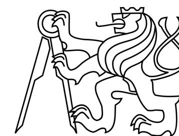
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název:	Uživatelská databáze dopravních p estupk
Student:	Jan Kirchner
Vedoucí:	Mgr. Petr Matyáš
Studijní program:	Informatika
Studijní obor:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2018/19

Pokyny pro vypracování

Navrhn te, realizujte a d kladn otestujte aplikaci na sb r záznam o p estupcích idi podpo ených fotografií i videozáznamem. Systém bude um t:
* spravovat databázi uživatel , vozidel a p estupk , každý uživatel bude mít seznam svých vozidel s SPZ,
* p ijmout záznam o p estupku, který bude mj. obsahovat SPZ vozidla, místo p estupku a typ p estupku, záznam bude nutné podložit fotografií nebo videozáznamem,
* vypsat místa s nejv tším po tem p estupk ,
* vypsat všechny p estupky v zadaném okruhu od vybraného místa (bez SPZ),
* vypsat uživateli p estupky vázané na jeho SPZ,
* administrátorovi vypsat seznam p estupk k zadané SPZ.

Seznam odborné literatury

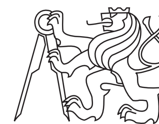
Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D. vedoucí katedry	prof. Ing. Pavel Tvrdík, CSc. d kan

V Praze dne 23. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství



Bakalářská práce

Uživatelská databáze dopravních přestupků

Jan Kirchner

Vedoucí práce: Mgr. Petr Matyáš

15. května 2017

Poděkování

Děkuji rodině a všem přátelům za podporu po celou dobu studia. Díky patří také Mgr. Petru Matyášovi za cenné rady při vedení této práce a nakonec speciální díky pro Dory za motivaci.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 15. května 2017 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Jan Kirchner. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Kirchner, Jan. *Uživatelská databáze dopravních přestupků*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato bakalářská práce popisuje proces návrhu a implementace webové aplikace Road Hogs sloužící k evidenci dopravních přestupků samotnými účastníky silničního provozu. Zaznamenané události podložené důkazem v podobě fotografie/videa mohou sloužit buď běžným uživatelům jako ukazatel bezpečnosti dané pozemní komunikace, nebo správcům vozových parků pro sledování svěřených automobilů. Výsledkem je funkční prototyp schopný nasazení a běhu v reálném prostředí.

Klíčová slova React, .NET, ASP.NET, Google maps, webová aplikace, dopravní přestupek, Road Hogs

Abstract

This bachelor thesis describes the process of designing and implementing the Road Hogs web application for recording traffic violations by drivers themselves. Recorded events supported by image/video evidence can serve either for users as road safety indicators or for car park managers as a vehicles tracking service. The result is a functional prototype capable of deploying and running in a real environment.

Keywords React, .NET, ASP.NET, Google maps, web application, traffic violation, Road Hogs

Obsah

Úvod **1**

1 Analýza a návrh **3**
1.1 Konkurenční aplikace 3
1.2 Požadavky 4
1.3 Případy užití 5
1.4 Architektura aplikace 6
1.5 Datový model 8
1.6 Návrhy obrazovek a design 9
1.7 Bezpečnost 12
1.8 Návrh technologií 15

2 Implementace **19**
2.1 Implementace serverové části 19
2.2 Implementace klientské části 26
2.3 Závěr 32

3 Testování a nasazení **33**
3.1 Testování serverové části 33
3.2 Testování klientské části 37
3.3 Nasazení 38

Závěr **41**

Literatura **43**

A Seznam použitých zkratek **45**

B Manuál na spuštění aplikace **47**

C Obsah přiloženého CD **49**

Seznam obrázků

1.1 Diagram případů užití . . . 7
1.2 Třívrstvá architektura aplikace . . . 8
1.3 Základní databázový model . . . 8
1.4 Rozšířený databázový model . . . 9
1.5 UI návrh hlavní stránky . . . 11
1.6 UI návrh obrazovky uživatelského účtu . . . 11
1.7 UI návrh obrazovky managementu dat . . . 12
1.8 Schéma autentizace pomocí cookies a tokenů . . . 13

2.1 React komponenty . . . 28
2.2 Redux data flow . . . 29

3.1 Snímek obrazovky aplikace . . . 42

Úvod

Mnoho velice úspěšných aplikací, které v posledních několika let vznikly, pracují na poměrně jednoduchém principu – spojují lidi, kteří mají přebytek nějakého prostředku s lidmi, které trápí naopak nedostatek. Role mediátora v tomto procesu může přinést opravdu silnou pozici a dokonce změnit fungování trhu tak, jak jej známe. Jako příklad se dá jmenovat AirBnB – služba spojující majitele volné nemovitosti nebo pokoje s někým, kdo shání ubytování; Uber, který spojí majitele ne stoprocentně vytíženého vozidla s lidmi, kteří se chtějí přepravit z bodu a do bodu B, nebo třeba mnohem starší Ebay – internetový bazar. Nemusí to však být jen věci, popřípadě služby, které se mezi uživateli dají vyměnit nebo zprostředkovat. Mohou to být i informace jakéhokoliv charakteru.

Cílem této bakalářské práce je uvést do provozu aplikaci, díky které by měli účastníci silničního provozu možnost zaznamenávat nepříjemné události, kterým byli svědky. Nápad vznikl především jako reakce na neustále větší množstvích kamer do auta, které si řidiči pořizují jako spolehlivého svědka pro případ nehody. Takto získaná data se v dnešní době většinou nenávratně ztrácí – většina incidentů naštěstí nemá závažné důsledky a jakékoliv zpracování nahraného materiálu tak nemá cenu. i tak se ale jedná o cenné informace, které mohou sloužit v podstatě komukoliv, kdo se se silničního provozu sám účastní k identifikaci rizikových úseků neznámé cesty.

Nejedná se o aplikaci, která by měla za cíl vydělávat peníze, ale spíše o komunitní projekt v první řadě pomáhající samotným řidičům. Důležitý aspekt je ovšem i ono samotné uchování informací, které by v budoucnu mohly mít potenciál a být využity. Jen čas ukáže, zda vůbec a jakým způsobem.

Kapitola 1

Analýza a návrh

Analýza a návrh aplikace představuje nástroj, jak za asistence zadavatele zhmotnit co nejjasněji představu o budoucí aplikaci nebo systému. Výsledkem by měly být dokumenty přesně popisující funkcionality a veškeré chování systému včetně technických aspektů – tedy toho, jakými metodami, jazyky a s pomocí jakých nástrojů bude software vyvinut. Finální specifikace je pak důležitým podkladem pro vývojáře a zároveň slouží pro ověření, zda dodaný software naplňuje požadavky a cíle zákazníka.

1.1 Konkurenční aplikace

Po prvotní konzultaci zadání jako první proběhl průzkum, jestli již neexistuje aplikace, která by alespoň v základních bodech splňovala cíle této bakalářské práce. Průzkum byl veden dvěma směry. v prvním případě šlo o to najít webové služby pomocí vyhledávače Google[1] a v případě druhém nativní mobilní aplikace pomocí Google Play[2], nebo Apple Store[3].

V obou případech hledání skončilo neúspěchem. Většina nalezených služeb se totiž zaměřuje výhradně na aktuální dopravní situaci obohacenou například informacemi o počasí. Na jednu stranu má uživatel šanci získat přehled o dění na silnici, na druhou stranu hlavní smysl této aplikace, tedy dlouhodobá evidence dopravních přestupků, zůstává nenaplněn. Dalším příkladem může být pár nalezených dopravních databází, ve kterých jsou k dispozici základní údaje o přestupcích zaznamenaných různými lokálními státními institucemi. Ani v tomto případě se však o splnění cílů nedá mluvit. Jedná se zaprvé jen o lokální data na úrovni okresů a zadruhé samotná forma znemožňuje běžným uživatelům jejich rozumné využití. k dispozici totiž nebylo žádné uživatelské rozhraní, ale jen možnost stáhnout textové JSON/CSV soubory.

1.2 Požadavky

Požadavky mají za úkol popsat vlastnosti navrhované aplikace tak, aby ji zadavatel mohl nakonec převzít. Dělí se na požadavky funkční, které mají zafixovat představu o funkcionalitách, které by program měl naplňovat, a na nefunkční, které mají na druhou stranu popsat kvalitu dodaného řešení. Stručně řečeno, první druh požadavků popisuje obsah a druhý formu dodaného softwaru.

Všechny požadavky by měly být proveditelné, měřitelné a zároveň testovatelné tak, aby se případné spory daly jednoznačně rozhodnout.

1.2.1 Funkční požadavky

- Registrace uživatele – vyžadující základní informace o uživateli, jako je jméno, heslo, e-mailová adresa a unikátní uživatelské jméno.
- Přihlášení uživatele – pomocí správné kombinace uživatelského jména a hesla.
- Odhlášení uživatele.
- Změna informací o uživateli – data o uživateli až na uživatelské jméno bude možné měnit.
- Editace registračních značek v seznamu sledovaných automobilů – uživatelé budou mít možnost přidávat, nebo odebírat RZ do/ze seznamu sledovaných automobilů.
- Zobrazení evidovaných přestupků – Uživatelé budou mít možnost zobrazit aplikací evidované dopravní přestupky. Ty dále bude možné několika způsoby filtrovat. To půjde pomocí:
  - místa, kde se přestupek odehrál,
  - data, kdy se přestupek odehrál,
  - typu dopravního přestupku,
  - pomocí seznamu sledovaných automobilů,
- Zobrazení detailu přestupku – uživatel bude mít možnost zobrazit detail přestupku.
- Přidání nového přestupku – aplikace umožní přidání nového dopravního přestupku. Kromě povinného udání místa, kde se situace stala, typu a SPZ, bude nutné záznam podložit důkazním materiálem. Buď fotografií, nebo videem.
- Smazání přestupku – aplikace umožní smazání evidovaného dopravního přestupku nesplňujícího pravidla.

- Zobrazení registrovaných uživatelů – aplikace umožní zobrazení přehledného seznamu všech registrovaných uživatelů.
- Editace rolí registrovaných uživatelů – aplikace umožní přidat/odebrat roli registrovanému uživateli.

1.2.2 Nefunkční požadavky

- Dostupnost, rychlost a stabilita – aplikace by měla být neustále dostupná z celého světa a poskytovat odpovědi bez dlouhých prodlev, které by mohly odradit uživatele od používání.
- Bezpečnost – aplikaci by měla odpovídat běžným bezpečnostním standardům.
- Bezúdržbovost – aplikaci by mělo být možné spravovat s minimálním možným úsilím.
- Design – aplikace by měla být minimalistická a poskytovat uživatelům informace v graficky příjemné, jednoduché a intuitivní formě.
- Responzivní design – aplikace by měla být responzivní tak, aby ji mohli snadno používat i uživatelé přistupující pomocí mobilního telefonu.
- Lokalizace – aplikace by měla po prvním kole implementace podporovat český a anglický jazyk. Dále by měla být připravena na rozšíření o další světové jazyky.
- Možnost používání bez registrace – aplikace by měla být přístupná i uživatelům, kteří se nechtějí registrovat.

1.3 Případy užití

Vymodelování případů užití je další způsob, jak zachytit požadavky klienta na dodávaný software a najít díky tomu to nejlepší řešení. Oproti obecnému výčtu požadavků reflektuje tento model i role, vztahy mezi nimi a také případné hranice systémů.

1.3.1 Role

Návrh aplikace v současné době počítá se třemi rolemi a akcemi vyvolanými nepřihlášeným uživatelem. Jsou jimi:

- User,
- Admin,
- SuperAdmin.

1.3.1.1 Nepřihlášený uživatel

Aplikaci mohou využívat a lidé, kteří se z jakéhokoli důvodu nechtějí registrovat. Jejich možnosti jsou ale omezené – k datům o evidovaných dopravních přestupcích mohou jenom přistupovat bez možnosti je měnit, nebo do nich jakkoli zasahovat.

