

STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor č. 79-41-K/41: Gymnázium (Programování)

Autonomní Parkovací Systém

Tomáš Černý
Hlavní město Praha

Praha 2020

STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor č. 79-41-K/41: Gymnázium (Programování)

Autonomní Parkovací Systém

Autonomous Parking System

Autoři: Tomáš Černý

Škola: Gymnázium, Praha 6, Arabská 14

Kraj: Hlavní město Praha

Konzultant: Tomáš Černý

Praha 2020

Prohlášení

Prohlašuji, že jsem svou práci SOČ vypracoval/a samostatně a použil/a jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů.

Prohlašuji, že tištěná verze a elektronická verze soutěžní práce SOČ jsou shodné.

Nemám závažný důvod proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů.

V Praze dne

Název práce: Autonomní Parkovací Systém

Autoři: Tomáš Černý

Abstrakt: Cílem projektu bylo navrhnout a implementovat informační systém pro správu komerčních parkovišť, parkovišť supermarketů, obchodních center a podobných míst. Systém je napsán moderními technologiemi a vyžaduje málo obsluhy za pomoci automatického rozpoznávání SPZ díky knihovně OpenALPR a levným Android zařízením. Systém umožňuje vytvářet flexibilní pravidla určující cenu parkování za jednotku času, jednotky času zadarmo a jejich interval platnosti. Také jsou schopna filtrovat vozidla na základě rozpoznané SPZ. Zároveň umožňuje monitorovat stav v reálném čase a číst statistiky.

Systém by po několika úpravách a přidání komunikace s platebním terminálem a závorou měl být reálně použitelný.

Title: Autonomous Parking System

Authors: Tomáš Černý

Abstract: The goal of this project was to design and implement an information system for administration of commercial parking lots, parking lots of supermarkets, shopping centers and similar. The system is written using modern technologies and requires little maintenance by automatically recognizing license plates thanks to the OpenALPR library and cheap Android devices. It allows to create flexible rules that specify price per unit time, free units of time. They can also be enabled during certain time periods and can filter vehicles by their license plate. The administrator can also watch both live and past statistics.

After the addition of a payment terminal and a parking barrier and some other changes, the system should be usable in real-life.

Obsah

1	Úvod	3
2	Architektura a technologie	5
2.1	Architektura řešení	5
2.2	Technologie	5
2.2.1	Databáze	5
2.2.2	Backend	5
2.2.3	Frontend	7
2.2.4	Mobilní aplikace	7
2.2.5	Detekce SPZ	8
2.3	Metodika vývoje	8
3	Webová aplikace	9
3.1	Obrazovky	10
3.2	GraphQL resolvers	10
3.2.1	Obecné resolvers	11
3.3	Autentizace a autorizace	11
3.4	Parkovací pravidla	12
3.4.1	Algoritmus filtru vozidel	13
3.4.2	Algoritmus pro aplikaci pravidel	13
3.5	Uživatelské rozhraní	15
3.5.1	Redux a persistence stavu	15
3.5.2	Typy komponent	15
3.5.3	Grafy	16
3.5.4	Obecná vyběrátko modelů	16
3.6	Snímky obrazovky obrazovek	16
4	Rozpoznávání SPZ	21
4.1	Zvyšování přesnosti	21
4.1.1	Cachování výsledků	21
4.1.2	Filtrování podle geometrického obsahu	21
4.2	Autentifikace	21
4.3	Volba zařízení pro mobilní aplikaci	21
4.4	Životní cyklus mobilní aplikace	22
4.5	Uživatelské rozhraní mobilní aplikace	22
4.6	Implementační detaily mobilní aplikace	23
4.6.1	Komunikace s backendem	23
4.6.2	Ukládání snímků	23
5	Instalace	25
	Závěr	27
	Seznam použité literatury	29
	Seznam obrázků	31

1. Úvod

Tento dokument se zabývá koncepcí a implementací informačního systému pro uzavřená parkoviště s minimální obsluhou a automatickým rozpoznáváním SPZ. Tento projekt zároveň slouží k vyzkoušení si moderních webových technologií a frameworků s přesahem k mobilnímu vývoji.

Práce se nezabývá operací se závorou a platebním terminálem, neb by to zvyšovalo obtížnost už tak obtížného projektu a vyžadovalo by to poměrně vysoké náklady na hardware.

Zadání

Parkovací systém by měl mít následující funkcionalitu:

- Naskenování SPZ.
- Vytváření dostatečně flexibilních pravidel pro většinu použití.
 - Od času A do B za tarif X/jednotka času.
 - Různý provoz o svátcích a víkendech.
 - Limity na hodiny zdarma.
 - Filtrování vozidel – pro různá vozidla mohou platit jiná pravidla.
- Statistiky počtu aut a výdělku s grafy.

2. Architektura a technologie

2.1 Architektura řešení

Parkovací systém se skládá z následujících částí, které si nyní popíšeme stručně a detailněji v následujících kapitolách. Jednotlivé komponenty spolu komunikují pomocí HTTP. Obrázek 2.1 ukazuje tyto části a nastiňuje obsah komunikace.

- **Backend** je středobodem celého systému – komunikuje se všemi ostatními komponentami. Zajišťuje business logiku aplikace, autentizaci i autorizaci uživatelů i zařízení a persistenci dat do databáze.
 - **Databáze** slouží k ukládání a čtení dat.
 - **OpenALPR Server** obstarává přístup ke knihovně OpenALPR, která rozpoznává SPZ, přes protokol HTTP.
- **Mobilní aplikace** je určená pro platformu Android a posílá obrazová data na backend, kde jsou zpracována.
- **Frontend** je rozhraní mezi celým systémem a správcem parkoviště.

2.2 Technologie

2.2.1 Databáze

Databáze MongoDB byla vybrána, protože data se budou převážně zapisovat a bude potřeba v nich rychle hledat a provádět agregační dotazy. Mimo jiné umožňuje provoz několika spolupracujících instancí, zálohování apod. (viz MongoDB Inc., 2020)

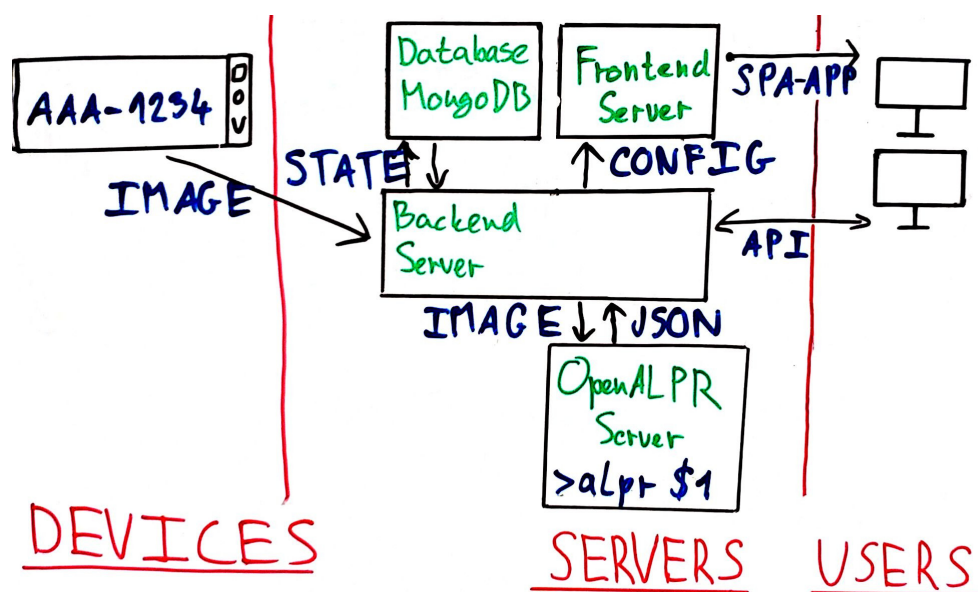
2.2.2 Backend

Jako programovací jazyk pro backend byl zvolen staticky typovaný Typescript kvůli rychlosti vývoje a množství knihoven, které poskytuje ekosystém Node.js. (Microsoft, 2019) (OpenJS Foundation, 2019)

