

STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor č. 18: Informatika

Parkovací systém

**Tomáš Černý
Hlavní město Praha**

Praha 2020

STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor č. 18: Informatika

Parkovací systém

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: 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 pomocí automatickým rozpoznáváním 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. To vše v přehledném uživatelském rozhraní.

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

Klíčová slova: rozpoznávání SPZ, počítačové vidění, React, SPA, Typescript, GraphQL

Title: 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. All in a well-arranged user interface.

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

Keywords: license plate recognition, computer vision, React, SPA, Typescript, GraphQL

Obsah

1	Úvod	3
2	Architektura a technologie	4
2.1	Architektura řešení	4
2.2	Technologie	4
2.2.1	Databáze	4
2.2.2	Backend	4
2.2.3	Frontend	6
2.2.4	Mobilní aplikce	7
2.2.5	Detekce SPZ	7
2.3	Metodika vývoje	7
3	Webová aplikace	8
3.1	Obrazovky	9
3.1.1	Správa účtu	9
3.1.2	Zařízení	9
3.1.3	Pravidla a Filtry	9
3.1.4	Vozidla a Parkování	9
3.1.5	Statistiky	9
3.2	Autentizace a autorizace	11
3.3	Komunikace	13
3.3.1	Endpointy backendu	13
3.3.2	GraphQL	14
3.3.3	Top-level GraphQL typy	15
3.3.4	Obecné resolvency	16
3.3.5	Tvorba Apollo klienta	16
3.3.6	Získávání obrázků	17
3.4	Ukládání a mazání výřezů SPZ	17
3.5	Parkovací pravidla	18
3.5.1	Algoritmus filtru vozidel	19
3.5.2	Algoritmus pro aplikaci pravidel	19
3.5.3	Algoritmus pro mazání přiřazení pravidel	21
3.6	Uživatelské rozhraní	21
3.6.1	Redux a persistence stavu	21
3.6.2	Typy komponent	23
3.6.3	Základní prvky uživatelského rozhraní	23
3.6.4	Grafy	24
3.6.5	Obecná vybírátko modelů	24
4	Rozpoznávání SPZ a mobilní aplikace	27
4.1	Zvyšování přesnosti	27
4.1.1	Cachování výsledků	27
4.1.2	Filtrování podle geometrického obsahu	27
4.2	Komunikace zařízení a backendu	27
4.2.1	Autentizace	27

4.2.2	Zabezpečení komunikace s mobilní aplikací	28
4.2.3	Optimalizace velikosti přenesených dat	28
4.3	Mobilní aplikace	28
4.3.1	Volba zařízení pro mobilní aplikaci	28
4.3.2	Životní cyklus mobilní aplikace	29
4.3.3	Uživatelské rozhraní mobilní aplikace	29
4.3.4	Implementační detaily mobilní aplikace	29
5	Instalace	31
5.1	Soubory nastavení	31
5.2	Instalace Dockerem	32
5.3	Instalace bez Dockeru	32
Závěr		34
Bibliografie		35
Seznam obrázků		37
A	GraphQL Schema	38

1. Úvod

Tento dokument se zabývá koncepcí a implementací informačního systému pro uzavřená parkoviště s automatickým rozpoznáváním SPZ bez použití drahého kamerového hardware. Inspirací pro projekt bylo, že se po autorovi při výjezdu z parkoviště nechtěl parkovací lístek, nýbrž vozidlo bylo puštěno na základě SPZ.

Projekt 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.

Projekt zároveň slouží k vyzkoušení si moderních webových technologií a frameworků s přesahem k mobilnímu vývoji a počítačovému vidění.

Cíle

- Rozpoznávání SPZ levným hardwarem.
- Vytváření flexibilních parkovacích pravidel.
- Přívětivé uživatelské rozhraní.
- Statistiky a pěkné 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. Zajíšťuje 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 přes HTTP, která rozpoznává SPZ.
- **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. MongoDB je takzvaná NoSQL databáze – k dotazování se nepoužívá SQL, nýbrž vlastní způsob dotazování v JSON. Základní datovou jednotkou je JSON/BSON dokument. Mimo jiné umožňuje provoz několika spolupracujících instancí, zálohování apod. [1]

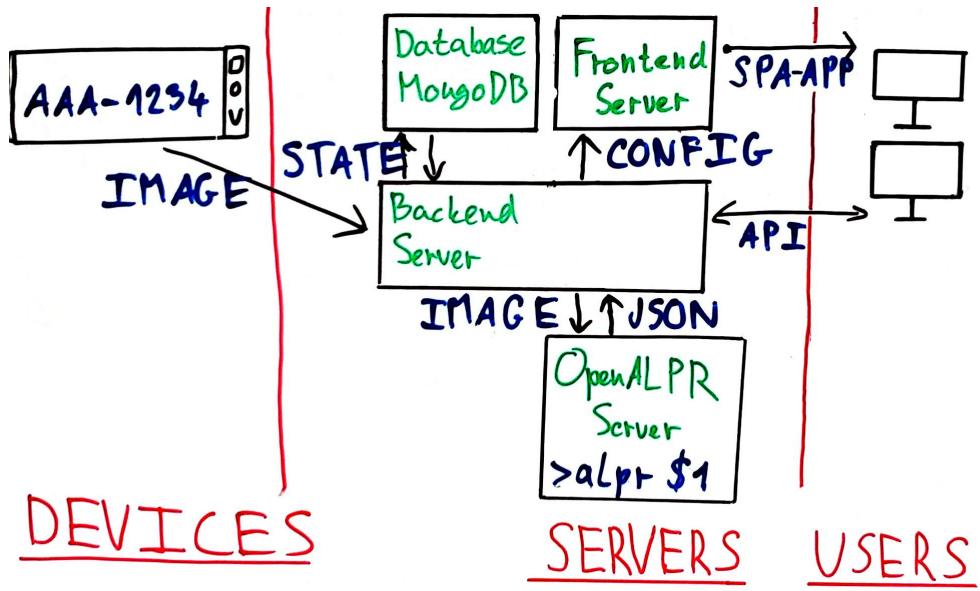
2.2.2 Backend

Jako programovací jazyk pro backend byl zvolen staticky typovaný Typescript kvůli rychlosti vývoje a množství knihoven, které poskytuje ekosystém Node.js. [2] [3]

Pro definici databázových modelů a komunikaci s databází byla kvůli své vyspělosti a skvělé funkcionality zvolena knihovna mongoose. [4]

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. [5] Pro tento projekt byla zvolena implementace Apollo, protože poskytuje knihovnu pro backend i frontend. [6]