1.3.1.2 User

Registrací získává uživatel automaticky roli User. Kromě toho, že je pak umožněno dopravní přestupky přidávat a aktivně tak tvořit obsah webové aplikace, se otevře možnost editovat svůj uživatelský účet – například seznam sledovaných automobilů pro rozšíření možnosti filtrování dat.

1.3.1.3 Admin

Role Admin je rozšířením běžného uživatelského účtu. Nově přibývá přístup do další sekce sloužící ke správě aplikačních dat, jako jsou registrovaní uživatelé a dopravní přestupky. Předpokládá se, že uživatelů s rolí Admin bude více a jejich starostí bude odstraňovat záznamy nesplňující podmínky aplikace – data o uživatelích jsou ale pro tuto roli stále chráněna před zápisem.

1.3.1.4 SuperAdmin

SuperAdmin je rozšířením role Admin. Předpokládá se, že počet uživatelů s touto rolí bude minimální. Mezi nově nabyté možnosti patří editace registrovaných uživatelů a přiřazování aplikačních rolí.

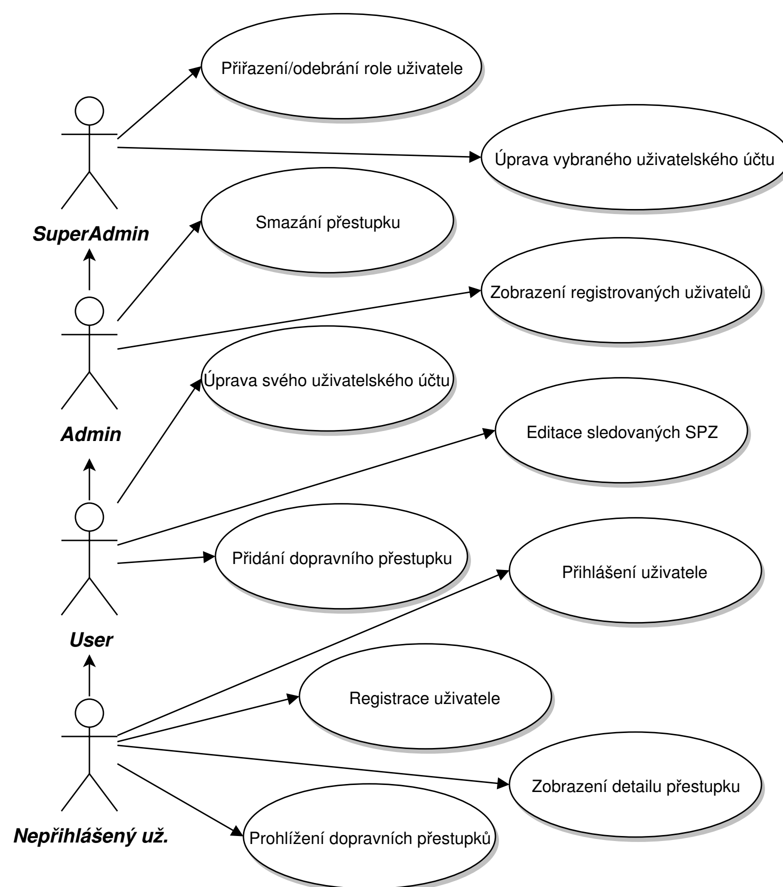
1.3.2 Model případů užití

Jak je z popisu rolí jasné, existuje mezi nimi jednoduchá hierarchie a nadřazená role je vždy jen rozšířením role podřazené. To značně zjednodušuje vytváření samotného modelu případů užití (viz obrázek 1.1) a v konečném důsledku i implementaci.

1.4 Architektura aplikace

Navrhovaná webová aplikace klasické architektury typu klient-server bude rozdělena do tří oddělených se sebou navzájem komunikujících vrstev. Těmito vrstvami jsou:

- Prezentační vrstva – grafické rozhraní mající za úkol prezentovat data v pro člověka dobře čitelné formě. Slouží rovněž jako brána pro vstup nových požadavků. Data ovšem nijak nezpracovává, ale jen zobrazuje



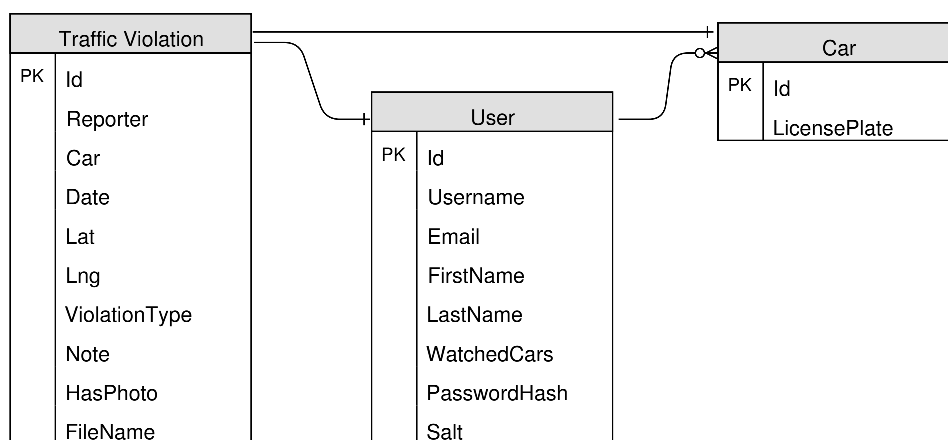
Obrázek 1.1: Diagram případů užití

- Aplikační vrstva – jádro aplikace, které zajišťuje zpracování veškerých příchozích požadavků
- Datová vrstva – vrstva zodpovědná za perzistentní uchování dat. Nejčastěji se jedná o databázi, ale může jí být i souborový systém, jejich kombinace, popřípadě jiné řešení

Toto rozdělení kromě toho, že umožňuje přehlednější dělení funkčnosti, reflektuje i skutečné hranice infrastruktury, na které aplikace bude běžet. o datovou vrstvu se postará databázový systém, o aplikační vrstvu webový server a o běh prezentační vrstvy webový prohlížeč na straně uživatele.

Kromě samotného rozdělení na jednotlivé vrstvy je třeba vyřešit i to, jak mezi sebou budou navzájem komunikovat. Na rozhraní klienta a aplikačního serveru bude fungovat zabezpečené aplikační rozhraní komunikující přes JSON a na pomezí aplikačního a databázového serveru bude implementováno rozhraní pomocí technologie Entity Framework. Mezi datovou a prezentační vrst-

Obrázek 1.2: Třívrstvá architektura aplikace



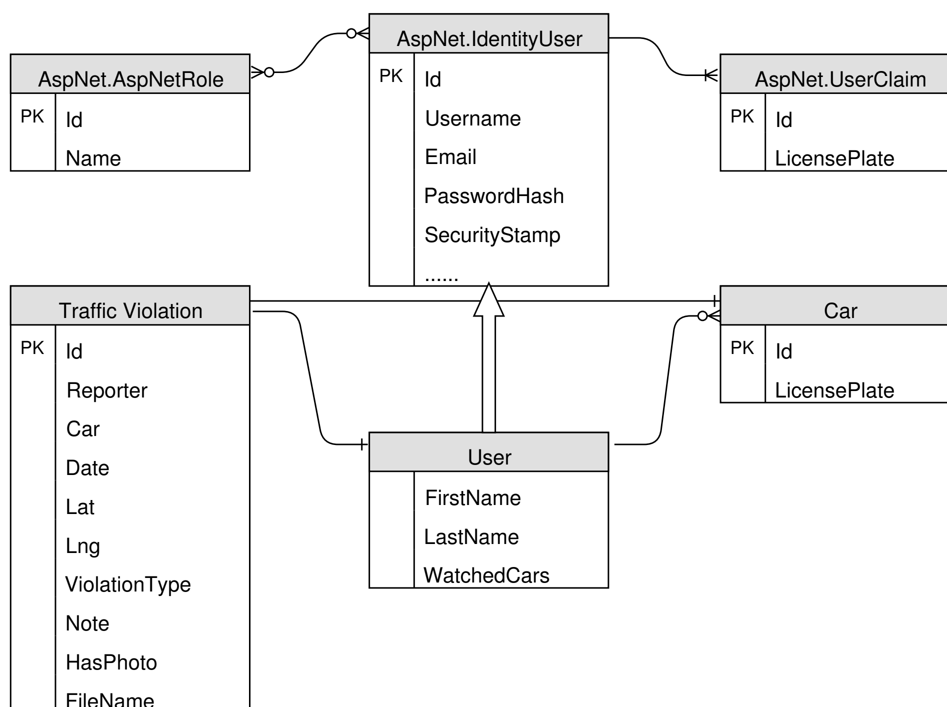
Obrázek 1.3: Základní databázový model

vou by na základě takto zvolené architektury k žádné komunikaci docházet nemělo, jak je vidět i na Obrázku 1.2.

1.5 Datový model

Datový model aplikace je opravdu jednoduchý. v programu figurují tři základní entity – uživatel, dopravní přestupek a automobil. Diagram takového modelu je zobrazen na Obrázku 1.3. Má ovšem zásadní vadu na kráse – a totiž fakt, že neumožňuje pokrytí jednoho ze zásadních požadavků na aplikaci, kterým je zabezpečení API rozhraní na straně serveru. Tyto funkcionality budou řešeny pomocí knihovny ASP.NET Identity, která model rozšířila do následující podoby. Viz obrázek 1.4.

Detailnímu popisu vytváření databáze je věnována vlastní kapitola 2.1.3



Obrázek 1.4: Rozšířený databázový model

1.6 Návrhy obrazovek a design

Při navrhování obrazovek bylo na základě požadavků kladeno za cíl udržet co nejprostší a nejčistší design. Tím hlavním důvodem pro tuto snahu je fakt, že jednoduché stránky se lépe používají, programují, a v neposlední řadě rozšiřují (jako příklad se dají jmenovat již v úvodu zmiňované služby: AirBnB, Uber a další). Dalším podstatným důvodem byl fakt, že aplikace bude mít responzivní design. Ten se v případě komplikovaného uživatelského rozhraní implementuje podstatně složitěji.

Jedním z významných funkčních požadavků aplikace je již zmíněný responzivní design. Ten je nutný především kvůli předpokládanému využívání aplikace na mobilních telefonech, pomocí kterých mohou být porušení dopravních předpisů často zaznamenány a v případě jednoho ze scénářů rovnou (nebo i později) z toho samého zařízení přímo nahrány do aplikace. v první části vývoje se tedy počítá s dvěma možnými rozloženími v závislosti na velikosti přístroje. První verze se zobrazí uživatelům používajícím mobilní telefony a druhá všem zbylým – tedy lidem přistupujícím na stránky přes tablet, notebook, počítač, nebo jakékoli jiné větší zařízení.

1.6.1 Společné prvky

Prvkem, který se v aplikaci vyskytuje vždy, je navigační lišta. Ta poskytuje minimální množství ovládacích prvků tak, aby působila přehledně a čistě. Kromě ikony pro návrat na hlavní stránku a názvu služby, nabízí lišta tlačítka pro přihlášení/odhlášení, změnu jazyka a také odkaz na správu uživatelského účtu. Při přihlášení uživatele v roli Admin/SuperAdmin pak dále odkaz na správu aplikačních dat.

Pro vzhled aplikace jsem zvolil specifikaci vytvořenou společností Google a pojmenovanou Material Design[4]. Koncepčně se jedná o velice jednoduché, čisté a přehledné motivy, které převzalo mnoho grafických knihoven – například Material-UI[5], která nabízí všechny potřebné grafické elementy a bude v aplikaci použita.