Pro definici databázových modelů a komunikaci s databází byla kvůli své vyspělosti a skvělé funkcionalitě zvolena knihovna mongoose. (viz LearnBoost, 2019)

Primárním způsobem komunikace s frontendem je dotazovací jazyk GraphQL, který přináší ucelený popis poskytovaných dat pomocí kontroly typů a expresivních dotazů, jejichž odpověď má stejný “tvar” v JSON formátu. Obrázek 2.2 ukazuje dotaz hledání uživatele podle jména.

Model uživatele může mít i další atributy, ale GraphQL vrátí přesně ty údaje, na které se klient zeptal. Tento triviální příklad neukazuje další funkce jako mutace dat, dědičnost typů, více dotazů v jedné HTTP žádosti a mnoho dalších funkcí. GraphQL je pouze specifikace vytvořená společností Facebook a má několik implementací. (viz The GraphQL Foundation, 2020) Pro tento projekt byla zvolena implementace Apollo (viz Meteor Development Group Inc., 2020).



Obrázek 2.1: Diagram komponent a jejich komunikace.

```

1 query {
2   userSearch(
3     search: { name: "user" }
4   ) {
5     data {
6       id
7       name
8       permissions
9     }
10  }
11 }
12

```

```

{
  "data": {
    "userSearch": {
      "data": [
        {
          "id": "5e37cd1ed732da2177b8a811",
          "name": "user1",
          "permissions": [
            "ALL"
          ]
        },
        {
          "id": "5e3e7bcb56ab06a93e41dab8",
          "name": "user2",
          "permissions": [
            "DEVICES"
          ]
        }
      ]
    }
  }
}

```

Obrázek 2.2: Příklad GraphQL dotazu (vlevo) a odpovědi (vpravo). Screenshot z nástroje GraphQL Playground.

Jelikož GraphQL posílá odpovědi v JSON, není vhodné pro posílání obrázků. Je to možné za využití base64 kódování, ale přes síť se přenesou více bytů, než při použití obvyklého způsobu přes HTTP. Z toho důvodu pro posílání obrázků bude mít backend i jiné endpointy.

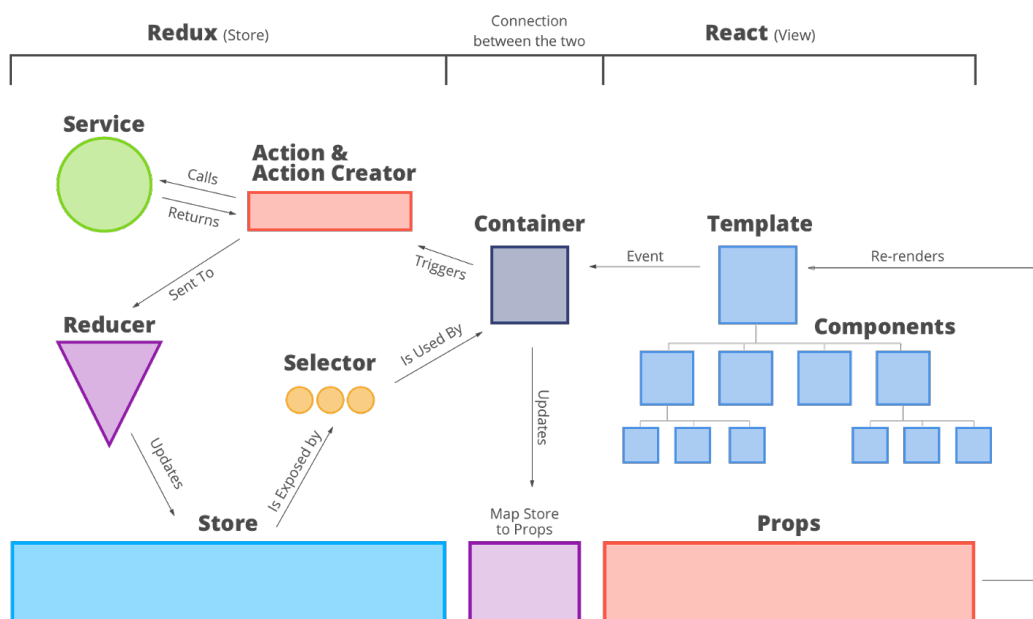
2.2.3 Frontend

Pro frontend byl jako u backendu vybrán Typescript ze stejných důvodů. Webové rozhraní je takzvaná SPA (z angl. Single-Page-Application), což znamená, že uživateli se obsah mění dynamicky bez načítání dalších stránek.

Renderování zajišťuje knihovna React, která od klasického přístupu, kdy se odděluje HTML a Javascript do separátních souborů, mandatuje, že v jednom souboru je jedna komponenta se vším svým HTML a logikou ve formě Javascriptu nebo Typescriptu. (viz Facebook, 2019) Pomocí další knihovny typestyle pak můžeme do stejného souboru psát i typované CSS (viz typestyle, 2020).

Aby bylo možné snadno sdílet mezi komponentami stav, byla pro takzvaný state-management zvolena knihovna Redux. (viz Dan Abramov et al., 2020) Diagram na obrázku 2.3 ukazuje tok dat mezi Reactem a Reduxem.

Jelikož je aplikace vyvíjena pro správce systému a ne pro velké množství uživatelů, můžeme si dovolit klást menší nároky na velikost aplikace (tj. můžeme přidávat i velké knihovny), což velice usnadní vývoj.



Obrázek 2.3: Spojení React a Redux. (viz Er Ajay Pratap, 2018)

2.2.4 Mobilní aplikace

Mobilní aplikace, která je určena pro platformu Android, měla volbu jazyka omezenou na Javu, Kotlin a C++. Rychlost C++ není potřeba a navíc autor s tímto nízkourovňovým jazykem nemá takové zkušenosti. Kotlin oproti Javě umožňuje přímočarejší přístup k prvkům uživatelského rozhraní, a proto byl zvolen.

Jediným úkolem mobilní aplikace je v pravidelném intervalu pořizovat snímky fotoaparátem a posílat je na backend, který je patřičně zpracuje.

2.2.5 Detekce SPZ

Detekci SPZ bude zajišťovat knihovna OpenALPR, jejímž vstupem je obrázek a popřípadě parametry jako úhel kamery apod. (viz OpenALPR, 2018)

Ke zbytku aplikace bude připojena malým HTTP serverem, jenž byl převzán a upraven. Ten umožňuje poslat pomocí protokolu HTTP obrázek a obdržet JSON s SPZ daty a souřadnicemi detekované SPZ. Samotný server ke knihovně přistupuje zavoláním binárky *alpr*, který jako argument přijme cestu k obrázku, ve kterém hledáme SPZ. (viz gerhardsletten, 2019)

Alternativní a lepší způsob přístupu by bylo mít v Node.js přímo takzv. *language-binding*, ale to se autorovi (a mnoho dalším, kteří se o to pokoušeli) nepodařilo.

