UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DIPLOMOVÁ PRÁCE

2017 Jan Matička

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Návrh a implementace systému pro efektivní plánování nákladní dopravy (Dispečerská aplikace)

Jan Matička

Diplomová práce

2017

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 16. 05. 2017

podpis autora

Jan Matička

PODĚKOVÁNÍ

Rád bych tímto poděkoval svému vedoucímu diplomové práce Ing. Janu Fikejzovi Ph.D. za jeho podnětné rady, postřehy a čas, který mi věnoval při plánování postupných kroků a společném řešení vzniklých problémů, které vedli k úspěšnému dokončení práce.

Dále bych rád poděkoval Mgr. Ivě Holubcové a Mgr. Zdence Modráčkové za pomoc při provádění jazykových korektur v diplomové práci.

ANOTACE

Cílem diplomové práce je návrh a implementace dispečerské aplikace. V úvodní části práce je nastíněna situace nákladní dopravy se zaměřením na interní procesy v malých a středních firmách, zejména pak plánování denních výkonů řidičů a následnou komunikaci mezi dispečery a řidiči. Praktická část práce se věnuje vytvoření dispečerské aplikace, která problém plánování řeší.

KLÍČOVÁ SLOVA

Dispečerská aplikace, plánování, silniční nákladní doprava, ASP.NET Core, AngularJS

TITLE

Design and implementation of a system for the effective planning of the freight transport (Dispatching application)

ANNOTATION

The aim of the diploma thesis is the design and implementation of a dispatcher application. The theoretical part of the diploma thesis presents a situation of freight transport focusing on an internal processes in small and medium-sized companies especially on planning of daily drivers' performance and following communication between dispatchers and drivers.The practical part of the thesis is devoted to creation of the dispatcher application aiming on solving the problem of planning daily drivers' performance.

KEYWORDS

Dispatcher application, planning, road freight transport, ASP.NET Core, AngularJS

OBSAH

Úvod....14
1 Současné trendy v nákladní Dopravě....16
1.1 Vývoj vnitrostátní nákladní dopravy v ČR....16
1.2 Nákladní silniční doprava....17
1.3 Procesy plánování v malých a středních firmách....17
1.4 Informační technologie v silniční nákladní dopravě....19
2 Použité technologie....21
2.1 .NET Core....21
2.1.1 .NET Standard....22
2.2 ASP.NET Core....24
2.2.1 ASP.NET Core MVC....24
2.2.2 ASP.NET Core a podpora prezentačních nástrojů....25
2.3 AngularJS....25
2.3.1 Data Binding....26
2.3.2 Controllery....27
2.3.3 Angular Route....28
2.4 Bootstrap....28
2.5 Syncfusion....29
3 návrh a Implementace dispečerské aplikace....30
3.1 Vrstvení aplikace....30
3.2 Aplikační vrstva....31
3.2.1 Autentizace a autorizace dispečerské aplikace....32
3.2.2 Aplikační rozhraní pro prezentační vrstvu....34
3.2.3 Mail Service....38
3.2.4 Klíčové NuGet balíčky....39
3.2.5 AutoMapper a ViewModely....40

3.3 Prezentační vrstva 41
3.3.1 Klíčové bower balíčky 41
3.3.2 Angular moduly 42
3.3.3 Modul jízdy 44
3.3.4 Administrátorské moduly 49
3.4 Shrnutí a využití dispečerské aplikace 51
ZÁVĚR 53
Zdroje 55
Přílohy 57

SEZNAM OBRÁZKŮ

Obrázek 1- Struktura nákladní dopravy v ČR v roce 2013 [mil. tkm, %] 16
Obrázek 2 - Dispečerský deník 18
Obrázek 3 - Průmysl 4.0 19
Obrázek 4 - Struktura .NET Core 22
Obrázek 5 - knihovna .NET Standard...23
Obrázek 6 - Struktura MVC..24
Obrázek 7 - AngularJS workflow 26
Obrázek 8 - Obousměrný data binding 27
Obrázek 9 - Navigace pomocí Angular Route 28
Obrázek 10 - Schéma aplikace pro efektivní plánování nákladní dopravy 31
Obrázek 11 - Editor bower balíčků 41
Obrázek 12 - Modul jízdy, dispečerský pohled 42
Obrázek 13 - Chybí závislost pro zobrazení alertu 44
Obrázek 14 - Přehled uživatelských účtů 49

SEZNAM TABULEK

Tabulka 1 - Vnitrostátní nákladní silniční doprava (pouze vozidly registrovanými v ČR)......17
Tabulka 2 - Kompatibilita .NET Standard...23

SEZNAM ZDROJOVÝCH KÓDŮ

Zdrojový kód 1 - nastavení ASP.NET Identity 32
Zdrojový kód 2 - akce Login 33
Zdrojový kód 3 - viewModel pro přihlášení 34
Zdrojový kód 4 - získání aut dle dispečera 35
Zdrojový kód 5 - vytvoření uživatele 36
Zdrojový kód 6 - zobrazení denního plánu řidičům 37
Zdrojový kód 7 - emailová služba 38
Zdrojový kód 8 - odeslání emailové zprávy řidiči 39
Zdrojový kód 9 - nastavení AutoMapperu 40
Zdrojový kód 10 - příklad viewModelu 40
Zdrojový kód 11 - nastavení modulu 43
Zdrojový kód 12 - validace vstupního atributu 44
Zdrojový kód 13 - manuální mapování jízd na objekty scheduleru 46
Zdrojový kód 14 - reakce na změnu datumu v rámci editoru jízdy 47
Zdrojový kód 15 - uložit a pokračovat 48
Zdrojový kód 16 - metoda pro získání uživatelských účtů 50
Zdrojový kód 17 - uložení zákazníka 51

SEZNAM ZKRATEK A ZNAČEK

API	Application Programming Interface
ASP	Active Server Pages
BCL	Base Class Libraries
CSS	Cascading Style Sheets
ČR	Česká republika
DOM	Document Object Model
GUID	Globally unique identifier
HTML	HyperText Markup Language
IT	Informační technologie
JSON	JavaScript Object Notation
MIT	Massachusetts Institute of Technology
MVC	Model-View-Controller
REST	Representational state transfer
SMS	Short message service
SMTP	Simple Mail Transfer Protocol
URL	Uniform Resource Locator

ÚVOD

Nákladní silniční doprava patří mezi nejrozšířenější způsob přepravy zboží, a to nejen u nás, ale i v zahraničí. S tímto souvisí počet firem, které tyto služby nabízejí. Každodenně se v dopravních firmách jen v České republice uskuteční několik desítek až stovek přepravních zakázek, jejichž počet je odvislý od velikosti firmy a počtu vozidel. To s sebou přináší tlak na logistiku, využívání firemních zdrojů a plánování. Vše je třeba využívat na maximum s nulovou chybovostí. Proto vyvstala potřeba, což je zároveň náplň této práce, zjistit, jak fungují zmíněné interní procesy v malých a středních firmách.

Vzhledem ke komplexnosti a rozsahu zadání bylo toto téma rozděleno do dvou diplomových prací, z nichž se každá zabývá řešením problémů pro zefektivnění práce dispečerů a řidičů za použití moderních technologií. Tato práce se věnuje dopravě jako celku, specifikaci procesů v dopravních firmách, návrhu řešení pomocí centrálního systému pro efektivní plánování a následné implementaci dispečerské části systému spolu s přehledem technologií použitých při vývoji. Diplomová práce kolegy Jana Holešínského se zaměřuje detailněji na možné využití informačních technologií v dnešním průmyslu a jeho možný vývoj. V praktické části diplomové práce Jana Holešínského se řeší problém uchovávání dat a automatizace komunikace mezi dispečerem a řidičem.

V první kapitole je čtenář seznámen se situací nákladní dopravy v České republice, která dále přechází v detailnější upřesnění silniční nákladní dopravy a její specifika. První dvě podkapitoly také zasazují tento problém do konkrétního prostředí pro území České republiky. Po tomto obecném základu je zaměřena pozornost na seznámení s vnitřními procesy ve firmách a na to, jak dnes u většiny firem probíhá objednání přepravy. Cílem je definovat jednotlivé kroky z pohledu dispečera, který musí objednávku zpracovat a naplánovat, a následně z pohledu řidiče, kdy může zakázku uskutečnit.

V průběhu zpracovávání přepravních zakázek, v kontextu dnešního způsobu jejich řešení, vyvstává celá řada problémů, které musí dispečeři, respektive řidiči operativně řešit. Touto částí se zabývá první část práce, a to stručným přehledem moderních technologií a jejich možnou aplikací pro vyřešení zmíněných úskalí. V samotném závěru je čtenář informován o následném návrhu softwarového řešení.

Další část pak pokračuje přehledem použitých technologií při implementaci dispečerské aplikace. Při výběru těchto technologií byl kladen důraz na možné rozdělení aplikace na autonomní moduly, rozšiřitelnost aplikace, rychlost zpracování a vytvoření přívětivého

uživatelského rozhraní. Z těchto požadavků vyplynulo použití moderní serverové technologie od společnosti Microsoft, dále využití JavaScriptového frameworku, čímž je zajištěna modularita, případná rozšiřitelnost a rychlost zpracovávání požadavku. Dále použití knihoven se sadou komponent zajišťující grafické zobrazení uživatelům.

Nejpodstatnější částí práce je vlastní návrh a implementace dispečerské aplikace, kterému se věnuje druhá část tohoto díla. Aplikace je rozdělena na dva bloky: (i) aplikační vrstvu zahrnující zpracovávající požadavky na server, získávání dat a jejich zpracování a (ii) prezentační vrstvu zabezpečující interakci mezi uživatelem a serverem.

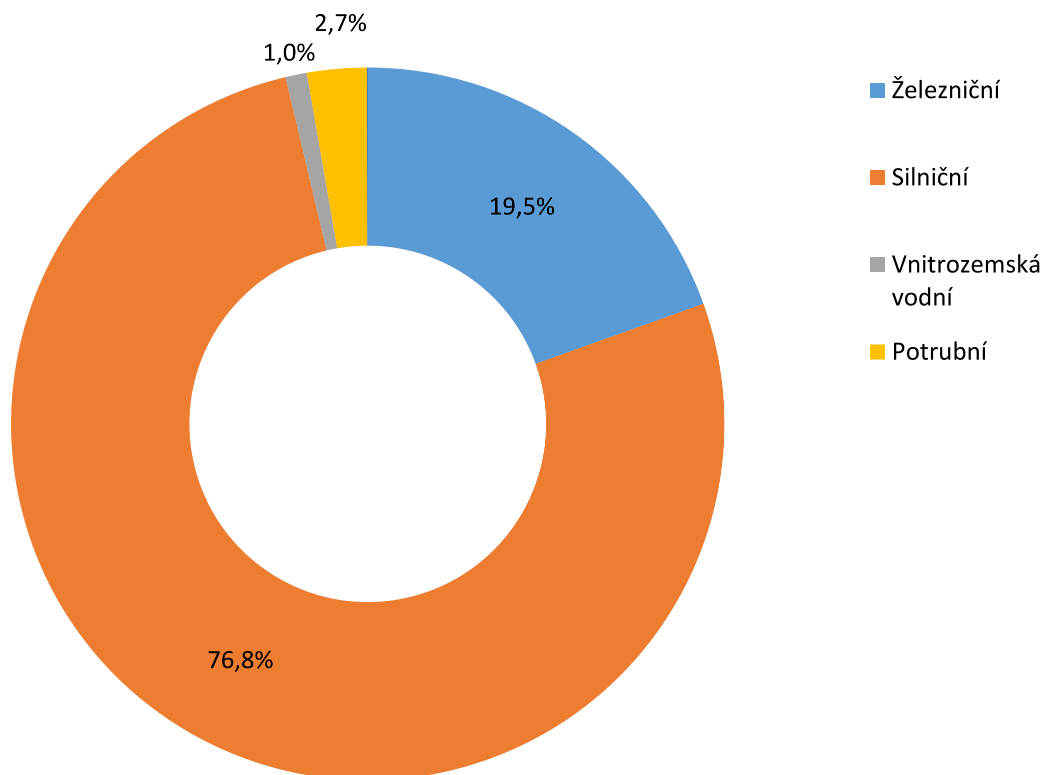
1 SOUČASNÉ TRENDY V NÁKLADNÍ DOPRAVĚ

Současná společnost požaduje stále více pestrou paletu poskytovaných služeb a produktů, z čehož plyne nutnost přepravovat stále větší množství zboží z jednoho místa na druhé v co nejkratší možné době. Tyto skutečnosti vedou ke zvyšujícím se objemům i k různorodosti přepravovaného zboží s důrazem na doručení ve stanoveném termínu.

1.1 Vývoj vnitrostátní nákladní dopravy v ČR

Rozloha České republiky je 78 866 km². Tím se řadíme mezi menší země v rámci Evropské unie. Nevzniká u nás tedy problém přepravy na dlouhé vzdálenosti, kde by nebylo možné přepravit zboží včas. Také u nás více převažuje přeprava již zhotovených produktů než jednotlivých surovin. Zároveň patříme k zemím s hustou sítí pozemní dopravní infrastruktury. Díky těmto faktům u nás převládá nákladní doprava silniční, jak je patrné z uvedeného grafu na **obrázku 1**.Tomuto tématu je věnována následující část.

Na druhém místě se dlouhodobě objevuje železniční nákladní doprava, a to zejména kvůli přepravování objemných nákladů hlavně surovin k jejich dalšímu zpracování [2].



Obrázek 1- Struktura nákladní dopravy v ČR v roce 2013 [mil. tkm, %], zdroj: [1]

1.2 Nákladní silniční doprava

Tento druh dopravy má výhody především v rychlosti doručení, protože firmy dokážou reagovat na poptávku po přepravě flexibilně. Dnes je možné si přepravu zboží objednat, a ještě tentýž den zakázku uskutečnit. Další předností je hustota dopravní sítě. Silniční sít je daleko hustší než železniční, a tak je možné doručit zboží na přesně stanovené místo. Není třeba se vázat jen na lokality, kde se vyskytuje nádraží nebo vykládkové místo pro složení nákladního vlaku.

Nevýhodou je nižší kapacita a vyšší cena za kilometr přepravovaného zboží, a proto se používá především pro transport již hotových věcí nebo produktů, které mají omezenou dobu spotřeby (zejména potraviny). Pro přepravování surových materiálů se využívá doprava železniční a vodní.

Po roce 1989 se v České republice začal průmysl a ekonomika zaměřovat na poskytování služeb a výrobu koncových produktů. Těžký průmysl začal ustupovat do pozadí. Navíc v posledních letech začala ekonomika opět růst a výroba se zvýšila. Tím se navýšily i objemy pro nákladní silniční dopravu. Přehled objemů v ČR naleznete v **tabulce 1**.

Stoupá počet firem podnikajících právě v oboru silniční nákladní dopravy, vídáme stále více nákladních automobilů. S tím roste počet lidí zaměstnaných jako řidiči, dispečeři a mechanici, kteří jsou nutní pro efektivní plánování, obsluhu a údržbu těchto vozidel [1].