1.6.2 Hlavní stránka

Navrhnout hlavní stránku tak, aby ve všech ohledech vyhovovala, byl oříšek. Stránka nakonec bude rozdělena na dvě části. Tou první, dominující, bude mapa obsahující záznamy o všech přestupcích. Využito bude Google maps APIs[6], které je velice rozšířené a poskytuje všechny pro svůj návrh potřebné funkce. Vedlejší oblast pak bude také rozdělena – na část zodpovědnou za filtrování zobrazovaných přestupků a na detail, který bude mít dva módy. Prvním je zobrazení detailu přestupku a druhým je formulář pro uložení nového záznamu. Výsledek ilustruje obrázek 1.5.

1.6.3 Uživatelský účet

Stránka s detailem o uživatelském účtu je velice jednoduchá a bude obsahovat tři základní oblasti:

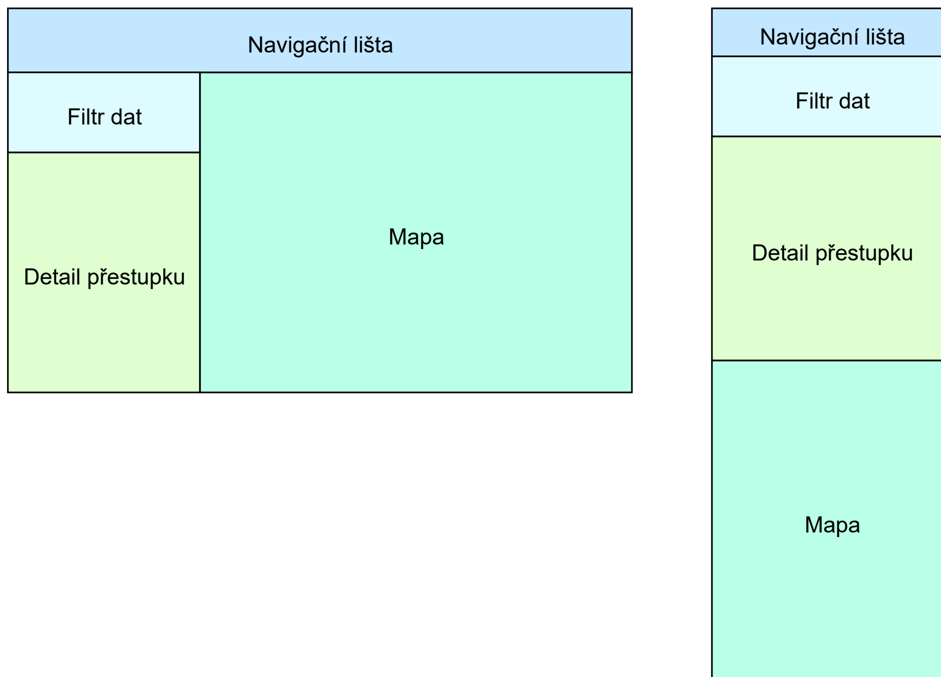
- detail účtu,
- sledované SPZ,
- informace o aplikaci.

Návrh najdete na obrázku 1.6

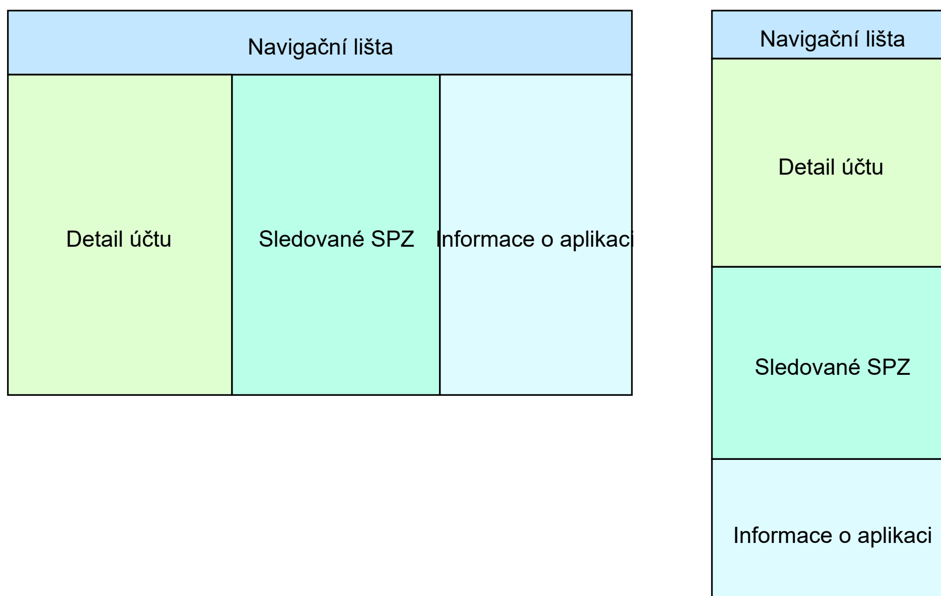
1.6.4 Management dat

Poslední stránka, která v aplikaci je k dispozici, je určena pouze pro uživatele v roli Admin/SuperAdmin. Již z názvu podkapitoly je jasné, že smyslem je poskytnout administrátorům prostředí pro správu dat aplikace – tedy možnost prohlížení a editace uživatelů a zaznamenaných dopravních přestupků. Obrazovka je rozdělena na tři základní sekce:

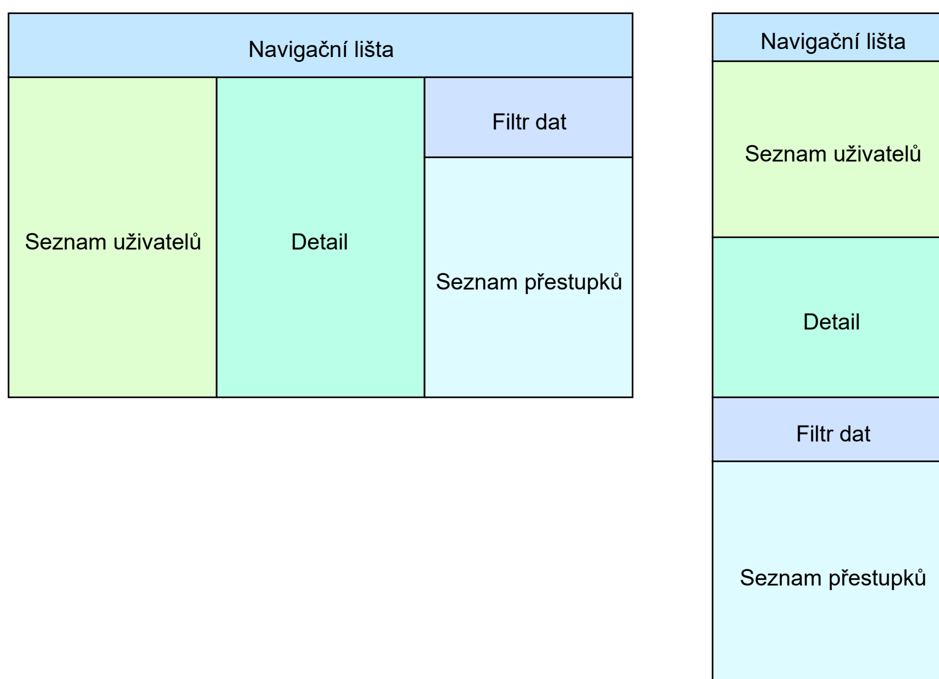
- seznam uživatelů,



Obrázek 1.5: UI návrh hlavní stránky



Obrázek 1.6: UI návrh obrazovky uživatelského účtu



Obrázek 1.7: UI návrh obrazovky managementu dat

- seznam dopravních přestupků,
- detail vybrané položky (může jím být obojí).

a modely jsou vyobrazeny na obrázku 1.7.

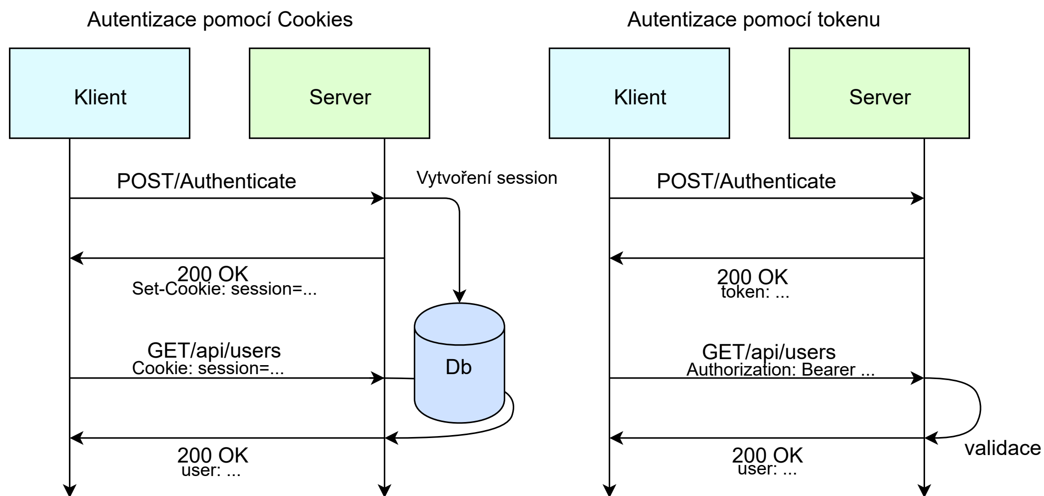
1.7 Bezpečnost

Jelikož se jedná o webovou aplikaci schraňující data, otázka bezpečnosti byla jednou z těch zásadních. v první řadě bylo důležité navrhnout datový model tak, aby umožňoval autorizaci a autentizaci přicházejících požadavků. Na straně serveru bylo nasnadě použít již hotové řešení integrované přímo v ASP.NET. Širší možnost výběru pak nabízel princip ověřování. Zde připadaly v úvahu především dvě nejrozšířenější technologie vhodné pro zabezpečení API – Cookies a tokeny.

1.7.1 Cookies

Autentizace pomocí cookies probíhá následovně (viz obrázek 1.8):

1. Uživatel odešle na server své uživatelské jméno a heslo.



Obrázek 1.8: Schéma autentizace pomocí cookies a tokenů

2. Server údaje zvaliduje a vytvoří v databázi sezení obsahující data o probíhající relaci – její identifikátor je pak odeslán zpět klientovi.
3. Klient uloží do cookie obdržené session id.
4. k dalším dotazům prohlížeč automaticky připojí tento identifikátor.
5. Server v databázi ověří validitu id a pokud je klient rozpoznán, provede dotaz – všechna potřebná data jsou uložena do daného sezení.

Autentizace pomocí cookies je tím starším řešením, které v posledních letech ustupuje. Mezi hlavní nevýhody patří především nutnost komunikovat s databází víc, než je nutné, což s sebou nese otázku efektivity tohoto přístupu. Dalšími nevýhodami je nutnost komplikovaného řešení v případě, že má aplikace využívat více webových služeb v rámci jedné stránky – cookie se totiž váže vždy jen k jedné doméně. Jako mínus by se dal brát i fakt, že id sezení je součástí každého dotazu, i když to třeba není u dotazů nevyžadujících autentizaci nutné.

1.7.2 Tokeny

Přestože existuje celá řada možností, jak token implementovat, standardem se stal JWT token. Proces autentizace probíhá následovně (viz obrázek 1.8):

1. Uživatel odešle na server své uživatelské jméno a heslo.
2. Server data zvaliduje a v odpovědi vrátí vygenerovaný token.
3. Klient token uloží.

4. k dalším dotazům na server může klient do hlavičky Authorization přiložit uložený token.

5. u takových dotazů jej server zvaliduje a provede dotaz s pomocí dat obsažených v tokenu.

