouseOutStartPoint,
                }
                : null;
        return (
            <React.Fragment key={index}>
                <Circle
                    key={index}
                    x={x}
                    y={y}
                    width={width + 2}
                    height={width + 2}
                    fill="white"
                    stroke={selectedPoint === index ? "red" : "#5DD27A"}
                    strokeWidth={3}
                    onDragStart={handleDragStartPoint}
                    onDragMove={(event) => handleDragMoveNewPoint(event, index)}
                    onDragEnd={(event) => {
                        event.cancelBubble = true;
                        handleDragEndNewPoint(event, index)
                    }
                    }
                    draggable
                    {...startPointAttr}
                />
            </React.Fragment>
        );
    })
}
```

Kód 35 – Polygonní anotace a její zobrazení

```
{
    newPoints && newPoints.length > 1 && (
        <Rect
            ref={rectRef}
            x={newPoints[0] ? newPoints[0][0] : 0}
            y={newPoints[0] ? newPoints[0][1] : 0}
            width={newPoints[1] ? newPoints[1][0] - newPoints[0][0] : 0}
            height={newPoints[1] ? newPoints[1][1] - newPoints[0][1] : 0}
            stroke="black"
            fill={'orange'}
            opacity={0.5}
        />
    )
}
{
    transform && (
        <Transformer
            ref={trRef}
            flipEnabled={false}
            rotateEnabled={false}
            anchorStrokeWidth={2}
        />
    )
}
```

Kód 36 – Bounding box anotace a její zobrazení

Funkce:

Proměnná `annotationCoordinatesBox` nebo `annotationCoordinatesPoly` si udržuje aktuální stav anotací a případně ho propaguje dále na backend, Pro práci s anotacemi byly implementovány tyto funce:

- Vybrání zvolené anotace
- Úprava polohy
- Úprava velikosti
- Uložení změn
- Odstranění anotace
- Vrácení změn
- Omezení pohybu

Vybrání zvolené anotace

Obdobně jako v případě listu anotací, i na obrázku lze vybrat konkrétní anotaci (např. pro její úpravu) a ta se označí. Implementačně toho je docíleno tak, že jakmile na stávající anotaci klikneme, nastaví se proměnná `selectedAnnotation` na odpovídající ID anotace.

V Kód 37 je celá funkce handleClick spravující jakékoliv kliknutí myší na Stage, na kterou se budu dále odkazovat.

```
const handleClick = (event: Konva.KonvaEventObject<any>) => {
        const stage = event.target.getStage();
        if (!stage) return;
        const mousePos = getMousePos(stage);

        if (isDrawingMode) {
                if (isFinished) {
                    const existingPointIndex = findExistingPointIndex(mousePos);
                    if (existingPointIndex === -1) { // clicking on line NOT point
                        if (projectType === 'poly') {
                            const index = findLineSegmentIndex(mousePos);
                            if (index !== -1) {
                                const currNewPoints = [...newPoints];
                                currNewPoints.splice(index + 1, 0, mousePos);
                                setNewPoints(currNewPoints);
                            }
                        }
                    }
                    else { //clicking on existing point -> select that point
                        //if(selectedPoint==existingPointIndex)
                        setSelectedPoint(existingPointIndex);
                    }
                    return;
                }

                if (projectType === 'poly') {
                    if (isMouseOverStartPoint && newPoints.length >= 3) {
                        setIsFinished(true);
                    } else {
                        setNewPoints([...newPoints, mousePos]);
                    }
                }
                else {
                    if (newPoints.length >= 2) {
                        setIsFinished(true);

                    } else {
                        setNewPoints([...newPoints, mousePos]);
                    }
                }
        }
        //editing current annotation
        else {

            // unselect annotation
            const clickedOnEmpty = (event.target === stage || isImage(event.target));

            if (clickedOnEmpty) {
                setSelectedAnnotation(null);
                return
            }

            const existingPointIndex = findExistingPointIndex(mousePos);
            if (existingPointIndex === -1) { // clicking on line NOT point
                if (projectType === 'poly') {
                    const index = findLineSegmentIndex(mousePos);

                    if (index !== -1) {
                        const currNewPoints = [...newPoints];
                        currNewPoints.splice(index + 1, 0, mousePos);
                        setNewPoints(currNewPoints);
                    }
                }
            }
```