Tabulka 1 - Vnitrostátní nákladní silniční doprava (pouze vozidly registrovanými v ČR), zdroj: [2]

	2010	2011	2012	2013	2014	2015
Přeprava věcí celkem (tis. tun)	***301 453***	***288 581***	***281 398***	***289 146***	***324 129***	***375 106***
podle kategorií přepravní vzdálenosti						
0 - 50 km	214 943	199 786	196 684	198 617	221 771	241 961
50 - 150 km	54 115	54 924	52 777	55 911	66 138	90 532
150 - 300 km	24 239	26 301	24 903	27 033	28 075	33 602
300 - 500 km	7 668	6 981	6 608	7 156	7 853	8 426
500 km a více	489	589	427	429	292	584

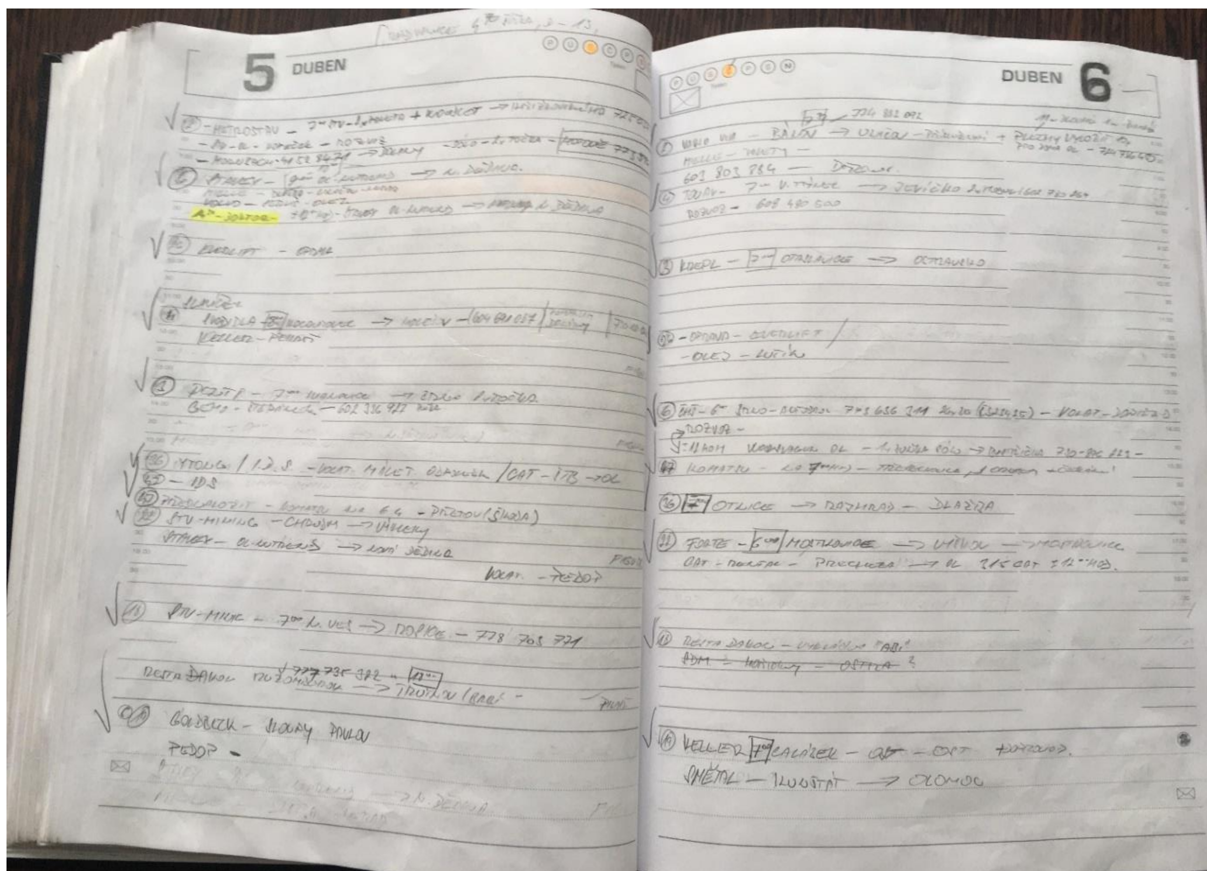
1.3 Procesy plánování v malých a středních firmách

Moderní zařízení jako chytré mobilní telefony, tablety či televize nebo IT technologie se v průmyslu a podnikání využívají, nicméně v oblasti nákladní dopravy v rámci malých a středních firem to většinou neplatí.

Většina dispečerů stále využívá pro plánování tužku a papír. Auta se často opravují, jakmile se vyskytne nějaká závada nebo před technickou prohlídkou. Průběžné vyhodnocování výkonů v takovém případě není možné, a nelze se tak zaměřit na slabá místa v procesech plánování a zlepšit je. Kontrola aktuálního dění se tak často omezuje pouze na sledování vozidel pomocí GPS jednotek vozidel na trase. A to je jen minimální výčet skutečností, které je možné díky moderním technologiím změnit.

Plánování přepravy v menší dopravní firmě:

- Objednání zakázky z místa A na místo B
- Zapsání informací o zakázce dispečerem
- Naplánování zakázky a zapsání do dispečerského deníku, viz **obrázek 2**
- Informování řidiče o vzniku zakázky



Obrázek 2 - Dispečerský deník, zdroj: vlastní

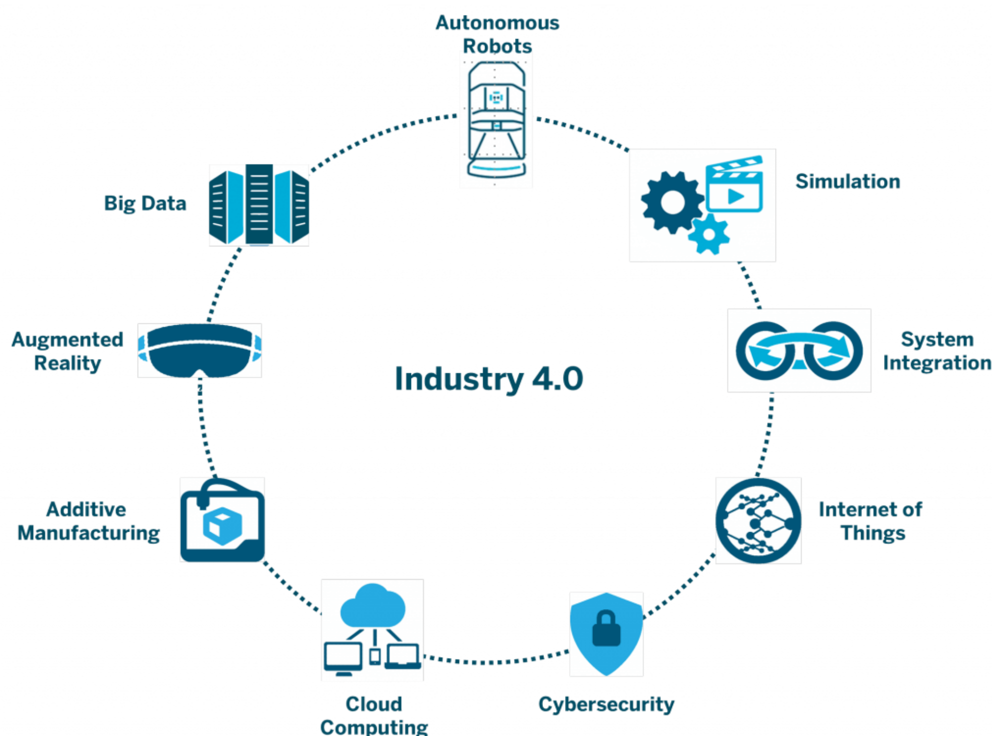
V tomto způsobu plánování vzniká celá řada problémů, které vedou k neefektivitě a přetěžování jednotlivých pracovníků. Mezi tyto nesnáze patří zejména změna jednotlivých jízd v den uskutečnění, potřeba záznam opravit nebo vytvořit zcela nový rozvrh. Následně je nutné zavolat řidičům a informovat je o změnách v jejich harmonogramu práce. Dále je zde riziko, že řidič ztratí informace o své práci, opět musí zavolat dispečerovi a informace od něj

získat. Prakticky není možné se v historii vracet k předešlým dnům a vyhodnocovat jednotlivé dny, týdny nebo měsíce. Dalším důležitým faktorem je kooperace více dispečerů v jedné firmě. Je nutné, aby seděli v jedné místnosti a mohli spolu komunikovat, neboť jinak by bylo složité vysvětlovat, kdy a jak je možné jízdu naplánovat.

S větším množstvím zakázek se zákonitě zvyšuje počet aut a zaměstnanců, což vede k přetížení stávajících dispečerů a k nutnosti zaměstnávat další. Najít kvalifikovaného zaměstnance je v dnešní době potíž. Zároveň není možné toto opakovat donekonečna, jelikož problém efektivity plánování exponenciálně narůstá, přibývají zmatky a chyby, nutnost stále více komunikovat, a tím se neúnosně zvyšují náklady.

1.4 Informační technologie v silniční nákladní dopravě

Dnes již lze tyto nesnáze řešit pomocí nástrojů, které nám moderní doba nabízí. Stále více věcí se digitalizuje a uchovává v datových skladech, klade se důraz na automatizaci procesů s důrazem na šetření zdrojů. Vzniká potřeba získaná data vyhodnocovat, nalézt slabá místa a navrhnout plán na jejich zlepšení.



Obrázek 3 - Průmysl 4.0, zdroj: [3]

Souhrn těchto moderních technologií se nazývá 4. průmyslová revoluce nebo také průmysl 4.0. Ta přináší celou řadu nových technologií a možností, které lze využít. Patři mezi ně internet věcí a služeb, machine learning, cloudová řešení, big data a další. Podrobné vysvětlení těchto

pojmů a podrobný popis průmyslu 4.0 naleznete v diplomové práci kolegy Jana Holešínského, přehled naleznete na **obrázku 3**.

Návrh řešení problému plánování výkonů a následné komunikaci mezi dispečerem a řidičem v rámci průmyslu 4.0 je představen v **kapitole 3**.

2 POUŽITÉ TECHNOLOGIE

Při implementaci centrálního dispečerského systému byl použit moderní přístup v kombinaci s nejnovějšími technologiemi. Byl kladen důraz na snadnou rozšiřitelnost, kterou tyto technologie umožňují. Jelikož je celá aplikace rozdělena na jednotlivé moduly, lze snadno nahradit jednu technologii jinou.

V rámci aplikační vrstvy (tzv. back-endu) byla zvolena nová platforma ***.NET Core***, která byla v počátcích vývoje stále ještě v beta verzi, nicméně pomohla odlehčit aplikaci od nežádoucích částí, jež by s sebou původní ***.NET Framework*** nesl.

V rámci prezentační vrstvy nebyl využit čistý ***JavaScript***, ale framework nad ním postavený ***AngularJS***. Šlo o zásadní rozhodnutí, které umožnilo intuitivnější, rychlejší a snazší ovladatelnost uživateli. Důležitou součástí je také knihovna komponent ***Syncfusion*** a možnost responzivního designu díky ***Bootsrap***.

2.1 .NET Core

Platforma ***.NET Core*** je zcela nová technologie představená v létě roku 2016. Tým zabývající se vývojem ***.NET Core*** začal budovat tuto platformu znovu od začátku tak, aby vyhovovala novým trendům v oblasti dnešní evoluce. Zcela se oprošťuje od původního ***.NET Frameworku*** a neobsahuje spoustu nepotřebných věcí, které tento prvotní framework obsahoval.

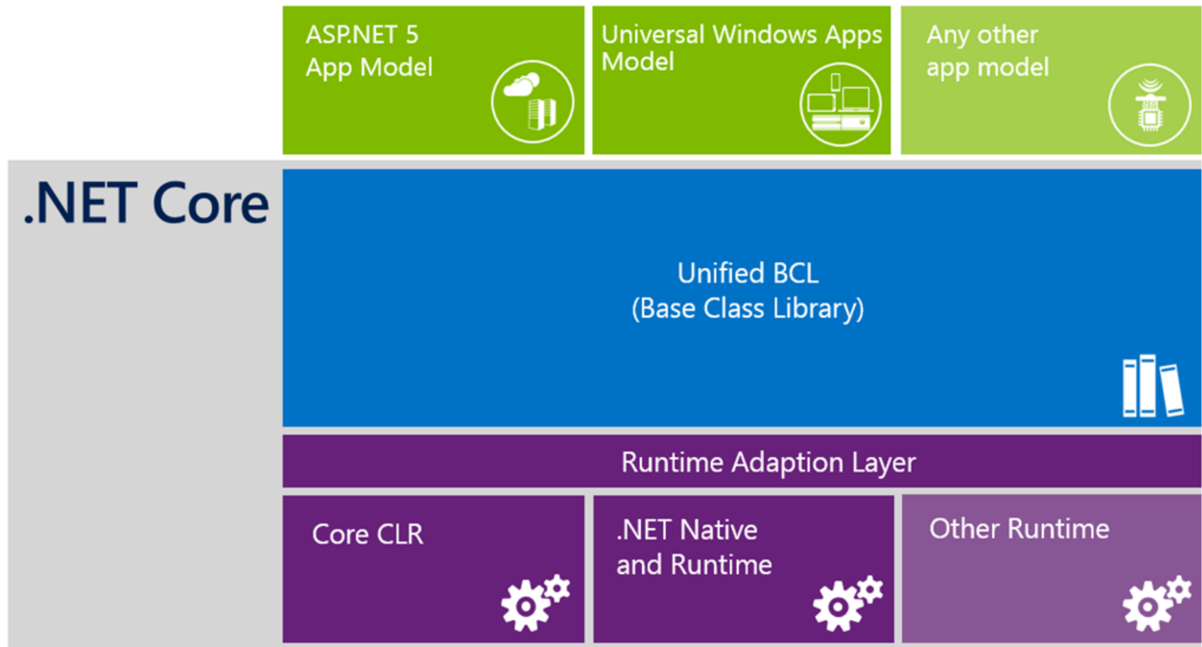
Microsoft tímto také jasně dokázal, že změnil svou strategii a tato platforma je kompletně open source. Dal ji tedy k dispozici komunitě, aby se mohla podílet na jejím vývoji a vylepšení, čímž si zajistila mnohem rychlejší růst a otevírá nové možnosti vývoje pro komunitu.

Jedná se o multiplatformní technologii, takže je možné aplikace hostovat a používat nejen na systémech Windows, ale i na Linux, macOS nebo v cloudových systémech. Je také plně kompatibilní s ostatními platformami jako jsou původní ***.NET Framework***, ***Xamarin*** či ***Mono***, díky knihovnám napsaných v ***.NET Standard***, tento přístup je popsán níže [5].

Charakteristické prvky definující ***.NET Core***:

- Pružné nasazování – může být součástí aplikace nebo nainstalovaný vedle jiné.
- Multiplatformní systém – může běžet na Windows, Linux, macOS a jiných.
- Možnost ovládání přes příkazový řádek – všechny pokyny lze napsat jako příkaz v příkazovém řádku.
- Kompatibilita – je plně kompatibilní s jinými systémy díky ***.NET Standard***.

- Open Source – MIT a Apache 2 licence a je možné platformu volně rozšiřovat.
- Podpora od Microsoft – Microsoft ***.NET Core*** stále posouvá kupředu a schvaluje nová vylepšení od komunity.



Obrázek 4 - Struktura .NET Core, zdroj: [4]

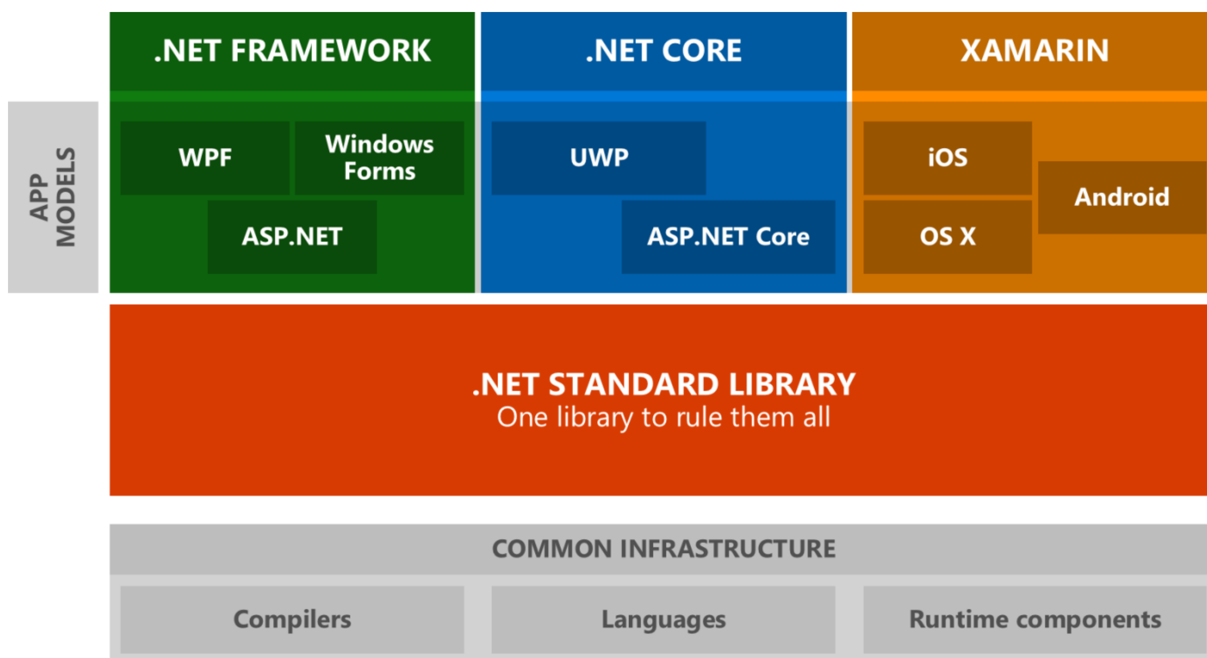
Platforma ***.NET Core*** se skládá z .NET běhového prostředí, které poskytuje typ systému, garbage collectoru, načítání zdrojů a další běžné služby, z knihoven frameworku poskytující základní datové typy, sady SDK nástrojů a kompilátorů nezbytných pro vývoj a příkazu „dotnet“ pro spouštění aplikace. Architektura .NET Core je na **obrázku 4.**

V ***.NET Core*** lze vyvíjet několika jazyky. Nejvíce používaným jazykem je ***C#***, variantou může být ***F#*** a připravuje se podpora ***Visual Basicu***. Kompilátory ***.NET Core*** umožňují vývoj všude tam, kde běží. Vývojáři je nepoužívají přímo, ale s využitím SDK nástrojů.