Obrázek 2.1: Diagram komponent a jejich komunikace. Serverové části nemusí běžet na separátních strojích.

```

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 přenese více bytů, než při použití obvyklého způsobu přes HTTP. Z toho důvodu pro posílání obrázků bude mít backend i jiné endpointy.

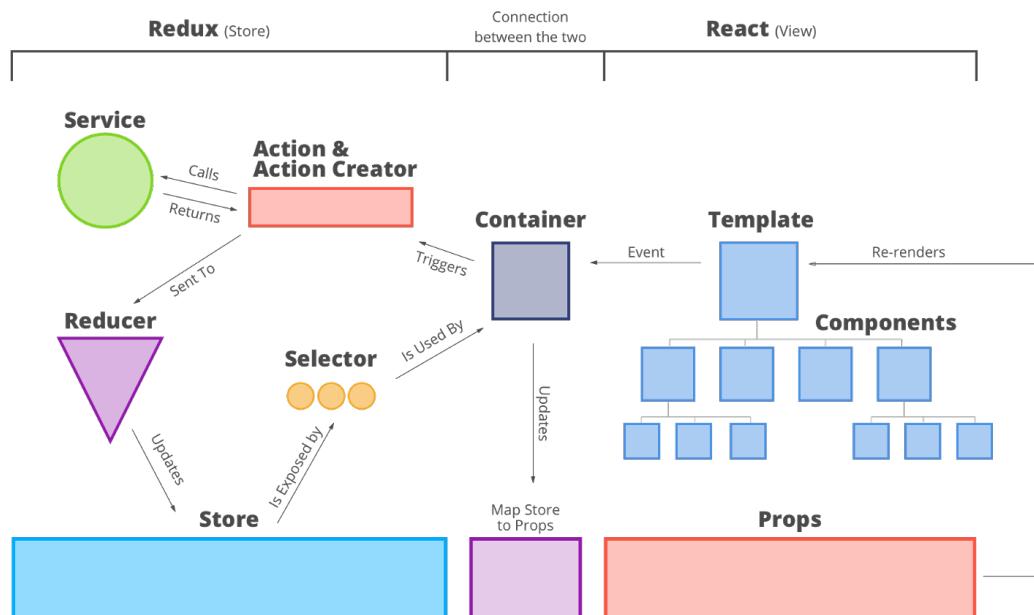
2.2.3 Frontend

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. [7] Pomocí další knihovny typestyle pak můžeme do stejného souboru psát i typované CSS.[8]

Aby bylo možné snadno sdílet mezi komponentami stav, byla pro takzvaný *state-management* zvolena knihovna Redux. [9] Diagram na obrázku 2.3 ukazuje tok dat mezi Reactem a Reduxem. Tato architektura zároveň umožňuje testovatelnost – můžeme jednoduše ověřit, zdali byla akce například po kliknutí na tlačítko komponenty vyvolána, a jak komponenta reagovala na změnu stavu.

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. [10] Komponenty vyvolávají akce pomocí *action-creators*, akce jsou předány prostým funkčním zvaným *reducers*, ty aktualizují stav. Každá komponenta napojená na Redux má *selector* funkci, pokud se výstup z *selector* funkce aktualizuje, komponenta je přerenderována Reactem. V obrázku je prvek zvaný *Container*, ten je HOF (abbrv. Higher-Order-Function) z knihovny react-redux pro komponentu a dodává jí přístup ke stavu a vytváření akcí. Reakce na akci může být i dotaz na server nebo jiný vedlejší efekt mimo komponentu.

2.2.4 Mobilní aplikce

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 nízkoúrovňovým jazykem nemá takové zkušenosti. Kotlin oproti Javě umožnuje 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. [11]

Ke zbytku aplikace bude připojena malým HTTP serverem, jenž byl převzat a upraven. Ten umožnuje poslat pomocí 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. [12]

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 vyhlaďí a zlepší. To však neznamená, že bychom si neměli záležet na kvalitním a udržitelném kódu, naopak. Cílem je mít flexibilní základ, na kterém lze stavět. Pro jistotu 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). Nejprve ukazuje dokončené části a dále vysvětluje zvolené postupy.

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** – helper 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** – helper funkce pro získávání obrázků z backendu
 - **components** – komponenty uživatelského rozhraní
 - **constants** – GraphQL dotazy, barvy, apod
 - **helpers** – helper 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.

Některé konfigurovatelné hodnoty jsou brané 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. [13] Například cesta `{ mongo: { host } }` lze přepsat proměnnou prostředí s názvem `mongo__host`.

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

3.1 Obrazovky

Mějme po přihlášení do webového rozhraní následující obrazovky, mezi kterými bude uživatel přepínat pomocí hlavního menu.

3.1.1 Správa účtu

Tato stránka umožňuje změnu údajů a hesla současně přihlášeného uživatele.

3.1.2 Zařízení

Na této stránce je správa zařízení zachycujících snímky SPZ, která lze autentifikovat do systému pomocí QR kódu. Pokud však zařízení není aktivováno a QR kód vyprší, tak lze vygenerovat nový. Obrazovka také zobrazuje čas posledního kontaktu se zařízením. Nastavení zařízení obsahuje směr (dovnitř/ven), parametry pro detekci SPZ popsané v podkapitole 4.1 a parametry pro zmenšení snímky popsané v podkapitole 4.2.3.

Screenshot obrazovky je na obrázku 3.3.

3.1.3 Pravidla a Filtry

Tato stránka umožňuje definici parkovacích pravidel a filtrů vozidel (popsáno v 3.5). Pro ověření je možné parkovací pravidla a filtry simulovat. Obrazovka obsahuje pohled v rámci dne a měsíce. Je-li potřeba mít ta samá pravidla několik týdnů, měsíců nebo let, lze požadovaný rozsah (týden, měsíc, pář dnů) vybrat a zkopírovat buď na několik míst, nebo zvolit počáteční datum a počet zkopírování do budoucnosti.

Screenshoty obrazovky jsou na obrázcích 3.4, 3.5 a 3.6.

3.1.4 Vozidla a Parkování

Na této stránce lze zobrazit záznamy parkování, které obsahují čas vjezdu a výjezdu, aplikovaná pravidla, vypočítanou částku k zaplacení pro konkrétní záznam a také vozidla, kdy parkovala a kolik bylo celkem naúčtováno jednomu vozidlu.

Screenshoty obrazovky jsou na obrázcích 3.7 a 3.8.

3.1.5 Statistiky

Tato stránka detailně zobrazuje údaje o počtu parkování a výdělku podle roku, měsíce a dne s grafy. Zároveň zobrazuje počet vozidel na parkovišti. Jeden z grafů je zobrazen na obrázku 3.19 v podpodkapitole 3.6.4.

Devices

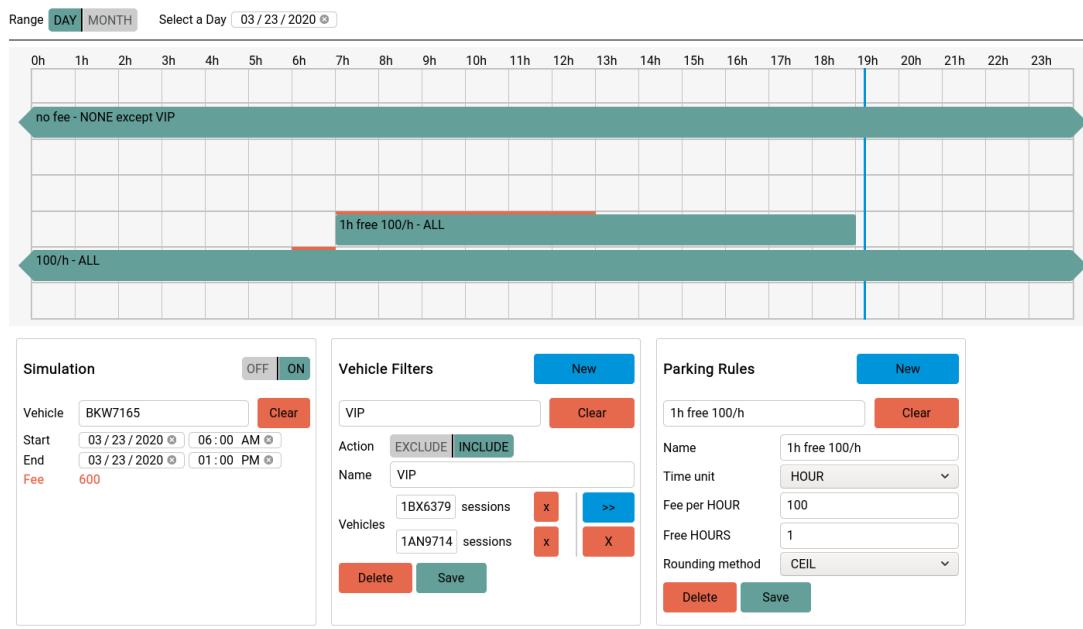
Reload (in 12s)

Name	Activated	Activated At	Last Contact	Actions
lg1	YES	3/23/2020, 6:38:19 PM	1.1 mins ago	Details Delete
lg2	YES	3/23/2020, 6:39:48 PM	never	Details Delete
Type <input type="radio"/> OUT <input checked="" type="radio"/> IN Capturing <input type="radio"/> NO <input checked="" type="radio"/> YES Minimal Area <input type="text" value="0"/> Resize X <input type="text" value="1000"/> Resize Y <input type="text" value="1000"/>				
Save				
lg3	NO	never	never	Details Delete
<i>Activation expires in <u>27 seconds</u></i>				
Name <input type="text"/> Create				

© Tomáš Černý
<tmscer@gmail.com>

Obrázek 3.3: Stránka pro zařízení s hlavním menu. Stránka se pravidelně aktualizuje. Pokud byl poslední kontakt se zařízením před více jak minutou, čas posledního kontaktu se podbarví červeně. Pokud byl poslední kontakt se zařízením před více jak hodinou, místo relativního časo se zobrazí čas absolutní.

Rules & Filters



Obrázek 3.4: Stránka s pravidly a filtry – pohled na jeden den. Na horizontální ose je čas a na vertikální ose je priorita přiřazení pravidla. Kliknutím na blok v kalendáři se otevře editace přiřazení pravidla. Oranžová čára značí simulaci a vlevo dole lze spatřit výpočítanou částku k zaplacení. Napravo od panelu se simulací jsou editory definic filtrů a pravidel. Modrá horizontální čára značí aktuální čas.

3.2 Autentizace a autorizace

Autentifikace lidských uživatelů probíhá pomocí standardního hesla a autentifikace zařízení pomocí dlouhého, náhodně generovaného hesla, které je posíláno na frontend ve formě QR kódu, které zařízení naskenuje. Po úspěšné autentizaci obdrží klient token. Kdyby klient posílal token v souboru Cookie, vystavoval by se riziku útoku CSRF/XSRF. Místo toho bude klient posílat token v hlavičce *Authorization*.

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

Rules & Filters

Range **DAY** **MONTH** Select a Month 03 / 31 / 2020 ⏺

Mon	Tue	Wed	Thu	Fri	Sat	Sun
24	25	26	27	28	29	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Select a Rule Assignment

Quick Actions

Currently Selecting SOURCE | DESTINATION

Copy Destinations

2020-03-08 Delete

2020-03-15 Delete

2020-03-29 Delete

Copy **Delete**

Obrázek 3.5: Stránka s pravidly a filtry – pohled na jeden měsíc. Kliknutí na tlačítko *Copy* by zkopiřovalo přiřazení pravidel mezi žlutě označenými dny na černě označené dny. Výsledek zkopiřování je na následujícím obrázku.

Rules & Filters

Range **DAY** **MONTH** Select a Month 03 / 31 / 2020 ⏺

Mon	Tue	Wed	Thu	Fri	Sat	Sun
24	25	26	27	28	29	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Active 03 / 08 / 2020 ⏺ 02:00 AM ⏺
Start 03 / 08 / 2020 ⏺ End 03 / 08 / 2020 ⏺ Priority 2

Filter Mode **NONE** **ALL**

Filters INC VIP **>>** **X**

Rules TimedFee no fee **>>** **X**

Delete **Save** **Reset** **Close**

Quick Actions

Currently Selecting SOURCE | DESTINATION

Copy Destinations

Empty

Copy Mode **MULTI** **REPEAT**

Copy **Delete**

Obrázek 3.6: Stránka s pravidly a filtry – výsledek kopírování. Vpravo nahoře lze navíc editovat zvolené přiřazení pravidel.

Vehicle Page

The screenshot shows the Vehicle Page for vehicle 1BX6379. It includes sections for 'Parking Session of 1BX6379' and 'Applied Parking Rules'. The parking session table has columns: Start, End, Total, Vehicle, and a highlighted row for 3/4/2020, 5:25:04 PM - 5:40:00 PM, 100, 1BX6379. The applied rules table has columns: A#, Name, Type, Permit, Fee, T unit, Free units, and Rounding, with one rule for 100/h TimedFee.

Parking Session Picker:

Date	Description
03/23/2020	0, 3/18/2020, 5:11:04 PM – still active 11111
03/18/2020	0, 3/8/2020, 4:31:17 PM – still active 4S502
03/8/2020	0, 3/8/2020, 4:18:57 PM – still active 4S50233
03/15/2020	0, 3/15/2020, 11:44:04 PM – 11:44:34 PM 1AN9714

Vehicle Picker:

Vehicle	Sessions
1BX6379	sessions
DDD1234	0 sessions
CCC1234	0 sessions
BKW7165	0 sessions
BBB1234	0 sessions
AF6533	2 sessions
AAA1234	0 sessions

Obrázek 3.7: Stránka se záznamy parkování. V levém horním kvadrantu se nachází záznam parkování. Při najetí myši na čas vjezdu/výjezdu se zobrazí výřez rozpoznané SPZ. Navíc lze zkontořovat použitá pravidla. Zároveň lze kliknout na SPZ a zobrazit si ostatní záznamy onoho vozidla na stejně obrazovce. Vpravo lze vidět vybírátka záznamů a vozidel. V levém dolním kvadrantu je pak detail jednoho vozidla.

The screenshot shows a detailed view of a parking session for vehicle 1AN9714. It includes sections for 'Parking Session' and 'Applied Parking Rules'. The parking session table has columns: Start, End, Total, Vehicle, and a highlighted row for 2/29/2020, 5:00:23 PM - 6:58:14 PM, 200, 1AN9714. The applied rules table has columns: A#, Name, Type, Permit, Fee, T unit, Free units, and Rounding, with one rule for 100/h TimedFee.

Obrázek 3.8: Záznam parkování s výřezem naskenované SPZ.

3.3 Komunikace

3.3.1 Endpointy backendu

Backend má pouze několik endpointů. Pro komunikaci s webovým rozhraním přes GraphQL. Mobilní zařízení GraphQL nevyužívají, protože GraphQL není k nahrávání snímků vhodné a míchání více způsobů komunikace pro velice malou mobilní aplikaci by vývoj spíše ztížilo. Máme tedy další endpointy:

- POST */graphql* je určen k primární komunikaci s webovým rozhraním včetně autentifikace.
- GET */devices/qr/:id* slouží k získání QR kódu ve formě PNG obrázku z

webového rozhraní pro autentifikaci zařízení. Přijímá *id* zařízení jako parametr v URL.

- POST */devices/activate* slouží k autentifikaci zařízení, které právě naskeñovalo QR kód s aktivačním heslem, které je posláno v těle žádosti.
- PUT */devices/config* slouží zařízení k získání své konfigurace a případné změně konfigurace ze strany zařízení.
- POST */devices/capture* přijímá obrazová data od zařízení.
- GET */captureImage/:id* slouží k získání výřezu naskenované SPZ z webového rozhraní. Přijímá *id* obrázku jako parametr v URL.

3.3.2 GraphQL

Protože GraphQL je primárním způsobem komunikace mezi frontendem a backendem, stojí za to si vysvětlit základní stavební bloky GraphQL – resolversy, které tvoří orientovaný graf, ale pro zjednodušení lze o této struktuře přemýšlet jako o stromu. Jaké resolversy máme, jaké přijímají typy, jaké vracejí typy definujeme v SDL (abbrv. Schema-Definition-Language). V Typescriptu (v tomto projektu) se pak resolversy implementují. Graf resoverů a tedy i GraphQL schema v SDL je v příloze A. Na obrázku 3.9 lze vidět příklad definice v SDL.