```
            else { //clicking on existing point -> select that point
                setSelectedPoint(existingPointIndex);
            }
            return;
        }
    };
```

Kód 37 – Funkce handleClick

Úprava anotace

Úprava anotací byla podmínkou správného fungování aplikace. V případě bounding box umožňuje manipulaci s rohy a tím změnu velikosti ohraničujícího rámečku. Implementačně již vybráním anotace proběhne změna proměněné `transform` na `true` a dojde k automatickému dosazení komponenty `Transformer` ke komponentě `Rect`. Tato komponenta ji obklopuje na hranicích a vytváří uchopitelné body pro změnu ohraničujícího rámečku. Využita byla výchozí komponenta `Transformer` z knihovny Konva, kde lze chování vhodně definovat. Komplikací bylo, že `Transformer` ke změně rozměru používá zvětšení a zmenšení, ne úpravy výšky nebo délky. Je tedy nutno s tímto počítat a po transformaci resetovat zoom `Transformer`.

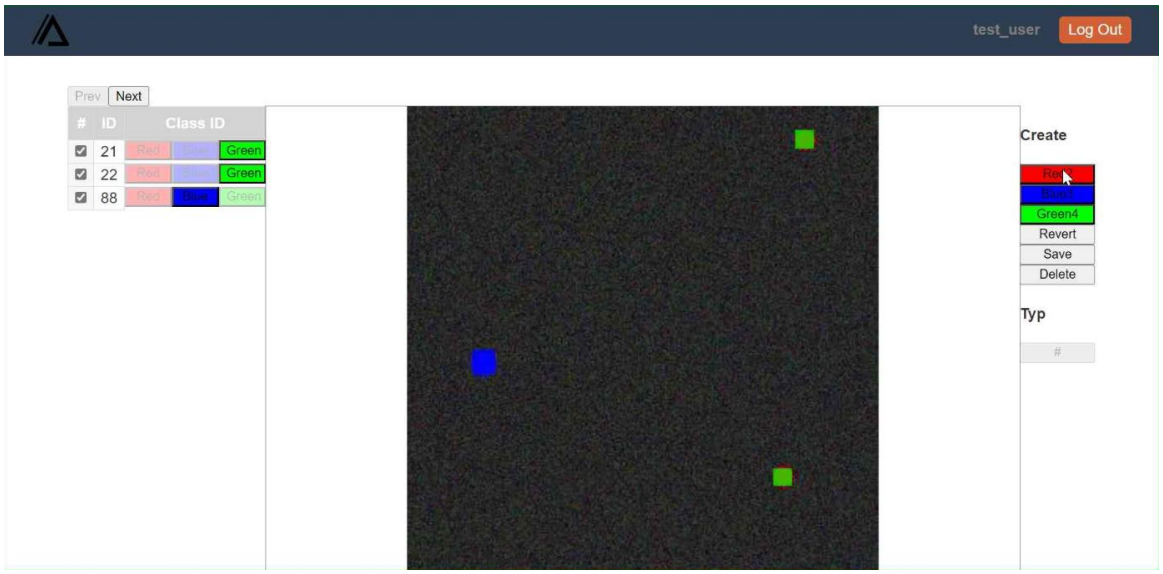
Pro polygony byl k úpravě využit posun bodů a přidání bodu mezi dva stávající nebo možnost odstranění vybraného. Interakce se `Stage` je podobně jako u bounding box řešena pomocí funkce `handleClick`. Odstranění je prováděno pomocí klávesy „DEL“, která pomocí `useEffectu` volá funkci `deleteSelectedPoint`.

Tvorba nových

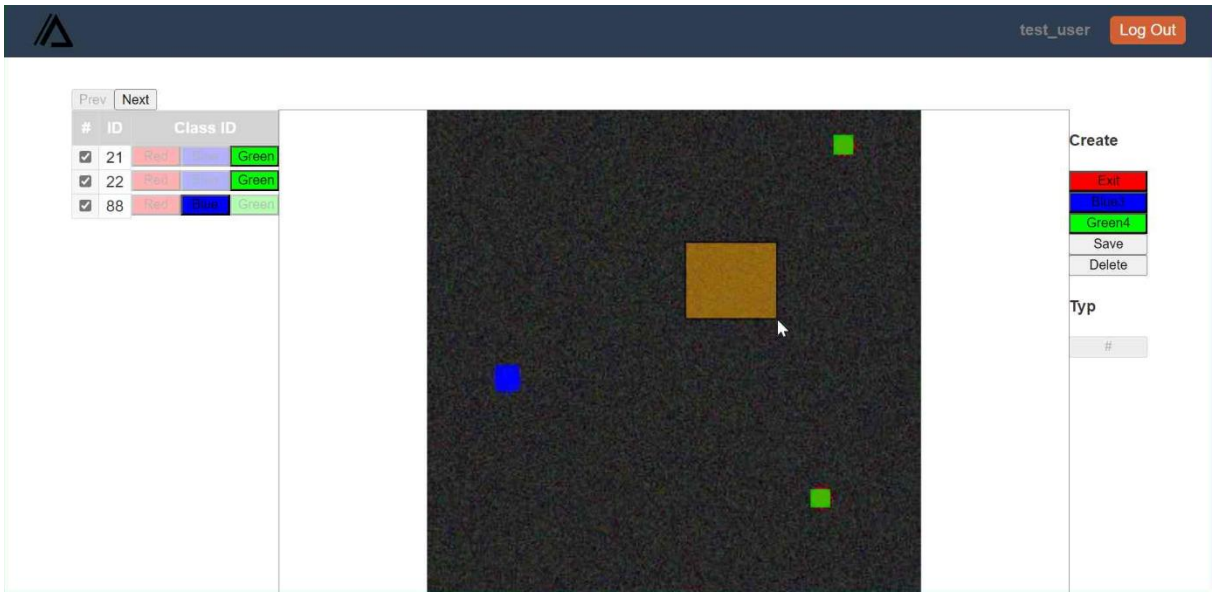
Pomocí vybrání anotační třídy se nastaví proměnná `isDrawingMode` a zahájí se tvorba nové anotace.