Pomocí ***.NET Core*** lze psát unifikované BCL knihovny, které se následně mohou používat na jiných platformách implementujících daný ***.NET Standard***. To umožňuje napsat si funkcionalitu pouze jednou a sdílet ji napříč různými aplikacemi, kde bude fungovat konzistentně a omezí se redundantní řádky kódu.

2.1.1 .NET Standard

.NET Standard je formální specifikace .NET API, které je dostupné napříč .NET běhovými prostředími. Cílem je standardizace knihoven, jejich přenositelnost a použití stejné



Obrázek 5 - Knihovna .NET Standard, zdroj: [6]

funkcionality napříč technologiemi .NET. Schéma návrhu těchto knihoven je možné vidět na **obrázku 5**.

Možnosti ***.NET Standard***:

- Definice sady univerzálního .NET API přístupného na všech .NET platformách.
- Psaní BCL přenositelných knihoven spustitelných na všech .NET běhových prostředích.
- Snížení a v budoucnu i eliminace různého překladu stejného zdrojového kódu.

.NET Standard byl představen v roce 2016. Ačkoliv již poskytuje celou řadu možností, čímž pomáhá vývojovým pracovníkům psát aplikace efektivněji, vyskytují se v něm ještě potíže. Různé verze .NET technologií momentálně podporují různý ***.NET Standard***. Tyto problémy by měla vyřešit verze ***.NET Standard 2.0***, která s sebou přinese celou řadu novinek, bude cílit na nové verze a vyjde v průběhu roku 2017 [7]. Přehled kompatibility ***.NET*** technologií s knihovnami ***.NET Standart*** je v **tabulce 2**.

Tabulka 2 - Kompatibilita .NET Standard, zdroj: [7]

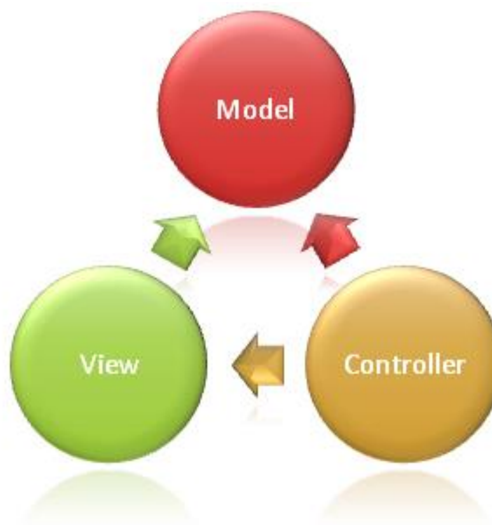
.NET Standard	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**2.0**
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	vNext
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	vNext

2.2 ASP.NET Core

Jde o jednu z technologií z výše zmíněné platformy ***.NET Core*** se zaměřením na vývoj multiplatformních aplikací založených na cloudovém řešení, ke kterým se uživatelé připojují pomocí internetu. Tato technologie byla použita pro tvorbu datové a aplikační vrstvy v praktické realizaci této diplomové práce a práce kolegy Jana Holešínského. Ta se zabývá touto technologií více do hloubky, jelikož přímo ***ASP.NET Core*** je hlavní součástí datové vrstvy. Pro aplikační vrstvu byla využita jedna ze součástí ***ASP.NET Core MVC***.

2.2.1 ASP.NET Core MVC

Jedná se o framework s širokou paletou nástrojů pro tvorbu aplikační případně prezentační vrstvy webových aplikací používající návrhový vzor *Model-View-Cotroller*. Návrhový vzor MVC umožňuje rozdělit aplikaci na funkční bloky jako je ukládání a získávání dat, jejich zpracování a následně jejich zobrazení uživateli. Uživatel vyšle svou akcí požadavek z *View* na server. Tam ho odchytí příslušný *Cotroller*, který se spojí s kompetentním *Modelem*, jenž získá data. Ta mohou být dále zpracována, než jsou předána zpět do *View* přes *Controller*, kde se zobrazí uživateli. Propojení jednotlivých vrstev je patrné z **obrázku 6**.



Obrázek 6 - Struktura MVC, zdroj: [8]

Aplikace napsané v ***ASP.NET Core MVC*** se snadno rozšiřují pomocí dalších modulů, dobře se testují a jsou optimalizovány pro spolupráci s jinými aplikacemi z platformy ***ASP.NET Core***. Mezi důležité funkce patří trasování, vázání dat na modely, validace dat, používání služeb pomocí dependency injection a podpora ***JavaScriptových*** frameworků pro prezentační vrstvu, které byly využity v praktické části této práce.

Trasování zajistí mapování URL adres, aby byly co nejlépe optimalizované pro prohlížeče. Díky tomu se zajistí zobrazování stránek na vyšších pozicích a uživatel je může snadno najít.

Adresy lze nakonfigurovat dle šablon (např. název *Controller* a poté název akce), nebo je přesně definovat a přiřadit jim příslušnou akci [8].

Validace dat a jejich následné přiřazování do připravených šablon objektů, aby s nimi mohl dále pracovat, se provádí za pomoci dekorování příslušných atributů objektů anotacemi. Ty specifikují přesné požadavky, např. minimální a maximální řetězce, zdali je atribut povinný, nebo odpovídá vzoru.

Dependency Injection, neboli poskytování služeb napříč aplikací, je technika pro zajištění volných vazeb tak, aby bylo možné službu zaměnit za jinou. Této problematice se opět věnuje jedna z kapitol v práci Jana Holešínského

2.2.2 ASP.NET Core a podpora prezentačních nástrojů

Podstatnou součástí ***ASP.NET Core*** je široká podpora ***JavaScriptových*** frameworků, jež mohou zajistit plynulejší chod celé aplikace oproti použití rozsáhlých *Views*, které se musejí celé znovu vykreslovat po každé akci uživatele.

Podporuje orchestraci ***JavaScriptového*** kódu pomocí dvou nejpoužívanějších nástrojů ***Gulp*** a ***Grunt***. Tyto nástroje se používají pro zmenšování ***JavaScritových*** souborů, kontrolu kvality kódu, sestavení ***TypeScriptu***, což je nadstavba ***JavaScriptu*** a mnoho dalších.

Pro přehlednou správu ***JavaScriptových*** knihoven lze využít ***Bower***, jenž byl použit v aplikaci této práce. Jde o balíčkovací systém, přes který je možné balíčky vyhledávat, instalovat a hlídat si aktuální verze. Odpadává vyhledávání knihoven na internetu, jejich stahování a následné kopírování do projektů. Vše je řešeno centrálně z jednoho místa.

Dále je uplatněna široká podpora ***JavaScriptových*** či ***CSS*** frameworků jako je ***Bootstrap***, ***AngularJS*** či ***ReactJS***. V již zmiňované aplikaci byl použit ***Bootstrap*** zajišťující responzivní design a ***AngularJS*** pro vykreslování dat v prezentační vrstvě [8].

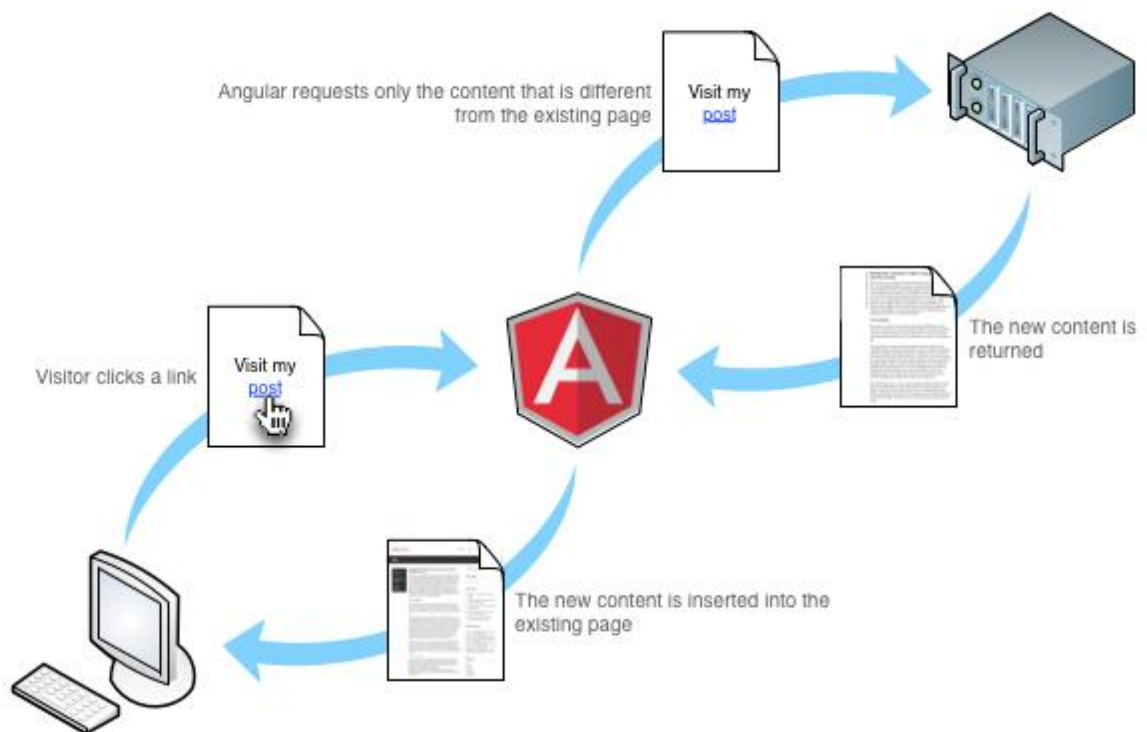
2.3 AngularJS

AngularJS je rozsáhlý framework pro tvorbu dynamických webových aplikací. Umožňuje vývojáři používat ***HTML*** jako šablony a nad ním vytváří logiku, která reaguje na akce uživatelů. Díky data bindingu a dependency injection se stává zdrojový kód čistší a čitelnější, zároveň eliminuje spoustu zbytečných řádků, jež by bylo jinak nutné napsat.

Celá logika naprogramovaná v ***AngularJS*** se provádí na straně klienta, tím šetří zdroje a čas nutné pro znovunačtení stránky a provedení příkazů na server. Request na server je

implementován pro každou akci, čímž se zlepšuje dojem uživatele z aplikace. Je tedy plně připraven pro spolupráci se serverovou částí aplikace.

Jazyk ***HTML*** byl navržen spíš pro tvorbu statických prezentačních webových dokumentů než pro tvoření dynamických aplikací. Již od počátku vznikala potřeba vytváření dynamického obsahu, tudíž se začalo využívat serverových možností případně ***JavaScriptu***. ***AngularJS*** ale přináší kompletní řešení této problematiky [9]. Na **obrázku 7** je možné vidět schéma ***AngularJS***



Obrázek 7 - AngularJS workflow, zdroj: [10]

Důležité aspekty ***AngularJS***:

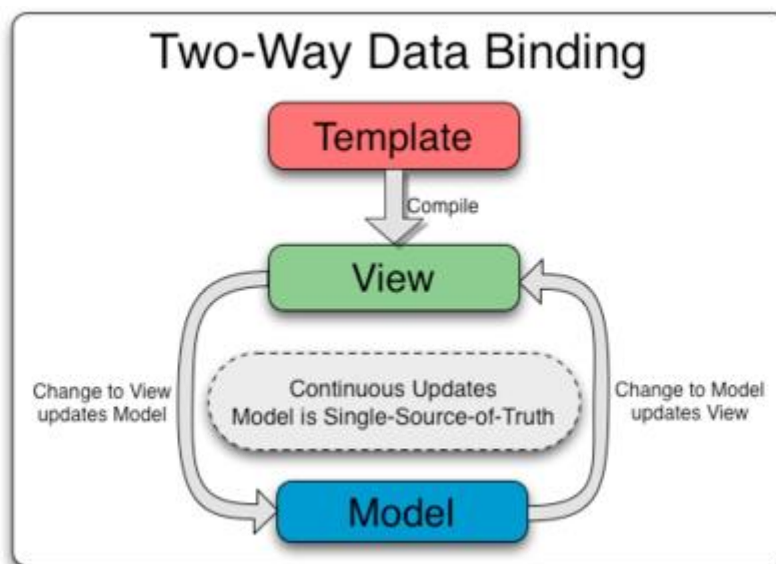
- Data binding – zobrazení dat získaných ze serveru.
- Správa DOM – vykreslování, schovávání nebo opakování elementů.
- Podpora formulářů a jejich validace.
- Napojení akcí uživatelů na DOM elementy.
- Vytváření znovupoužitelných komponent.

2.3.1 Data Binding

Díky obousměrnému data bindingu je zajištěna synchronizace komponent s daty uchovanými v klientské logice aplikace, která mohou být dále odeslána na server k jejich výslednému

zpracování a případně uchování v perzistentním uložišti. Projekce těchto dat je instantní a data jsou stále aktuální.

Obousměrný data Binding funguje na principu kompilování šablony, kde jeden z kroků vytvoří live view. V live view je odchycená každá změna, která se okamžitě promítne do modelu, a zároveň každá změna v modelu je hned zobrazena v šabloně. Tato technika odstraňuje složité získávání dat z jednotlivých komponent. Schéma obousměrného data bindingu je zobrazeno na **obrázku 8**. To usnadňuje práci s daty, jelikož jsou ihned dostupná a připravená na další zpracování [11].



Obrázek 8 - Obousměrný data binding, zdroj: [11]

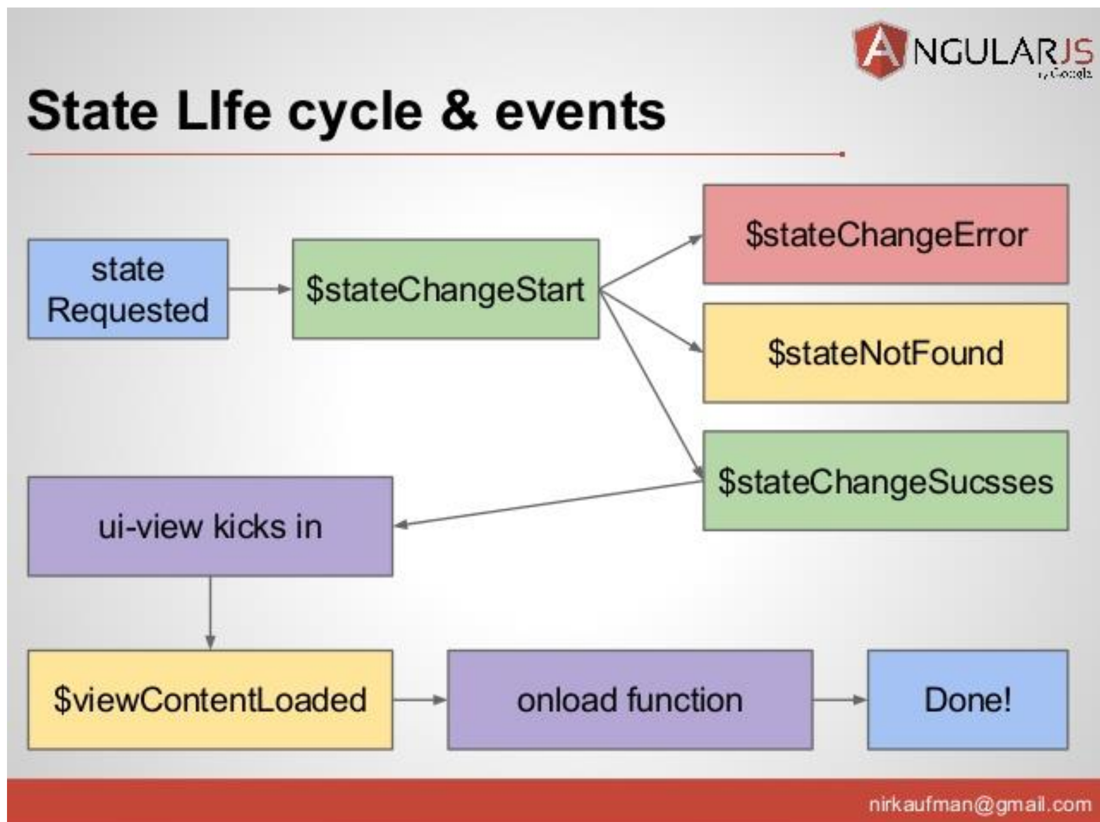
2.3.2 Controllery

Controllery jsou jedním ze stěžejních prvků pro ovládání ***HTML*** šablon a zpracování akcí uživatelů pomocí navržené business logiky. Jsou součástí větších funkčních celků tzv. ***Angluar*** aplikací. V principu ***Angular*** aplikace je jeden funkční celek (plánování jízd) a *Controller* obsluhuje jednu její část (editaci jízdy).