```
1  type UserSearchResult {  
2      data: [User!]  
3      page: PositiveInt!  
4      limit: PositiveInt!  
5  }  
6  
7  input UserSearchInput {  
8      name: String!  
9      # Default is 50  
10     limit: PositiveInt  
11     # Default is 1  
12     page: PositiveInt  
13 }  
14  
15 extend type Query {  
16     # If not logged id, an error is returned  
17     currentUser: User!  
18     userSearch(search: UserSearchInput!): UserSearchResult!  
19     userSearchByEmail(search: UserByEmailSearchInput!):  
20         UserSearchResult!  
21 }
```

Obrázek 3.9: Příklad definice resoverů v SDL. Klíčové slovo `extend` umožňuje definovat části typů v několika souborech, což umožňuje flexibilnější dělení projektu. Kromě obvyklých typů, které dostává klient, máme i `input` typy, ve kterých se definují povolené vstupní hodnoty.

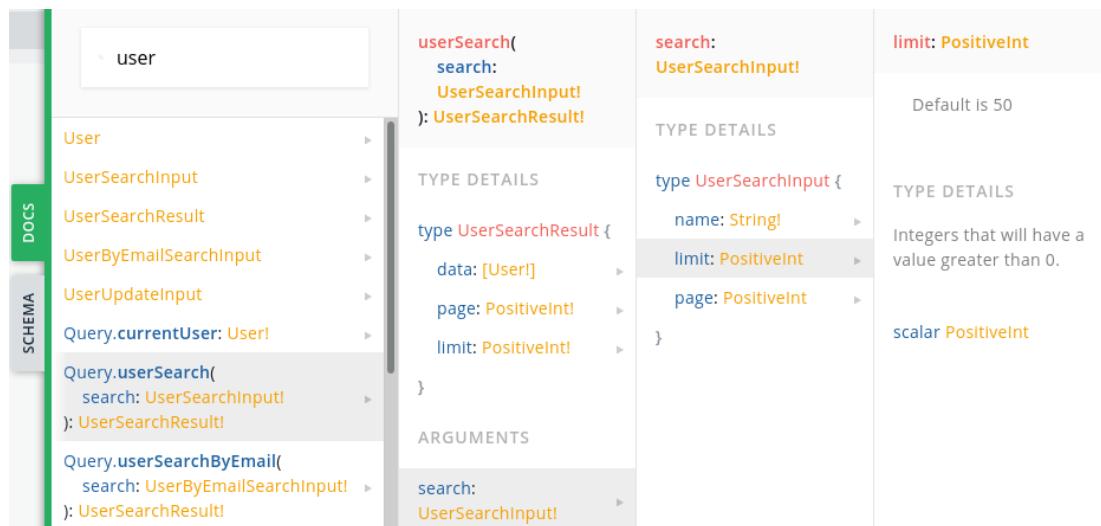
Vzpomeňme si na příklad z předchozí kapitoly na obrázku 2.2. Zde vstupním vrcholem grafu je resolver `userSearch`, jenž přijímá argumenty potřebné pro jeho běh, ty jsou v tomto případě povinné (ale nemusí). Kořenový resolver je funkce

vraťející nějaký typ, v tomto případě typ *UserSearchResult*, který udává nalezené uživatele (*UserSearchResult.data*) a stránkování (v příkladu vynecháno). Máme dva druhy typů: skaláry (“listy” grafu) a objekty (“nelisty” grafu). 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.

Pro každý kořenový resolver se musí napsat funkce, která budě přímo vrátí požadovaný objekt s atributy, a nebo k typu, který nás kořenový resolver vratí, napíšeme další resolvers, které vrací příslušné 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. [5]

Zároveň lze provádět virtualizaci atributů – přidávat atributy, které v databázi nemáme, ale pro webové rozhraní se hodí. Hypotetickým příkladem budiž celková zaplacená částka zákazníkem, která se v případě potřeby vypočítá nebo aktuální agregované statistiky.

SDL zároveň umožňuje komentáře, které lze pak zobrazit v různých nástrojích pro práci s GraphQL – dokumentace je vestavěná a téměř zadarmo. V tomto projektu byl použit nástroj GraphQLPlayground. Snímek dokumentace lze vidět na obrázku 3.10.



Obrázek 3.10: Screenshot dokumentace z nástroje GraphQLPlayground. CSS bylo upraveno, aby byl obrázek čitelnější.

3.3.3 Top-level GraphQL typy

Již zmíněný graf resolverů má vstupní vrcholy – *Query*, *Mutation* a popřípadě *Subscription* (v tomto projektu nepoužito). *Query* slouží k obalení operací čtení, *Mutation* slouží k obalení operací úpravy/mutace dat a *Subscription* slouží ke komunikaci přes websockety. [5] [14]

Tedy tvrzení z předchozí podpodkapitoly, že resolver *userSearch* je vstupem do grafu byla lež. Přesněji k resolveru *userSearch* se dostaneme přes top-level typ *Query*.

Graf resolverů a tedy i GraphQL schema v SDL (abbrv. Schema-Definition-Language) je v příloze A.

3.3.4 Obecné resolvency

Jelikož se většina operací nad modely opakuje, byly napsány obecné resolvency jako HOF (abbrv. Higher-Order-Function) pro vytváření, úpravu, mazání, vyhledávání a získávání modelů v relaci. Měnící se část je pak pouze definice databázového modelu. Díky GraphQL se neumí ani ověřovat datové typy – GraphQL toto udělá podle schema, které tvoří programátor v SDL (abbrv. Schema-Definition-Language). Na obrázku 3.11 lze vidět funkci vracející resolver, který vrátí objekt v relaci (provede JOIN v SQL terminologii), pokud ho vlastní objekt ještě nemá.