I když se na první pohled může zdát, že je proces podobný jako u cookies, je zde několik sice nenápadných, ale zásadních změn. Tou nejdůležitější je, že všechna data potřebná pro komunikaci jsou obsažena v tokenu samotném a server tak nemusí vůbec komunikovat s datovou vrstvou. Autentizace je tedy bezestavová a nezatěžuje zbytečně databázi.

Data, která token obsahuje, se nazývají nároky (claims) a může jimi být prakticky cokoliv, co server potřebuje k ověření a vykonání příchozích dotazů – v případě této webové aplikace to jsou především role a také unikátní identifikátor uživatele. Při rozhodování, co by token měl obsahovat, je ale vždy třeba myslet na to, že jeho velikost s každým přidaným nárokem roste a příliš mnoho obsažených dat by mohlo při velkém množství dotazů zpomalit komunikaci – ideální je tedy snažit se udržet množství nároků co nejmenší.

1.7.3 Závěr

Rozhodnout se mezi zmíněnými řešeními se při seznamování s problematikou zdálo jako větší problém, než tomu nakonec bylo. Jako technologie vhodnější pro tuto aplikaci byla nakonec vybrána pro v posledních letech stále populárnější autentizaci pomocí JWT tokenů a to hned z několika důvodů. Tím hlavním je především jednodušší implementace, dále bezestavovost celého procesu a také hledisko výkonu. Oproti tomu jako hlavní vadu na kráse JWT tokenů vidím jejich velikost. Ta je i v případě tokenu obsahujícího jeden jediný nárok mnohem větší, než velikost id sezení, což může zpomalovat komunikaci klienta se serverem především v případě pomalého internetového připojení. u této aplikace se ale nepočítá s vytvářením tolika dotazů, aby se problém stal i reálnou hrozbou. Navíc JWT token na rozdíl od cookies nemusí být obsažen v každému dotazu a množství přenášených dat se tak dá jistým způsobem regulovat.

Další nespornou výhodou je, že pokud by se v budoucnu rozhodlo o vývoji nativních mobilních aplikací, byla by implementace řádově snadnější. API by v takovém případě nekomunikovalo pouze s prohlížeči, ale i s mobilními klienty. Rozdíl je ovšem v tom, že internetové prohlížeče technologii cookies standardně podporují, ale u mobilní aplikace by ji bylo třeba od základu naimplementovat. To je řádově složitější problém, než obyčejné uložení JWT tokenu do libovolného úložiště.

1.8 Návrh technologií

V této kapitole je popsán seznam technologií, které budou použity pro implementaci celé aplikace. Při jejich výběru byla v maximální možné míře snaha držet se výsledků analýzy a zohledňovat především funkční a nefunkční požadavky. u těch by při špatné volbě mohlo hrozit jejich nenaplnění v tom horším případě, v tom lepším pak práce navíc při vymýšlení řešení, jak problém obejít.

1.8.1 Server

Na straně serveru jsem pro implementaci zvolil .NET framework – kód aplikace pak bude v jazyce C#. Jedná se o produkt společnosti Microsoft[7], který poskytuje dostatečně silný ekosystém pro vývoj i mnohem složitějších aplikací. Další alternativou by mohl být jazyk Java, který je s jazykem C# v mnohém podobný a při výběru často hraje největší roli preference zadavatele, nebo zkušennosti vývojového týmu. Obě varianty jsou totiž pro takto velké webové aplikace vhodné a nedá se snadno rozhodnout, která je lepší. Principy obou jazyků jsou totiž prakticky stejné, množství a kvalita nabízených knihoven srovnatelná a ani úsilí potřebné pro implementaci aplikace neukazuje zásadní rozdíl. v tomto případě rozhodlo především snadnější úvodní nastavení projektu, jelikož Visual Studio poskytuje celou řadu šablon předpřipravených pro projekty různého rázu.

1.8.1.1 Aplikační vrstva

Aplikační vrstva, která vyplňuje prostor mezi klientem a vrstvou datovou a je zodpovědná za výpočty a zpracování dat, bude implementovaná pomocí frameworku ASP.NET, který je součástí .NET. Tento framework poskytuje celou řadu knihoven sloužících ke snadnému vytvoření webových aplikací. Jedná se o robustní a spolehlivé řešení, které by mělo poskytovat veškeré nutné technologie a funkcionality.

1.8.1.2 Datová vrstva

Rozhraní pro datovou vrstvu bude implementováno za pomoci Entity Frameworku [8] verze 6.1.3. Jedná se o další produkt společnosti Microsoft fungující jako objektově relační mapovač, s kterým může programátor k entitám v databázi přistupovat stejně jako by se jednalo o kolekce v paměti. Kromě toho je pomocí EF možné databázi i navrhnou a vytvořit. Princip bude detailně vysvětlen v kapitole implementace. Hlavní výhodou je dokonalé odstínění programátora od celého databázového stroje, intuitivní přístup k datům a také široké možnosti konfigurace tohoto řešení.

Samotná databáze pak bude běžet na MSSQL[9] serveru rovněž od Microsoftu. v budoucnu ovšem není problém přejít na jinou technologii.

1.8.2 Klient

Výběr technologie pro klientskou stranu aplikace byl o něco komplikovanější. Ne snad proto, že by jen málokterá z nich byla schopna naplnit všechny požadavky vzešlé z analýzy, ale hlavně kvůli nepřebernému množství JavaScriptových frameworků, které v posledních několika letech vznikly. Vzhledem k trendu posledních let jsem se rozhodl pro koncept Single Page Application. Jak název napovídá, jedná se o jedinou stránku, která se však tváří jinak – chováním napodobuje stránky klasické, ale za asistence javascriptu jsou viditelné pouze části, které by měly být, a ostatní obsah je skryt. Mezi hlavní výhodu patří prakticky nulová odezva při přechodu na jinou podstránku, jelikož se ze serveru nestahuje žádný HTML kód. To se zdá možná jako maličkost, ale ve skutečnosti se jedná o vlastnost, která má velmi pozitivní vliv na UX.

Na poli javascriptových frameworků, které SPA podporují, jsem vybral tři kandidáty:

- React[10],
- AngularJS[11],
- Vue.js[12].

Všechny výše zmíněné technologie je poměrně těžké srovnávat. Prvním pohledem na situaci může být to, jak jsou si navzájem blízké. Pravdou totiž je, že React a Vue.js jsou si principiálně velice podobné. Platí, že Vue.js je o něco menší knihovna, která je podle autorů výkonnostně lépe optimalizovaná, ale na druhou stranu nenabízí programátorům tolik možností. Pro srovnání by mohlo posloužit přirovnání k cihlám (reprezentujícím React) a prefabrikovaným stěnám (reprezentujícím Vue.js). v druhém případě může jít sice stavba rychleji a snáz, ale při specifických požadavcích na výsledek aplikace může dojít k problémům. Dalším faktem je, že zatímco přechod z Reactu na Vue.js je otázka pár hodin praxe, tak v opačném směru je tomu jinak. s určitým nadhledem by se tak dalo říct, že člověk, který se naučí React, se naučí obě technologie naráz. Tou nejzásadnější slabinou je ale stáří dané technologie. Vue.js vzniklo na přelomu let 2015 a 2016 a základna uživatelů a velikost celého ekosystému je oproti Reactu nesrovnatelně menší. i z tohoto důvodu bylo Vue.js z možných kandidátů vyřazeno jako první.

V druhém kole šlo o to rozhodnout se mezi Angularem, za kterým stojí společnost Google, a mezi Reactem vyvinutém v sídle Facebooku[13]. Obě technologie jsou silnými hráči na trhu a může být těžké mezi nimi rozhodovat. Nabízejí sice koncepčně jiný pohled na vývoj frontendu aplikace, ale výkonnostně jsou na tom podobně (pokud mluvíme o novější verzi Angular 2) a zázemí obou technologií je rámcově stejné. Faktem je, že obě varianty jsou pro řešení vhodné a výběr je stejně jako v případě C# a Javy ovlivněn hlavně zkušenostmi vývojového týmu. i v tomto případě je úsilí nutné pro vyvinutí

funkční aplikace stejné a výsledek nerozpoznatelný. Nakonec bylo rozhodnuto pro použití Reactu, který je podle mnohých názorů mírně vhodnější pro menší aplikace.

Kapitola 2

Implementace

Po dokončení analýzy přišla na řadu implementace – tedy převedení myšlenek zachycených ve specifikaci do reálného výstupu, který může být otestován a nasazen tak, aby mohl začít sloužit prvním reálným uživatelům. Vzhledem k tomu, že proces implementace serverové části byl značně odlišný od implementace strany klienta, bude tento fakt reflektovat i rozdělení podkapitol.

2.1 Implementace serverové části

2.1.1 Použité nástroje

Použití vhodných nástrojů během vývoje aplikace může značně usnadnit práci. Zde je příklad těch, které sehrály při implementaci serverové části aplikace největší roli:

- Visual Studio 2013[14] – IDE, které mezi .NET vývojáři suverénně patří mezi ty nejpopulárnější,
- SQL Server Management Studio[15] – průzkumník databáze,
- Insomnia[16] – provolávač REST API,
- Microsoft TFS[17] – nástroj pro správu verzí.

2.1.2 Začátek vývoje

Vzhledem k tomu, že serverová část programu představuje poměrně jednoduché aplikační rozhraní se základními operacemi nad několika málo entitami, bylo prvotní nastavení projektu nejobtížnějším bodem. Na konci tohoto procesu by měl být projekt v takové stavu, aby další rozšiřování o nové funkcionality znamenalo pouze programování a žádné zasahování do konfiguračních souborů a tříd. Práce byla zahájena vytvořením prázdné šablony WEB API

projektu, která je ve vývojovém prostředí Visual Studia k dispozici. Následující kapitoly popisují funkční popis řešení především složitějších problémů, kterým bylo během vývoje aplikace třeba čelit.

2.1.3 Vytvoření databáze

Databáze je vytvářená na základě principu Code first technologie Entity Framework. v takovém případě naprosto odpadá jakákoli práce se skripty a vše se odehrává přímo v kódu. Při prvním spuštění aplikace se vytvoří databáze kopírující schéma tříd programu. Kroky, pomocí nichž lze dosáhnout vytvoření takové databáze jsou:

- v konfiguračním souboru přidat textový řetězec pro přístup do databáze,
- vytvořit třídu definující kontext aplikace – data jsou zde definována jako kolekce, takže se s nimi dá velice snadno pracovat bez nutnosti znát jakýkoliv dotazovací jazyk typu SQL. Dále metoda může obsahovat další nastavení – například úpravu vazeb mezi tabulkami, nebo třeba způsob mazání objektů z databáze,
- optimálně vytvořit Seed metodu s iniciálními daty.

Vzhledem k tomu, že je v případě tohoto programu třeba zabezpečit některé metody API proti neautorizovaným voláním, je nezbytné myslet na to již v prvních krocích. u .NET aplikací je nejsnažší využít knihovny ASP.NET Identity, která sice řeší spoustu důležitých funkcionalit, ale jen pokud programátor dokáže správně naimplementovat třídu definující databázový kontext aplikace. Ta by normálně byla snadno rozpoznatelná díky dědění třídy DbContext. Tentokrát ovšem bude děděna třída obohacená o další metody a je jí generická IdentityDbContext<TUser>, kde TUser musí implementovat rozhraní IIdentityUser obsahující všechny nutné vlastnosti pro robustní zabezpečení. Prvním krokem tedy bylo rozšíření třídy User o IdentityUser (Třída knihovny Identity) pomocí dědičnosti. Další změna pak proběhla v již zmínění třídě definující kontext. Nejpodstatnější části zdrojového kódu pak vypadají následovně:

```
public class AuthContext : IdentityDbContext<User>
{
  public AuthContext() : base("AuthContext")
  {
    Database.SetInitializer<AuthContext>(
    new AuthDbInitializer());
  }

  public DbSet<Car> Cars { get; set; }
  public DbSet<TrafficViolation>
    TrafficViolations { get; set; }
```

```
    protected override void OnModelCreating(
      DbModelBuilder modelBuilder)
    {
      modelBuilder.Entity<User>()
        .HasMany(m => m.WatchedCars)
        .WithMany();
    }
}
```

String, který je předán do konstruktoru předka definuje, jaký řetězec pro připojení do databáze z konfiguračního souboru bude použit. Pro programátora to znamená snadné přepínání mezi databázemi, pokud je to třeba. Dále je v konstruktoru identifikována inicializační třída databáze (obsahující i Seed metodu pro iniciální naplnění databáze daty). Její předpis vypadá následovně:

```
public class AuthDbInitializer
  : CreateDatabaseIfNotExists<AuthContext>
{
  protected override void Seed(AuthContext context)
  {
  ...
  }
}
```

Jak je vidět, v případě této aplikaci dojde k vytvoření databáze jen pokud žádná neexistuje. Při vývoji je však často šikovné využít módu vytvoření databáze ve chvíli, kdy dojde ke změně datového modelu. Pokud se tak totiž stane a databázový model neodpovídá definici tříd, stejně dojde při startu k chybě.