Obecně platí, že každý *controller* obsluhuje právě jedno *view* a neměl by obsahovat více logiky, než je nutné pro jeho obsluhu. Objekt *controlleru* je inicializován a přiřazen do DOM zkompilované stránky se svým rozsahem působení (scope) pomocí volání konstruktoru. V rámci scope se provádí inicializace a zpracování dat, odchytávání událostí, volání serveru. Pomocí dependency injection lze používat již vytvořenou funkcionalitu napříč moduly [12].

2.3.3 Angular Route

Modul ***Angular Route*** slouží pro používání části webové aplikace jako jednostránkové aplikace, čímž se šetří zdroje a čas. Obvykle jedna funkční část (***Angular*** aplikace) obsahuje svou navigaci pomocí ***Angular Route***. Vytváří iluzi změny stránek pro uživatele, ale ve skutečnosti nenačítá celou stránku znovu, nýbrž získává jen potřebná data, která se dále projektují do připravené ***HTML*** šablony. Cyklus trasování v rámci ***Angular Route*** je na **obrázku 9**.



Obrázek 9 - Navigace pomocí Angular Route, zdroj: [13]

Funkce ***Angular Route*** je přepínání mezi *controllery* pomocí definovaných URL šablon. V konfiguraci každé ***Angular*** aplikace je specifikována URL adresa, cesta k ***HTML*** šabloně a controller obsluhující tuto ***HTML*** šablonu. V rámci jedné aplikace lze snadno sdílet data. To je zajištěno pomocí parametrů v URL adrese nebo díky definování stejného rozsahu [13].

2.4 Bootstrap

Je moderní framework pro vývoj responzivních aplikací, což zajistí správné zobrazení aplikace na zařízeních s různým rozlišením obrazovky. Pomocí tohoto nástroje lze udržet jednotný styl ***HTML*** šablon. ***Bootstrap*** významně urychluje práci s designem, jelikož obsahuje předpřipravené styly a ***JavaScriptové*** knihovny pro práci s běžnými ***HTML*** elementy.

2.5 Syncfusion

Jde o knihovnu komponent umožňující rozšířit možnosti běžných ***HTML*** šablon. Poskytuje nástroje pro prezentaci dat uživateli, rozšiřuje funkcionalitu běžných součástí a přináší komponenty, které nejsou v ***HTML*** jazyce dostupné. Se ***Syncfusion*** lze pracovat na více platformách. Nejznámější jsou ***PHP***, čistý ***JavaScript***,***AngularJS***, ***ReactJS*** a mnoho dalších.

Mezi prvky, jež ***Syncfusion*** přináší, je třeba zmínit různé možnosti grafů, data *gridy*, *date* a *time pickery* či *schedule*. Schedule neboli rozvrh byl použit jako stěžejní komponenta umožňující správu a přehled jízd v praktické části této práce. Zde byl požadavek na specifické zobrazení dat uživateli (zejména dispečerovi) a toto zobrazení poskytla tato knihovna jako jedna z mála [14].

3 NÁVRH A IMPLEMENTACE DISPEČERSKÉ APLIKACE

Hlavním záměrem této diplomové práce a její praktické části je návrh a implementace dispečerské centrální aplikace, která poslouží pro efektivní plánování dopravy a správu výkonů řidičů, tedy vlastních jízd. Cílem je převést psanou formu, jak byla popsána v **kapitole 1,** do formy elektronické v rámci moderních IT technologií.

Efektivní plánování dopravy a správu nákladních automobilů, jejich přívěsů a řidičů, případně zákazníků lze vyřešit návrhem centrálního softwarového řešení, který bude moci dispečer obsluhovat a který bude napojen na další systémy v rámci celistvého řešení. Konkrétním návrhem a implementací se zabývá praktická část této práce. Hlavním motivem je snadná ovladatelnost, snadný přistup k softwaru a modularita.

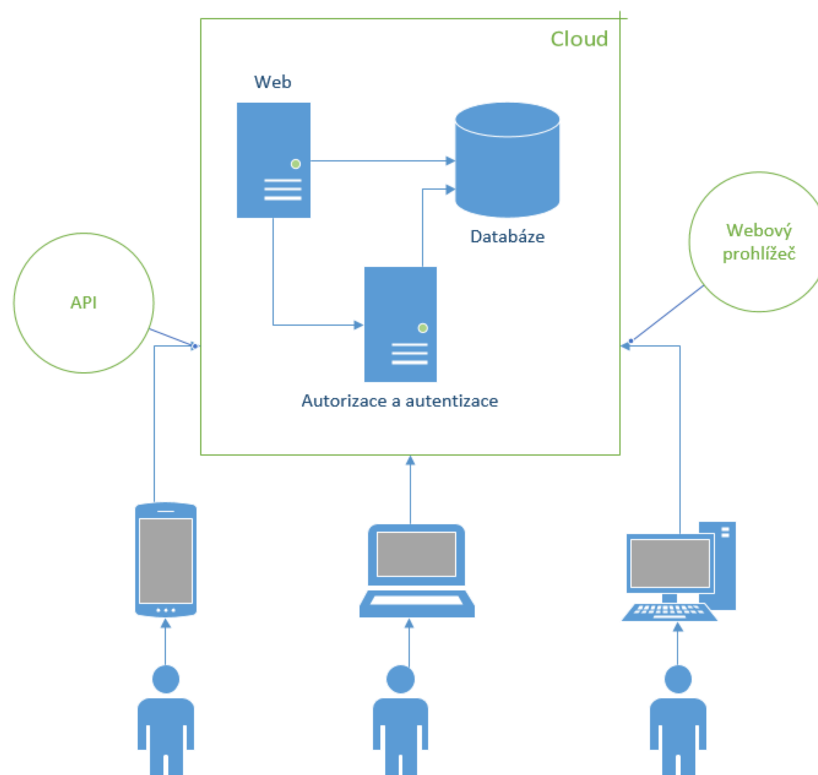
Pro veškerá data vytvořená dispečery, případně získaná z jiných zdrojů, je přiměřené použít relační databázi, která bude obsluhována databázovým serverem a knihovnou, z níž je vhodné použít moderní frameworky, aby bylo možné tato data dále zpracovávat.

Pro potřeby komunikace mezi dispečerem a řidiči, aby nebylo nutné jim každou změnu telefonovat, je vhodnější využít platformy mobilních zařízení a navrhnout pro ně vhodnou aplikaci. Aplikace musí splňovat snadné ovládání a kompatibilitu napříč nejpoužívanějšími systémy, jimiž jsou Android a iOS.

Návrhem a implementací datové vrstvy a mobilní aplikace se podrobně věnuje praktická část diplomové práce Jana Holešínského.

3.1 Vrstvení aplikace

Návrh centrálního softwarového řešení umožňuje rozdělení odpovědností aplikace do tří vrstev, a to vrstvy datové, aplikační a prezentační. Schéma aplikace naleznete na **obrázku 10.** Každá vrstva obsahuje sadu funkcionalit. Datová vrstva slouží k ukládání, získávání a třídění dat. Této části práce se věnuje kolega Jan Holešínský ve své praktické části. Aplikační vrstva slouží pro aplikaci business logiky, která se získanými daty dále pracuje, a prezentační vrstva slouží pro zobrazení těchto dat a bezprostřední komunikaci s uživatelem.



Obrázek 10 - Schéma aplikace pro efektivní plánování nákladní dopravy, zdroj: vlastní

Díky vrstvení aplikace se zajistí její snadná rozšiřitelnost o další moduly, zdrojový kód je přehlednější a snadno testovatelný. Samotná aplikační vrstva je rozdělena na jednotlivé funkční bloky zajišťující správu různých dat, stejným způsobem byla navržena i vrstva prezentační. Prezentační vrstva spolupracuje pouze s vrstvou aplikační, také datová vrstva je napojena jen na aplikační. Z toho plyne, že aplikační vrstva je centrálním bodem celé aplikace. Díky této architektuře je zajištěno volné spojení a vrstvy jsou lehce nahraditelné, což umožňuje snadnou změnu technologie, pokud by tato potřeba nastala. V **příloze C** se nachází diagram tříd celé aplikace.

3.2 Aplikační vrstva

Aplikační vrstva slouží k aplikování business logiky spolu se zajištěním propojení mezi prezentační a datovou vrstvou. Nacházejí se zde servisy, které rozšiřují možnosti celé aplikace. Může to být například služba zajišťující odesílání SMS zpráv nebo, jak je tomu v tomto případě, služba pro odesílání emailových zpráv.

Další částí aplikační vrstvy je komunikace s prezentační vrstvou, jež je zajištěna pomocí API nebo MVC strukturou. Z **kapitoly 2** již víme, že tato vrstva běží na ***ASP.NET Core MVC***,

což zefektivňuje a zpřehledňuje tuto komunikační činnost. Vlastní předávání a zpracování dat provádí převod na ViewModely, což jsou jednoduché třídy odrážející význam dat neboli databázových entit. Pomocí těchto objektů je prováděna validace dat na této úrovni, tzn. pokud zaslaná data nesplňují předem stanovené požadavky, tak se nezpracují a aplikace provede další kroky pro informování uživatele, který musí provést nápravu.

Za vlastní komunikaci jsou odpovědné *controllery*, jejichž názvy jsou vždy voleny tak, aby z jejich názvu byly zřejmé jejich zodpovědnosti. V dispečerské aplikaci jsou *controllery* rozděleny do dvou skupin, kde jedna komunikuje v rámci ***ASP.NET Core MVC*** a druhá pomocí API komunikuje s prezentační vrstvou, případně s mobilní aplikací.

V neposlední řadě je důležité vlastní zabezpečení, autentizace a autorizace, která je zajištěna pomocí ***ASP.NET Identity***. Jde o komplexní řešení, jež umožní navigaci napříč aplikací jen přihlášeným uživatelům, kteří mají na příslušnou akci právo.

3.2.1 Autentizace a autorizace dispečerské aplikace

Jelikož softwarový systém bude obsahovat citlivá firemní data, nejdůležitější součástí aplikace je její zabezpečení spolu s autentizací a autorizací tak, aby nedošlo k jejich zneužití třetí stranou. I když z pohledu uživatele zabezpečení nepřináší žádná rozšíření, bylo nutné věnovat mu velkou pozornost. Z tohoto důvodu bylo zvoleno zabezpečení pomocí ***ASP.NET Identity***

```
services.AddIdentity<ApplicationUser, IdentityRole>(option =>
{
    option.Password.RequireNonAlphanumeric = false;
    option.Password.RequireUppercase = false;
    option.Password.RequireLowercase = false;

    option.Cookies.ApplicationCookie.LoginPath = "/account/login";
    option.Cookies.ApplicationCookie.Events =
    new CookieAuthenticationEvents
    {
        OnRedirectToLogin = async ctx =>
        {
            if (ctx.Request.Path.StartsWithSegments("/api") &&
                ctx.Response.StatusCode == 200)
            {
                ctx.Response.StatusCode = 401;
            }
            else
            {
                ctx.Response.Redirect(ctx.RedirectUri);
            }
            await Task.Yield();
        }
    };
})
```

Zdrojový kód 1 - nastavení ASP.NET Identity

a webových tokenů. Kompletní informace o zabezpečení naleznete v práci mého kolegy Jana Holešínského. Tato kapitola je dále věnována autentizaci a autorizaci uživatelů.

První věcí je nastavení samotného ***ASP.NET Identity*** (**zdrojový kód 1**) ve spojení s uživatelskými účty. Je provedeno v souboru *Startup.cs*, podobně jako další nastavení aplikace, jelikož je tento soubor zpracováván při spuštění aplikace. Stejně jako u ostatních balíčků je nutné službu v aplikaci zaregistrovat pomocí příkazu *services.AddIdentity* Vlastní nastavení se provádí pomocí lambda výrazu, kde je nutné zvolit sílu hesla, cestu k login stránce a v případě použití kontaktování API chování, když uživatel není přihlášen a chce provést nějakou akci. Zde mu bude vrácen status *code 401*, kde uživatel je informován, že není autorizován akci provést.

Akce uživatelů jako přihlášení, odhlášení nebo registrace jsou implementovány pomocí metod *Login, Logoff, Register* v *AccountController* (**zdrojový kód 2**) jako výsledek vrací *ActionResult*, což může být příslušný view, jak je patrno ze zdrojového kódu níže. Důležitou součástí všech akcí *controllerů* je anotace jako v tomto případě *[AllowAnonymous]*, což umožňuje provést akci, i když není uživatel přihlášen. Opakem je *[Authorize]*, kde akci může provést pouze přihlášený uživatel. Také zde lze specifikovat uživatelské role, které mají

```
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model,
string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        var result = await
        _signInManager.PasswordSignInAsync(model.UserName,
        model.Password,
        model.RememberMe,
        lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation(1, "User logged in.");
            return RedirectToLocal(returnUrl);
        }
        else
        {
            ModelState.AddModelError
            (string.Empty, "Invalid login attempt.");
            return View(model);
        }
    }
    return View(model);
```

Zdrojový kód 2 - akce Login

na akci právo. Druhá anotace *[ValidateAntiForgeryToken]* provede validaci tokenu aplikace a zabraňuje vynucení spuštění akce vně aplikace.

Před provedením pokusu o přihlášení je nejprve zkontrolováno, zdali jsou zadané informace správné, a to pomocí validace proti viewModelu (*LoginViewModel*), který je předán jako argument metody. *LoginViewModel* (**zdrojový kód 3**) je obyčejná třída obsahující vlastnosti objektu a anotace, pomocí nichž se specifikuje například, zda je daná vlastnost povinná, délka řetězce nebo co se má zobrazit uživateli v ***HTML*** šabloně. Pokud jsou splněny podmínky, provede se pokus o přihlášení, kde je poté vráceno příslušné *view* na základě výsledku pokusu.

```
public class LoginViewModel
{
    [Required]
    [Display(Name = "Uživatelské jméno")]
    public string UserName { get; set; }

    [Required]
    [Display(Name = "Heslo")]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Display(Name = "Zapamatovat si?")]
    public bool RememberMe { get; set; }
}
```

Zdrojový kód 3 - viewModel pro přihlášení

Podobně jako akce přihlášení v podobě metody *Login* jsou implementovány i ostatní metody pro ovládání uživatelských účtů. *Controllery* odpovědné za správu účtů jsou *AccountController* a *ManageController*.

3.2.2 Aplikační rozhraní pro prezentační vrstvu

Klíčovou zodpovědností aplikační vrstvy je propojení mezi datovou a prezentační vrstvou. V podstatě je to most získávající data z perzistentního uložiště, která jsou následně předána ke konečnému zpracování a zobrazení uživateli. Jelikož hlavní úlohu v prezentační vrstvě představuje ***JavaScriptový*** framework, bylo nutné vytvořit aplikační rozhraní (API) pro snadnou komunikaci mezi těmito vrstvami. Teoretický detail je zpracován v diplomové práci Jan Holešínského, jelikož aplikační vrstva komunikuje také s mobilní aplikací.

Zodpovědnost za firemní zdroje v aplikaci reprezentovanými entitami jako auta, přívěsná vozidla, uživatelé či zákazníci mají následující *controllery*: *CarsController*, *TrailersController*, *UserAccountsController* a *CustomersController*. V rámci aplikačního rozhraní byly dodrženy principy ***REST***.

Pro čtení dat z datové vrstvy a následné předání zpět k zobrazení uživateli bylo použito dotazovací metody *GET*. Pojďme se detailněji podívat na metodu pro získání aut, která spravuje určitý dispečer. Metoda je implementována v *CarsController* (**zdrojový kód 4**) a je jednoznačně určena pomocí názvu *GET* a atributu *id*, což představuje jednoznačný identifikátor dispečera pomocí *GUID*.

```
[HttpGet("owner/{id}")]
public IActionResult Get(string id)
{
    try
    {
        var results = _carRepository.GetCarsByOwner(id);
        return Ok(Mapper.Map<IEnumerable<UpdateCarViewModel>>(results));
    }
    catch (Exception e)
    {
        var message =
        $"Problém se získáním aut pro dispečera s {id}: {e.Message}";
        _logger.LogError(message);
        return BadRequest(message);
    }
}
```

Zdrojový kód 4 - získání aut dle dispečera

Opět zde můžeme vidět anotaci, která se liší syntaxí i významem. Anotace *[HttpGet("owner/{id}")]* představuje adresu, kdy bude tato metoda zavolána, a určení dotazovací metody, která musí být použita, aby byl požadavek správně zpracován. Označení *{id}* očekává, že v adrese se bude nacházet atribut, který se následně předá jako parametr právě do této metody. Pokud je požadavek zachycen, pokusí se získat auta dispečera z datové vrstvy a namapuje výsledky na příslušný *viewModel*. Problematice mapování a viewModelům je věnována kapitola **3.2.5 AutoMapper a ViewModely**.