2.3 Metodika vývoje

Nejprve se vyvine veškerá funkcionalita v základní podobě (na způsob MVP – z angl. Minimal-Viable-Product) a později se vše vyhladí a zlepší. To však neznamená, že by se nemělo dát si záležet na kvalitním a udržitelném kódu, naopak. Cílem je mít flexibilní základ, na kterém lze stavět. Pro klidný spánek budeme psát dle uvážení automatizované testy, aby se předešlo nežádoucímu chování aplikace po úpravě kódu – regresi.

Backend i frontend budou vyvinuty současně. Dokud není mobilní aplikace pro zařízení, lze ji simulovat například nástrojem *curl*. Pro rychlejší prototypování bude použita technika zvaná *hot-reload*.

3. Webová aplikace

Tato kapitola popisuje zvolené postupy při psaní webové aplikace (backend a frontend).

Obrázek 3.1 a obrázek 3.2 popisují adresářové struktury backendu a frontendu. Některé méně důležité adresáře byly vynechány.

- **config** – konfigurační soubory
- **scripts** – skripty pro vývoj
- **test_assets** – obrázky pro testování rozpoznávání SPZ
- **src** – zdrojový kód
 - **apis** – hepler funkce pro komunikaci s externím API pro rozpoznávání SPZ
 - **auth** – autentifikace a autorizace
 - **cache** – implementace mezipaměti
 - **db** – připojení k databázi a nastavení GraphQL serveru Apollo
 - **endpoints** – endpointy (ne GraphQL)
 - **types** – business logika aplikace, definice GraphQL schema, resolverů a mongoose modelů
 - **utils** – datové struktury a ostatní helper funkce i pro testy

Obrázek 3.1: Adresářová struktura backendu.

- **config** – konfigurační soubory
- **src/app** – zdrojový kód
 - **apis** – hepler funkce pro získávání obrázků z backendu
 - **components** – komponenty uživatelského rozhraní
 - **constants** – GraphQL dotazy, barvy, apod
 - **helpers** – hepler funkce
 - **images** – obrázky
 - **layouts** – rozložení obrazovek
 - **pages** – komponenty obrazovek
 - **redux** – definice reducerů a stavů aplikace
 - **routes** – definice cest obrazovek
 - **sagas** – vedlejší efekty Redux akcí

Obrázek 3.2: Adresářová struktura frontendu.

Nastavení frontendu je upravitelné ve složce s konfigurací (`/config`). Některé hodnoty jsou brány z proměnných prostředí, mají ale předdefinovanou hodnotu, pokud konkrétní proměnná prostředí je prázdná.

Nastavení backendu je řešeno knihovnou `nconf`, která umožňuje kteroukoliv hodnotu upravit pomocí proměnných prostředí – stačí jen vzít cestu k hodnotě a každé vnoření nahradit dvěma podtržítky. (viz Charlie Robbins, 2019) Například cesta `{mongo : {host}}` lze přepsat proměnou prostředí s názvem `mongo__host`.

Seznam všech závislostí je uveden v souborech `package.json` a `package-lock.json`.■

3.1 Obrazovky

Dle požadavků z úvodu mějme po přihlášení do webového rozhraní následující obrazovky, mezi kterými bude uživatel přepínat pomocí hlavního menu. Snímky obrazovek lze vidět v sekci 3.6.

- **Zařízení** – správa zařízení zachycujících fotografie SPZ, která lze autentifikovat do systému pomocí QR kódu.
- **Pravidla a Filtry** – definice parkovacích pravidel a filtrů vozidel (popsáno v 3.4). Pro ověření půjde si parkovací pravidla a filtry odsimulovat.
- **Statistiky** – detailněji zobrazené údaje o počtu parkování a výdělku podle roku, měsíce a dne s grafy.
- **Vozidla a Parkování** – prohlížení vozidel a jejich parkování.
- **Správa účtu** – změna údajů a hesla současně přihlášeného uživatele.

3.2 GraphQL resolvers

Protože GraphQL je primárním způsobem komunikace mezi frontendem a backendem, stojí za to si vysvětlit základní stavební bloky GraphQL – resolvers, které tvoří strom. Vzpomněme si na příklad z předchozí kapitoly na obrázku 2.2. Zde kořenem stromu je resolver `userSearch`, jenž přijímá argumenty potřebné pro jeho běh, ty jsou v tomto případě povinné (ale nemusí). Kořenový resolver je funkce vracející nějaký typ, v tomto případě typ `UserSearchResult`, který udává nalezené uživatele (`UserSearchResult.data`) a stránkování (v příkladu vynecháno). Při návrhu tohoto resolveru bychom si mohli rozmyslet, že údaje o stránkování vynecháme a vrátíme pouze pole nepřázdňých uživatelů – typ `[User!]`. Máme dva druhy typů: skaláry (listy stromu) a objekty (uzly stromu). Typ `User` je objekt a má v sobě také nějaké hodnoty a vrátí se nám jen ty, na které se zeptáme. Měl-li by typ `User` další objekty, mohli bychom se zeptat i na jejich hodnoty. Kdybychom psali aplikaci pro veterinární kliniku, mohli bychom se zeptat třeba na jména mazlíčků.

Jak toto implementujeme? Pro každý kořenový resolver musíme napsat funkci, která buď přímo vrátí požadovaný objekt s atributy a nebo k typu, který náš kořenový resolver vrací, napíšeme další resolvers, které vrací opět objekty, nebo skaláry. Kdybychom se databáze ptali na uživatele i s mazlíčky (v SQL terminologii bychom provedli JOIN), tak bychom nemuseli pro typ `User` psát resolver

pro mazlíčky, kde bychom se dotázali na mazlíčky našeho uživatele. (viz The GraphQL Foundation, 2020)

Zároveň můžeme provádět virtualizaci atributů – přidávat atributy, které v databázi nemáme, ale pro webové rozhraní se hodí. Hypotetickým příkladem buď celková zaplacená částka zákazníkem, která se v případě potřeby vypočítá nebo živé, stále měnící se statistiky.

3.2.1 Obecné resolvers

Jelikož se většina operací nad modely se opakuje, byly napsány obecné resolvers jako HOF (Higher-Order-Function) pro vytváření, úpravu, mazání, vyhledávání a získávání modelů v relaci. Mění se část je pak pouze definice databázového modelu. Díky GraphQL se neumí ani ověřovat datové typy – GraphQL toto udělá podle schema, které tvoří programátor.

```
1 function gqlPopulate<D extends mongoose.Document,
2     K extends keyof D>(
3     modelGetter: ModelGetter<D>,
4     key: K
5 ): Resolver {
6     return async function(obj: D, args, ctx: Context, info) {
7         const model = modelGetter(ctx);
8         const keyStr = key.toString();
9         if (obj.populated(keyStr)) {
10             return obj[key];
11         } else {
12             const populated: D = await model.populate(obj, {
13                 path: keyStr
14             });
15             return populated[key];
16         }
17     };
18 }
```

Ukázka obecného resolveru v Typescriptu.