```

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         // Get the model
8         const model = modelGetter( ctx );
9         const keyStr = key.toString();
10        if (obj.populated(keyStr)) {
11            // Already populated
12            return obj[ key ];
13        } else {
14            // Fetch the field K on D only
15            const populated: D = await model.populate( obj, {
16                path: keyStr
17            });
18            // Resolver returns just the field K
19            return populated[ key ];
20        }
21    };
22 }

```

Obrázek 3.11: Ukázka obecného resolveru v Typescriptu. Na osmém řádku se získá model z funkce v *lexical-scope* funkce vracející resolver. Proměnná *ctx* definuje kontext resolveru a obsahuje modely a klienta, aby bylo snažší resolvency 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.

3.3.5 Tvorba Apollo klienta

Webové rozhraní komunikující s backendem využívá již zmíněnou klientskou verzi knihovny Apollo, aby si vytvořila klienta, který má mnoho parametrů.

Nejdůležitější z nich jsou funkce, které řeší zpracování žádostí i odpovědi, jimž se říká *links*. [15] Ty řeší zpracování chyb a poslání chyb do Reduxu, pře-

čtení URL backendu z nastavení, také vložení tokenu do hlavičky žádosti (viz podkapitola 3.2). Jejich způsob použití lze přirovnat k návrhovému vzoru *chain-of-responsibility*.

Další parametr je určen ke cachování a vložení GraphQL schema získaného z backendu pomocí endpointu */graphql* (viz podpodkapitola 3.3.1), aby bylo možné používat v dotazech fragmenty.

3.3.6 Získávání obrázků

Jelikož se k autorizaci používá header *Authorization* místo souborů cookies, způsob získávání obrázků z endpointů zmíněných v podpodkapitole 3.3.1 není obvyklý. Namísto nastavení atributu *src* HTML tagu *img* na normální URL obrázku, použijeme Blob URL. Obrázek stáhneme v Javascriptu a vytvoříme Blob URL pomocí API prohlížeče. Přibližná implementace je ukázána na obrázku 3.12.

```

1 function imageGetter(url: string): Promise<Response> {
2   const url = `${config.backendApi.root}/${url}`;
3   return fetch(url, {
4     method: "GET",
5     headers: {
6       Authorization: `Bearer ${localStorage.getItem("token")}`
7     }
8   });
9 }
10
11 // In a component, setBlobUrl and setError are React hooks
12 function blobImage(url: string, setBlobUrl: string => void,
13                     setError: any => void) {
14   imageGetter(url)
15     .then(response => response.blob())
16     .then(blob => {
17       setBlobUrl(URL.createObjectURL(blob));
18     })
19     .catch(setError);
20 }
```

Obrázek 3.12: Získání obrázků za použití hlavičky *Authorization* a jejich zpracování.

3.4 Ukládání a mazání výřezů SPZ

Pokud je rozpoznána SPZ, nalezená SPZ se z obrázku vyřízne, a pak je dostupná k zobrazení ve webovém rozhraní. Výřezy jsou získány pomocí knihovny sharp a souřadnic SPZ od OpenALPR. Zvolený způsob uložení je standardní dokument v MongoDB. Data jsou zakódována do base64, a handler endpointu */capture-Image/:id* (viz podpodkapitola 3.3.1) je dekóduje zpět.

První alternativou bylo využít GridFS MongoDB, ten je určen pro soubory nad 16MB (limit formátu BSON) nebo pokud je potřeba číst jen části souborů. [16] Výřezy přibližně 500x100 pixelů jsou však mnohonásobně menší a vždy jsou načítány celé. Druhou alternativou by byl souborový systém, ale mít všechna data v databázi je praktičtější.

3.5 Parkovací pravidla

Nechť umí parkovací pravidla následující:

- 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 jednoduchost je zapotřebí oddělit samotná pravidla od jejich priority, časového intervalu platnosti i filtrů, k čemuž bude sloužit objekt typu ParkingRuleAssignment.

Filtrování bude mít dva módy: začne se se všemi vozidly (ALL) a začne se bez vozidel (NONE). Následné filtry mohou být přidávat, nebo odstraňovat jednotlivá vozidla. Hodí se mít filtry uložené separátně, aby mohly být využity několikrát. Datový návrh lze vidět na obrázku 3.13.