Pokud dotaz skončí chybou, je zachycena a je vytvořena zpráva pomocí šablony, která slouží pro identifikaci chyby a její následné odstranění. Zpráva je zapsána do logů a též poslána zpět uživateli, který je následně informován, že jeho požadavek nebyl zpracován z důvodu zachycení chyby.

Metody *POST* bylo použito pro vytváření nových entit a *PUT* při úpravách již existujících entit. Co se týče zachycení požadavku, který má být zpracován touto metodou, funguje na podobném principu jako metoda *GET* s tím rozdílem, že data odeslána od klienta mohou být uložena v těle požadavku, ta jsou pak získána pomocí anotace *[FromBody]*. Získané entitě jsou přiděleny nezbytné informace a následně opět provedeno mapování, tentokrát opačným směrem na entitu datové vrstvy. Detail je možné vidět ve **zdrojovém kódu 5** pro vytvoření nového uživatele.

Tato metoda je oproti ostatním specifická v tom, že používá pro vytváření nového účtu ***ASP.NET Identity*** a její metody. Nejprve je vytvořen uživatelský účet spolu s přidělením dočasného hesla pro první přihlášení. Pokud tato operace proběhne úspěšně, je uživatelskému účtu přidělena jeho role a odeslána zpráva s informací o vytvoření uživatelského účtu. Zachycení chyb funguje na podobném principu jako u metody *GET*.

```
[HttpPost("useraccounts")]
public async Task<IActionResult> Post(
[FromBody]UserAccountViewModel user)
{
    user.LastEditationDateTime = DateTime.Now;
    user.CreatedDateTime = DateTime.Now;
    try
    {
        var userEntity = Mapper.Map<ApplicationUser>(user);
        userEntity.LastEditedById =
        User.FindFirstValue(ClaimTypes.NameIdentifier);
        var password = "heslo";
        //save to database
        var result = await _userManager.CreateAsync(userEntity,
                                     password);
        if (result.Succeeded)
        {
            await _userManager.AddToRoleAsync(userEntity,
                    user.RoleName);
            return Created($"api/useraccounts", userEntity);
        }
        return BadRequest(
        "Problém s uložením nového uživatele do databáze");

    }
    catch (Exception e)
    {
        var message = $"Problém při ukládání uživatele: {e.Message}";
        _logger.LogError(message);
        return BadRequest(message);
    }
}
```

Zdrojový kód 5 - vytvoření uživatele

Důvodem vzniku této aplikace je efektivnější plánování. Za tuto důležitou funkci odpovídá *RidesController*. Jde o nejrozsáhlejší funkční celek v rámci aplikační vrstvy využívající zdroje aplikace představené dříve v rámci vytváření jízd spolu s emailovou službou, představenou v následující kapitole pro informování uživatelů.

Controller pracuje na podobném principu pro získávání, vytváření a změnu jízd jako *controllery* pro správu zmíněných zdrojů v rámci aplikace. Obsahuje metody obohacující aplikaci o funkce vyžadované pro plánování. Patří mezi ně schválení jízdy s auty nepřidělenými dispečerovi, který jízdu naplánoval nebo zobrazení plánu jízd na určitý den po jeho dokončení.

V rámci metody pro zobrazení jízd je získána kolekce jednoznačných identifikátorů jízd na určitý den. Jelikož se jedná o úpravu již vytvořených entit, byla použita metoda *PUT* (**zdrojový kód 6**). Jak je patrné z hlavičky metody není vázána na určité datum. Může tak být použita i pro schválení plánu za určité období, pokud by takový požadavek nastal. V rámci metody je nejprve uložen uživatel, který změnu provedl, poté je odeslán požadavek na změnu jízd do datové vrstvy. Pokud vše proběhlo v pořádku, jsou odeslány emailové zprávy řidičům, aby se podívali do mobilní aplikace, co je čeká. Opět se zde zachytávají chyby, které případně nastanou, a jsou dále zpracovány.

```
[HttpPut("show-drivers")]
public async Task<IActionResult> Put([FromBody]ICollection<int> rideIds)
{
    try
    {
        var actualUser = await _userManager.GetUserAsync(User);
        //save to database
        _rideRepository.ShowRidesToDrivers(rideIds, actualUser);
        if (await _rideRepository.SaveChangesAsync())
        {
            _emailSender.SendEmailAsync(
                new List<string> { "st40409@student.upce.cz" },
                "Máte nové jízdy",
                $"Pozor byla vám zpřístupněna nová jízda
                {(DateTime.Now).ToString("HH:mm dd.MM.yyyy")}");

            return Created($"api/rides", rideIds);
        }
        return BadRequest("Problém s uložením jízd," +
        " které se mají ukázat řidiči v aplikaci, do databáze");
    }
    catch (Exception e)
    {
        var message =
        $"Problém při změně jízd, co se mají ukázat řidičům v aplikaci:"
        + "{e.Message}";
        _logger.LogError(message);
        return BadRequest(message);
    }
}
```

Zdrojový kód 6 - zobrazení denního plánu řidičům

Jak je patrné z výše uvedeného (implementace jednotlivých *controllerů* rozdělených do jednotlivých funkčních celků a odpovědných za určité oblasti), je komunikace mezi klientem a serverem z pohledu uživatelské funkcionality nejdůležitější částí aplikační vrstvy. Ostatní služby a nástroje vhodně doplňují a rozšiřují možnosti aplikace.

3.2.3 Mail Service

Aplikační vrstva obsahuje služby, které slouží k různým účelům. Jednou z těchto služeb může být mailová služba *EmailSender*, jež byla použita v rámci dispečerské aplikace a která je zobrazena ve **zdrojovém kódu 7**. Slouží k odesílání emailových zpráv v případě, že dispečer

```
public Task SendEmailAsync(List<string> UsersToSend,
 string Subject,
 string Text)
{
    var message = new MimeMessage();
    message.From.Add(
    new MailboxAddress("smartcargoplanning@gmail.com"));
    foreach (var item in UsersToSend)
    {
        message.To.Add(new MailboxAddress(item));
    }
    message.Subject = Subject;
    message.Body = new TextPart("plain")
    {
        Text = Text
    };
    using (var client = new SmtpClient())
    {
        client.Connect("smtp.gmail.com",
        587,
        SecureSocketOptions.StartTlsWhenAvailable);
        client.Authenticate("smartcargoplanning@gmail.com", "SCPs520.");
        client.Send(message);
        client.Disconnect(true);
    }
    return Task.FromResult(0);
}
```

Zdrojový kód 7 - emailová služba

naplánuje jízdu na vozidlo, které spravuje jiný dispečer. Ve zprávě jsou obsaženy informace pro danou jízdu jako vozidlo, případně přívěs, místo nakládky a vykládky, spolu s časem jízdy.Druhý případ je odeslání emailové zprávy řidiči. To se děje v případě, že si je dispečer jízdou jistý a nebude provádět žádné změny. Případně pokud nějaké změny udělá a jízda je již ve stavu pro zobrazení řidiči, je opět informován, aby se podíval do mobilní aplikace na změny.

V rámci metody *SendEmailAsync*, kterou naleznete ve **zdrojovém kódu 8**. se posílá text, jenž se má odeslat, předmět zprávy a kolekce emailových adres, na které má být zpráva doručena. Objekty, obsluhujícími daný požadavek na odeslání, jsou *message* a *client*. *Message* je vlastní zpráva a *client* je služba komunikující pomocí protokolu SMTP, jež se nejprve připojí k příslušnému serveru, pokusí se o přihlášení pod uživatelským jménem a heslem, pošle zprávu a následně se odpojí.

```
_emailSender.SendEmailAsync(
    new List<string> { "st40409@student.upce.cz" },
    "Máte nové jízdy",
    $"Pozor byla vám zpřístupněna nová jízda
{(DateTime.Now).ToString("HH:mm dd.MM.yyyy")}");
}
```

Zdrojový kód 8 - odeslání emailové zprávy řidiči

Posílání zpráv v rámci celé aplikace je důležité pro informování příslušných uživatelů, aby měli stále aktuální informace o věcech, které se jich týkají.

3.2.4 Klíčové NuGet balíčky

NuGet je balíčkovací systém, jímž lze získávat již vytvořené nástroje pro rozšiřování vyvíjené aplikace v rámci ***.NET Frameworku***. V prostředí ***.NET Core*** je již součástí celé platformy, tudíž funguje lépe než v klasickém ***.NET Frameworku***. Teoretickému popisu ***NuGet*** balíčkovacího manažeru se věnuje jedna z kapitol diplomové práce kolegy Jana Holešinského.

Nejdůležitějšími balíčky v rámci dispečerské aplikace jsou hlavní komponenty v rámci ***.NET Core***. Mezi nejvýznamnější patří ***Microsoft.NETCore.App***, ***Microsoft.AspNetCore.Mvc***, ***Microsoft.AspNetCore.Routing*** a ***Microsoft.AspNetCore.Server.IISIntegration***, které umožňují vlastní vývoj aplikace na této platformě, její spouštění pomocí serveru a zobrazení v prohlížeči.

Důležité jsou také balíčky, které rozšiřují ***ASP.NET*** aplikaci, ***Microsoft.AspNetCore.Authentication.Cookies*** pro autentizaci uživatele pomocí cookies, ***Microsoft.ApplicationInsights.AspNetCore*** sbírající data z běhu aplikace pro jejich další zpracování, ***Microsoft.AspNetCore.Diagnostics*** umožňující odhalení slabých míst, ***Microsoft.Extensions.Logging*** pro uchovávání logů a ***Microsoft.Extensions.Configuration.Json*** nastavující komunikaci pomocí JSON formátu.

Dalšími balíčky jsou ***MailKit*** popsaný v předcházející kapitole a ***AutoMapper***, jemuž je věnována následující část. ***BundlerMinifier.Core*** minifikuje ***CSS***, HTML a ***JavaScriptové*** soubory pro nasazení na produkci.

NuGet balíčkovací systém v rámci aplikace umožňuje efektivní správu nad rozšiřujícími nástroji, které se snadno aktualizují, přidávají nebo odstraňují, díky čemuž činí aplikaci více škálovatelnou.

3.2.5 AutoMapper a ViewModely

Nástroj umožňující konvenční mapování objektu na objekt, odstraňuje redundantní kód při vytváření objektů na základě objektu jiného. Páruje na sebe shodné atributy objektu, chybějící nastaví na implicitní hodnotu a přebytečné ignoruje. ***AutoMapper*** lze konfigurovat a přizpůsobovat potřebám aplikace. Podobně jako nastavení ***ASP.NET Identity*** se nastavuje ve *Startup.cs*. Pro běžné mapování objektů podle klíče stačí specifikovat objekty k mapování, případně nastavení obousměrného mapování, viz. **zdrojový kód 9**.

```
Mapper.Initialize(config =>
    {
        config.CreateMap<CreateRideViewModel, Ride>().ReverseMap();
        config.CreateMap<UpdateRideViewModel, Ride>().ReverseMap();
        config.CreateMap<CreateCarViewModel, Car>().ReverseMap();
        config.CreateMap<UpdateCarViewModel, Car>().ReverseMap();
    });
```

Zdrojový kód 9 - nastavení AutoMapperu

S použitím ***AutoMapperu*** souvisí *viewModely* reprezentující objekty aplikace odrážející aplikační data. Modely jsou jednoduché třídy, určené pomocí atributů, odrážející vlastnosti z reálného světa a obohacené o anotace určující, kdy je model validní. Například délka řetězce, nebo povinné vyplnění pole a zprávy pro informování uživatele v případě, když model nesplňuje zadaná kritéria. Pro ilustraci následuje příklad modelu ve **zdrojovém kódu 10.**

```
public class CreateCustomerViewModel : TrackableViewModel
{
    [Required(ErrorMessage = "Musí být zadáno jméno zákazníka.")]
    [StringLength(100, MinimumLength = 3,
    ErrorMessage = "Jméno zákazníka musí mít alespoň 3 a maximálné 100
                    znaků.")]
    public string Name { get; set; }
    [Required(ErrorMessage = "Musí být zadána ulice.")]
    [StringLength(100, MinimumLength = 3,
    ErrorMessage = "Ulice musí mít alespoň 3 a maximálné 100 znaků.")]
    public string Email { get; set; }
    public string PhoneNumber { get; set; }
    public string Street { get; set; }
    [StringLength(100, MinimumLength = 2,
    ErrorMessage = "Město musí mít alespoň 2 a maximálné 100 znaků.")]
    public string City { get; set; }
    [StringLength(10, MinimumLength = 5,
    ErrorMessage = "PSČ musí mít alespoň 5 a maximálné 10 znaků.")]
    public string ZipCode { get; set; }
    [StringLength(25, MinimumLength = 5,
    ErrorMessage = "IČO musí mít alespoň 5 a maximálné 25 znaků.")]
    public string IdentificationNumber { get; set; }
    [StringLength(25, MinimumLength = 5,
    ErrorMessage = "DIČ musí mít alespoň 5 a maximálné 25 znaků.")]
    public string VatNumber { get; set; }
}
```

Zdrojový kód 10 - příklad viewModelu

3.3 Prezentační vrstva

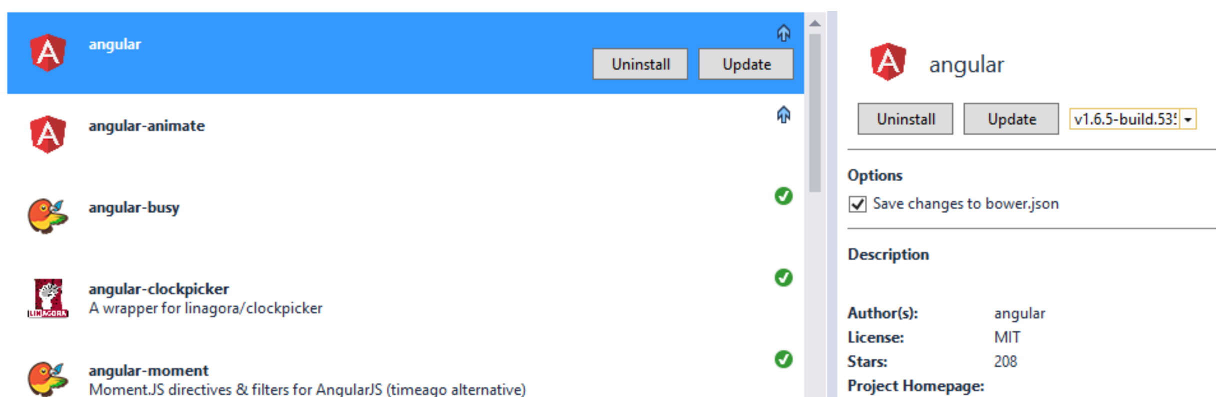
Zásadní částí aplikace pro uživatele je prezentační vrstva neboli též front-end, protože umožňuje zobrazení dat čitelně na obrazovku počítače, chytrého telefonu nebo jiného zařízení. Úlohou front-endu je spolupráce s aplikační vrstvou, od které získává data, následně je finálně upraví do čitelné podoby a vypíše do připravené ***HTML*** šablony.

Hlavním jazykovým nástrojem pro tvorbu vizualizace je tedy jazyk ***HTML***, kde se ***CSS*** stylováním zajišťuje příjemné grafické prostředí pro uživatele. V případě dispečerské aplikace bylo využito ***JavaScriptového*** frameworku ***AngluarJS***. Důvodem použití je vytvoření příjemného uživatelského prostředí a snadná práce s nástroji pro tvorbu plánů jízd, aby aplikace byla skutečně přínosem a posunem od fyzického vyplňování jízd do papírového dispečerského deníku, jak bylo možné vidět v úvodu diplomové práce.

AngularjJS se stará o vlastní správu dat a navigaci napříč aplikačními moduly jako plánování jízd či administrace. Jde převážně o to, aby se zbytečně nezatěžoval server a nevykreslovala se stránka při každé akci, co uživatel udělá. Naopak cílem je obsluhovat jenom tu část business logiky, se kterou je momentálně pracováno.

3.3.1 Klíčové bower balíčky

Podobně jako aplikační vrstvu lze rozšířit pomocí balíčků v rámci ***NuGet*** balíčkovacího systému, tak front-end balíčky lze spravovat systémem pracujícím na podobném principu. Nazývá se ***bower***. Všechny použité knihovny jsou uloženy v souboru **bower.json**, kde je možné balíčky ručně spravovat. Balíčky lze spravovat též pomocí editoru, viz. **obrázek 11**. Mezi klíčové balíčky použité v rámci vývoje dispečerské aplikace patří ***Bootstrap***, ***AngularJS***, kterému bude převážně věnována zbylá část této kapitoly, a ***Moment*** pracující s formáty data a času v ***JavaScriptu***. Nesmíme zapomenout na komponentovou knihovnu ***Syncfusion*** obohacující šablony nadstandartními prvky.