Pro bounding box se dvojitým klinutím se označí první bod a poté druhým klinutím výsledný bounding box. Implementačně je to řešené tak, že každý klik vyvolá funkci `handleClick` a dále se pomocí levého horního bodu a spodního pravého přepočítává potřebný obdélník s x,y (horního levého rohu), šířkou a výškou, který se ukládá do proměnné `newPoints`. Po druhém kliknutí se nastavuje proměnná `isFinised` na `true` a to vyvolává `useEffect`, který volá API funkci `fetchCreateAnnotation`, která v případě úspěchu dále pomocí API volá `fetchUpdatedAnnotations` a tímto způsobem je automaticky přidána nová anotace ke stávajícím, aby byla možnost pokračovat v tvorbě.

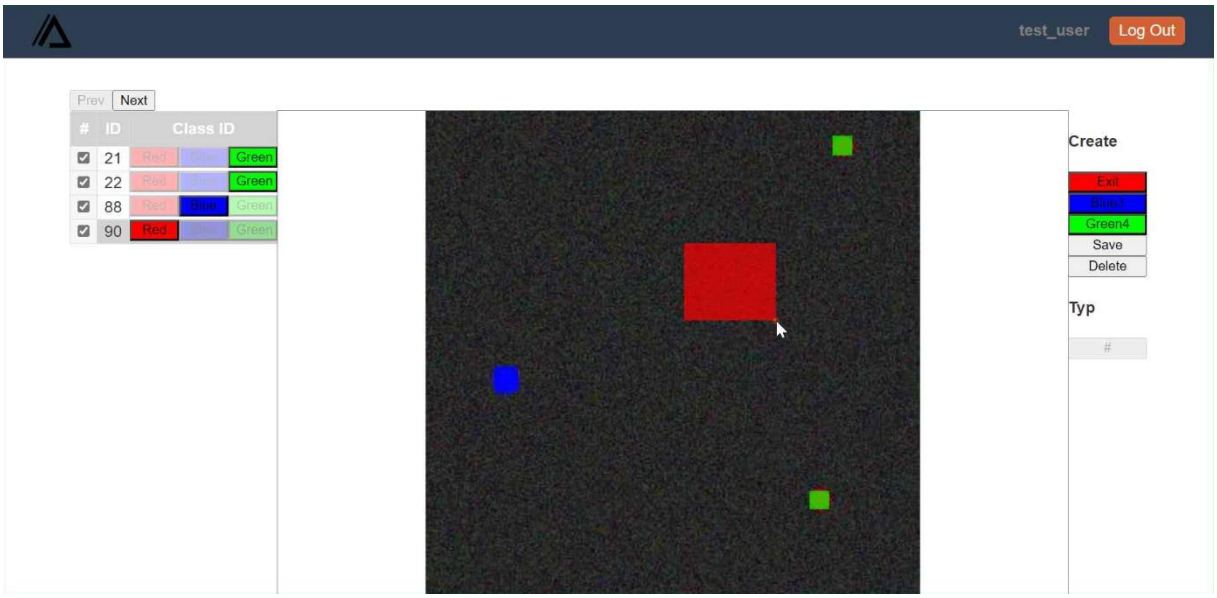
Pro polygony funguje tvorba nové aplikace tak, že postupně kliknutím označujeme požadované body kudy má výsledný polygon vést. Nový polygon se uzavře po kliknutím na první bod nebo pomocí klávesy „ESC“. Poté je stejně jako u bounding box anotace přidána ke stávajícím.



Obr. 19 – Přechod do módu kreslení nové anotace



Obr. 20 – Kreslení bounding boxu



Obr. 21 – Automatické přidání ke stávajícím anotacím

Zoom

Jak už bylo řečeno, pro trénování modelu je požadavek na co nejkvalitnější anotaci, a proto je důležitá možnost zoomu daného obrázku. Z funkčních požadavků plynula potřeba na to, aby se tak dělo za kurzorem myši, což bude pro uživatele nejpohodlnější.

Pro implementaci jsem se rozhodla nepoužít žádné knihovny a bylo zvoleno vlastní řešení pomocí dopočítávání aktuální požadované polohy `Stage` elementu v závislosti na poloze kurzoru. Tato implementace je vidět v Kód 38. Zároveň bylo potřeba určit horní a dolní mez zoomu, aby se po dosažení této meze už komponenta nezvětšovala/nezmenšovala. Důležitým bodem také bylo dopočítání nové polohy `newPosX` a `newPosY` (viz. Kód 38), tyto řádky hlídají případ, kdy narazíme na konec obrázku a při dalším zoomu by obrázek mohl částečně opustit `Stage`.