```
1 type ParkingRuleAssignment {
2   id: ID!
3   rules: [ParkingRule]!
4   start: DateTime!
5   end: DateTime!
6   vehicleFilterMode: VehicleFilterMode!
7   vehicleFilters: [VehicleFilter!]
8   priority: NonNegativeInt!
9 }
10
11 type VehicleFilter {
12   id: ID!
13   name: String!
14   action: VehicleFilterAction!
15   vehicles: [Vehicle!]
16 }
```

Obrázek 3.13: Definice typů ParkingRuleAssignment a VehicleFilter v GraphQL SDL. Tučné jsou označeny rezervovaná slova a skáláry již obsažené v základním GraphQL. Kurzívou jsou označeny skaláry z knihovny graphql-scalars. Vykríčník značí, že atribut je povinný (nesmí být null v odpovědi). Typ VehicleFilterMode je enum, jehož možné hodnoty jsou ALL, NONE. Typ VehicleFilterAction je taktéž enum, jehož možné hodnoty jsou INCLUDE, EXCLUDE.

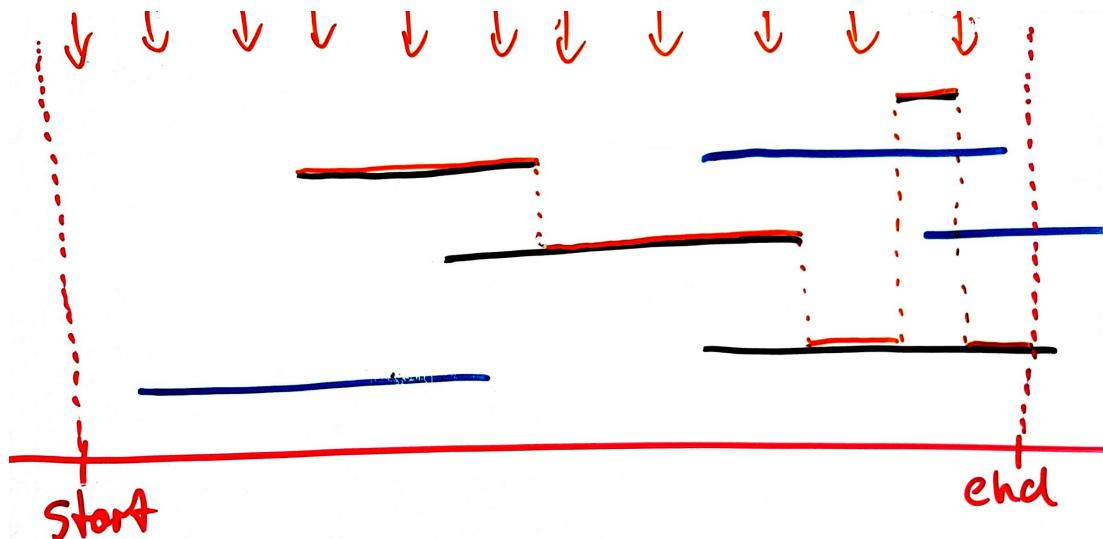
3.5.1 Algoritmus filtru vozidel

Vstup: objekt *ParkingRuleAssignment* s filtry, vozidlo
Výstup: boolean určující zdali vozidlo prošlo filtrem

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

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

3.5.2 Algoritmus pro aplikaci pravidel



Obrázek 3.14: Ilustrace problému úseček. Modré úsečky neplatí kvůli filtrům nebo protože jsou deaktivované. Oranžová čára značí výstup požadovaného algoritmu.

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

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. Na obrázku 3.14 je ilustrace tohoto geometrického problému.

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

Vstup: seznam $\text{ParkingRuleAssignment}$ platných v intervalu pobytu vozidla na parkovišti, vozidlo

Výstup: seznam $\text{ParkingRuleAssignment}$ s časy platnosti

1. Seřadíme si začátky a konce úseček podle jejich času. Začátek úsečky se uchovává pointer na svůj objekt typu $\text{ParkingRuleAssignment}$.
2. Vytvoříme si haldu pro odkládání úseček, která řadí podle $\text{ParkingRuleAssignment.priority}$ – 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 lepsí 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 $\mathcal{O}(N \cdot M)$.

3.5.3 Algoritmus pro mazání přiřazení pravidel

Pro mazání je praktičtější, když se nemažou všechny objekty typu `ParkingRuleAssignment`, které jsou některou svou částí v časovém úseku, který chceme vyprázdnit. Pak by se mohlo stát, že se smažou i jiné, jejichž začátek a/nebo konec je mimo tento úsek. Tedy místo toho, abychom mazali podle podmínky,

$$start \leq deleteEnd \wedge end \geq deleteStart$$

budeme mazat pouze ty, které jsou celé v mazaném úseku (*a*). Ty, které mají některou svoji část i mimo mazaný úsek ($b_1 \vee b_2$), zkrátíme. Ty, co mají začátek i konec mimo mazaný úsek (*c*), rozdělíme na dva – jeden vytvoříme a druhý dostaneme zkrácením původního.

$$deleteStart \leq start \wedge end \leq deleteEnd \text{ (a)}$$

$$start < deleteStart \wedge deleteStart < end \leq deleteEnd \text{ (} b_1 \text{)}$$

$$deleteStart \leq start < deleteEnd \wedge deleteEnd < end \text{ (} b_2 \text{)}$$

$$start < deleteStart \wedge deleteEnd < end \text{ (c)}$$

Případy *a* smažeme. U případů b_1 nastavíme $end = deleteStart$ a u případů b_2 nastavíme $start = deleteEnd$. U případů *c* původnímu objektu typu `ParkingRuleAssignment` nastavíme $end = deleteStart$ a kopii, která má původní hodnotu end , nastavíme $start = deleteEnd$.

Při implementaci využijeme věstavěnou schopnost Javascriptu provádět asynchronně na pozadí několik akcí, což můžeme, protože případy jsou disjunktní. Pokračujeme v programu, jakmile jsou všechny dokončené, aniž bychom blokovali vlákno. Akcí je nyní myšlena úprava dokumentů v databázi.

3.6 Uživatelské rozhraní

Jelikož autor neměl zkušenosti s vývojem webového uživatelského rozhraní, rozhol si využít starter projekt společnosti Crazy Factory GmbH, který pojí vybrané technologie dohromady a mandatuje projektu základní strukturu. Navíc podporuje překlady, reportování chyb apod., což je obtížné vytvářet z ničeho. [17]

Na obrázku 3.16 lze vidět `favicon.ico` webového rozhraní.

3.6.1 Redux a persistence stavu

Jak již bylo zmíněno v podkapitole 2.2.3, 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. [9] Prostost reducerů zajišťuje testovatelnost.

Jako příklad konkrétního využití budí sdílení některých údajů mezi obrazovkami. Například sdílení vozidla mezi stránkou s vozidly a stránkou s pravidly, schopnost seznamu se záznamy o parkování zvolit vozidlo, ukládní části stavu obrazovky mezi přechody mezi ostatními obrazovkami a mnoho dalších.