Obrázek 11 - Editor bower balíčků, zdroj: vlastní

Bower usnadňuje a zpřehledňuje práci s ***JavaScritovými*** knihovnami, které by jinak musely být vkládány do projektu ručně. Díky výše zmíněným balíčkům bylo dosaženo požadované funkcionality k zajištění, co možné nejlepší zkušenosti uživatele z používání aplikace.

3.3.2 Angular moduly

Celá prezentační vrstva je rozdělena do několika funkčních celků, o něž se starají ***AngularJS*** moduly. Moduly vytvářejí iluzi znovu načítání webového obsahu, nicméně pouze zpracovávají požadovanou sadu dat, které následně vkládají do připravených ***HTML*** šablon.

Stěžejním modulem front-endu, stejně jako tomu bylo u *controlleru* v aplikační vrstvě, je modul pro práci s jízdami. Skládá se z dispečerského pohledu, kde dispečer nalezne přehled o naplánovaných jízdách. Druhou částí je editor jednotlivých jízd. Více o tomto modulu naleznete v kapitole **3.3.3 Modul jízdy**. Na **obrázku 12** je zobrazen dispečerský pohled.

Obrázek 12 - Modul jízdy, dispečerský pohled, zdroj: vlastní

Každý modul má svůj ***AngularJS*** *controller* aplikující front-end logiky, například pro komunikaci se serverem, dále sadu ***HTML*** šablon, se kterou *controller* pracuje a kam vkládá data, jež se následně zobrazí uživateli. Když se vrátíme zpět k ***ASP.NET Core MVC*** a k jeho *controllerům*, tak každá akce vrací zpět *view*, které by mělo odrážet výsledek této akce.

V tuto chvíli přebírá kontrolu nad zobrazováním modul ***AngularuJS***, jemuž je pouze specifikován prostor, kam vykresluje již zmiňované šablony s daty pro uživatele.

V rámci každého modulu je navrženo trasování pomocí ***Angular Route***. V konfiguraci modulu je specifikováno, jaká cesta odpovídá určitému *controlleru* a jeho šabloně. Například může jít o *controller* starající se o editor jízd, který má za úkol vykreslit uživateli sadu nástrojů pro tvorbu zamýšlené jízdy. Ve **zdrojovém kódu 11** v části *.config* se pomocí funkce specifikuje pro každou adresu její *controller* a šablona. V části definování*.module* nalezneme služby, které jsou do modulu injektovány a jsou v nich využívány.

```
(function () {
    "use strict";
    angular.module("admin-customers", ["simpleControls", "ngRoute",
      "ngAnimate", "ui.bootstrap", "ui.bootstrap", "cgBusy"])
        .config(function ($routeProvider, $locationProvider) {
            $routeProvider.when("/", {
                controller: "customersController",
                controllerAs: "vm",
                templateUrl: "views/admin/customer/customersView.html"
            });
            $routeProvider.when("/editor", {
                controller: "customerEditorController",
                controllerAs: "vm",
                templateUrl:
                  "views/admin/customer/customerEditorView.html"
            });
            $routeProvider.when("/editor/:customerId", {
                controller: "customerEditorController",
                controllerAs: "vm",
                templateUrl:
                  "views/admin/customer/customerEditorView.html"
            });
            $routeProvider.otherwise({ redirectTo: "/" });
            $locationProvider.hashPrefix("");
        });
})();
```

Zdrojový kód 11 - nastavení modulu

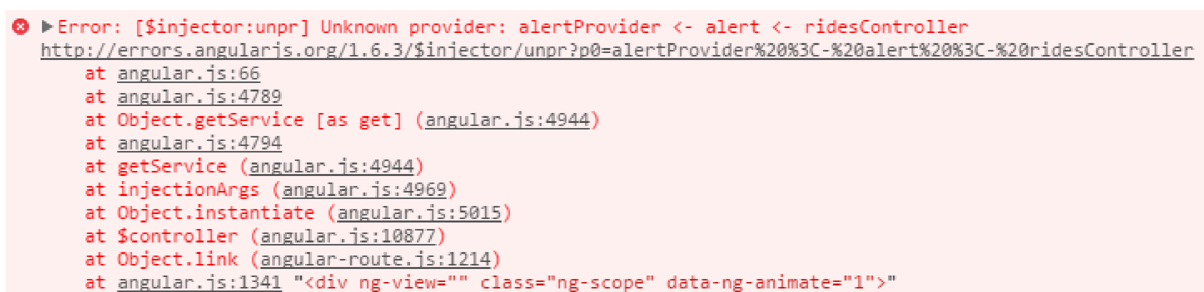
Další součástí modulu je validace jeho šablon před odesláním sady dat ke zpracování na server. Na rozdíl od validace modelových objektů v ***ASP.NET Core MVC*** pomocí anotací, v ***AngularJS*** modulech se validace provádí na úrovni ***HTML*** šablon. U každého elementu je specifikováno, jaké jsou na něj požadavky. Pokud není validní celá sada dat, například jízda, tak je uživateli znemožněno tato data poslat. Samozřejmostí jsou zprávy informující uživatele, co a kde je špatně. Tyto zprávy se zobrazí pouze v případě, že zadané parametry neodpovídají požadavkům. Ve **zdrojovém kódu 12** nalezneme příklad této validace. Byl zde použit speciální vstup pro zadávání data pomocí editoru. Podstatnou částí je, že input je obohacen o direktivy *required* a *dateTimeTo,* které zajišťují, že datum musí být vyplněno a že je větší než datum

odjezdu. Pokud není některá z podmínek splněna, zobrazí se uživateli zpráva definovaná v párovém tagu *<span>*. V případě splnění všech podmínek ***AngularJS*** umožní uživateli odeslat formulář s daty tím, že zaktivuje tlačítko pro odeslání.

```
<datepicker class="ride-picker" date-format="dd.MM.yyyy">
    <input class="form-control" id="dateTo"
    name="dateTo" readonly="readonly"
    ng-model="vm.pickerDateTo" type="text" required dateTimeTo />
    <span ng-show="rideEdit.dateTo.$error.required"
      class="text-warning">
    Musí být zadáno datum příjezdu.
    </span>
    <span ng-show="rideEdit.dateTo.$error.dateTimeTo"
      class="text-warning">
    Čas a datum příjezdu musí být až po času a datu odjezdu.
    </span>
</datepicker>
```

Zdrojový kód 12 - validace vstupního atributu

AngularJS, podobně jako ***ASP.NET Core***, umožňuje používání jiných knihoven či již vytvořených funkčních celků pomocí dependency injection popsané v práci Jana Holešínského. Díky dependency injection není třeba se starat o inicializaci knihoven a správu jejich zdrojů. To vše za programátora obstará poskytovatel závislostí (dependency provider). Stačí pouze specifikovat, jaké služby má modul využívat. Příklad lze nalézt ve **zdrojovém kódu 11** v části *.module* a specifikovat cestu ke knihovně v ***HTML*** šabloně v části *scripts*. Pokud se stane, že závislost zapomeneme přidat, ***AngularJS*** programátora informuje pomocí konzole v prohlížeči, že používaný modul nerozeznal, ať zkontrolujeme dostupné závislosti. Detail je možné nalézt na **obrázku 13**.



Obrázek 13 - Chybí závislost pro zobrazení alertu, zdroj: vlastní

3.3.3 Modul jízdy

Modul jízdy v kontextu prezentační vrstvy umožňuje interakci s uživatelem. Je první, kdo reaguje na uživatelovu akci, a poslední, kdo upravuje data před jejich zobrazením uživateli. Spolu s *cotrollerem* v aplikační vrstvě a entitou v databázové vrstvě představuje obsáhlé řešení pro přehled a plánování denních výkonů aut, řidičů a přívěsů a plně nahrazuje tužku a papír.

Používání tohoto modulu by mělo v důsledku vést k zefektivnění, zpřehlednění a v budoucnu i vyhodnocování každodenních procesů v dopravních firmách, a tím šetření zdrojů.

Základním pohledem v rámci modulu je dispečerské zobrazení. Je možné je filtrovat dle dispečerů, zvolit si den, případně denní výkony nastavit pro zobrazení řidičům, které se následně zobrazí v mobilní aplikaci. V neposlední řadě je zde tlačítko pro vytvoření nové jízdy, kde se po uskutečnění této akce zobrazí editor jízdy.

Hlavní komponentou je rozvrh jízd (*scheduler*), v němž jsou jednotlivé jízdy zobrazeny. Podle požadavku mělo zobrazení odpovídat papírovému dispečerskému deníku, kde ve vertikální poloze budou zobrazena vozidla a v horizontální denní čas. Objekty jízd pak odrážejí časový úsek, jenž zaberou a zobrazí se u příslušného vozidla. Vozidla jsou dále seskupena podle účelu použití.

Scheduler je interaktivní, čímž se značně ulehčuje práce s ním. Každá buňka odpovídá třicetiminutovému úseku. Pokud na ni uživatel dvakrát klikne, je přenesen do editoru, kde se předvyplní zvolený čas. Každý objekt jízdy je také interaktivní. Po kliknutí na něj se zobrazí detail jízdy, kde je možné přejít na editaci jízdy, případně ji smazat. V případě mazání jízdy je uživatel nejprve dotázán, zdali chce jízdu skutečně smazat. To samé pravidlo platí u zobrazení denního plánu řidičům, čímž se zamezuje chyba způsobená náhodným kliknutím. Náhled scheduleru je možné vidět na **obrázku 12**.

Použitý scheduler je komponentou z knihovny ***Syncfusion***, lze ho nastavit dle potřeb aplikace. Běžné věci jako šířka, výška v rámci ***HTML*** šablony nebo režim zobrazení, kde v našem případě je požadovaný pouze přehled na den, je možné nastavit při deklaraci. Lze také nadefinovat funkce, které budou vyvolány v případě zachycení určité události, např. mazání, dvojklik na buňku nebo zobrazení detailu jízdy.

Pokročilým nástrojem v rámci scheduleru je seskupování zdrojů do funkčních celků, jako tomu je v případě vozidel rozdělených do skupin podle výbavy či způsobu jejich použití. Pokročilou technikou je též šablonování objektů jízd nebo způsob zobrazení zdrojů (vozidel). Kompletní nastavení lze nalézt v **příloze A**.

Při prvním načítání stránky nebo změně data dojde k načtení příslušných jízd pro určité datum. Během načítání je uživateli ztmavena obrazovka, znemožněno pracovat se schedulerem a je informován o provádění činnosti. Po získání dat ze serveru je nutné data ručně napárovat do objektů, se kterými může scheduler pracovat. Mapování jízd je patrné na **zdrojovém kódu 13**. Jedná se o ruční přiřazení atributů získaných dat na atributy objektu scheduleru. Pokud by zde byl nástroj jako v ***ASP.NET Core AutoMapper***, tento zdrojový kód by zde nemusel být.

```
function bindRidesToScheduleEvents(responseData) {
    if (vm.rides.length > 0) {
        vm.rides = [];
    }
    for (var i = 0; i < responseData.length; i++) {
        var ride = {
            rideId: responseData[i].rideId,
            subject: responseData[i].customer.name + ": " +
                responseData[i].placeFrom + " - " +
                responseData[i].placeTo,
            description: responseData[i].placeFrom + " - " +
                responseData[i].placeTo,
            startsAt: moment(responseData[i].timeFrom).toDate(),
            endsAt: moment(responseData[i].timeTo).toDate(),
            rideState: responseData[i].rideState.ridesStateId,
            carId: responseData[i].car.carId,
            ownerId: responseData[i].car.owner.id,
            carTypeId: responseData[i].car.carTypeId,
            color: responseData[i].rideState.color
        }
        if (responseData[i].isReadyToShow) {
            ride.color = "#2E8B57";
        }
        vm.rides.push(ride);
    }
}
```

Zdrojový kód 13 - manuální mapování jízd na objekty scheduleru

Pro práci s vlastními jízdami byl dle dnešních požadavků vytvořen editor. V jeho rámci je umožněno vybrat datum, čas nakládky a vykládky, údaje se nezaznamenávají ručně, ale pomocí editorů. Aplikační zdroje, což jsou vozidla, přívěsná vozidla, řidiči a zákazníci se načítají do *select boxů*, z nichž lze vybírat. U řidičů byla provedena modifikace na *token box*, protože na jízdu může být naplánován více než jeden řidič.

Specifický výběr je i u zákazníka. Dispečer může plánovat jízdu pro zákazníka uloženého v adresáři, ale také může chtít zadat nového zákazníka přímo při vytváření. Pokud se rozhodne pro vytvoření nového, je zákazník přidán do adresáře ve chvíli, kdy je jízda ukládána do databáze. Tato funkce zpříjemňuje plánování jízdy, pokud se jedná o jednorázovou zakázku pro zákazníka. Po naplánování se dispečer může přepnout do administrace zákazníků a doplnit

chybějící údaje. Popsanou funkci si lze detailně prostudovat v **příloze B**, kde jsou zobrazeny metody pro práci se zákazníky v rámci editoru jízd.

Systém také hlídá při změně data a času jízdy, zdali načtené objekty již nebyly použity v rámci jiné jízdy, což zabraňuje naplánování vozidla, přívěsného vozidla nebo řidiče na jinou jízdu v rámci stejného časového úseku. Aplikace okamžitě po potvrzení změny data nebo času zavolá metodu, která tuto skutečnost ověří, jak je patrné ze **zdrojového kódu 14**. V rámci této metody byl také zpracován požadavek, aby datum příjezdu bylo větší než datum odjezdu. Pouze pokud je požadavek splněn, tak se zdroje znovu načtou.

```
vm.onDateTimeChange = function () {
    vm.times.dateTimeFrom = vm.ride.timeFrom.format("HH:mm");
    vm.times.dateTimeFrom = moment.utc(vm.times.dateTimeFrom +
        " " + vm.pickerDateFrom, "HH:mm DD.MM.YYYY");
    vm.times.dateTimeTo = vm.ride.timeTo.format("HH:mm");
    vm.times.dateTimeTo = moment.utc(vm.times.dateTimeTo +
        " " + vm.pickerDateTo, "HH:mm DD.MM.YYYY");

    if ($scope && $scope.rideEdit) {
        var isDateTimeFromValid =
        vm.times.dateTimeFrom.isBefore(vm.times.dateTimeTo);
        var isDateTimeToValid =
        vm.times.dateTimeTo.isAfter(vm.times.dateTimeFrom);
        $scope.rideEdit.dateFrom.$setValidity
            ("dateTimeFrom", isDateTimeFromValid);
        $scope.rideEdit.timeFrom.$setValidity
            ("dateTimeFrom", isDateTimeFromValid);
        $scope.rideEdit.dateTo.$setValidity
            ("dateTimeTo", isDateTimeToValid);
        $scope.rideEdit.timeTo.$setValidity
            ("dateTimeTo", isDateTimeToValid);
        if (!isDateTimeFromValid || !isDateTimeToValid) {
            return;
        }
    }

    getCars();
    getTrailers();
    getDrivers();
}
```

Zdrojový kód 14 - reakce na změnu datumu v rámci editoru jízdy

Dispečer má možnost vidět nejenom jemu přidělená vozidla, nýbrž i vozidla přidělená ostatním kolegům. Toto je důležité zejména ve chvíli, kdy dispečerovi zavolá zákazník s novou zakázkou na přepravu, ale on nemá již na daný termín dostupné vozidlo, nicméně zakázku chce přijmout, a tak využije vozidlo svého kolegy. Pokud taková jízda vznikne, je uložena ve stavu blokovaný, a dispečer, kterému je vozidlo přiděleno, je informován mailem, aby si jízdu zkontroloval

a následně schválil. Pokud je vozidlo dispečera, který jízdu vytvořil, je uložena ve stavu schváleno.

Nezávislým stavem jízdy je, kdy se má a nemá ukázat řidičům v mobilní aplikaci. V průběhu plánování se běžně stává, že se jednotlivé jízdy postupně mění a většinou plán na následující den je v pořádku až na konci dne. Z toho plyne, že dispečer nechce zobrazovat výkony řidičům, dokud není přesvědčen o konečném stavu. Tento stav může být nastaven pomocí *check boxu* v rámci vytváření nebo editace jízdy, ale také hromadně v rámci jednoho dne pomocí tlačítka při dispečerském pohledu.