```
    /* Zoom - calculating scale and shift */
    const [scale, setScale] = useState(1);
    const handleWheel = (e: Konva.KonvaEventObject<WheelEvent>) => {
        e.evt.preventDefault();

        const stage = stageRef.current?.getStage();
        if (!stage) return

        const oldScale = stage.scaleX();
        const pointer = stage.getPointerPosition(); //on stage

        if (!pointer) return
```

```
        const mousePointTo = { // relative to original image
            x: (pointer.x - stage.x()) / oldScale,
            y: (pointer.y - stage.y()) / oldScale,
        };

        const scaleBy = 1.05;
        let newScale = e.evt.deltaY < 0 ? oldScale * scaleBy : oldScale / scaleBy;

        // max/min zoom
        if (newScale > 4) return;
        else if (newScale < 1) return;

        setScale(newScale);

        const newPos = {
            x: pointer.x - mousePointTo.x * newScale,
            y: pointer.y - mousePointTo.y * newScale,
        };

        const image = konvaImageRef.current
        if(!image) return

        const stageWidth = stage.width();
        const stageHeight = stage.height();

        const newPosX = Math.min(0, Math.max(newPos.x, -image.width() * newScale + stageWidth));
        const newPosY = Math.min(0, Math.max(newPos.y, -image.height() * newScale + stageHeight));

        stage.scale({ x: newScale, y: newScale });
        stage.position({ x: newPosX, y: newPosY });

        stage.batchDraw();
    };
```

Kód 38 – Funkce zoom

Posouvání

Stejně tak podstatnou funkcí jako je zoom, je funkce posouvání. Celkem byly implementovány tyto typy posunu:

- Stage
- Group (skupina) – reprezentuje jednu anotaci
- Bod

U Stage je pouze props atribut Konva komponenty `draggable`, který je nastaven na `true`, Jinak jsou využívá `props` atributy `onDragMove` a `onDragEnd`, které definují, co se má stát potom co je objekt chycen kurzorem myši a co poté, co je puštěn. Především onDragEnd je důležitý, protože aktualizuje hodnotu stavové proměnné `annotationCoordinatesBox` nebo `annotationCoordinatesPoly`. Příklad takové funkce pro skupinu je vidět v Kód 39.