```

1 const promises: Promise<IParkingRuleAssignment[]> = Promise.all([
2   // (a) Delete those that are wholly between deleteStart and
3   //      deleteEnd
4   ParkingRuleAssignment.deleteMany({
5     start: { $gte: deleteStart },
6     end: { $lte: deleteEnd }
7   }),
8   // (b1) Start is outside, end is inside
9   ParkingRuleAssignment.updateMany(
10    { start: { $lt: deleteStart }, end: { $gt: deleteStart, $lte:
11      deleteEnd } },
12    { $set: { end: deleteStart } }
13  ),
14  // (b2) End is outside, start is inside
15  ParkingRuleAssignment.updateMany(
16    { start: { $gte: deleteStart, $lt: deleteEnd }, end: { $gt:
17      deleteEnd } },
18    { $set: { start: deleteEnd } }
19  ),
20  // (c) Both start and end are outside -> divide into two
21  //      assignments
22  ParkingRuleAssignment.find({
23    end: { $gt: deleteEnd },
24    start: { $lt: deleteStart }
25  }).then(results => {
26    const copies = results.map(original => {
27      const copy = duplicateWithoutId(original);
28      copy.start = deleteEnd;
29      return copy;
30    });
31    results.forEach(a => (a.end = deleteStart));
32    return Promise.all([
33      ParkingRuleAssignment.create(copies),
34      Promise.all(results.map(r => r.save()))
35    ]);
36  });
37]);
38 // ... continue once everything is finished

```

Obrázek 3.15: Mazání časového úseku objektů typu ParkingRuleAssignment. V kódu je IParkingRuleAssignment interface a ParkingRuleAssignment je mongoose model, pomocí kterého se přistupuje ke kolekci dokumentů.



Obrázek 3.16: Ikona

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 (abbrv. Higher-Order-Function), které dáme reducer, a ona změny, které způsobí, někam uloží. [18] V našem případě se jedná o *localStorage*.

3.6.2 Typy komponent

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

- **Bez vnitřního stavu** – může se jednat o tlačítka, vstupy textu apod. Tyto komponenty jsou zcela řízeny nadřazenou komponentou. S nadřazenou komponentou komunikují pomocí callback funkcí, které komponenta přijímá jako parametry.
 - **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 údávají 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.

3.6.3 Základní prvky uživatelského rozhraní

Každá složitější webová aplikace používá různé druhy tlačítek a vstupů. Bohužel v každém prohlížeči vypadají základní prvky jinak. Na obrázku 3.17 jsou screenshoty z webové stránky <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/checkbox> prohlížečích Mozilla Firefox a Google Chrome.

Pokud se chce navíc sjednotit tyto prvky se zbytkem uživatelského rozhraní, je vyvinutí vlastních elementárních prvků nevyhnutelné. Vývoj se řídil principem minima informací – omezí se v rámci jednoho prvku počet nápadných částí. V praxi to například znamená žádné nebo méně výrazné orámování (CSS vlastnost *border*) a zahlé rohy (CSS vlastnost *border-radius*). Na obrázku 3.18 lze vidět prvky využité v projektu.



Obrázek 3.17: Checkboxy v Mozilla Firefox a Google Chrome.



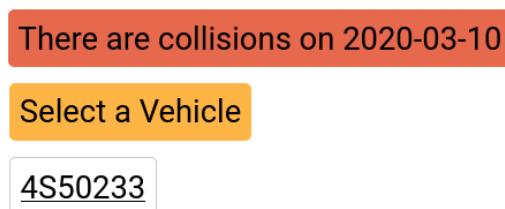
(1) Tlačítka – Delete je vypnuto/disabled.



(2) Přepínač – zapnut a vypnuto.



(3) Checkbox ve stádiích vypnuto, přejíždění myší, zapnuto a opět přejíždění myší.

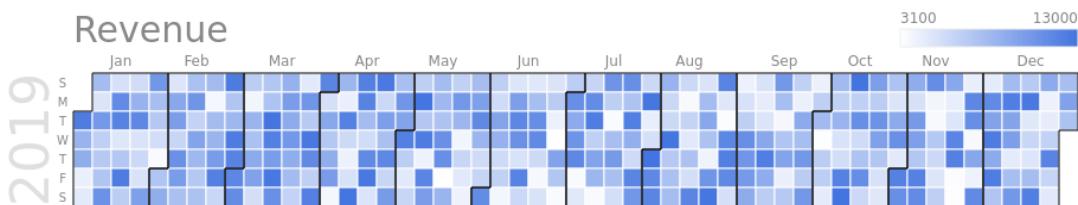


(4) Zvýrazňovač – chyba, upozornění a SPZ (podtržení značí interaktivnost).

Obrázek 3.18: Základní prvky uživatelského rozhraní.

3.6.4 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 3.19.



Obrázek 3.19: Kalendářový graf z knihovny react-google-charts.

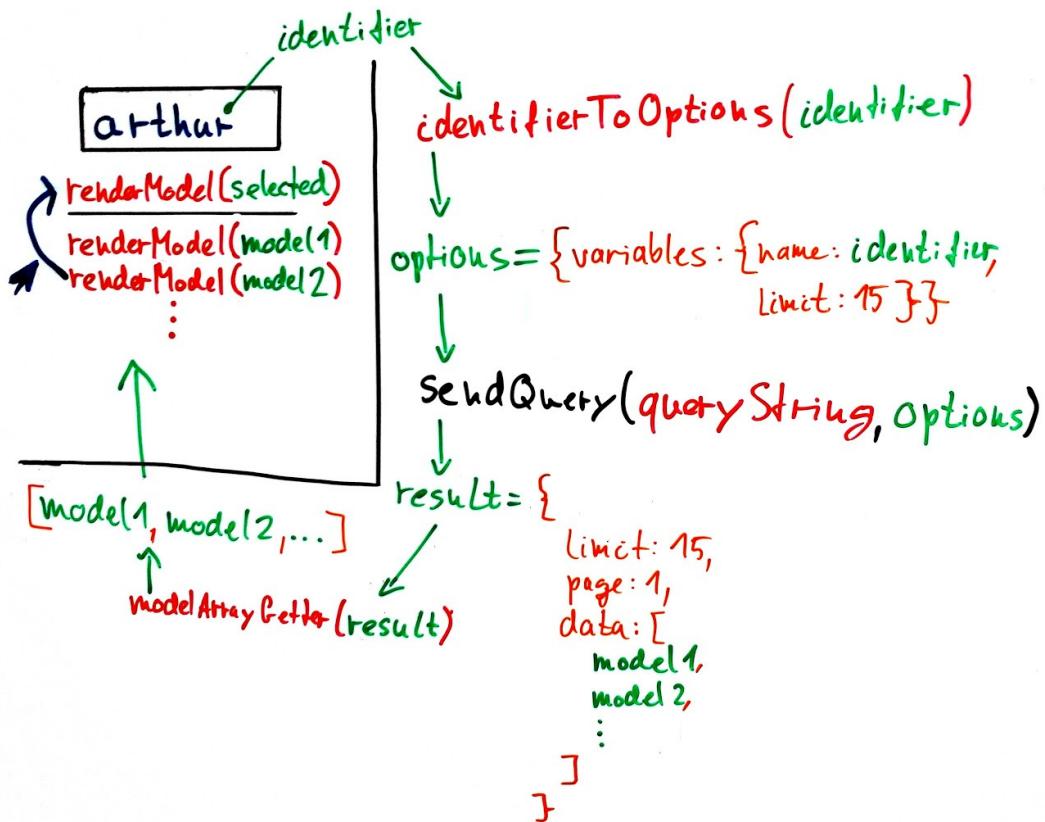
3.6.5 Obecná vybírátka modelů

Dle principu neopakování se (DRY – Don-Repeat-Yourself) 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 (aabrv. Higher-Order-Function), což je funkce vracející další funkce – konkrétní komponenty. Měnící se části, které se do těchto obecných komponent

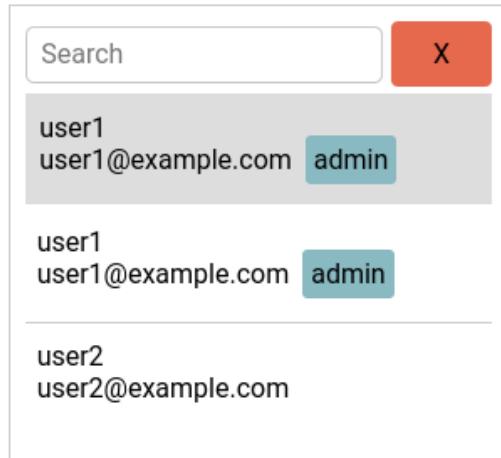
budou vkládat, je komponenta renderující jediný model (*renderModel*), GraphQL dotaz pro získávání modelů (*queryString*), funkce, jejímž vstupem je odpověď na GraphQL dotaz a výstupem je samotné pole modelů (*modelArrayGetter*), a funkce, jejímž vstupem je dotazovací řetězec a výstupem jsou argumenty pro GraphQL dotaz (*identifierToOptions*).

Obrázek 3.20 ukazuje tok dat v jednom z obecných vybírátek. Obrázek 3.21 ukazuje vyrenderovanou komponentu vybírátko.

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



Obrázek 3.20: Tok dat v jednom z obecných vybírátek.



Obrázek 3.21: Vyrezenovaná komponenta jednoho vybírátka. Šedivě podbarvený uživatel je zvolený. Při psaní do textového se ihned vyhledává hledaný objekt – v tomto případě uživatel.

4. Rozpoznávání SPZ a mobilní aplikace

Jak již bylo řečeno v kapitole 2, mobilní aplikace pořídí snímek, pošle ho backendu, 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

Přesnost je hodně ovlivněna osvětlením, proto při nasazení v prostředí s neideálním osvětlením je dobré použít dodatečné světlo osvětlující SPZ.

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 Komunikace zařízení a backendu