Zde lze vidět funkci vracející resolver, který vrátí objekt v relaci (provede JOIN v SQL terminologii), pokud ho vlastníci objekt ještě nemá. Na sedmém řádku se získá model z funkce v *lexical-scope* funkce vracející resolver. Proměnná *ctx* definuje kontext resolveru a obsahuje modely a klienta, aby bylo snazší resolvery testovat – jedná se o *dependency-injection*.

Na šestém řádku lze vidět parametry, které dostává každý resolver. První parametr je objekt vrácený nadřazeným resolverem. Druhý parametr obsahuje klientem specifikované argumenty, například *id* hledaného uživatele. Třetí parametr je již zmíněný kontext, jeho obsah si lze definovat dle libosti. Poslední parametr obsahuje informace o dotazu samotném – jaké parametry si klient vyžádal, AST (angl. Abstract-Syntax-Tree) dotazu apod.

3.3 Autentizace a autorizace

Autentifikace lidských uživatelů probíhá pomocí standardního hesla a zařízení pomocí dlouhého, náhodně generovaného hesla, které je posíláno na frontend ve

formě QR kódu, které zařízení naskenuje. Po úspěšné autentizaci obdrží klient token, který kdyby klient posílal v souboru Cookie, tak bychom se vystavovali riziku útoku CSRF/XSRF. Místo toho bude klient posílat token v hlavičce *Authorization*.

Podle tokenu je klient jednoznačně identifikovatelný. Endpointy a GraphQL resolvers jsou zabezpečeny tak, že každý endpoint nebo resolver, jenž vyžaduje oprávnění, dáme jako argument HOF (Higher-Order-Function) *checkPermissions* společně s požadovanými oprávněními. *checkPermissions* pak zavolá původní endpoint nebo resolver pouze pokud má klient dostatečná oprávnění.

3.4 Parkovací pravidla

- Různá vozidla mohou podléhat různým pravidlům.
- Pravidla mají prioritu.
- Pravidla mají časové omezení.
- V jednu chvíli může platit více pravidel.

Mechanismus, kterým umožníme vozidlům být ovlivněna některými pravidly, budou filtry. Pro dostatečnou flexibilitu je zapotřebí oddělit samotná pravidla od jejich priority, časového intervalu platnosti i filtrů, k čemuž bude sloužit objekt typu *ParkingRuleAssignment*.

```
type ParkingRuleAssignment {
  rules: [ParkingRule]!
  start: DateTime!
  end: DateTime!
  # ALL nebo NONE
  vehicleFilterMode: VehicleFilterMode!
  vehicleFilters: [VehicleFilter!]
  priority: NonNegativeInt!
}
```

```
type VehicleFilter {
  id: ID!
  name: String!
  # INCLUDE nebo EXCLUDE
  action: VehicleFilterAction!
  vehicles: [Vehicle!]
}
```

Filtrování bude mít dva módy: začneme se všemi vozidly (ALL) a začneme bez vozidel (NONE). Následné filtry mohou buď přidávat, nebo odstraňovat jednotlivá vozidla. Hodí se mít filtry uložené separátně, aby mohli být využity několikrát.

Pro zjednodušení algoritmů, uvalíme omezení: ve stejný čas nesmí existovat více *ParkingRuleAssignment* se stejnou prioritou.

3.4.1 Algoritmus filtru vozidel

Vstup: objekt `ParkingRuleAssignment` s filtry, vozidlo

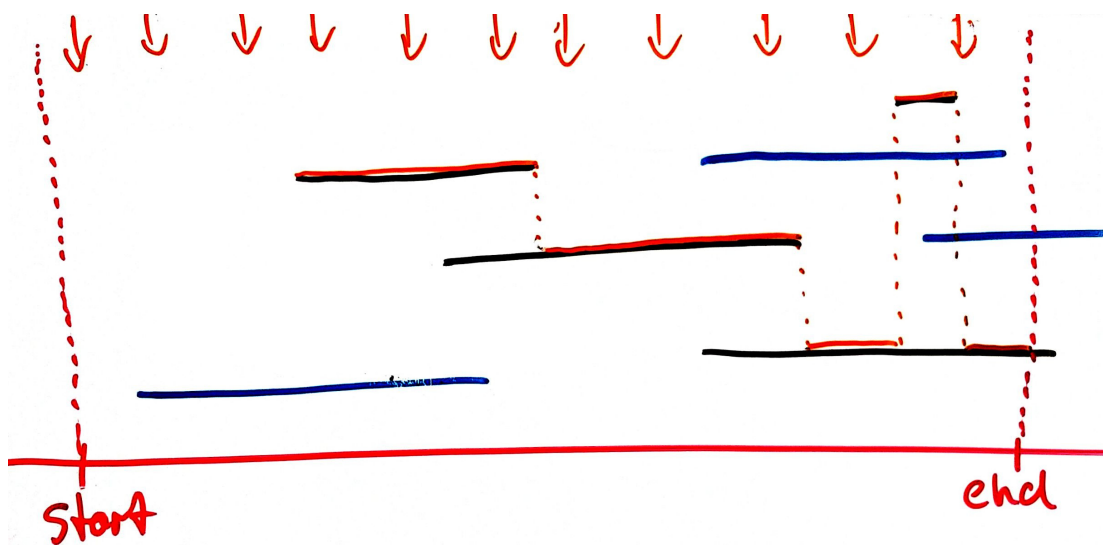
Výstup: boolean určující platnost

1. Na základě módu filtrování si budeme udržovat množinu buď odstraněných vozidel (mód ALL), nebo přidanych vozidel (mód NONE).
2. Podle příslušné akce filtrů (odstranit nebo přidat) budeme množinu našich vozidel manipulovat. Např. je-li mód ALL a filtr odstraňuje, do množiny si vozidla přidáme.
3. Pokud je vozidlo ve výsledné množině, tak pro něj *ParkingRuleAssignment* platí pokud je filtrovací mód NONE, ale neplatí pokud je mód ALL. Opačné výsledky nastanou, pokud vozidlo v množině není.

Časová i paměťová složitost algoritmu je $\mathcal{O}(N)$, kde N je počet filtrů.

Tento algoritmus lze potenciálně rozdělit na předvýpočet (kroky 1. a 2.) a ověření (krok 3.). Tato možnost zvýší paměťovou náročnost, protože bychom si museli pamatovat množiny, což je nepraktické. Je-li uživatel přičetný, počet použitých filtrů nebude obrovský, a tudíž je lepší zvolit následující způsob cachování. Zapamatujeme si výsledky pro určitý seznam filtrů pro konkrétní vozidlo pouze při běhu algoritmu, který je vysvětlen v následující podkapitole, čímž následující algoritmus zrychlíme.

3.4.2 Algoritmus pro aplikaci pravidel



Modré úsečky neplatí kvůli filtrům nebo protože jsou deaktivované. Oranžová čára značí výstup požadovaného algoritmu.

Obrázek 3.3: Ilustrace problému úseček.

V jednom čase může existovat více objektů typu *ParkingRuleAssignment* avšak s různou prioritou. Může se stát, že aplikovaných *ParkingRuleAssignment* bude několik (různé priority, vyprší platnost, etc.).

Situaci si lze představit jako několik navzájem rovnoběžných úseček v různých výškách, které se neprotínají. Nás nyní zajímá, na které a v jakých intervalech na ně dopadne světlo, pokud na ně kolmo zeshora posvítíme. Situaci lze vidět na obrázku 3.3.

Pro zjednodušení předpokládejme, že všechny *ParkingRuleAssignment*, které zpracováváme, platí pro naše vozidlo. Přidat tuto kontrolu později je triviální.