```
const handleDragEndGroup = (event: Konva.KonvaEventObject<DragEvent>, annotationIndex: number) =>
{
        const group = event.target;
        const deltaX = group.x();
        const deltaY = group.y();

        if (projectType === 'box') {
            const deltaW = group.width();
            const deltaH = group.height();
            const updatedPoints = [
                {
                    ...annotationCoordinatesBox[annotationIndex],
                    values: {
                        ...annotationCoordinatesBox[annotationIndex].values,
                        xRel: (deltaX + deltaW / 2) / imageDimensions.width,
                        yRel: (deltaY + deltaH / 2) / imageDimensions.height
                    }
                }
            ];
            const newAnnotationCoordinatesBox = [...annotationCoordinatesBox];
            newAnnotationCoordinatesBox[annotationIndex] = updatedPoints[0];
            setAnnotationCoordinatesBox(newAnnotationCoordinatesBox);
        } else {
            const updatedPoints = annotationCoordinatesPoly[annotationIndex].values.map(point =>
({
                xRel: point.xRel + deltaX / imageDimensions.width,
                yRel: point.yRel + deltaY / imageDimensions.height
            }));

            const newAnnotationCoordinatesPoly = [...annotationCoordinatesPoly];
            newAnnotationCoordinatesPoly[annotationIndex] = {
                ...annotationCoordinatesPoly[annotationIndex],
                values: updatedPoints
            };
            setAnnotationCoordinatesPoly(newAnnotationCoordinatesPoly);
        }
};
```

Kód 39 – Funkce handleDragEndGroup

Bounds

Pro posouvání a zoom bylo nejprve vyzkoušeno použití props `onBoundFunc` u dané komponentu z knihovny react-konva, ale tento přístup se příliš neosvědčil, jelikož očekává vrácení upravených relativních souřadnic vůči počáteční poloze, a tudíž je bylo náročné přepočítávat nejprve do absolutních hodnot, kde je hranice `Stage`, a poté zase zpět dosazovat relativní souřadnice k vrácení. Jediné, kde byl jednoduše použitelný bylo právě u samotné `Stage`. Implementace je vidět v Kód 40.

```
// použití props dragBoundFunc
dragBoundFunc={pos => {
   return {
       x: clamp(pos.x, -(stageSize.width * scale - stageSize.width), 0),
       y: clamp(pos.y, -(stageSize.height * scale - stageSize.height), 0)
   }
}}
```

```
// funkce clamp
const clamp = (value: number, min: number, max: number) => {
   return Math.max(min, Math.min(max, value));
};
```

Kód 40 – Omezení posuvu Stage

Z toho důvodu jsem se rozhodla použít zmíněnou funkci `handleDragMove` a uvnitř ní doplnit kontrolu, zda je kurzor myši již na hraně obrázku a případně zastavit další růst souřadnic přes tuto mez. Toho principu bylo využitu posouvání skupiny i bodu. V Kód 41 je příklad takové implementace pro skupinu, kde v konstantách `deltaX` a `deltaY` probíhá omezení.

```
const handleDragMoveGroup = (event: Konva.KonvaEventObject<DragEvent>) => {
        const group = event.target;

        let deltaX = group.x();
        let deltaY = group.y();

        const stage = group.getStage();
        if (!stage) return

        const stageWidth = stage.width();
        const stageHeight = stage.height();
        const stagePosition = stage.position();

        // Get the current scale
        const scaleX = stage.scaleX();
        const scaleY = stage.scaleY();

        // Calculate boundaries considering the current scale
        const groupWidth = group.width() * group.scaleX() * scaleX;
        const groupHeight = group.height() * group.scaleY() * scaleY;

        const minX = 0;
        const minY = 0;
        const maxX = stageWidth - groupWidth;
        const maxY = stageHeight - groupHeight;

        // Clamp values
        deltaX = Math.max(minX, Math.min(deltaX, maxX));
        deltaY = Math.max(minY, Math.min(deltaY, maxY));

        group.position({ x: deltaX, y: deltaY });