Jízda může být zobrazena řidiči, pokud se nalézá ve stavu schváleno. Tato podmínka zabraňuje naplánovat jízdu a hned ji ukázat řidiči v případě, že ji plánuje dispečer s vozidlem, jež mu nebylo přiděleno. Totéž platí v případě zobrazení denního plánu řidičů, v němž jsou změněny stavy jen u schválených jízd.

```
vm.saveAndContinue = function () {
    vm.isSafeAndContinue = true;
    vm.submitRide();
}
vm.submitRide = function () {
    $http.put("api/rides/" + vm.ride.rideId, vm.ride)
    .finally(function () {
        if (vm.isSafeAndContinue) {
            vm.onDateTimeChange();

            if (vm.ride.customerId) {
                var result =
                vm.customers.filter(function (customer) {
                    return customer.customerId
                    === vm.ride.customerId;
                });

                if (result.length > 0) {
                    vm.ride.customer = result[0];
                } else {
                    vm.ride.customer = vm.customers[0];
                }
            }
            vm.isSafeAndContinue = false;
        }
    });
}
```

Zdrojový kód 15 - uložit a pokračovat

Pokud nastane situace, že dispečer obdrží zakázku na přepravu, která zabere více než jedno vozidlo, chce mít možnost již jednou připravenou jízdu přenést do jízdy nové, aby nemusel znovu zadávat stejné informace. Pro tuto situaci bylo vytvořeno tlačítko uložit a pokračovat,

kde se nejprve první jízda uloží, a pokud nastala tato událost, jsou data do polí znovu vyplněna, tyto metody naleznete ve **zdrojovém kódu 15**.

3.3.4 Administrátorské moduly

Jak již bylo nastíněno v aplikační vrstvě, dispečeři potažmo administrátoři musí mít možnost spravovat jednotlivé zdroje aplikace, jimiž jsou řidiči, vozidla, přívěsná vozidla či zákazníci. Pro každou sadu těchto zdrojových objektů byl vytvořen samostatný nezávislý modul. Tato modularita umožňuje snadné rozšiřování nejenom zdrojů, ale dalších různých modulů, pokud tato potřeba vyvstane.

Každý z těchto modulů se skládá ze dvou pohledů – přehledu a editoru, podobně jako tomu je u modulu jízd. Nicméně použité komponenty v rámci administrace jsou odlišné z důvodů rozdílného použití. V přehledu se nalézá seznam všech dostupných zdrojů daného druhu a základní informace o nich. Dále jsou umožněny základní operace s objekty jako vytvoření, úprava nebo mazání. Při zvolení vytvoření nebo úpravě se otevře editor, při mazání nejprve potvrzovací hláška. U uživatelských účtů je navíc přidána možnost pro reset hesla. Náhled přehledu uživatelských účtů je vidět na **obrázku 14**.

Obrázek 14 - Přehled uživatelských účtů, zdroj: vlastni

Načítání dat do seznamu uživatelských účtů je provedeno pomocí metody *GET*, kde je specifikována adresa vedoucí k příslušné akci API *controlleru* v aplikační vrstvě. Pokud je požadavek úspěšný, provede se automatické nakopírování objektů do kolekce v modulu. Příklad jednoduché metody pro načtení uživatelských účtů je viditelný ve **zdrojovém kódu 16**.

```
vm.getUserAccounts = function () {
    vm.loading = $http.get("/api/useraccounts")
        .then(function (response) {
            //success
            angular.copy(response.data, vm.users);
        },
        function (error) {
            //failure
            vm.errorMessage =
                "Failed to load data for user accounts" + error;
        });
}
```

Zdrojový kód 16 - metoda pro získání uživatelských účtů

Při mazání nebo u správy uživatelů resetu hesla je nejprve vyvolán potvrzovací dialog. Pokud ho uživatel potvrdí, je následně zavolána příslušná metoda, například *vm.resetPassword (id, closeEvent)* nebo *vm.deleteCar (carId, closeEvent)*, kde jsou v parametrech předány unikátní identifikátory a delegát funkce pro zavření dialogu. Nejprve je proveden příslušný požadavek na server. V případě úspěšného provedení operace je zavřen dialog a zobrazena informační zpráva o úspěšném provedení akce. Pokud ne, zobrazí se chybová hláška.

V případě vytvoření nového nebo úpravy stávajícího objektu je pomocí ***Angular Route*** načten editor s příslušnými vstupy pro úpravu nebo vytvoření určitého objektu. V rámci těchto editorů je vždy provedena validace dle požadavků, než je povoleno uživateli objekt uložit. Pro vybírání z číselníku, jako je typ vozidla, bylo použito *select boxů* a pro ostatní informace jsou použity textové vstupy.

Při ukládání úpravy nebo vytvoření nového objektu je zavolána příslušná funkce, kde se nejprve rozlišuje, zda se jedná o vložení nového objektu nebo úpravu stávajícího. Podle toho je odeslán příslušný požadavek na server, tedy *POST* pro vytváření a *PUT* pro úpravu. Metodu pro odeslání úpravy naleznete ve **zdrojovém kódu 17.** Při zpracování požadavku serverem je uživateli znemožněna činnost, ztmavena obrazovka a je informován o provádění akce. To zabraňuje nepředvídatelnému chování aplikace v případě, že by uživatel začal hned znovu editovat a ukládat změny ještě před dokončením požadavku. Pokud je požadavek úspěšný, je uživatel přesměrován zpět na přehled a informován o úspěšné změně v aplikaci. Pokud nastane chyba, je vypsaná a uživatel zůstává v editoru. Na podobném principu jsou vytvořeny všechny metody pro tuto činnost v rámci administrace i plánování jízd. V některých případech jsou obohaceny o další kroky, které je nutné provést po uložení, jako tomu je u jízd při uložit a pokračovat.

```
vm.submitCustomer = function () {
    if (vm.customer.customerId) { //updating customer
        vm.loading = $http.put("api/customers/"
                        + vm.customer.customerId, vm.customer)
            .then(function (response) {
                //success
                var msg = "Zákazník " + response.data.name
                    + " byl upraven.";
                $location.path("/", msg).search({ msg: msg });
            },
            function (error) {
                //failure
                vm.errorMessage = "Failed to save changes in customer"
                    + vm.customer.customerId + ": " + error;
            });
    } else { //creating new customer
        vm.loading = $http.post("api/customers", vm.customer)
            .then(function (response) {
                //success
                var msg = "Zákazník " + response.data.name
                    + " byl vytvořen.";
                $location.path("/", msg).search({ msg: msg });
            },
            function (error) {
                //failure
                vm.errorMessage = "Failed to create customer: " + error;
            });
    }
}
```

Zdrojový kód 17 - uložení zákazníka

3.4 Shrnutí a využití dispečerské aplikace

V rámci praktické části této diplomové práce a diplomové práce Jana Holešínského bylo navrženo softwarového řešení interních procesů v dopravních firmách. Celá aplikace se skládá z několika modulů. Jde o mobilní aplikaci pro Android a iOS zařízení, kde si řidiči mohou zobrazit informace o výkonech, které je v určitý den čekají. Důležitým předpokladem nejen pro mobilní aplikaci, ale i pro ostatní části centrální dispečerské aplikace je návrh a implementace datové vrstvy, kde dochází k uchovávání a získávání dat. Tyto dvě části jsou součástí práce kolegy.

V rámci návrhu a implementace této práce je vytvořit aplikační vrstvu, která umožňuje napojení na mobilní telefon a prezentační vrstvu, čímž dovoluje přijímat a zpracovávat požadavky odeslané z těchto modulů. Cílem je co nejrychlejší vyřízení požadavku a odeslání dat případně informace o provedení operace zpět na mobilní zařízení nebo na front-end.

Druhou částí praktické práce bylo vytvoření prezentační vrstvy (fron-endu) pro interakci s dispečery. Důraz byl kladen především na co nejsnazší a intuitivní ovládání a přechod od „tužky a papíru“ k digitální podobě. Tato část se stará o zachycení uživatelských akcí,

zavolání správné metody, odeslání požadavku a vyčkání na odpověď, která se následně vykreslí uživateli.

Jednotlivé vrstvy a moduly aplikace jsou na sebe napojeny a předávají si mezi sebou informace. Ale díky navržené architektuře a dodržení zásady volných vazeb je možné aplikaci nadále rozšiřovat nebo nahrazovat stávající moduly jinými technologiemi, pokud to bude nutné.

Celé řešení má za cíl zefektivnit interní procesy napříč dopravními firmami, aby nebylo nutné zbytečně čerpat zdroje. Aplikace slouží jako centrální bod, kde jsou uchovány informace o naplánovaných jízdách nebo historie provedených jízd. Tento software má pomoci zaměstnancům těchto firem koncentrovat se na svou práci bez obav, že by mohli ztratit přehled o pracovní náplni, kterou mají vykonávat.

ZÁVĚR

V úvodní části práce bylo nejdříve představeno téma nákladní dopravy jako celku, následované stručným přehledem nákladní dopravy v České republice. Z uvedených údajů jasně vyplynulo, že nejrozšířenějším druhem nákladní dopravy je silniční nákladní doprava. Vzhledem ke každoročnímu nárůstu objemu přepravovaného zboží stoupá i počet zakázek pro dopravní firmy, a tím i tlak na jednotlivé zaměstnance, aby byli co nejproduktivnější s co nejmenší chybovostí.

Z tohoto důvodu byla provedena analýza každodenních procesů ve firmě s důrazem na plánování jízd dispečery a jejich následnou komunikaci s řidiči. Z této analýzy vyvstalo několik kritických bodů, ve kterých by mohly nastat chyby. Problematickými místy jsou hlavně dnešní způsob zpracování a naplánování zakázky, kde se často stále používá „tužka a papír“ ve formě dispečerského deníku, a následná komunikace denního výkonu řidiči, který si buď výkon zapamatuje, nebo zapíše. Tyto neefektivní činnosti mohou vést ke ztrátě informací, zmatkům naplánovaných výkonů a s nárůstem vozidel i k přetížení odpovědných pracovníků.

Protože je nutné tyto problémy řešit, vyvstala motivace vytvoření centrálního softwarového řešení pro efektivní plánování výkonů a následnou komunikaci mezi dispečery a řidiči. Důraz byl především kladen na škálovatelnost, možnost následného rozšíření či rychlost a přívětivost pro uživatele. Proto byly použity moderní technologie pro implementaci aplikace jako ASP.NET Core pro serverovou část, JavaScriptový framework AngularJS pro rychlou odezvu na straně klienta a komponentové knihovny Bootstrap a Syncfusion zajišťující snadno ovladatelné uživatelské prostředí. Všechny tyto technologie jsou detailně popsány v kapitole 2.

Jak již bylo zmíněno výše, cílem bylo navržení komplexní aplikace, která se skládá z datové vrstvy pro centrální uchovávání dat, dispečerské aplikace pro plánování denních výkonů a aplikace pro chytré telefony na platformách Android a iOS sloužící pro komunikaci denních výkonů mezi řidičem a dispečerem. Praktická část této práce se věnuje návrhu a implementaci dispečerské aplikace s použitím těchto technologií. Zbylé dvě části jsou implementovány v rámci práce Jana Holešínského.

Dispečerská aplikace byla rozlišena na dvě vrstvy, a to aplikační a prezentační. Pro aplikační vrstvu bylo použito zejména platformy ASP.NET Core MVC, kde díky návrhovému vzoru Model-View-Controller (MVC) bylo možné aplikaci modularizovat a lépe testovat. Každá část aplikační vrstvy je odpovědná za jinou činnost. Hlavní náplní aplikační vrstvy jako celku je zachycení požadavků z prezentační vrstvy dispečerské aplikace a mobilních aplikací, které jsou

následně zpracovány a dále postoupeny datové vrstvě, kde se zajistí potřebná data. Data jsou poté zpracována a předána zpět do aplikační vrstvy.

Prezentační vrstva je zodpovědná za zobrazení navrhnutých šablon uživatelského rozhraní, načtení příslušných dat do těchto šablon a bezprostřední reakce na činnost uživatele, kde pro zajištění vytvoření šablon slouží komponentové knihovny Bootstrap a Syncfusion a pro komunikaci s uživatelem a práci s daty framework AngularJS. Prezentační vrstva byla rozdělena na funkční celky (Angular moduly). Hlavním modulem je plánování jízd, kde dispečer má možnost zobrazení plánu pomocí rozvrhu, který se podobá dispečerskému deníku v papírové podobě. Pro vytváření a editaci jízd byl vytvořen intuitivní editor. Následují moduly pro administraci firemních zdrojů (vozidel, přívěsných vozidel, zákazníků a uživatelů), které vždy obsahují přehled vytvořených zdrojů a editor pro jejich úpravu.

Díky této diplomové práci a diplomové práci Jana Holešínského vzniklo komplexní softwarového řešení, které centralizuje činnost pracovníků dopravních firem s cílem na zefektivnění jejich práce a eliminaci chyb, které mohlo způsobovat stávající řešení pomocí „tužky a papíru“. Datovou vrstvu a dispečerskou aplikaci je možné okamžitě nasadit a začít používat. Vhodným řešením se jeví využití cloudové technologie Azure od společnosti Microsoft. Mobilní aplikace lze vystavit na Google Play nebo Apple Store.

ZDROJE

[1] Vývoj nákladní dopravy v ČR. In: *Vítejte na Zemi* [online]. Praha: CENIA, 2013 [cit. 2017-04-11]. Dostupné z: http://vitejtenazemi.cz/cenia/index.php?p=vyvoj_nakladni_dopravy_v_cr&site=doprava

[2] Vnitrostátní nákladní silniční doprava. In: *Ministerstvo dopravy ČR* [online]. Praha: Ministerstvo dopravy ČR, 2016 [cit. 2017-04-12]. Dostupné z: https://www.mdcr.cz/Statistiky/Silnicni-doprava/Preprava/Vnitrostatni-nakladni-silnicni-doprava

[3] MELANSON, Anthony. What Industry 4.0 Means for Manufacturers. In: *Aethon: Autonomous Mobile Robots and Tracking Solutions* [online]. Pittsburgh: Aethon, 2015 [cit. 2017-04-18]. Dostupné z: http://www.aethon.com/industry-4-0-means-manufacturers/

[4] MASSI, Beth. Understanding .NET 2015. In: *MSDN Blogs* [online]. Seattle: Microsoft, 2015 [cit. 2017-04-19]. Dostupné z: https://blogs.msdn.microsoft.com/bethmassi/2015/02/25/understanding-net-2015/

[5] ROTH, Daniel, Rick ANDERSON a Shaun LUTTIN. Introduction to ASP.NET Core. In: *Docs.microsoft.com* [online]. Seattle: Microsoft, 2016 [cit. 2017-04-19]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/

[6] LANDWERTH, Immo. Introducing .NET Standard. In: *MSDN Blogs* [online]. Seattle: Microsoft, 2016 [cit. 2017-04-19]. Dostupné z: https://blogs.msdn.microsoft.com/dotnet/2016/09/26/introducing-net-standard/

[7] PETRUSHA, Ron, Maira WENZEL, Petr ONDERKA, et al. .NET Standard. In: *Docs.microsoft.com* [online]. Seattle: Microsoft, 2017 [cit. 2017-04-19]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/articles/standard/library

[8] SMITH, Steve. Overview of ASP.NET Core MVC. In: *Docs.microsoft.com* [online]. Seattle: Microsoft, 2016 [cit. 2017-04-19]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/mvc/overview

[9] What Is AngularJS? In: *Angularjs* [online]. Mountain View: Google, 2010 [cit. 2017-04-21]. Dostupné z: https://docs.angularjs.org/guide/introduction

[10] RAVULAVARU, Arvind. ANGULAR.JS. In: *Jackalstack* [online]. Hyderabad: Arvind Ravulavaru, 2014 [cit. 2017-04-21]. Dostupné z: http://jackalstack.com/training/mindtree/Angularjs/#/20

[11] Data Binding. In: *Angularjs* [online]. Mountain View: Google, 2010 [cit. 2017-04-21]. Dostupné z: https://docs.angularjs.org/guide/databinding

[12] Understanding Controllers. In: *Angularjs* [online]. Mountain View: Google, 2010 [cit. 2017-04-21]. Dostupné z: https://docs.angularjs.org/guide/controller

[13] KAUFMAN, Nir. AngularJS exploring routing options. In: *SlideShare* [online]. Mountain View: Google, 2014 [cit. 2017-04-21]. Dostupné z: https://www.slideshare.net/nirkaufman/angular-js-routing-options

[14] ONLINE DOCUMENTATION. In: *Syncfusion* [online]. Morrisville: Syncfusion, 2001 [cit. 2017-04-26]. Dostupné z: https://help.syncfusion.com/

PŘÍLOHY

Příloha A – Nastavení scheduleru 58

Příloha B – Načtení zákazníků a události zachycené při jeho změně v rámci editoru jízd 59