Vstup: seznam ParkingRuleAssignment odpovídající pro interval pobytu vozidla na parkovišti, vozidlo

Výstup: seznam ParkingRuleAssignment s časy platnosti

1. Seřadíme si začátky a konce úseček podle jejich času.
2. Vytvoříme si haldu pro odkládání úseček, která řadí podle priority – větší výše.
3. Vytvoříme si seznam aplikovaných pravidel s časy (časy se mohou lišit od počátečních i koncových časů).
4. Nechť s je současná úsečka a t_s čas zvolení s (čas zvolení se může lišit od začátku úsečky).
5. Pro každou událost u značící začátek/konec úsečky (aplikaci pravidla) p :

(a) Pokud se jedná o začátek nové úsečky:

- i. Pokud není zvolená úsečka:

$t_s \leftarrow p.start$

$s \leftarrow p$

- ii. Pokud je zvolená úsečka a p má vyšší prioritu než s :

s dáme do seznamu aplikovaných pravidel se začátkem t_s a koncem $p.start$.

s dáme na haldu, pokud $s.end > p.end$.

$t_s \leftarrow p.start$

$s \leftarrow p$

- iii. Pokud je zvolená úsečka a p má nižší prioritu než s a $p.end > s.end$:

s dáme na haldu

(b) Jinak (jedná se o konec nějaké úsečky):

- i. Přidáme s do seznamu aplikovaných pravidel se začátkem t_s a koncem $s.end$.

- ii. Taháme z haldy, dokud nedostaneme úsečku s koncem později než koncem p , nebo dokud halda není prázdná.

- iii. Pokud jsme z haldy vhodnou úsečku vytáhli, použijeme ji. V opačném případě vyprázdníme s a t_s .

Algoritmus zajisté doběhne, protože máme konečný počet událostí a v každém cyklu jednu zpracujeme. Paměťová složitost je zřejmě lineární. Časová složitost bez filtrování je $\mathcal{O}(N \cdot \log N)$, kde N je počet úseček, protože využijeme některého z rychlých řazení a protože použijeme binární haldu nebo lepší haldu. Počet operací nad haldou, který by konečnou složitost mohl změnit, je naštěstí lineární. Největší počet operací nad haldou dostaneme tak, že každou přidanou úsečku umístíme

tak, aby při zpracování jejího konce i začátku došlo k přidání a odebrání z haldy. Jedno z takovýchto uspořádání je například pyramida, popřípadě zikkurat. Tedy s každou přidanou úsečkou se počet operací nad haldou zvýší maximálně o 2, což je lineární vzhledem k počtu úseček.

Přidáme-li filtrování, tak v nejhorším případě budeme ověřovat platnost každé úsečky. Bude-li M průměrný počet filtrů, pak je časová složitost $\mathcal{O}(N \cdot M)$.

3.5 Uživatelské rozhraní

Jelikož autor neměl zkušenosti s vývojem webového uživatelského rozhraní, rozhodl se využít starter projekt společnosti Crazy Factory GmbH, který pojí vybrané technologie dohromady a mandatuje projektu základní strukturu. Navíc podporuje překlady, reportování chyb apod., což vytvářet z ničeho je obtížné. (viz Crazy Factory GmbH, 2019)

3.5.1 Redux a persistence stavu

Jak již bylo zmíněno v předchozí kapitole, uživatelské rozhraní používá Redux pro sdílení stavu mezi komponentami. Stav v Reduxu je strom, jenž je upravován akcemi. Akce a současný stav je předán takzvaným *reducers*, což jsou prosté funkce, které vrátí stav nový. (viz Dan Abramov et al., 2020) Prostost reducerů zajišťuje testovatelnost.

Jako příklad konkrétního využití budiž sdílení některých údajů mezi obrazovkami. Například sdílení vozidla mezi stránkou s vozidly a stránkou s pravidly, schopnost seznamu se záznamy o parkování zvolit vozidlo a mnoho dalších.

Některé komponenty jsou pak na stavu závislé a dojde k jejich překreslení, pokud se jimi čtený stav změní, což je též zařízeno Reduxem. Jedná se vlastně o formu *dependency-injection*. Stejným mechanismem jsou do komponent vkládány funkce vytvářející akce a jiné vedlejší efekty, což dělá tyto komponenty snadno testovatelné.

K uložení stavu aplikace po zavření okna, použijeme knihovnu *redux-persist*, která poskytuje HOF funkci, které dáme reducer, a ona změny, které způsobí, někam uloží. (viz rt2zz, 2020) V našem případě se jedná o *localStorage*.

3.5.2 Typy komponent

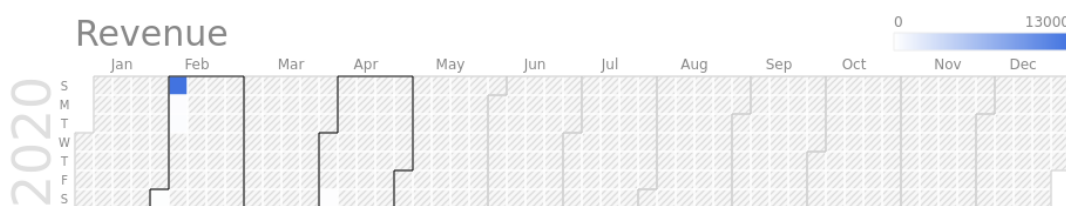
Níže je rozlišení komponent podle jejich interakce s ostatními a míry autonomie. Jedná se o spektrum, ne o definitivní kategorie.

- **Bez vnitřního stavu** – může se jednat o tlačítka, vstupy textu apod. Tyto komponenty jsou zcela řízeny nadřazenou komponentou.
 - **Strukturní** – delegují data dalším komponentám a mohou mít podřízené komponenty. Příkladem budiž sekce třeba v uživatelském rozhraní se stejným stylem.
- **S vnitřním stavem** – tyto komponenty údajují stav podřízeným komponentám.

- **Napojené na Redux** – jedná se o podtyp komponent s vnitřním stavem, které část stavu berou z Reduxu.
- **Komunikující se světem** – komunikují s vnějším světem. Například posílají GraphQL dotaz backendu.

3.5.3 Grafy

Grafy jsou potřeba na stránce se statistikami. K tomu byla použita knihovna `react-google-charts`, což, jak název napovídá, je port knihovny `google-charts` do Reactu. Ta kromě obvyklých grafů umí například i takzvaný kalendářový graf, jenž lze vidět na obrázku 3.4.



Obrázek 3.4: Kalendářový graf z knihovny `react-google-charts`.

3.5.4 Obecná vybíratka modelů

Dle principu neopakování se bylo vytvořeno několik obecných UI komponent, které umožňují vyhledávání libovolných modelů a jejich volbu a využití v ostatních komponentách. Ve zdrojovém kódu je implementujeme jako HOF (Higher-Order-Function), což je funkce vracející další funkce – konkrétní komponenty. Mění se části, které se do těchto obecných komponent budou vkládat je komponenta renderující jediný model (*renderModel*), GraphQL dotaz pro získávání modelů (*queryString*), funkce, jejímž vstupem je odpověď na GraphQL dotaz a výstupem je samotné pole modelů (*modelArrayGetter*), a funkce, jejímž vstupem je dotazovací řetězec a výstupem jsou argumenty pro GraphQL dotaz (*identifierToOptions*).