        group.getLayer()?.batchDraw();
    };
```

Kód 41– Omezení posuvu skupiny ve funkci handleDragMoveGroup

3.6.8. Not Found

Jedná se o stránku, na kterou probíhá přesměrování, pokud je neplatná URL. Zobrazuje chybový kód.

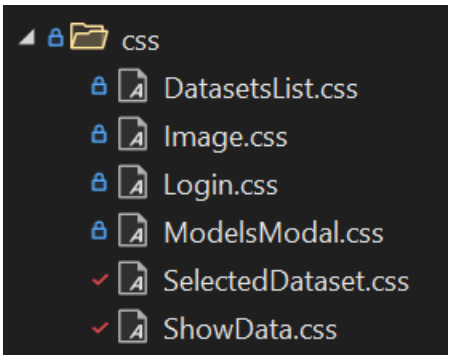
3.6.9. Další komponenty

3.6.9.1. Navigační lišta

Pro zobrazení konkrétního uživatel a případné odhlášení byla implementována navigační lišta. Pro získání dat o uživateli komponenta používá hook `useSession` a po zmáčknutí tlačítka volá funkci `SignOut`.

3.6.10. CSS

K aplikaci byl zároveň také vytvořen jednoduchý grafický vizuál definovaný pomocí samostatných CSS souborů ve stejnojmenné složce, která obsahuje definici pro jednotlivé stránky i některé komponenty. Na Obr. 23 je zobrazena struktura této složky.



Obr. 22 – Soubory CSS pro definice vizuální podoby aplikace

3.7. Uživatelský manuál – nasazení

Aplikace je v současnosti především lokálního charakteru, kdy serverová i klientská část běží pouze lokálně na jediném počítači, avšak vývoj byl prováděn tak, aby bylo v budoucnu možné nasazení na veřejnou adresu.

Ke zprovoznění je nutnou podmínkou operační systém podporující Python 3.9+, což Windows, Mac OS a většina Linux distribucí splňuje a dále pak webový prohlížeč. Testováno bylo nasazení pro Windows, pro který budou zde uvedeny všechny příkazy a pro prohlížeč Chrome a Edge.

3.7.1. Stažení repositářů

Repozitář je k dispozici ke stažení na https://github.com/CVUT-FS-12110/mv-dataset-editor/tree/master. Pokud máme naistalován verzovací systém Git, tak naklonovat repozitář můžeme například pomocí HTTPS příkazem `git clone`. Například, pro stažení repositáře do složky `projekt` lze použít příkazy:

```
cd C:\Users\projekt\
git clone https://github.com/CVUT-FS-12110/mv-dataset-editor.git
```

Jelikož se jedná v současnosti o privátní repozitář, podmínkou je být v git configu pod správným uživatelem. Pokud Git nemáme, nebo chceme zvolit co nejjednodušší verzi, tak je možnost stáhnout na GitHub odkazu ZIP projektu a extrahovat repozitář do požadované složky. Případně můžeme využít jinou preferovanou metodu.

3.7.2. Instalace knihoven

Proces instalace knihoven je pro backendovou a frontendovou část rozdělený a náležitosti obou jsou popsány níže.

Backend

Prvním krokem je ujištění se, že máme nainstalovanou odpovídající verzi Pythonu (např pomocí příkazu `python --version`) a případně její instalace.

Pro správnou instalaci knihoven je potřeba přejít do kořenové složky adresáře (např. pomocí příkazu `cd`) a dále si doporučuji vytvořit virtuální prostředí pomocí skriptu:

```
python -m venv venv
```

Poté využijeme souboru `requirements.txt` ze kterého pomocí následujícího příkazu nainstalujeme všechny potřebné knihovny. Pro instalaci do právě vytvořeného virtuálního prostředí spustíme:

```
.\venv\scripts\python.exe -m pip install -r requirements.txt
```

Pokud nechceme používat virtuální prostředí, tak postačí:

```
pip install -r requirements.txt
```

Frontend

Ujistíme se, že máme k dispozici node.js, případně naistalujeme i společně s npm.

Pro frontend přejdeme do adresáře src/ui, který je kořenovým adresářem FE a pustíme příkaz `npm install` pro instalaci všech knihoven z package.json a případně ještě `npm audit fix`, pokud to package manager doporučuje.

3.7.3. Konfigurace

Pro správnou funkci projektu je nezbytné nakonfigurovat několik jeho nastavení. Níže jsou rozděleny do celků zvlášť pro backend a frontend.

3.7.3.1. Konfigurace backendu

Patří sem:

- Proměnné prostředí
- Datasety

Proměnné prostředí

Je potřeba vytvořit soubor `.env` v rootu projektu, Jeho podoba je zobrazena v Kódu 42. Je zde nastavena databáze tak, aby bylo určeno, zda se mají data ukládat do paměti nebo na disk a na jaké místo. K tomuto účelu slouží proměnná `API_DATABASE_DNS`. V ukázce jsou obě možnosti volby mezi persistentní a neperzistentní verzí databáze.

Dále nastavení MLflow kam se mají ukládat data, na jakém serveru a portu má prostředí běžet apod. Je také nutné ověřit, zda projekt obsahuje složku `storage`, a pokud ne, tuto složku vytvořit, například v kořenovém adresáři projektu.

Posledním krokem nastavení je odpovídající adresa serveru a portu k frontendové aplikaci. Jelikož v současné době vše lokálního charakteru, je adresa 127.0.0.1.