2.1.4 Generické repozitáře

Jedním z principů pro psaní čistého kódu je i DRY[18]. Ten si klade za cíl omezit opakování informací jakéhokoliv typu. Jelikož se v aplikaci objevuje více míst, kdy je třeba provádět základní CRUD operace s několika různými druhy entit, bylo více než vhodné naimplementovat generický návrhový vzor Repository. Ten poskytuje nadstavbové rozhraní mezi datovou a aplikační vrstvou, které ve výsledku při minimální (či vůbec žádné) replikaci zdrojového kódu umožňuje:

- základní operace pro všechny entity implementující rozhraní,
- snadné rozšíření pro každé jednotlivé entity,
- trvalé uložení do databáze .

Základní kámen tohoto návrhového vzoru je generické rozhraní IRepository, ve kterém jsou definovány operace společné pro všechny entity.

```
public interface IRepository<T> where T : class
{
  IEnumerable<T> GetAll();
  IEnumerable<T> FindBy
        (Expression<Func<T, bool>> predicate);
  T GetById(long id);
  void Add(T entity);
  void Delete(T entity);
  void Edit(T entity);
  void Save();
}
```

Z tohoto rozhraní bude děděno v třídě Repository, která pak bude funkce genericky implementovat. Výsledek (s jednou ukázkovou metodou) vypadá následovně:

```
public class Repository<T>
  : IRepository<T> where T : class
{
  protected readonly DbContext Context;

  public Repository(DbContext context)
  {
    Context = context;
  }

  public void Add(T entity)
  {
    Context.Set<T>().Add(entity);
  }
}
```

V tuto chvíli už stačilo vytvořit interface pro repozitář konkrétní entity – zde v příkladu pro CarRepository, který bude dědit generický interface IRepository a případně rozšiřovat nabídku dostupných metod pro konkrétní třídu – v tomto případě Car.

```
public interface ICarRepository : IRepository<Car>
{
  Car GetCarBySpz(string licensePlate);
}
```

Na samotný závěr pak přijde implementace samotného repozitáře pro konkrétní entitu. Taková třída dědí jak generickou třídu Repository(takže pro danou entitu budou dostupné základní operace) tak rozhraní rozšiřující funkcionality, které je ovšem právě zde nutné implementovat. Kód vypadá následovně:

```
public class CarRepository
  : Repository<Car>, ICarRepository
{
```

```
  public CarRepository(AuthContext context)
    : base(context)
  {
  }

  public Car GetCarBySpz(string licensePlate)
  {
    return FindBy(o => String.Compare(
      o.LicensePlate, licensePlate, true)== 0)
        .FirstOrDefault();
  }
}
```

2.1.5 BaseApiController

Všechny kontrolery, které přijímají a obsluhují požadavky klienta nějakým způsobem zasahují do databáze. Aby byla práce s daty příjemnější a aby se v tomto směru dala aplikace snadno rozšiřovat z jediného místa, byla vytvořena třída BaseApiController, kterou dědí všechny kontrolery a je zodpovědná za:

- inicializaci repozitářů,
- inicializaci User a Role managerů (třídy patřící pod ASP.NET Identity),
- překlad objektů do vhodné formy pro odpověď serveru.

BaseApiController dědí z ApiController, což je třída definující vlastnosti a metody API kontroleru – vytvořená třída ji jen rozšiřuje o další výše zmíněné funkcionality.

V konfugurační třídě Startup.cs je možné nadefinovat, jaké zdroje budou uvnitř kontroleru k dispozici pro každý dotaz. Samotná konfigurace vypadá následovně:

```
app.CreatePerOwinContext(AuthContext.Create);
app.CreatePerOwinContext<ApplicationUserManager>
  (ApplicationUserManager.Create);
app.CreatePerOwinContext<ApplicationRoleManager>
  (ApplicationRoleManager.Create);
```

Jak je vidět, k dispozici budou managery starající se o uživatele a jejich role a také aplikační kontext, který je třeba pro inicializaci všech dalších repozitářů. Ty je třeba v rámci aplikace inicializovat jen jednou, aby všechny dotazy přicházející na server pracovaly se stejnou verzí dat. Toho je docíleno pomocí následujícího zdrojového kódu:

```
private ICarRepository _CarRepo = null;
protected ICarRepository CarRepo
{
```

```
  get
  {
    if (_CarRepo == null) _CarRepo = new CarRepository
      (Request.GetOwinContext().Get<AuthContext>());
    return _CarRepo;
  }
}
```

Jak je vidět, kód z dotazu získá databázový kontext, pomocí kterého inicializuje repozitář. Tento nově vytvořený objekt je potom vracen při každém dalším volání CarRepo

Alternativou pro popsané řešení by mohlo být použití některého z Dependency Injection kontejnerů, jako je třeba Unity[19]. Pro kombinaci vlastních repozitářů a manager tříd převzatých z ASP.NET Identity je ale pro menší projekty, u nichž se nepředpokládají časté zásahy do architektury právě tento způsob jednodušší a přitom stoprocentně funkční.

2.1.6 Autorizace a autentizace

Jak již bylo v analytické části práce vysvětleno, autorizace a autentizace je řešena pomocí JWT tokenů. Implementovat tuto funkcionalitu především na straně serveru může být poměrně obtížné, ale výsledné řešení je elegantní a snadno použitelné. Zde je nástin procesu, kterým je třeba projít.

Prvním krokem je nadefinovat v aplikaci způsob, jakým bude autentizace provedena. To je třeba provést v konfigurační třídě Startup.cs. Zde se mimo jiné určí, na jaké URL bude možné token převzít a také jeho doba platnosti. v případě této služby je to jeden den. v následujícím kódu je ukázka, jak lze nastavení docílit.

```
public void ConfigureOAuth(IAppBuilder app)
{
  OAuthAuthorizationServerOptions OAuthServerOptions =
    new OAuthAuthorizationServerOptions()
  {
    TokenEndpointPath = new PathString("/token"),
    AccessTokenExpireTimeSpan = TimeSpan.FromDays(1),
    Provider = new SimpleAuthorizationServerProvider()
  };
  app.UseOAuthAuthorizationServer(OAuthServerOptions);
  app.UseOAuthBearerAuthentication(
    new OAuthBearerAuthenticationOptions());
}
```

Další částí, kterou je třeba naimplementovat, je třída SimpleAuthorizationServerProvider, která je zodpovědná za vytvoření autorizačního tokenu v případě úspěšného ověření uživatele. Zjednodušený kód této třídy vypadá takto:

```
public override async Task GrantResourceOwnerCredentials(
  OAuthGrantResourceOwnerCredentialsContext context)
```

```
{
  var userManager = context.OwinContext
    .GetUserManager<ApplicationUserManager>();
  User user = await userManager
    .FindAsync(context.UserName, context.Password);

  if (user == null)
  {
    context.SetError(
      "invalid_grant",
      "The␣user␣name␣or␣password␣is␣incorrect.");
    return;
  }

  ClaimsIdentity oAuthIdentity = await user
    .GenerateUserIdentityAsync(userManager, "JWT");

  var authenticationProps =
    new AuthenticationProperties(
      new Dictionary<string, string>());
  var roles = await userManager.GetRolesAsync(user.Id);
  authenticationProps.Dictionary
    .Add("superAdmin",
      roles.Contains(Roles.SuperAdmin.ToString())
      .ToString());
  authenticationProps.Dictionary
    .Add("admin",
      roles.Contains(Roles.Admin.ToString())
      .ToString());
  var ticket =
    new AuthenticationTicket(
      oAuthIdentity, authenticationProps);
  context.Validated(ticket);
}
```

V kódu si lze všimnout, že do objektu authenticationProps.Dictionary jsou přidávána dvě pole (superAdmin a admin) a odpovídající pravdivostní hodnota podle toho, jestli je tato role uživateli přiřazena, nebo nikoli. To slouží pro správné chování uživatelského rozhraní, kde má například administrátor k dispozici více ovládacích prvků v hlavní liště aplikace. JSON objekt, který při ověření server odešle, má následující tvar:

```
{
  "access\_token": "HwGvQSwPiYR5-n09oyk␣<truncated>",
  "token\_type": "bearer",
  "expires\_in": 86399,
  "superAdmin": "False",
  "admin": "False",
  ".issued": "Tue,␣04␣Apr␣2017␣19:04:48␣GMT",
  ".expires": "Wed,␣05␣Apr␣2017␣19:04:48␣GMT"
```

```
}
```

V tuto chvíli už uživatel může získat token, ze kterého je server obratem schopen vyčíst id uživatele a jeho role. Použití v praxi je pak jednoduché a intuitivní díky anotacím metod API (popřípadě celého kontroleru). Přístup k nim je řízen pomocí metod [Authorize] pro přihlášené uživatele, [Authorize(Roles = "SuperAdmin")] pro přihlášené uživatele s přiřazenou danou rolí, nebo pro případ, kdy je metoda Authorize použita pro celý kontroler, [AllowAnonymous] pro přístup i nepřihlášených uživatelů.

2.1.7 Práce se soubory

Jelikož aplikace pracuje s multimediálními soubory, které slouží jako důkaz spáchání přestupku, bylo nutné se s tímto faktem vypořádat. v první fázi dokonce vznikl kontroler zpracovávající veškerý multimediální obsah. Jako mnohem snadnější a elegantnější řešení se ale nakonec ukázalo využití souborového systému samotného serveru. Pro fungování stačí v požadavku na detail přestupku vrátit URL s adresou důkazního souboru a ve složce Data, kam jsou ukládány, povolit prohlížení zdrojů. Toto řešení je nejen snadnější na implementaci, ale zároveň mnohem efektivnější Pro přijetí souboru spolu s dalšími daty bylo nutné pracovat s požadavkem typu multipart, který na pozadí odešle formu ve více částech. Jak se s takovým požadavkem pracuje na straně serveru ukazuje následující ukázka kódu.

```
[Authorize]
[HttpPost]
[Route("violations")]
public async Task<IHttpActionResult> AddViolation()
{
  if (!Request.Content.IsMimeMultipartContent())
    throw new HttpResponseException(
      HttpStatusCode.UnsupportedMediaType);

  var fileToSave = HttpContext.Current.Request.Files[0];
  var form = HttpContext.Current.Request.Form;
}
```

2.2 Implementace klientské části

Stejně jako kapitola o implementaci serverové části, i tato si klade za cíl především popsat princip fungování zvolených technologií a také nastínění možných řešení problémů, na které bylo během vývoje aplikace naraženo.