Obrázek 3.5 ukazuje tok dat v jednom z obecných vybíratek. Obrázek 3.6 ukazuje vyrenderovanou komponentu vybíratka.

Další vybíratka abstrahuje i vstup od uživatele, takže lze použít například kalendář.

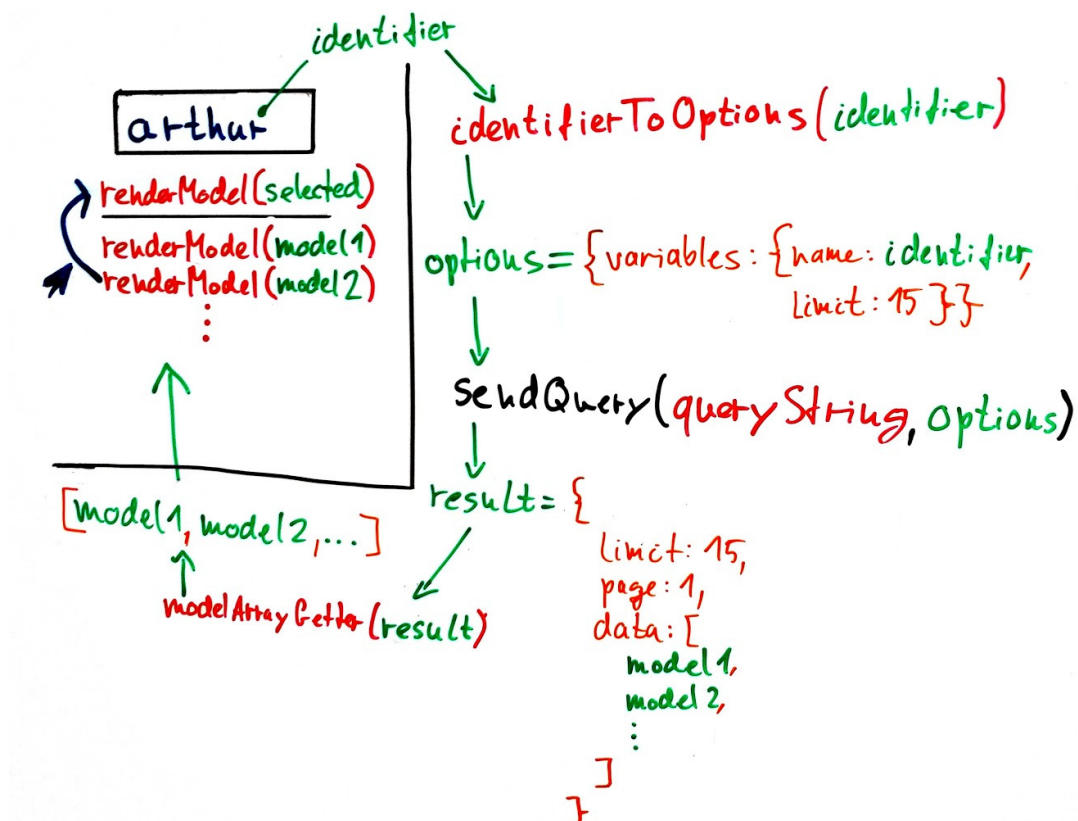
3.6 Snímky obrazovky obrazovek

Zařízení

Viz obrázek 3.7. Po vypršení QR kódu je uživateli nabídnuto vytvořit nový.

Pravidla a Filtry

Viz obrázek 3.8. Oranžová linka v kalendáři značí simulaci pravidel, jejíž výsledek je cena viditelná vlevo dole v sekci Simulation.



Obrázek 3.5: Tok dat v jednom z obecných vybírátek.

Obrázek 3.6: Vyrenderovaná komponenta jednoho vybírátko.

Vozidla a parkování

Viz obrázek 3.9. Pokud uživatel najede myší na čas vjezdu nebo výjezdu, tak se mu zobrazí výřez SPZ z pořízeného snímku. Zároveň lze kliknout na SPZ a zobrazit si ostatní záznamy onoho vozidla na stejné obrazovce. Součástí záznamu je i kopie použitých pravidel, protože mohou být zpětně změněna správcem.

Statistics

Rules & Filters

Devices

Vehicles

User Management

user1

Logout

Profile

Devices

Activated

any

Fetch Devices

Name	Activated	Activated At	Actions	
Ig1	YES	2/17/2020, 8:24:43 AM	<div>Details</div>	<div>Delete</div>

Type

OUT

IN

Capturing

NO

YES

Minimal Area

3000

Resize X

1000

Resize Y

1000

Save


Ig2

NO

Details

Delete

Activation expires in 3 seconds



Name

Create

Obrázek 3.7: Stránka pro zařízení.

Vehicle Selector Mode <any> Day 02/29/2020

0h	1h	2h	3h	4h	5h	6h	7h	8h	9h	10h	11h	12h	13h	14h	15h	16h	17h	18h	19h	20h	21h	22h	23h
no fee - NONE include 1AN9714																							

Delete Save Reset Close

Active ☐ NO ☒ YES
Start 02/04/2020 12:00 PM
End 06/04/2020 01:00 PM
Priority 8
Filter Mode NONE ALL
Filters INC include 1AN9714
Rules TimedFee no fee

1h free 100/h - ALL
100/h - ALL

Simulation OFF ON
Vehicle CCC1234
Start 02/29/2020 02:00 PM
End 02/29/2020 06:29 PM
Fee 400

Vehicle Filters New
include 1AN9714
Action EXCLUDE INCLUDE
Name include 1AN9714
Vehicles 1AN9714
Delete Save

Parking Rules New
100/h
Name 100/h
Time unit HOUR
Fee per HOUR 100
Free HOURS 0
Rounding method CEIL
Delete Save

Obrázek 3.8: Stránka s pravidly, filtry a simulací pravidel.

Parking Session				Applied Parking Rules							
Start	End	Total	Vehicle	A#	Name	Type	Permit	Fee	T unit	Free units	Rounding
2/29/2020, 5:00:23 PM	6:58:14 PM	200	1AN9714	1	100/h	TimedFee	-	100	H	0	CEIL
Applied Rule Assignments											
#	Start	End	Subtotal	# of rules							
1	2/29/2020, 5:00:23 PM	6:58:14 PM	200	1							

Obrázek 3.9: Záznam o parkování.

4. Rozpoznávání SPZ

Jak již bylo řečeno v kapitole 2, mobilní aplikace pořídí snímek, pošle ho na Backend, jenž ho pošle serveru s knihovnou OpenALPR, která rozpozná SPZ a výsledek pošle zpět na backend. V této kapitole si popíšeme mobilní aplikaci a server s OpenALPR.

4.1 Zvyšování přesnosti

4.1.1 Cachování výsledků

Výsledek z knihovny OpenALPR je seznam dvojic udávající SPZ a šanci, že konkrétní SPZ je správně – jak si OpenALPR věří ve výsledek. Je tudíž logické měření udělat víc a provést aritmetický průměr a zvolit nejlepší výsledek.

K ukládání takto dočasných dat (přibližně počet měření krát 1 sekunda) se databáze nehodí, a proto bylo zavedeno ukládání do mezipaměti. V současné chvíli se využívá prostá paměť backendu, kde klíčem je *id* zařízení. Díky tomu, že Node.js běží na jednom vlákně, nemusíme se bát souběhu (angl. race-condition). Externí mezipaměť by bylo vhodné využít (např. Redis), pokud by se spouštělo více instancí backendu a prováděl by se takzvaný *load-balancing*.