4.2.1 Autentizace

Zařízení se autentifikuje naskenováním QR kódu, jenž lze vytvořit 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 způsob komunikace s touto externí aplikací byl převzán z [19].

4.2.2 Zabezpečení komunikace s mobilní aplikací

Jelikož mobilní aplikace naivně komunikuje s nastaveným počítačem (nastavení viz obrázek 4.2), tak je potřeba zajistit zabezpečení této komunikace. Autor vidí tři varianty:

- Použít HTTPS.
 - S certifikátem podepsaným certifikační autoritou.
 - S certifikátem podepsáným sám sebou (angl. *self-signed*). U této subvarianty je potřeba do Android zařízení importovat daný certifikát v nastavení.
- Použít VPN.
- Použít uzavřenou a důvěryhodnou síť.

Tyto varianty lze samozřejmě kombinovat.

4.2.3 Optimalizace velikosti přenesených dat

Aby se ušetřilo na přenesených datech a aby rozpoznání proběhlo rychleji, mobilní aplikace zmenší snímek na nastavitelnou hodnotu. Tuto funkci poskytuje metoda *createScaledBitmap* v třídě *android.graphics.Bitmap*.

V závislosti na poměru stran snímků největšího rozlišení fotoaparátu, může být potřeba hodnotu zmenšení změnit. Základní hodnota je 1000x1000 pixelů, ale podle poměru stran fotoaparátu se může vyplatit jiná hodnota.

Ušetření je opravdu veliké. Při testu s fotoaparátem o rozlišení 4356x3492 pixelů byla průměrná velikost nezmenšených snímků 1,9MB (JPG, $N = 50$, $\sigma = 0,325$). Průměrná velikost snímků zmenšených na 1300x1000 pixelů byla 0,15MB (JPG, $N = 50$, $\sigma = 0,024$). Kdyby byla frekvence snímání jeden snímek za sekundu, tak by bez optimalizací byl objem přenesených dat za jeden den 164,16GB ($3600 \cdot 24 \cdot 1,9\text{MB}$). Při stejné frekvenci s optimalizacemi by byl objem přenesených dat za jeden den 12,96GB ($3600 \cdot 24 \cdot 0,15\text{MB}$), což je 7,9% objemu bez optimalizací.

4.3 Mobilní aplikace

4.3.1 Volba zařízení pro mobilní aplikaci

Co se týče hardwarového vybavení snímacího zařízení, tak je vyžadována přední kamera s rozlišením alespoň 1000 na 1000 pixelů. Bližší informace ohledně a zdůvodnění zmenšení jsou v podkapitole 4.2.3. 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).

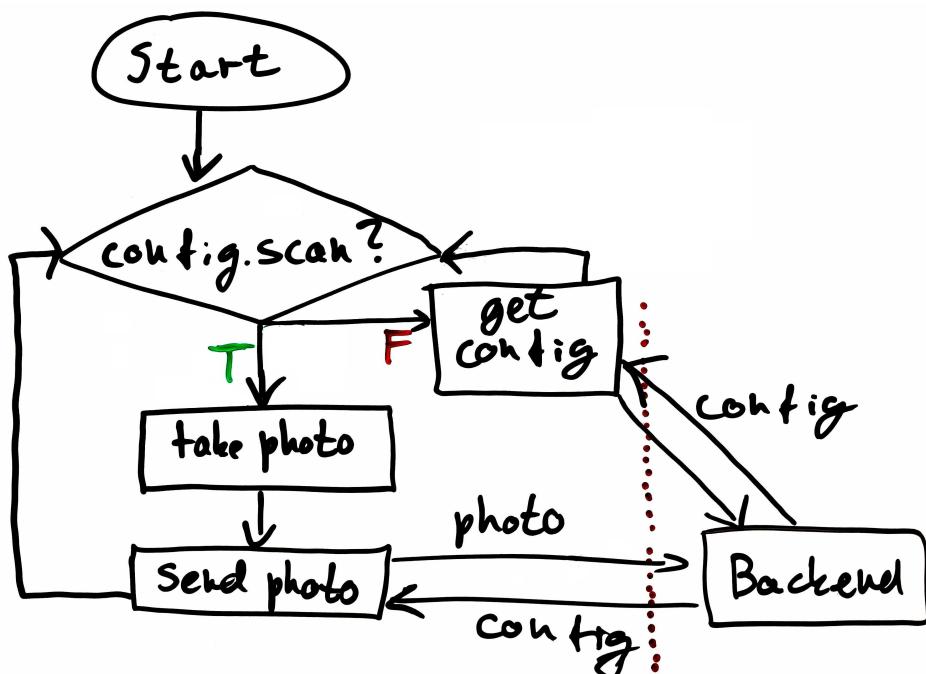
Autorovi se nepodařilo najít způsob, jak zároveň pořizovat v pravidelném intervalu snímky a mít zařízení uzamknuté proti přístupu. K zajištění pořizování snímků si hlavní obrazovka aplikace řekně systému Android o zabránění uzamknutí.

To má dva následky. První je, že zařízení by nemělo mít OLED displej, aby nedošlo k takzvanému *burn-in*. [20] Druhý je, že zařízení by mělo být v produkčním provozu bezpečně uzavřeno v krabičce, nebo by se mělo nacházet na bezpečném místě, aby se předešlo nepovolené manipulaci.

Na místě instalace je zároveň nutné napájení, nepředpokládá se, že by zařízení mohlo vydržet delší provozní dobu.

4.3.2 Životní cyklus mobilní aplikace

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



Obrázek 4.1: Životní cyklus mobilní aplikace.

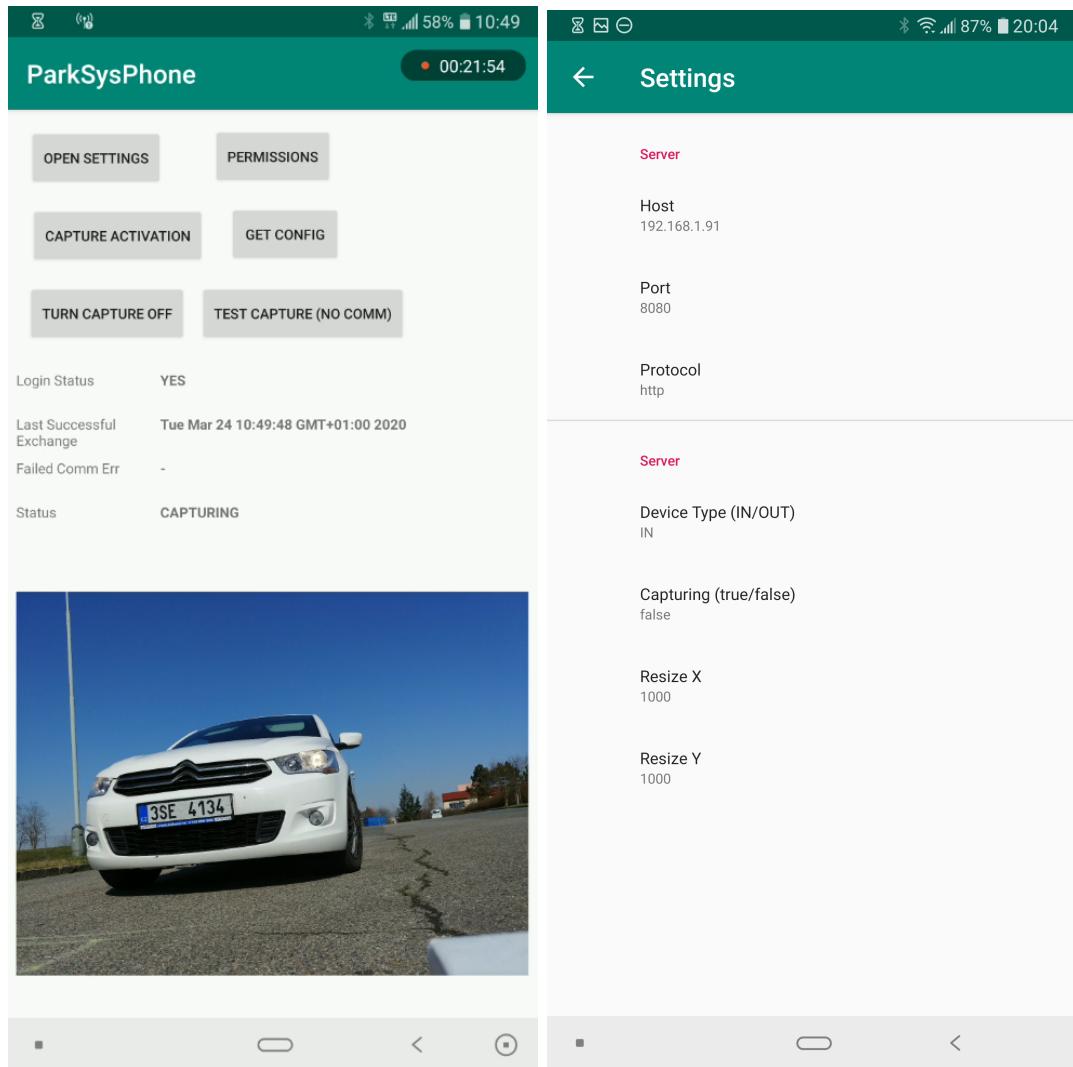
4.3.3 Uživatelské rozhraní mobilní aplikace

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

4.3.4 Implementační detaily mobilní aplikace

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. [21]



Obrázek 4.2: Rozhraní mobilní aplikace. Hlavní obrazovka zobrazuje poslední snímek a status komunikace s backendem. V nastavení lze nastavit adresu backendu a vybrat si mezi HTTP a HTTPS a zobrazit nastavení typu zařízení a úpravy snímku.

Ukládání snímků

Snímky se ukládají do paměti určené zařízením pro aplikaci bez potřeby permisí navíc. Kdyby se použily dočasné soubory, mohlo by se stát, že je systém před posláním nemilosrdně smaže. [22] Aplikace tedy musí obstarávat i mazání souborů, což se provádí ihned po odeslání snímku na backend.

5. Instalace

Defaultně potřebuje aplikace porty 4500 (server rozpoznávající SPZ), 8080 (backend) a 8889 (frontend). Pokud je MongoDB instalována systémově, běží na portu 27017, a pokud v Dockeru, tak běží na portu 5432. Porty lze změnit v konfiguračních souborech jednotlivých komponent. Při změně portu komponenty A, na které jsou jiné komponenty B závislé, je potřeba změnit port závislých komponent B pro danou komponentu A. Mobilní aplikace se kompiluje vývojovým prostředím Android Studio.

5.1 Soubory nastavení

Soubory pro nastavení jsou:

- Backend
 - /backend/Dockerfile
 - /backend/config/production.json
 - /backend/config/development.json
 - /backend/src/config.ts
- Frontend
 - /frontend/Dockerfile
 - /frontend/config/main.js
 - /frontend/config/main.local.js
- Server rozpoznávající SPZ
 - /express-openalpr-server/Dockerfile
 - /express-openalpr-server/processes.json

5.2 Instalace Dockerem

Instalace celého systému Dockerem je velice jednoduchá, ten zajistí komplaci, spuštění, propojení všech částí a odhalení potřebných služeb ven na internet. [23]

Postup instalace

1. Po stažení git repozitáře se zdrojovým kódem nastavíme submoduly (lze vynechat, pokud máte kód jako zip archiv):

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

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