2.2.1 Použité nástroje

I při vývoji klientské části aplikace nebylo možné efektivně pracovat bez vhodných programů. Zde je několik z nich seřazených podle důležitosti:

- Visual Studio Code[20] – IDE vynikající svou přehledností, jednoduchostí a také stabilitou,
- GIT[21] – nástroj pro správu verzí,
- Google Chrome[22] – webový prohlížeč vybavený kartou pro vývojáře usnadňující debugging,

2.2.2 Začátek vývoje

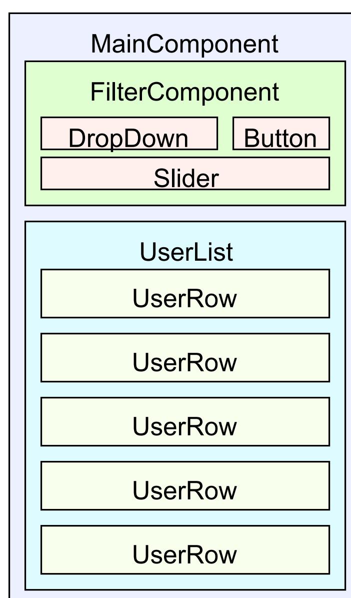
Při startu nového projektu má vývojář na výběr dvě možnosti. Buď začít úplně od začátku sám, nebo využít předpřipravených ukázkových aplikací, které velmi razantně snižují čas potřebný pro prvotní nastavení. Krom toho obsahují mnoho užitečných nástrojů pomáhajících při vývoji. V tomto případě byla zvolena druhá varianta, tedy použití startovacího kitu React Slingshot[23], který se podle hodnocení na GitHubu[24] řadí mezi ty nejpopulárnější. Ze zmíněných již v základní verzi obsažených funkcí a nástrojů je dobré zmínit například:

- hot reloading – automatická projekce uložených změn kódu bez nutnosti znovunačítání stránky nebo dokonce rekompilace zdrojového kódu,
- hotové npm skripty – skripty umožňující start, vyčištění aplikace, nebo dokonce nasazení pomocí jediného příkazu,
- babel[25] – nástroj kompilující novější verze javascriptu, které nemusí být ještě podporovány všemi prohlížeči do starší verze JS, která má podporu plošnou,
- webpack[26] – nástroj, který zjednodušeně transformuje mnoho souborů do několika málo, které jsou staženy klientem,
- eSLint[27] – nástroj pro udržení čistoty kódu.

React-Slingshot ale kromě toho všeho obsahuje i malou ukázkovou aplikaci, která dobře slouží jako základní kostra projektu. Ta se po prvotní modifikaci rozšiřovala opravdu snadno a rychle.

2.2.3 React

Knihovna React, která byla zvolena jako technologie vhodná pro implementaci klientské části, nabízí poměrně inovativní způsob, jak vytvořit uživatelské rozhraní pomocí deklarativního zápisu.



Obrázek 2.1: React komponenty

Základním pojmem při programování v Reactu je slovo komponenta. Jedná se v podstatě o funkci/třídu, která převezme na vstupu libovolná data a vrátí objekt reprezentující HTML uzel. Nejedná se ale o opravdový HTML element. React totiž pracuje tak, že vytvoří virtuální DOM, který je při jakékoliv změně porovnán s reálným DOMem a jen uzly, které byly afektovány, budou přerenderovány. Tímto přístupem je dosaženo vysoké efektivity, protože pokud jde o vykreslování internetových stránek v prohlížečích, manipulace s DOMem je kritickým místem.

Jak si komponentu představit a jak se dají skládat ilustruje obrázek 2.1 zobrazující ukázkovou mikro aplikaci umožňující vypsání uživatelů a navíc vybavenou ovládacím prvkem na jejich filtrování. Další silnou stránkou je již zmíněná deklarativnost – tedy programování stylem "jak to bude vypadat". Té je dosaženo především pomocí syntaxe JSX, s kterou se dá React element vytvořit například následovně:

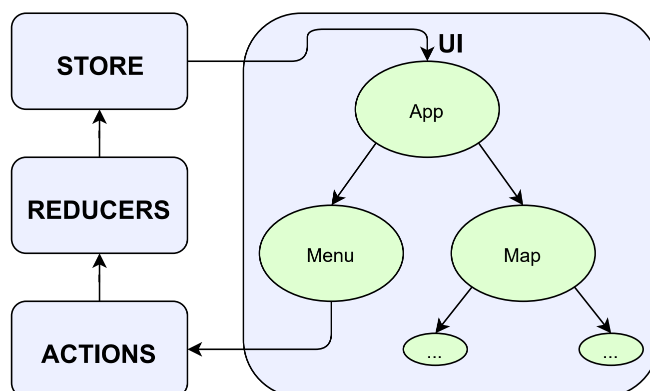
```
const element = <h1>Ahoj svete!</h1>;
```

Celá ukázková komponenta s využitím JSX a ES6 pak vypadá například takto:

```
const Greetings =  ({name}) => (<h1>Ahoj {name}!</h1>);
```

A její volání následovně:

```
<Greetings name="Jan" />;
```



Obrázek 2.2: Redux data flow

2.2.4 Redux

React sice umí vykreslit uživateli stránku, ale na další věci, které jsou pro komplexní aplikaci potřeba, je krátký. Jedná se totiž "pouze"o knihovnu a ne o celý framework (jako je tomu například v případě Angularu). Je tudíž třeba sáhnout po další technologii, která by se starala o stav programu a tok dat. Těch je hned několik, ale pro tuto aplikaci bylo rozhodnuto pro tu nejrozšířenější, kterou je samotným Facebookem podporovaný Redux. Alternativou by mohl být o něco starší Flux, nebo například MobX.

Princip Reduxu je poměrně jednoduchý. Stav aplikace se dá představit jako klasický javascriptový objekt, z kterého React komponenty získávají data a zobrazují relevantní výsledky. Takový stav ale není stálý a je třeba do něj často zasahovat. a právě to má na starosti Redux. Věc je komplikovanější o to, že stav je v pojetí Reduxu imutabilní. Není tedy možné jej měnit, ale jen vytvářet na základě starých informací nové objekty. Jak celý proces změny stavu probíhá nejlépe ilustruje přiložený diagram 2.2.

V datovém toku figurují následující objekty.

1. Store – Objekt uchovávající stav aplikace.
2. Komponenta – zobrazuje data ze Storu a vyvolává akci.
3. Akce – funkce, která je vyvolaná komponentou. Má povinný typ a můhou jí být předána libovolná data.
4. Reducer – Objekt zodpovědný za změnu stavu. Obsluhuje část stavu a může reagovat na libovolný typ události.

V celém procesu je nejdůležitější a nejkomplikovanější práce s Reducery. Musí se jednat o takzvané čisté (pure) funkce, které by při volání se stejnými parametry nad stejným objektem měly vždy vést ke stejnému výsledku bez

jakéhokoliv vedlejšího efektu. Dále je nutné mít na paměti, že stav je imutabilní – nelze tedy měnit, ale jen vytvořit nový. Oba tyto na první pohled nenápadné principy byly převzaté z funkcionálního programování a mají dalekosáhlé důsledky. Tím nejoceňovanějším je fakt, že v jakémkoliv bodě běhu aplikace může programátor zjistit, v jakém je stavu a do jakého stavu směřuje. Dokonce se dá "vracet"v čase, což vede ke snadnému debugování i v případě velmi komplexních aplikací.

2.2.5 Použité moduly

Jelikož správně navržená komponenta je znovupoužitelná, nebrání programátorům nic ve sdílení svých výsledků práce. i díky tomu bylo využito mnoha npm modulů, které mnohdy dramaticky snížily obtížnost řešených problémů. Některé z nich byly obsažené již ve startovacím balíčku React-Slingshot, jiné, ty specifičtější, bylo nutné dohledat na internetu. Zde je výčet několika málo z nich, které při implementaci sehrály nejdůležitější roli:

- react-redux – sada funkcí sloužících pro propojení Reactu s Reduxem,
- react-router – navigační komponenta sloužící pro přecho mezi podstránkami,
- react-google-maps – komponenta zobrazující klasické rozhraní Google maps. Jednou z používaných komponent je i Marker Clusterer, díky kterému je možné dosáhnout větší přehlednosti díky shlukování několika značek do jedné,
- redux-form – modul usnadňující práci s formuláři,
- redux-api-middleware – modul usnadňující volání API rozhraní,
- material-ui – sada grafických komponent navržených na základě material designu.

Všechny výše vypsané balíčky jsou snadno dohledatelné pomocí manageru balíčků npm[28].

2.2.6 Automatické přihlašování

Jelikož HTTP je bezestavový protokol, bylo by teoreticky třeba, aby klient pro každou akci vyžadující autentizaci přikládal své uživatelské jméno a heslo. To by bylo samozřejmě velmi nepraktické a naštěstí existuje řešení popsané v kapitole bezpečnost 1.7. Posledním nevyřešeným problémem je uložení JWT tokenu na straně klienta. v tomto řešení bylo využito objektu LocalStorage, což je místo spravované webovým prohlížečem, kam mohou být ukládána data. Samotné uložení je potom opravdu jednoduché:

```
localStorage
  .setItem('jwtToken', action.payload.access_token);
```

Problém nastává při znovuotevření prohlížeče, kdy aplikace musí zjistit současný stav přihlášení. Proces probíhá následovně:

1. Klient zkontroluje, jestli je v LocalStorage uložený JWT token.
2. Pokud ne, uživatel je nepřihlášený.
3. Pokud ano, odešle se dotaz na server, zda je token validní (existuje speciální metoda API, která vrací mimo jiné i informaci o přiřazených rolích(viz 2.1.6)).
4. Pokud ne, uživatel je nepřihlášený.
5. Pokud ano, uživatel je přihlášený.

Pro odhlášení uživatele stačí z lokálního úložiště daný jwt token smazat. To se provede následovně.

```
localStorage.removeItem('jwtToken');
```

Tato operace je provedena bez serverového volání.

2.2.7 Práce s Google maps

Existuje sice celá řada veřejných React komponent, které dokáží pracovat s Google mapami, ale jen málokterá z nich zvládá například shlukování bodů na mapě tak, jak je vidět na obrázku 3.1. Další funkcí, která sehrála při výbru nejvhodnějšího řešení roli, byla geolokace – tedy vycentrování mapy v bodě, kde se uživatel zrovna nachází. Nakonec bylo rozhodnuto pro využití velice snadno ovladatelného modulu react-google-maps obsahujícího mimo jiné i třídu MarkerClusterer schopnou právě zmíněného shlukování. Tento modul je následně obalen komponentou react-localization schopnou získat a předat GPS souřadnice přistupujícího uživatele. Výsledná struktura dané části aplikace je následující:

```
<GoogleApiWrapper>
  <GoogleMap>
    <MarkerClusterer/>
  </GoogleMap>
</GoogleApiWrapper>
```

GoogleApiWrapper je komponenta, která je při načítání stránky zodpovědná za stažení skriptů potřebných pro správný běh mapy a za identifikaci této aplikace pomocí API key, což je textový řetězec získaný na stránkách společnosti Google. Pokud by došlo k překročení limitu denních požadavků, bylo by nutné přistoupit na placený program spolupráce.

2.2.8 Responzivní design

Responzivní design je vyřešen pomocí Media queries, které se staly standardem v CSS3. Jedná se o možnost vytváření podmínek uvnitř CSS souborů založených na velikosti zařízení. v případě takto jednoduché aplikace se často vyplatí nespoléhat na knihovny třetích stran, ale implementovat vlastní řešení. Jak se s MQ pracuje odhaluje následující ukázka kódu.

```
@media only screen and (min-width: 1001px) {
  .main{
    width: 60vw;
    float: right;
  }
  .minor{
    width: 40vw
    float: left;
  }
}
@media only screen and (max-width: 1000px) {
  .main{
    width: 100vw;
  }
  .minor{
    width: 100vw;
  }
}
```

2.3 Závěr

Během implementace obou částí programu byla snaha se nanejvýš držet základních programovacích principů a pravidel, aby bylo dosaženo nejen funkčního programu, ale zároveň aplikace připravené pro budoucí rozšíření. Zásadní bylo přehledné členění souborů, snaha o co nejvíce sebepopisný kód a v neposlední řadě také použití návrhových vzorů.