Výchozí počet měření je 2, a lze ho upravit v konfiguraci backendu.

4.1.2 Filtrování podle geometrického obsahu

Pokud OpenALPR nalezne SPZ, udá i její pozici ve zdrojovém obrázku. Aby se tedy předešlo naskenování SPZ, které jsou například daleko, lze odfiltrovat SPZ podle jejich obsahu v pixelech čtverečních. Konkrétní hodnota je potřeba odladit na místě skenování a lze změnit ve webové aplikaci pro kterékoliv zařízení.

4.2 Autentifikace

Zařízení se autentifikuje naskenováním QR kódu, jenž lze najít ve webové aplikaci. Ten obsahuje JSON řetězec s aktivačním heslem, pomocí kterého se zařízení přihlásí do systému a získá svou konfiguraci.

Samotné skenování QR kódu je provedeno externí aplikací Barcode Scanner od vývojáře Zxing Team, která lze nainstalovat z Play Store.

Konkrétní mechanismus komunikace s touto externí byl převzán. (viz Seshu Vinay, 2019)

4.3 Volba zařízení pro mobilní aplikaci

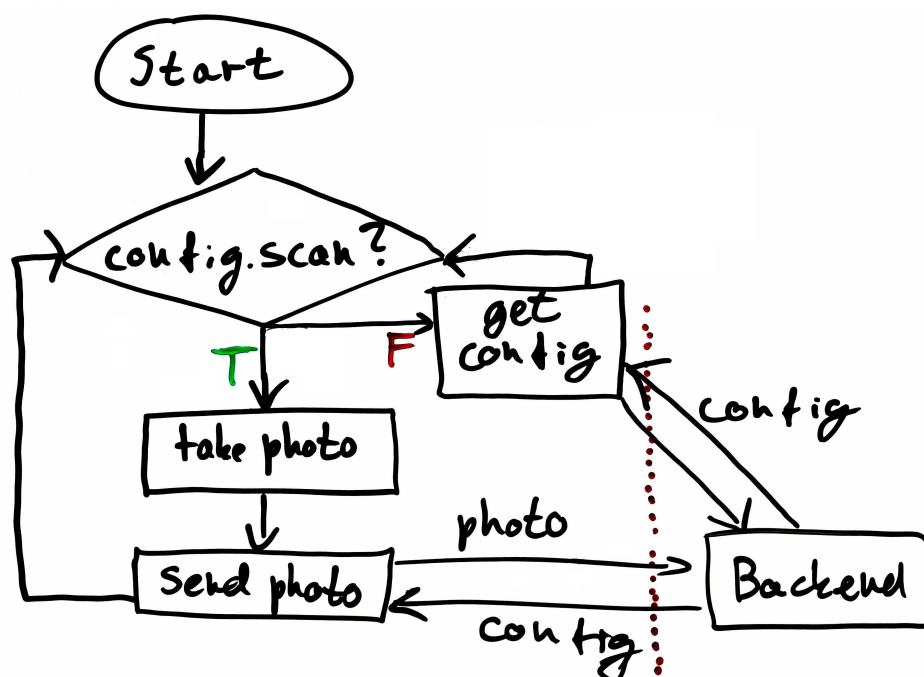
Co se týče hardwarového vybavení snímacího zařízení, tak je vyžadována přední kamera s rozlišením alespoň 1000 na 1000 pixelů. Důvod pro toto rozlišení je, že backend obdržený snímek stejně zmenší na 1000x1000 pixelů (lze však změnit ve webové aplikaci), aby knihovna OpenALPR provedla rozpoznání co nejrychleji, a zároveň aby bylo rozpoznání dostatečně přesné. Procesor, RAM i

vnitřní paměť může být libovolná – kterékoliv dnešní nové zařízení bohatě postačí (za předpokladu, že vnitřní paměť není zaplněná). Minimální verze Androidu je 5 (SDK 21).

Autorovi se nepodařilo najít způsob, jak zároveň pořizovat v pravidelném intervalu snímky a mít zařízení uzamknuté proti přístupu. K zajištění pořizování snímků si hlavní obrazovka aplikace řekne systému Android o zabránění uzamknutí. To má dva následky. První je, že zařízení by nemělo mít OLED displej, aby nedošlo k takzvanému *burn-in* (viz Geoffrey Morrison, 2019). Druhý je, že zařízení by mělo být v produkčním provozu bezpečně uzavřeno v krabici, nebo by se mělo nacházet na bezpečném místě, aby se předešlo nepovolené manipulaci.

4.4 Životní cyklus mobilní aplikace

Na obrázku 4.1 lze vidět životní cyklus mobilní aplikace. Proces neprobíhá na jednom vlákně. Jakmile se pořídí fotografie, tak začne odpočet kolem jedné sekundy, po kterém se pořídí další, a zároveň se už posílá první fotografie. Změnili se konfigurace na backendu, tak je posílána zařízení při dalším kontaktu, jinak konfigurace posílána není.