```
$ docker-compose up
```

4. Zjistíme počáteční přihlašovací údaje z výstupu předchozího příkazu. Pokud byla proměnná prostředí **NODE_ENV=development**, bude počáteční heslo 1234.

```
mongo_1      | 2020-03-23T20:21:44.660+0000 I NETWORK  [listener]  Connected to: mongodb://127.0.0.1:27017 (conn1) for admin  
mongo_1      | 2020-03-23T20:21:44.661+0000 I NETWORK  [conn2]  received query: { find: "admin.system.users", filter: { user: "admin", password: "1234", db: "admin" }, projection: {}, ns: "admin.$cmd" } on connection: 0x55c110bd3  
ejs|Mongoose", version: "3.3.5|5.8.0" }, os: { type: "Linux", name: "Ubuntu", version: "18.04 LTS", architecture: "x64", release: "18.04.19-Ubuntu", distro: "Ubuntu", major: 18, minor: 0, patch: 49, env: "NODE_ENV=development" }  
mongo_1      | 2020-03-23T20:21:44.661+0000 I SHARDING [conn2]  Sharding command: { find: "admin.system.users", filter: { user: "admin", password: "1234", db: "admin" }, projection: {}, ns: "admin.$cmd" }  
backend_1    | =====  
backend_1    | > ADMIN ACCOUNT: admin:5ddc110bd3  
backend_1    | =====  
mongo_1      | 2020-03-23T20:21:44.663+0000 I NETWORK  [listener]  
mongo_1      | 2020-03-23T20:21:44.664+0000 I NETWORK  [conn3]  received query: { find: "admin.system.users", filter: { user: "admin", password: "1234", db: "admin" }, projection: {}, ns: "admin.$cmd" } on connection: 0x55c110bd3  
ejs|Mongoose", version: "3.3.5|5.8.0" }, os: { type: "Linux", name: "Ubuntu", version: "18.04 LTS", architecture: "x64", release: "18.04.19-Ubuntu", distro: "Ubuntu", major: 18, minor: 0, patch: 49, env: "NODE_ENV=development" }  
mongo_1      | 2020-03-23T20:21:44.664+0000 I SHARDING [conn3]  Sharding command: { find: "admin.system.users", filter: { user: "admin", password: "1234", db: "admin" }, projection: {}, ns: "admin.$cmd" }  
mongo_1      | 2020-03-23T20:21:44.664+0000 I SHARDING [conn1]  Sharding command: { find: "admin.system.users", filter: { user: "admin", password: "1234", db: "admin" }, projection: {}, ns: "admin.$cmd" }
```

5. Otevřeme prohlížeč na adresu <http://localhost:8889>.

Krok číslo 3 může trvat i několik minut v závislosti na rychlosti interneto-vého připojení. Protože Docker zabalí vše včetně systémových závislostí, velikost výsledných imagů je kolem 1,4GB.

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

5.3 Instalace bez Dockeru

Pro spuštění bez Dockeru je potřeba nainstalovat několik programů. Jmenovitě se jedná o Typescript pro komplaci, Node.js a NPM pro knihovny a spuštění backendu, frontendu a serveru rozpoznávající SPZ, pm2 pro load-balancing serveru rozpoznávající SPZ, MongoDB, Android Studio pro kompliaci mobilní aplikace a pro rozpoznávání SPZ je potřeba OpenALPR .

Instalace knihovny OpenALPR je nejproblematičejší, protože probíhá kompliací velkého množství C++. Instrukce ke kompliaci jsou dostupné z <https://github.com/openalpr/openalpr#compiling>. Další alternativou je spouštět server rozpoznávající SPZ Dockerem a zbytek bez Dockeru – stačí ve složce */express-openalpr-server* spustit příkazy: **asddddddddd**
ddddd

```
$ npm install  
$ docker build -t express-openalpr:latest ".."  
$ docker run express-openalpr:latest
```

Postup spuštění bez Dockeru

1. Nainstalujeme závislosti.
2. Po stažení git repozitáře se zdrojovým kódem nastavíme submoduly (lze vynechat, pokud máte kód jako zip archiv):

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

3. Spustíme MongoDB.
4. Spustíme server rozpoznávající SPZ.

```
$ cd express-openalpr-server  
$ npm install  
$ pm2 start processes.json
```

5. Spustíme backend buď následujícími příkazy, nebo bash skriptem */backend/sdev.sh -c*.

```
$ cd backend  
$ npm install  
$ npm run compile  
$ npm start
```

6. Spustíme frontend (poslední příkaz může chvíli trvat).

```
$ cd frontend  
$ npm install  
$ npm start
```

7. Přihlašovací údaje jsou **admin:1234**, protože nespouštíme s proměnnou prostředí **NODE_ENV=production** jako při spuštění Dockerem.
8. Otevřeme prohlížeč na adresu <http://localhost:8889>.

Závěr

Výsledný produkt se vyvinul nad očekávání autora. Rozpoznávání SPZ pomocí běžného telefonu funguje skvěle a občas i za poměrně nehostinných podmínek (viz obrázek 5.1).



Obrázek 5.1: Rozpoznaná SPZ v nehostinných podmínkách.

Co se týče uživatelského rozhraní, tak to vypadá jednotně a přehledně. Je nabité funkcemi a dohromady s backendem umožňuje ovládat parkovací systém. Zejména monitorovat zařízení a měnit jejich nastavení na dálku, simulovat vytvořená parkovací pravidla a filtry, číst statistiky a záznamy parkování.

Vývoj byl díky vhodně zvoleným technologiím poměrně rychlý a bezbolestný. V jeho průběhu nedošlo k žádnému backtrackování kvůli předchozím rozhodnutím. I automatické testování se vyplatilo. Projekt je bez velkých obtíží rozšířitelný a pozměnitelný. Projekt by šlo rozšířit o komunikaci se závorou a platebním terminálem, což by z projektu udělalo plnohodnotný parkovací systém.

Bibliografie

- [1] MongoDB Inc., 2020. <https://www.mongodb.com> [cit. 2020-02-16].
- [2] Microsoft, 2019. <http://www.typescriptlang.org/> [cit. 2020-03-01].
- [3] OpenJS Foundation, 2019. <https://nodejs.org/en/> [cit. 2020-03-01].
- [4] LearnBoost, 2019. <https://mongoosejs.com/> [cit. 2020-02-29].
- [5] The GraphQL Foundation. Queries and mutations, 2020. <https://graphql.org/learn/queries/#fields> [cit. 2020-02-24].
- [6] Meteor Development Group Inc., 2020. <https://www.apollographql.com> [cit. 2020-02-16].
- [7] Facebook, 2019. <https://reactjs.org/> [cit. 2020-02-29].
- [8] typestyle, 2020. <https://github.com/typestyle/typestyle> [cit. 2020-02-29].
- [9] Dan Abramov et al. Core concepts, 2020. <https://redux.js.org/introduction/core-concepts> [cit. 2020-02-24].
- [10] Er Ajay Pratap. React redux tutorials: React redux data flow and redux lifecycle methods with examples, 2018. <https://www.reactreduxtutorials.com/2018/02/redux-tutorial-for-beginners-redux-data-flow-redux-lifecycle.html> [cit. 2020-02-08].
- [11] Rekor Recognition Systems Inc., 2018. <https://github.com/openalpr/openalpr> [cit. 2020-02-08].
- [12] gerhardsletten, 2019. <https://github.com/gerhardsletten/express-openalpr-server> [cit. 2020-02-08].
- [13] Charlie Robbins, 2019. <https://github.com/indexzero/nconf> [cit. 2020-03-14].
- [14] Meteor Development Group Inc., 2020. <https://www.apollographql.com/docs/react/data/subscriptions/> [cit. 2020-03-30].
- [15] Meteor Development Group Inc., 2020. <https://www.apollographql.com/docs/link/> [cit. 2020-03-30].
- [16] MongoDB Inc., 2020. <https://docs.mongodb.com/manual/core/gridfs/> [cit. 2020-03-28].
- [17] Crazy Factory GmbH, 2019. <https://github.com/crazyfactory/ts-react-boilerplate> [cit. 2020-02-24].
- [18] Zack Story, 2020. <https://github.com/rt2zz/redux-persist> [cit. 2020-02-29].

- [19] Seshu Vinay. How to scan qrcode in android, 2019. <https://stackoverflow.com/questions/8830647/how-to-scan-qrcode-in-android/8830801> [cit. 2020-03-01].
- [20] David Katzmaier Geoffrey Morrison. Oled screen burn-in: What you need to know now, 2019. <https://www.cnet.com/how-to/oled-screen-burn-in-what-you-need-to-know-now> [cit. 2020-02-16].
- [21] Google LLC. Set up requestqueue, 2019. <https://developer.android.com/training/volley/requestqueue.html#singleton> [cit. 2020-03-01].
- [22] Google LLC. Access app-specific files, 2019. <https://developer.android.com/training/data-storage/app-specific> [cit. 2020-03-01].
- [23] Docker Inc., 2020. <https://docs.docker.com> [cit. 2020-02-16].

Seznam obrázků

2.1	Diagram komponent a jejich komunikace.	5
2.2	Příklad GraphQL dotazu z nástroje GraphQLPlayground.	5
2.3	Spojení React a Redux.	6
3.1	Adresářová struktura backendu.	8
3.2	Adresářová struktura frontendu.	8
3.3	Stránka pro zařízení s hlavním menu.	10
3.4	Stránka s pravidly a filtry – pohled na jeden den.	11
3.5	Stránka s pravidly a filtry – pohled na jeden měsíc.	12
3.6	Stránka s pravidly a filtry – výsledek kopírování.	12
3.7	Stránka se záznamy parkování.	13
3.8	Záznam parkování.	13
3.9	Příklad definice resolverů v SDL.	14
3.10	Screenshot dokumentace z nástroje GraphQLPlayground.	15
3.11	Ukázka obecného resolveru v Typescriptu.	16
3.12	Získání obrázků za použití hlavičky <i>Authorization</i> a jejich zpracování.	17
3.13	Definice typů <code>ParkingRuleAssignment</code> a <code>VehicleFilter</code> v GraphQL SDL.	18
3.14	Ilustrace problému úseček.	19
3.15	Mázání.	22
3.16	Ikona.	22
3.17	Checkoxy v Mozilla Firefox a Google Chrome.	23
3.18	Základní prvky uživatelského rozhraní.	24
3.19	Kalendářový graf z knihovny react-google-charts.	24
3.20	Tok dat v jednom z obecných vybírátek.	25
3.21	Vyrenderovaná komponenta jednoho vybírátka.	26
4.1	Životní cyklus mobilní aplikace.	29
4.2	Rozhraní mobilní aplikace.	30
5.1	Rozpoznaná SPZ v nehostinných podmínkách.	34

A. GraphQL Schema