Kapitola 3

Testování a nasazení

Testování je důležitým procesem při vývoji softwaru nejenom proto, že díky němu může dodavatel najít vzniklé chyby, ale také proto, že lze díky němu zafixovat funkční stav aplikace. Pokud jsou testy správně navržené a implementované, případné budoucí rozšíření nepředstavuje takovou hrozbu. Po dokončení vývoje stačí spustit staré testy, které pokud projdou, měl by být program v pořádku. v tomto případě, vzhledem k rozličnosti vybraných technologií, byla testována serverová a klientská část zvlášť.

3.1 Testování serverové části

Testování serveru byl poměrně náročný proces, který odhalil nejen několik menších chyb, ale také poukázal jak na drobné slabiny návrhu, tak naopak na jeho silné stránky. Je to tím, že ačkoli programátoři často slýchají řadu zásad, jak psát kód správně, tak ne vždy je hned vidět smysl takového řešení. Testování ale vrhá světlo právě na takové situace a objasňuje přístupy, které jsou považovány za ty správné. Myšlen je především správný návrh architektury systému včetně použití návrhových vzorů.

3.1.1 Testování zabezpečení

Zabezpečení API je vyřešeno pomocí atributu Authorize. Jedná se o část MVC frameworku, který je otestován samotnými vývojáři a není důvod tomuto řešení nevěřit. Proces ověření uživatele probíhá tak, že program nejdříve otestuje identitu a až v případě, že ji potvrdí, vyvolá samotnou metodu. z tohoto důvodu není možné volat tyto metody napřímo z kódu Unit testu, protože pak atribut vůbec není použit. Další možností je vytvářet přímo HTTP dotazy a službu volat zvenčí. Tímto způsobem, pomocí nastroje Insomnia, bylo postupováno i v tomto případě. Byly vytvořeni tři uživatelé s různými rolemi a s jim přiřazenými tokeny pak byly volány ukázkové metody tak, aby povolovaly:

- přístup i anonymním uživatelům,
- přístup autentizovaných uživatelů,
- přístup uživatelům s rolí Admin, nebo SuperAdmin,
- přístup uživatelům v roli SuperAdmin.

Testování dopadlo přesně tak, jak bylo předpokládáno a nevedlo k odhalení žádných chyb.

3.1.2 Testování funkcionalit

Funkcionality programu byly testovány po větších celcích – vždy jako testování jedné API metody. Kontrolery silně závisí na několika vnějších objektech, ke kterým přistupují a z kterých berou data. Jedním z nich je databázový kontext dané aplikace, dále to jsou třídy zodpovědné za management uživatelů a rolí a poslední zdroj dat jsou samotné objekty reprezentující HTTP požadavky. Princip takového testování je jednoduchý – stačí třídě podstrčit falešné objekty chovající se stejně, jako ty pravé, a po každé vyvolané metodě kontrolovat, jestli se data změnila přesně tak, jak bylo očekávané. Vytváření takových napodobenin se nazývá mockování a bez patřičných knihoven se jedná o netriviální problém. v případě této aplikace byla vybrána knihovna Moq[29], která se v .NET komunitě těší vysoké popularitě. Samotné používání je již poměrně intuitivní – pro jednotlivé metody nebo vlastnosti objektu se jasně deklaruje, co na takové volání objekt vrátí. Princip je zřejmý především z ukázek, které budou použity v následujících kapitolách.

3.1.2.1 Mockování databáze

Vytvářet testy nad produkční databází by rozhodně nebylo vhodné. z tohoto důvodu je možné použít buď další databázi s testovacími daty, nebo implementovat objekt napodobující chování reálné databáze, který by ovšem existoval jen v paměti. Druhé řešení řešení je praktičtější a elegantnější. s knihovnou Moq pak není složité postup realizovat. Databáze samotná je jeden objekt, obsahující několik tříd typu DbSet<T>, které je také nutné mockovat. Práci jsem zahájil právě vytvořením falešných DbSetů. Úkol byl o to komplikovanější, že se databázové kolekce chovají jinak než kolekce v paměti. Pro jejich vytvoření byla naimplementována následující generická metoda, která z jakéhokoliv listu vytvoří objekt vlastnostmi simulující právě DbSet.

```
private static Mock<DbSet<T>> GetQueryableMockDbSet<T>
        (List<T> sourceList) where T : class
{
  var queryable = sourceList.AsQueryable();
```

```
  var dbSet = new Mock<DbSet<T>>();
  dbSet.As<IQueryable<T>>().Setup(m => m.Provider)
        .Returns(queryable.Provider);
  dbSet.As<IQueryable<T>>().Setup(m => m.Expression)
        .Returns(queryable.Expression);
  dbSet.As<IQueryable<T>>().Setup(m => m.ElementType)
        .Returns(queryable.ElementType);
  dbSet.As<IQueryable<T>>()
        .Setup(m => m.GetEnumerator())
        .Returns(() => queryable.GetEnumerator());
  dbSet.Setup(d => d.Add(It.IsAny<T>()))
        .Callback<T>((s) => sourceList.Add(s));
  dbSet.Setup(x => x.Remove(It.IsAny<T>()))
        .Returns<T>(x => { if (sourceList.Remove(x))
               return x;
               else return null;
        });
  return dbSet;
}
```

Pro vytvoření celého falešného databázového kontextu, který by následně mohl být použit v konstruktoru kontrolerů, byla vytvořena metoda GetContext. Zajímavé jsou především poslední tři řádky, které vycházejí z implementace návrhového vzoru Repository, kde se pro přístup k jednotlivým kolekcím databáze používá volání Context.Set<T>. Kód třídy je možné vidět v následující ukázce.

```
private AuthContext GetContext
        (List<Car> cars, List<User> users,
        List<TrafficViolation> trafficViolations)
{
  var mockContext = new Mock<AuthContext>();
  var carSet = GetQueryableMockDbSet(cars);
  var userSet = GetQueryableMockDbSet(users);
  var trafficViolationSet =
        GetQueryableMockDbSet(trafficViolations);

  mockContext.Setup(c => c.Set<TrafficViolation>())
        .Returns(trafficViolationSet.Object);
  mockContext.Setup(c => c.Set<Car>())
        .Returns(carSet.Object);
  mockContext.Setup(c => c.Set<User>())
        .Returns(userSet.Object);

  return mockContext.Object;
}
```

3.1.2.2 Mockování user/role managerů

V metodách kontrolerů je často manipulováno se záznamy uživatelů a jejich rolí. Ty obsluhují speciální Manager třídy, jejichž chování bylo třeba simulovat. Proces mockování byl podobný jako ve výše zmíněném případu s databází. Pro metody manageru, které byly uvnitř třídy volány, bylo nutné definovat, co bude vráceno. Jako ukázka může posloužit asynchronní metoda FindByIdAsync, která v tomto případě vracela vždy podstrčeného ukázkového uživatele. Bylo by možné implementovat i opravdové hledání v seznamu na základě Id, ale pro tyto účely to nebylo nutné už proto, že se jedná o funkci důvěryhodné třetí strany.

```
var context =
        GetContext(_cars, _users, _trafficViolations);
var userStoreMock = new Mock<UserStore<User>>();
userStoreMock
        .Setup(x => x.FindByIdAsync(It.IsAny<string>()))
        .ReturnsAsync(() => expectedUser);

var controller = new MainController
        (context,
         new RoleStore<IdentityRole>(context),
         userStoreMock.Object);
```

3.1.2.3 Vytváření falešného kontextu kontroleru

Poslední zdroj dat, ke kterým metody kontroleru přistupují, je jeho samotný kontext. Ten se v běžném scénáři plní při volání serveru a průchodu požadavku skrz jeho jednotlivé vrstvy. Při volání metod z kódu testu ale k ničemu takovému nedochází a je proto třeba data dodat jiným způsobem. v této aplikaci na těchto datech závisí jediná, ovšem hojně využívaná metoda, kterou je:

```
var userId = User.Identity.GetUserId();
```

Řešení tohoto problému bylo nakonec jednoduché. Při vytváření nového objektu testovaného kontroleru je totiž možné objekt User přímo definovat. Podstatná část kódu potom vypadá následovně:

```
var context = GetContext(_cars, _users, _trafficViolations);
var controller =
        new MainController(
                context,
                new RoleStore<IdentityRole>(context),
                new Mock<UserStore<User>>())
{
  User = UserInitializer("testId"),
};
```

A metoda vytvářející Mock objektu User vracející stejné id, jaké dostane jako parametr, vypadá takto:

```
private IPrincipal UserInitializer(string id)
{
  var claim = new Claim("test", id);
  var mockIdentity =
    Mock.Of<ClaimsIdentity>
        (ci => ci.FindFirst(It.IsAny<string>()) == claim);
  return Mock.Of<IPrincipal>
    (ip => ip.Identity == mockIdentity);
}
```

3.2 Testování klientské části

Pokud jde o testování React aplikace, může jít o několik okruhů. Dají se:

- testovat komponenty,
- testovat akce,
- testovat reducery,
- testovat middleware.

V případě této aplikace je ideální zaměřit se především na akce a reducery. Správná funkčnost komponent se zpravidla dá otestovat již na základě používání programu během vývoje. Navíc se nejedná o náročný program se složitým uživatelským rozhraním, kde by mohlo testování komponent sehrát stěžejní roli. Vlastní middleware pak vůbec není použit.

Jako testovací nástroj je používána knihovna Jest[30]. Ta je velice rozšířená a patří v React komunitě mezi vůbec ty nejoblíbenější. Krom toho je nativně podporována již samotným startovacím kitem React Slingshot. Testovací soubory jsou rozpoznány pomocí přípon test.js, nebo .spec.js a spouští se hromadně pomocí skriptu.

3.2.1 Testování reducerů

Jak již bylo řečeno v části práce věnující se implementaci, stav je imutabilní a neměl by proto být nikdy měněn. Zda k tomu nikde v aplikaci nedošlo by mělo být rovněž předmětem testů. Existuje ovšem snadnější řešení, které toto pravidlo hlídá již během vývoje. Je jím použití modulu reduxImmutableStateInvariant při vytváření React storu aplikace. Pokud detekuje nepovolenou změnu stavu, vypíše do chybové konzole hlášku informující o takové události. Navíc je snadné tento modul použít jen ve vývojovém prostředí, kde je kontrola potřeba.

Dalším předmětem zkoumání testů je již samotný výstup funkce reduceru v závislosti na vstupu. Jedná se o poměrně jednoduchý princip, který demonstruje následující ukázka kódu:

```
it('should␣handle␣FETCH_LICENSEPLATES_SUCCESS',
  () => {
  expect(
    reducer([], {
    type: types.FETCH_LICENSEPLATES_SUCCESS,
    payload: ['ABC', 'DEF']
    })
  ).toEqual(
    {
    availableLicensePlates: ['ABC', 'DEF']
    }
  )
})
```

Je jasně patrné, že vybranému reduceru předáme na vstup typ akce (který je povinný) a pak libovolné další objekty tak, jak by to při běhu programu skutečně udělala některá z vyvolaných akcí. Výstup reduceru, tedy nová část stavu aplikace nebo stav celý, je porovnán s očekávaným výstupem. Zde je nutné zahrnout pouze vlastnosti objektu, které byly změněny. Tímto způsobem je poměrně snadné pokrýt buď všechny větve reduceru, nebo jen kritická místa.