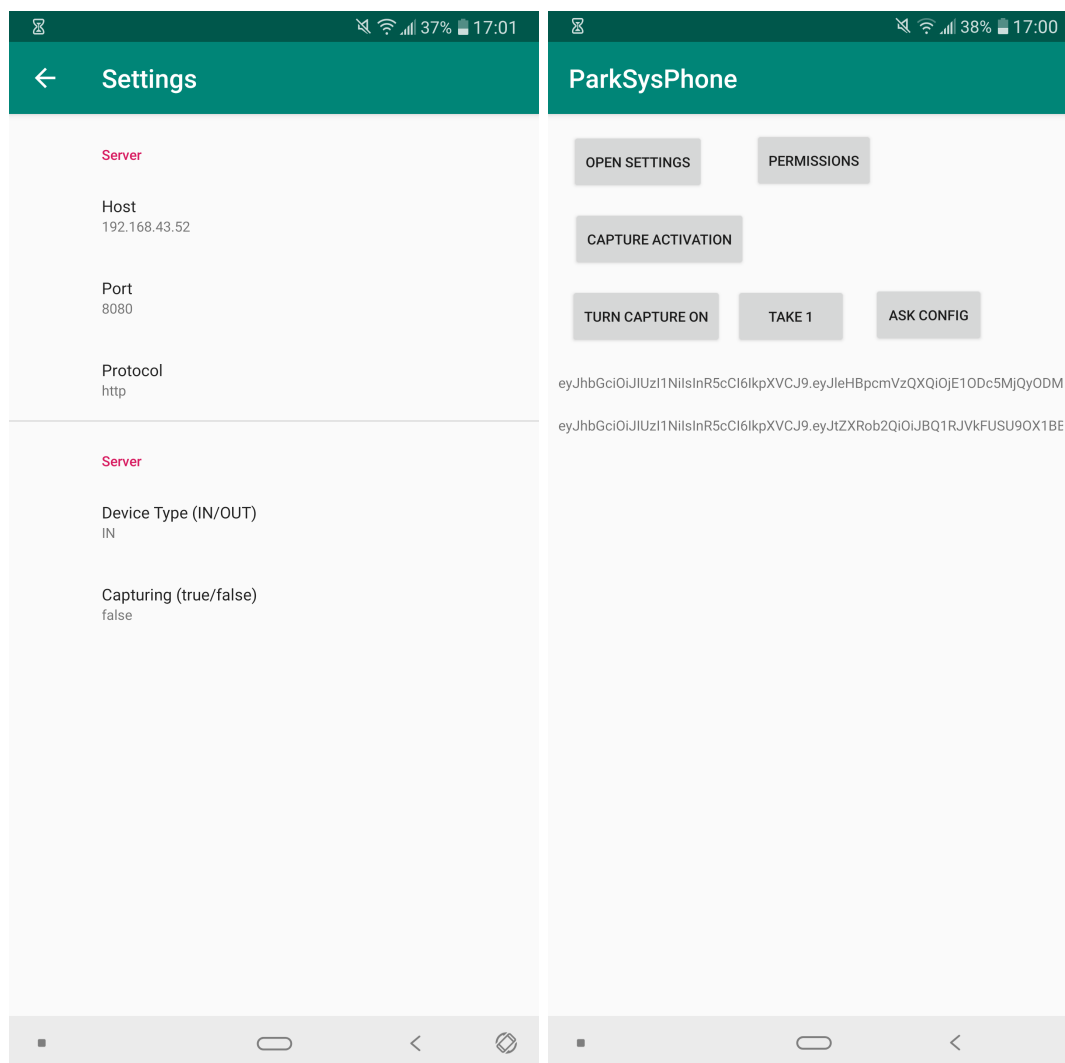


Obrázek 4.1: Životní cyklus mobilní aplikace.

4.5 Uživatelské rozhraní mobilní aplikace

Uživatelské rozhraní se skládá ze dvou obrazovek. Na obrázcích 4.2 lze vidět obě – hlavní obrazovku a nastavení.

V nastavení lze nastavit adresu backendu a vybrat si mezi HTTP a HTTPS.



Obrázek 4.2: Rozhraní mobilní aplikace.

4.6 Implementační detaily mobilní aplikace

4.6.1 Komunikace s backendem

Ke komunikaci přes HTTP využívá aplikace knihovnu Volley, která je doporučena v Android dokumentaci. Princip použití je takový, že si vytvoříme *singleton* obstarávající frontu žádostí, kterému předáváme HTTP žádosti s *callback* funkcí obsluhující odpověď. (viz Google LLC, 2019b)

4.6.2 Ukládání snímků

Snímky se ukládají do paměti určené pro aplikaci. Kdyby se použili dočasné soubory, mohlo by se stát, že je systém před posláním nemilosrdně smaže. (viz Google LLC, 2019a) Aplikace tedy musí obstarávat i mazání souborů, což se provádí ihned po odeslání snímku na backend.

5. Instalace

Instalace celého systému je velice jednoduchá. Pro distribuci je použit nástroj Docker, který zajistí kompilaci, spuštění, propojení všech částí a odhalení potřebných služeb ven na internet. (viz Docker Inc., 2020)

Postup Instalace

1. Po stažení git repozitáře se zdrojovým kódem nastavíme submodule:

```
$ git submodule init
$ git submodule update --remote --recursive
```

2. Dle potřeby upravíme soubor */docker-compose.yml*.
3. Celý systém spustíme.

```
$ docker-compose up
```

Poslední krok může trvat i několik minut v závislosti na rychlosti internetového připojení. Protože Docker zabalí vše včetně systémových závislostí, velikost výsledných imagů je kolem 1,4GB.

Pro detaily je potřeba konzultovat soubor */docker-compose.yml* a jednotlivé soubory zvané *Dockerfile* každé komponenty.

Závěr

Výsledný projekt splňuje celé zadání kromě jednoho bodu – konkrétně se jedná o různý provoz o svátcích a podobných dnech (viz 1). Implementace takovéto funkcionality vyžaduje implementovat kalendář. Co se týče rozpoznávání SPZ, statistik, samotných pravidel a filtrů vozidel, tak zde je vše implementováno a funguje skvěle. Při prohlížení záznamů parkování lze nahlédnout na výřez SPZ z pořízeného snímku. Navíc lze pravidla a filtry simulovat pro libovolné vozidlo.

Vývoj byl díky vhodně zvoleným technologiím poměrně rychlý a bezbolestný. V jeho průběhu nedošlo k žádnému backtrackování kvůli předchozím rozhodnutím. Projekt je bez velkých obtíží rozšiřitelný a pozměnitelný. Dalším rozšířením, aby projekt byl plnohodnotný parkovací systém, by byla integrace s platebním terminálem a závorou.

Seznam použité literatury

- CHARLIE ROBBINS (2019). URL <https://www.npmjs.com/package/nconf>.
- CRAZY FACTORY GMBH (2019). URL <https://github.com/crazyfactory/ts-react-boilerplate>.
- DAN ABRAMOV ET AL. (2020). URL <https://redux.js.org/introduction/core-concepts>.
- DOCKER INC. (2020). URL <https://docs.docker.com>.
- ER AJAY PRATAP (2018). React js redux js data flow. URL <https://www.reactreduxtutorials.com/2018/02/redux-tutorial-for-beginners-redux-data-flow-redux-lifecycle.html>.
- FACEBOOK (2019). URL <https://reactjs.org/>.
- GEOFFREY MORRISON, D. K. (2019). Oled screen burn-in: What you need to know now. URL <https://www.cnet.com/how-to/oled-screen-burn-in-what-you-need-to-know-now>.
- GERHARDSLETTEN (2019). URL <https://github.com/gerhardsletten/express-openalpr-server>.
- GOOGLE LLC (2019a). URL <https://developer.android.com/training/data-storage/app-specific>.
- GOOGLE LLC (2019b). URL <https://developer.android.com/training/volley/requestqueue.html#singleton>.
- LEARNBOOST (2019). URL <https://mongoosejs.com/>.
- METEOR DEVELOPMENT GROUP INC. (2020). URL <https://www.apollographql.com>.
- MICROSOFT (2019). URL <http://www.typescriptlang.org/>.
- MONGODB INC. (2020). URL <https://www.mongodb.com>.
- OPENALPR (2018). URL <https://github.com/openalpr/openalpr>.
- OPENJS FOUNDATION (2019). URL <https://nodejs.org/en/>.
- RT2ZZ (2020). URL <https://github.com/rt2zz/redux-persist>.
- SESHU VINAY (2019). URL <https://stackoverflow.com/questions/8830647/how-to-scan-qr-code-in-android/8830801>.
- THE GRAPHQL FOUNDATION (2020). URL <https://graphql.org/learn/queries/#fields>.
- TYPESTYLE (2020). URL <https://github.com/typestyle/typestyle>.

Seznam obrázků

2.1	Diagram komponent a jejich komunikace.	6
2.2	Příklad GraphQL dotazu (vlevo) a odpovědi (vpravo). Screenshot z nástroje GraphQL Playground.	6
2.3	Spojení React a Redux. (viz Er Ajay Pratap, 2018)	7
3.1	Adresářová struktura backendu.	9
3.2	Adresářová struktura frontendu.	9
3.3	Ilustrace problému úseček.	13
3.4	Kalendářový graf z knihovny react-google-charts.	16
3.5	Tok dat v jednom z obecných vybírátek.	17
3.6	Vyrenderovaná komponenta jednoho vybírátko.	17
3.7	Stránka pro zařízení.	18
3.8	Stránka s pravidly, filtry a simulací pravidel.	19
3.9	Záznam o parkování.	19
4.1	Životní cyklus mobilní aplikace.	22
4.2	Rozhraní mobilní aplikace.	23