```
API_DATABASE_DNS = 'sqlite://'
# API_DATABASE_DNS = 'sqlite:///db.sqlite3' # for persistent db
```

```
API_STORAGE = 'storage'

MLFLOW_DATA = "/data/models_data"
MLFLOW_TRACKING_URI = "http://127.0.0.1:5000"
MLFLOW_BACKEND_STORE_URI='sqlite:///storage/models_data/mlflow.db'
MLFLOW_DEFAULT_ARTIFACT_ROOT='./storage/models_data/artifacts'
MLFLOW_HOST='127.0.0.1'
MLFLOW_PORT='5000'

REACT_APP_BASE_URL='http://127.0.0.1:3000'
```

Kód 42 – Proměnné prostředí pro backendovou část

Datasety

V souboru fixtures.py popsaném v Kap. 3.4.1.1 je nutné nastavit uživatele a výchozí složku pro datasety, se kterými chceme pracovat, a případně cestu k této složce, pokud se nachází mimo projektový adresář. K práci zvlášť přikládám jako testovací dataset složku `fixtures` s testovacími datasety. V Kód 43 je příklad takového nastavení. Případně je také možné upravit soubor _project.yaml podle potřeb.

```
user = create_user("test_user", "pass123", True)
load_projects(user, "fixtures")
```

Kód 43 – Nastavení uživatele a výchozí složky datasetu v souboru fixtures.py

3.7.3.2. Konfigurace frontendu

V konfiguraci frontendu je nutné nastavení proměnné prostředí.

Proměnné prostředí

Podobně jako na backendu je potřeba vytvořit soubor `.env`, ale v adresáři src/ui. Obsah souboru je zobrazen v Kódu 44. Proměnná `REACT_APP_BASE_URL` slouží ke konfiguraci adresy backendového serveru.

```
REACT_APP_BASE_URL='http://127.0.0.1:8000'
```

Kód 44 – Proměnné prostředí pro frontendovou část

3.7.4. Spuštění programu

Jakmile máme nainstalovaly všechny knihovny a provedenou potřebnou konfiguraci, nic nám nebrání aplikaci spustit. To opět rozdělím na backendovou část a frontendovou část. Ke spuštění si otevřeme příkazovou řádku, případně zvolíme jiný preferovaný způsob.

3.7.4.1. Spuštění backendové části aplikace

Ke spuštění backendové části nám stačí v kořenovém adresáři projektu spustit soubor `run_api.py` a soubor `run_models.py`. a to pomocí příkazu:

```
python nazev_souboru.py
```

V tuto chvíli by mělo být dostupné API na lokální adrese a portu ze souboru `run_api.py`.

3.7.4.2. Spuštění frontendové části aplikace

Ke spuštění frontendové části přejdeme do složky src/ui a zde voláme příkaz pro vývojový mód:

```
npm start
```

Nebo produkční mód:

```
npm run build
```

4. Závěr

Byla vytvořena aplikace na interaktivní asistované anotování YOLO datasetů, která umožňuje uživatelům upravovat a vytvářet Cílem bylo zjednodušit proces anotace využitím trénování modelu a následné predikce.

Aplikace byla navržena s důrazem na uživatelskou přívětivost a intuitivní ovládání. Implementace zahrnovala vytvoření uživatelského rozhraní v Reactu s využitím knihovny Konva pro grafické práce s uživatelským rozhraním a backendovou části v Pythonu. Součástí práce byl také modul MLflow na správu a trénink YOLO modelu.

Budoucí směřování této aplikace může zahrnovat rozšíření funkcionalit jako podpory dalších modelu pro predikci a anotaci, jejich zvolení přímo z frontendu a případně podporu dalších typů anotací.

Výsledky a zkušenosti získané během vývoje této aplikace beru jako cenou zkušenost v hlubším pochopení anotačního procesu a vývoje webové aplikace.

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódu

Bibliografie

[1] Seed Scientific. How Much Data Is Created Every Day? [online]. [cit. 2024-08-02]. Dostupné z: https://seedscientific.com/how-much-data-is-created-every-day/

[2] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Las Vegas, NV, USA, 2016. s. 779-788. [online]. DOI: 10.1109/CVPR.2016.91. Keywords: Computer architecture; Microprocessors; Object detection; Training; Real-time systems; Neural networks; Pipelines.

[3] Purina. Understanding Your Cat's Body Language [online]. [cit. 2024-08-02]. Dostupné z: https://www.purina.co.uk/sites/default/files/styles/ttt_image_original/public/2020-12/Understanding%20Your%20Cat%27s%20Body%20LanguageHERO.webp?itok=W8bF8KSm

[4] Roboflow. Polygon vs Bounding Box: Computer Vision Annotation [online]. [cit. 2024-08-02]. Dostupné z: https://blog.roboflow.com/polygon-vs-bounding-box-computer-vision-annotation/

[5] Neubeck, A. and Van Gool, L. Efficient non-maximum suppression. In: 18th International Conference on Pattern Recognition (ICPR'06). Vol. 3. IEEE, 2006. s. 850–855.

[6] Zhang, F., Li, C. and Yang, F. Vehicle Detection in Urban Traffic Surveillance Images Based on Convolutional Neural Networks with Feature Concatenation. School of Mechanical Electronic and Information Engineering, China University of Mining and Technology, Beijing, Beijing 100083, China. [online]. Received: 4 January 2019; Accepted: 29 January 2019; Published: 30 January 2019. [email: celi@cumtb.edu.cn (C.L.); yangf@cumtb.edu.cn (F.Y.)] Correspondence: zhangfukaidream@163.com; Tel.: +86-188-0356-0539.

[7] Wen, L., Du, D., Cai, Z., Lei, Z., Chang, M.C., Qi, H., Lim, J., Yang, M.H. and Lyu, S. UA-DETRAC: A New Benchmark and Protocol for Multi-Object Detection and Tracking. arXiv [online]. 2015. [cit. 2024-08-02]. Dostupné z: https://arxiv.org/abs/1511.04136v3. arXiv:1511.04136v3.

[8] LearnOpenCV. Mastering All YOLO Models [online]. [cit. 2024-08-02]. Dostupné z: https://learnopencv.com/mastering-all-yolo-models/

[9] Tkrotoff. Gist: YOLOv4 Object Detection using TensorFlow 2.0 [online]. [cit. 2024-08-02].

[10] React. *React reference* [online]. [cit. 2024-08-02]. Dostupné z: https://react.dev/reference/react

[11] Telerik. Understand how rendering works in React [online]. [cit. 2024-08-02]. Dostupné z: https://www.telerik.com/blogs/understand-how-rendering-works-react

[12] React. Reconciliation [online]. [cit. 2024-08-02]. Dostupné z: https://legacy.reactjs.org/docs/reconciliation.html

[13] D3.js. D3.js [online]. [cit. 2024-08-02]. Dostupné z: https://d3js.org/

[14] Konva. Konva [online]. [cit. 2024-08-02]. Dostupné z: https://konvajs.org/

[15] Konva. react-konva [online]. [cit. 2024-08-02]. Dostupné z: https://github.com/konvajs/react-konva

[16] Label Studio. Label Studio [online]. [cit. 2024-08-02]. Dostupné z: https://labelstud.io/

[17] Roboflow. Roboflow [online]. [cit. 2024-08-02]. Dostupné z: https://roboflow.com/

[18] CVAT. CVAT [online]. [cit. 2024-08-02]. Dostupné z: https://www.cvat.ai/

[19] Elbaum, S., Rothermel, G., Karre, S. a Fisher, M. (2005). Leveraging user-session data to support web application testing. IEEE Transactions on Software Engineering, 31, s. 187-202. DOI: 10.1109/TSE.2005.36.

[20] MLflow. MLflow [online]. [cit. 2024-08-02]. Dostupné z: https://mlflow.org/

[21] FastAPI. FastAPI [online]. [cit. 2024-08-02]. Dostupné z: https://fastapi.tiangolo.com/

[22] Yashika. *Write robust APIs in Python with three-layer architecture, FastAPI, and Pydantic models* [online]. Medium, 2024. [cit. 2024-08-02]. Dostupné z: https://medium.com/@yashika51/write-robust-apis-in-python-with-three-layer-architecture-fastapi-and-pydantic-models-3ef20940869c

[23] SQLAlchemy. SQLAlchemy [online]. [cit. 2024-08-02]. Dostupné z: https://www.sqlalchemy.org/

[24] Pydantic. Pydantic [online]. [cit. 2024-08-02]. Dostupné z: https://pydantic.dev/

Přílohy

[1] CD disk