3.2.2 Testování akcí

Při testování akcí existují dva možné scénáře, kterými jsou synchronní a asynchronní funkce. První možnost je na implementaci o něco málo jednodušší, protože v druhém případě je nutné simulovat opožděnou odpověď serveru. Vzhledem k tomu, že všechny akce v aplikaci jsou triviální a fungují jen jako místo pro předání parametrů a definování správného typu vyvolané akce, nebyly žádné automatizované testy pro akce napsány. Nadefinování špatného typu funkce, nebo špatná práce s parametry je totiž chyba, kterou je snadné si ohlídat již během vývoje, jelikož je okamžitě vidět a zpravidla je i snadno opravitelná. Kritické místo je v tomto případě funkčnost reducerů, kterým byla věnována o to větší pozornost.

3.3 Nasazení

Poslední fází vývoje je samotné nasazení. v případě statických stránek, nebo nenáročných webových aplikací je často možné zajistit hostování zdarma. Jinak je tomu v případě tohoto programu, kde dochází ke sběru dat a tudíž ke zvýšeným nárokům na systémové úložiště. Vzhledem k tomu, že dosud nebylo rozhodnuto o využití konkrétní služby, nebyla aplikace zatím nasazena a veškeré testování probíhalo jen na lokálním pracovišti.

Pro možnost otestování aplikace i dalšími uživateli bylo naimplementováno falešné API pomocí modulu json-server. Tím vzniká možnost spustit program

i na počítačích bez nainstalovaného IIS serveru a MSSQL serveru. Toto řešení je vhodné i pro testování během vývoje. Funkcionalita rozhraní je ovšem značně omezená – například přihlášení proběhne vždy úspěšně a testovací uživatel má přiřazené všechny role.

3.3.1 Výhled do budoucna

Přestože je aplikace hotová, nejedná se o konečný stav a do budoucna je v plánu řada vylepšení. Jsou to jak nové požadavky, tak vylepšování stávajících funkcionalit. v tuto chvíli by mohlo velkou roli hrát rozšíření do dalších světových jazyků – teď totiž aplikace podporuje pouze češtinu a angličtinu. Další oblastní, na kterou by bylo vhodné se v budoucnu zaměřit, je libovolná forma propagace, aby se možnost zaznamenávání dopravních přestupků dostala do podvědomí řidičů a dalších účastníků silničního provozu. První dojem na aplikaci je možné vytvořit si na základě snímku obrazovky hlavní stránky, který je na obrázku 3.1

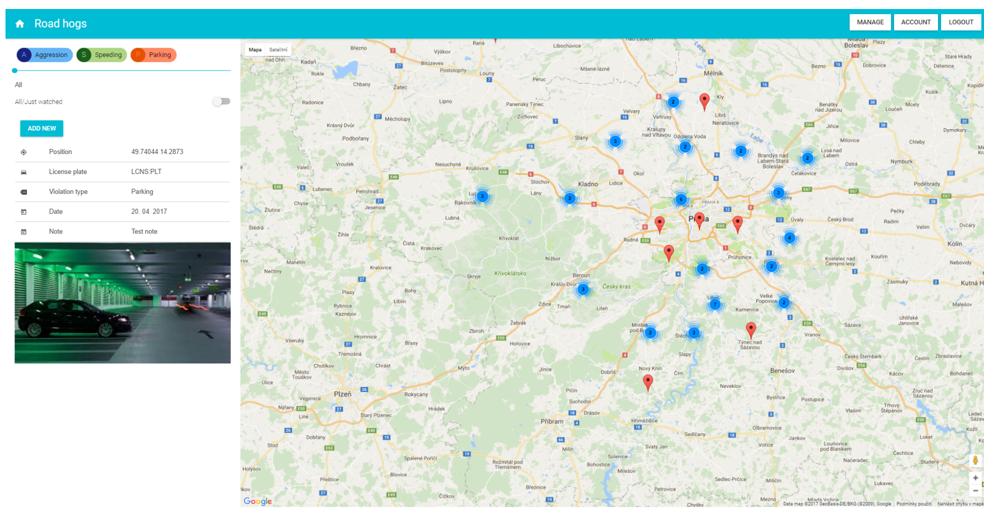
Další kapitolou k zamyšlení je potom nápad, jak aplikaci alespoň trochu komerčně zhodnotit. Je sice třeba zmínit, že primárním cílem není vydělávat, ale pokrytí nákladů na provoz by se dalo považovat za velký úspěch umožňující stát službě na pevných základech. Dosáhnou by toho šlo například využitím plochy detailu dopravního přestupku, která zůstává prázdná až do chvíle, kdy uživatel některý nevybere. Tímto způsobem by se dalo dosáhnout funkční reklamy schopné pokrýt náklady a zároveň nevytvářet zvlášť rušivý dojem – reklama by se znovu objevila až při další návštěvě stránky.

Závěr

Cílem této bakalářské práce bylo vytvořit uživatelskou databázi dopravních přestupků, pomocí které by měli účastníci provozu na pozemních komunikacích možnost trvale zaznamenat zachycená porušení dopravních předpisů, kterým byli svědky. Po průzkumu současných řešení byla vypracována důkladná analýza zachycující co nejvíce aspektů budoucího vývoje aplikace. Návrh vycházel především z funkčních a nefunkčních požadavků diskutovaných se zadavatelem.

Na základě tohoto návrhu byl následně vytvořen funkční prototyp pomocí .NET frameworku pro serverovou část a knihovny React.js pro klientskou část. Mezi hlavní výhody .NETu patří objektově orientovaný přístup, široká a ověřená základna technologií včetně ASP.NET pro vytváření webových aplikací a v neposlední řadě spousta silných nástrojů pro snadný vývoj. Nejzásadnějšími výhodami Reactu je potom moderní přístup k problematice, deklarativní zápis a také rychlost a efektivita celého řešení.

Po implementační fázi přišlo na řadu testování, které vedlo k nalezení a opravení několika málo drobných chyb a tím pádem i k finálnímu doladění aplikace. Na konci tohoto procesu již bylo možné mluvit o programu připraveném na nasazení do reálného provozu s reálnými daty. Posledním krokem tak zůstává vybrání vhodného poskytovatele hostování.



Obrázek 3.1: Snímek obrazovky aplikace

Literatura

[1] Google Inc. [online], [cit 16-03-2017]. Dostupné z: http://www.google.com

[2] Google Play. [online], [cit 16-03-2017]. Dostupné z: https://play.google.com/store

[3] App store. [online], [cit 16-03-2017]. Dostupné z: https://www.apple.com/itunes/charts/

[4] Material design. [online], [cit 16-03-2017]. Dostupné z: https://material.io/

[5] Material-UI. [online], [cit 16-03-2017]. Dostupné z: http://www.material-ui.com/

[6] Google Maps APIs. [online], [cit 16-03-2017]. Dostupné z: https://developers.google.com/maps/

[7] Microsoft. [online], [cit 16-03-2017]. Dostupné z: http://www.material-ui.com/

[8] Entity Framework. [online], [cit 16-03-2017]. Dostupné z: https://msdn.microsoft.com/en-us/library/aa937723(v=vs.113).aspx

[9] Microsoft SQL Server. [online], [cit 16-03-2017]. Dostupné z: https://www.microsoft.com/cs-cz/sql-server/sql-server-2016

[10] React. [online], [cit 16-03-2017]. Dostupné z: https://msdn.microsoft.com/en-us/library/aa937723(v=vs.113).aspx

[11] AngularJs. [online], [cit 16-03-2017]. Dostupné z: http://www.material-ui.com/

[12] Vue.js. [online], [cit 16-03-2017]. Dostupné z: https://www.microsoft.com/cs-cz/sql-server/sql-server-2016

[13] Facebook. [online], [cit 16-03-2017]. Dostupné z: https://www.facebook.com/

[14] Visual Studio. [online], [cit 16-03-2017]. Dostupné z: https://www.visualstudio.com/

[15] SQL Management Studio. [online], [cit 16-03-2017]. Dostupné z: https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms

[16] Insomnia. [online], [cit 16-03-2017]. Dostupné z: https://insomnia.rest/

[17] TFS. [online], [cit 16-03-2017]. Dostupné z: https://www.visualstudio.com/cs/tfs/

[18] HUNT, A.; THOMAS, D.: *Programátor pragmatik: jak se stát lepším programátorem a vytvářet kvalitní software*. Brno: Computer Press, první vydání, 2007, ISBN 978-80-251-1660-9.

[19] Unity. [online], [cit 16-03-2017]. Dostupné z: https://msdn.microsoft.com/en-us/library/ff647202.aspx

[20] Visual Studio Code. [online], [cit 16-03-2017]. Dostupné z: https://code.visualstudio.com/

[21] GIT. [online], [cit 16-03-2017]. Dostupné z: https://git-scm.com/

[22] Google Chrome. [online], [cit 16-03-2017]. Dostupné z: https://www.google.com/chrome/

[23] React-Slingshot. [online], [cit 16-03-2017]. Dostupné z: https://github.com/coryhouse/react-slingshot

[24] GitHub. [online], [cit 16-03-2017]. Dostupné z: https://github.com/

[25] Babel. [online], [cit 16-03-2017]. Dostupné z: https://babeljs.io/

[26] Webpack. [online], [cit 16-03-2017]. Dostupné z: https://webpack.github.io/

[27] ESLint. [online], [cit 16-03-2017]. Dostupné z: http://eslint.org/

[28] npm. [online], [cit 16-03-2017]. Dostupné z: https://www.npmjs.com/

[29] Moq. [online], [cit 16-03-2017]. Dostupné z: https://github.com/Moq/moq4/wiki/Quickstart

[30] Jest. [online], [cit 16-03-2017]. Dostupné z: https://facebook.github.io/jest/

[31] Node. [online], [cit 16-03-2017]. Dostupné z: https://nodejs.org/en/download/

Příloha A

Seznam použitých zkratek

JSON JavaScript Object Notation

CSV Comma-separated values

UI User interface

API Application Programming Interface

SPZ Státní poznávací značka

JWT JSON Web Token

EF Entity Framework

MSSQL Microsoft SQL

UX User Experience

REST Representational State Transfer

TFS Team Foundation Server

SQL Structured Query Language

DRY Don't repeat yourself

CRUD Create Read Update Delete

IDE Integrated Development Environment

JS JavaScript

CSS Cascading Style Sheets

MQ Media Queries

HTTP Hypertext Transfer Protocol

Příloha B

Manuál na spuštění aplikace

Pro spuštění aplikace v testovacím režimu změněném v kapitole 3.3 je nutné nejprve nainstalovat JavaScriptový engine Node [31]. Ten již v základní verzi obsahuje systém npm, pomocí kterého je možné uvést program do chodu. Postup je pro všechny systémy následující:

- Stáhnout oficiální instalační balíček.
- Nainstalovat Node podle návodu. v jednom z kroků je nutné zaškrtnout instalaci spolu s npm systémem.
- Stáhnout poslední verzi npm pomocí příkazu **npm install npm@latest -g** .
- Přejít v konzoli číslo 1 do adresáře src/client/dist.
- Spustit API pomocí příkazu npm run api.
- Přejít v konzoli číslo 2 do adresáře src/client/dist.
- Spustit klientskou část aplikace pomocí příkazu **npm start -s** .

Po spuštění posledního příkazu by se po krátké chvíli mělo otevřít okno prohlížeče s testovací verzí aplikace. v té je uživatel automaticky přihlášený (ale je možné se odhlásit a opět pomocí libovolné kombinace jména a hesla přihlásit) a poté program procházet. Data ovšem nejde měnit, nefunguje filtrování dat a například detail přestupku je vždy jeden a ten samý.

Příloha C

Obsah přiloženého CD

```
readme.txt....................stručný popis obsahu CD
src
  impl
    client..............zdrojové kódy implementace klienta
    server..............zdrojové kódy implementace serveru
  thesis................zdrojová forma práce ve formátu LaTeX
text............................text práce
  thesis.pdf............text práce ve formátu PDF
```