Příloha C – Diagram tříd 60

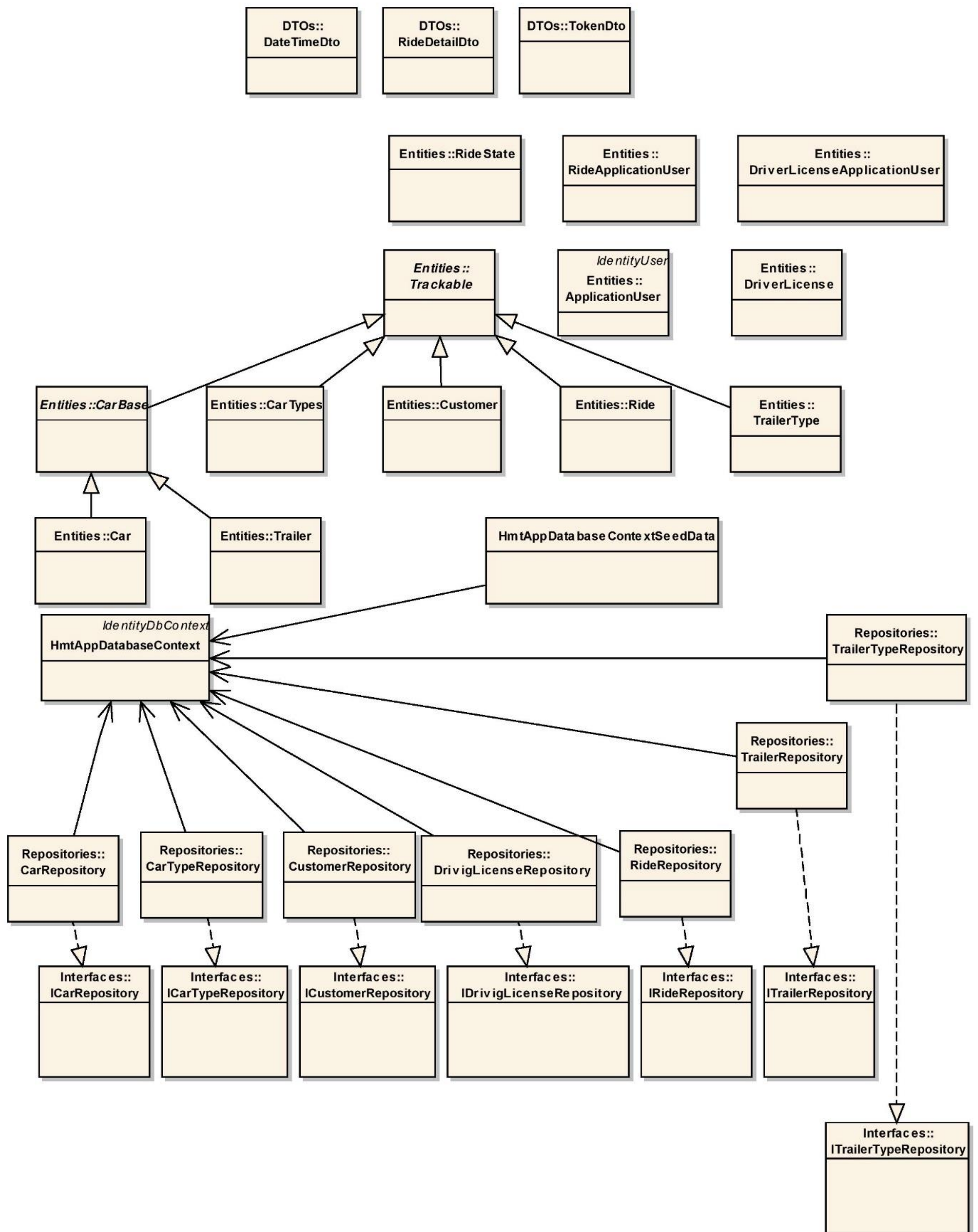
Příloha A – *Nastavení scheduleru*

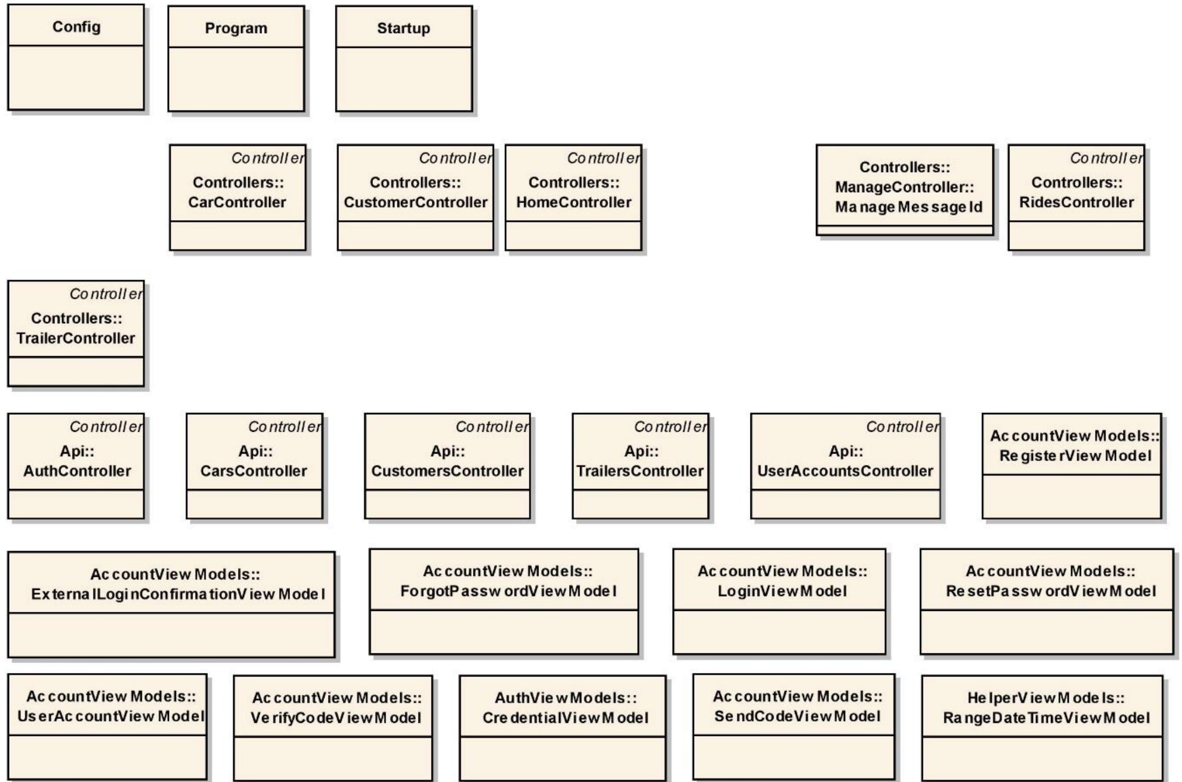
```
<ej-schedule id="RidesSchedule"
             e-appointmentTemplateId="#eventTemplate"
             e-resourceHeaderTemplateId="#resTemplate"
             e-workTemplateId=""
             e-allowDragAndDrop="false"
             e-enableAppointmentResize="false"
             e-locale="cs-CZ"
             e-width="100%"
             e-height="vm.scheduleHeight"
             e-orientation="horizontal"
             e-currentdate="vm.viewDate"
             e-views="vm.views"
             e-currentview="vm.calendarView"
             e-group="vm.group"
             e-create="onCreate"
             e-appointmentWindowOpen="vm.editEvent"
             e-appointmentClick="vm.editPopup"
             e-appointmentRemoved="vm.onAppointmentRemoved"
             e-cellClick="vm.editEvent"
             e-cellDoubleClick="vm.editEvent"
             e-appointmentEditSeriesButtonClicked="vm.validateRide"
             e-timeMode="24"
             e-appointmentsettings-datasource="vm.rides"
             e-appointmentsettings-id="rideId"
             e-appointmentsettings-resourcefields="carTypeId, carId"
             e-appointmentsettings-subject="subject"
             e-appointmentsettings-starttime="startsAt"
             e-appointmentsettings-endtime="endsAt"
             e-appointmentsettings-description="description"
             e-appointmentsettings-applytimeoffset="false"
             e-appointmentsettings-allday="allDay"
             e-appointmentsettings-recurrence="recurrence"
             e-appointmentsettings-recurrencerule="recurrenceRule">
    <e-resources>
        <e-resource e-allowmultiple="false"
                    e-field="carTypeId"
                    e-title="CarType"
                    e-name="CarTypes"
                    e-resourcesettings="vm.carTypeResourcedata">
        </e-resource>
        <e-resource e-allowmultiple="false"
                    e-field="carId"
                    e-title="Car"
                    e-name="Cars"
                    e-resourcesettings="vm.carResourcedata">
        </e-resource>
    </e-resources>
</ej-schedule>
```

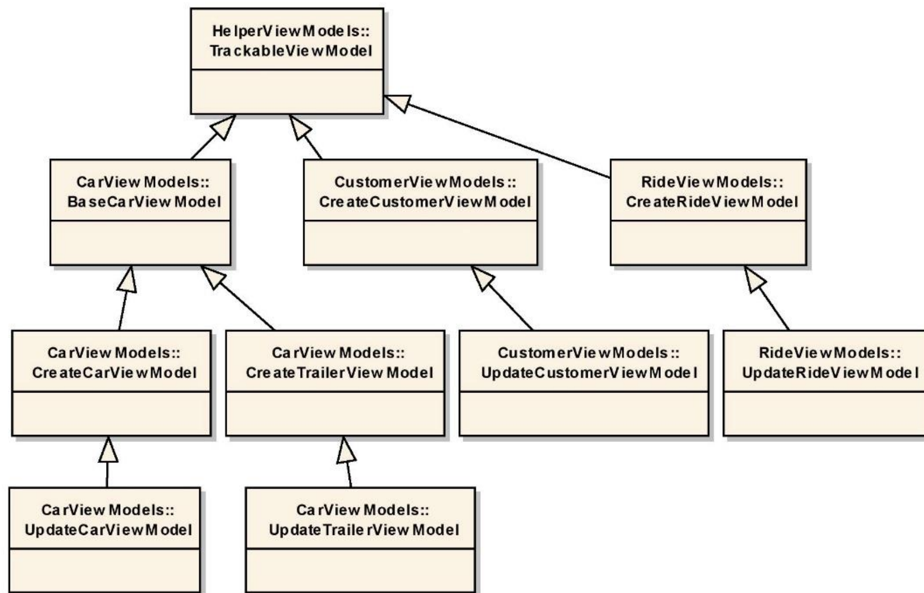
Příloha B – *Načtení zákazníků a události zachycené při jeho změně v rámci editoru jízd*

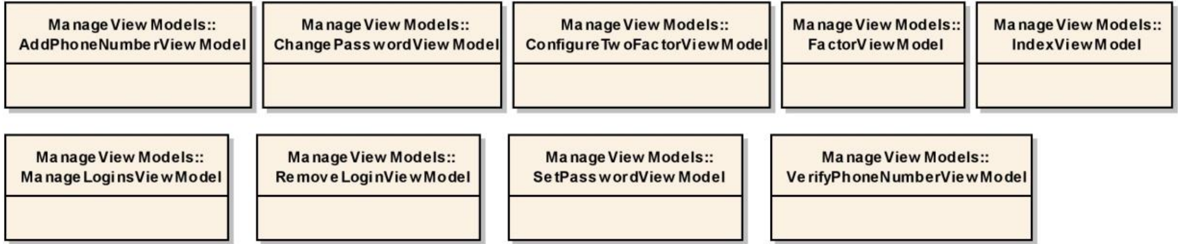
```
function getCustomers() {
    $http.post("/api/customers/get-all", (vm.rideId ? vm.rideId : null))
        .then(function (response) {
            //success
            if (vm.customers.length > 0) {
                vm.customers = [];
            }
            var customer = {
                name: "---Zadejte zákazníka---"
            };
            vm.customers.push(customer);
            for (var i = 0; i < response.data.length; i++) {
                customer = {
                    customerId: response.data[i].customerId,
                    name: response.data[i].name
                }
                vm.customers.push(customer);
            }
            if (!vm.ride.customer) {
                vm.ride.customer = vm.customers[0];
            }
        },
        function (error) {
            //failure
            vm.errorMessage = "Failed to load data for customers"
            + error;
        });
}
vm.onCustomerNameChanged = function () {
    if (vm.customerName) {
        vm.customers[0].name = vm.customerName;
        vm.ride.customer = vm.customers[0];
        vm.ride.customerId = null;
    } else {
        vm.customers[0].name = "---Zadejte zákazníka---";
        vm.ride.customer = vm.customers[0];
        vm.ride.customerId = null;
    }
}

vm.onCustomerChanged = function () {
    if (vm.ride.customer.customerId) {
        vm.ride.customerId = vm.ride.customer.customerId;
    } else {
        vm.ride.customerId = null;
    }
    if (vm.ride.customer
        && vm.ride.customer.name !== "---Zadejte zákazníka---") {
        vm.customerName = vm.ride.customer.name;
    } else {
        vm.customerName = "";
    }
}
```

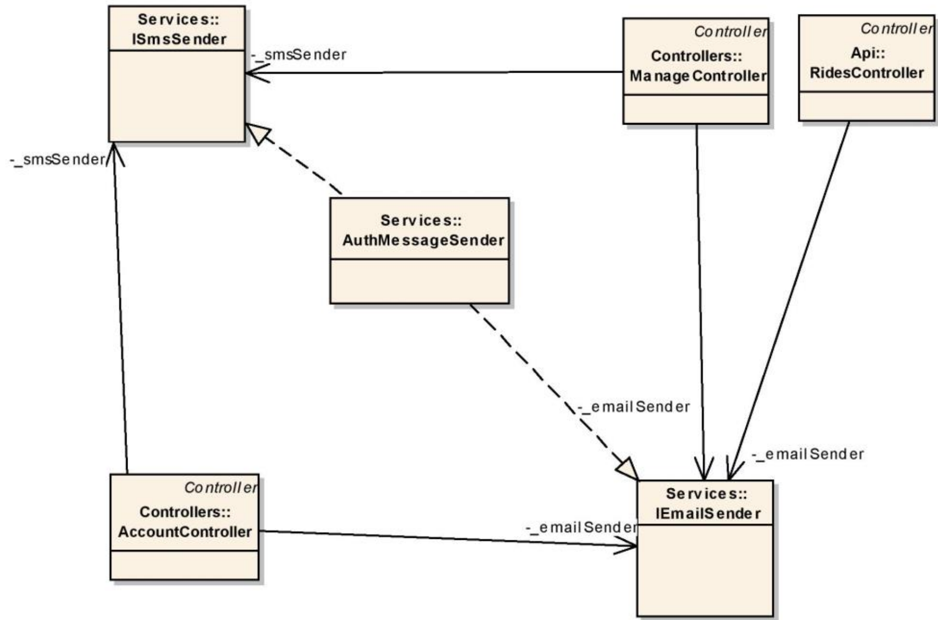
Příloha C – *Diagram tříd*




Config
Program
Startup
Controller
Controllers::
CarController
Controller
Controllers::
CustomerController
Controller
Controllers::
HomeController
Controllers::
ManageController::
ManageMessageId
Controller
Controllers::
RidesController
Controller
Controllers::
TrailerController
Controller
Api::
AuthController
Controller
Api::
CarsController
Controller
Api::
CustomersController
Controller
Api::
TrailersController
Controller
Api::
UserAccountsController
AccountViewModels::
RegisterViewModel
AccountViewModels::
ExternalLoginConfirmationViewModel
AccountViewModels::
ForgotPasswordViewModel
AccountViewModels::
LoginViewModel
AccountViewModels::
ResetPasswordViewModel
AccountViewModels::
UserAccountViewModel
AccountViewModels::
VerifyCodeViewModel
AuthViewModels::
CredentialViewModel
AccountViewModels::
SendCodeViewModel
HelperViewModels::
RangeDateTimeViewModel


HelperViewModels::
TrackableViewModel
CarViewModels::
BaseCarViewModel
CustomerViewModels::
CreateCustomerViewModel
RideViewModels::
CreateRideViewModel
CarViewModels::
CreateCarViewModel
CarViewModels::
CreateTrailerViewModel
CustomerViewModels::
UpdateCustomerViewModel
RideViewModels::
UpdateRideViewModel
CarViewModels::
UpdateCarViewModel
CarViewModels::
UpdateTrailerViewModel


ManageViewModels::
AddPhoneNumberViewModel
ManageViewModels::
ChangePasswordViewModel
ManageViewModels::
ConfigureTwoFactorViewModel
ManageViewModels::
FactorViewModel
ManageViewModels::
IndexViewModel
ManageViewModels::
ManageLoginsViewModel
ManageViewModels::
RemoveLoginViewModel
ManageViewModels::
SetPasswordViewModel
ManageViewModels::
VerifyPhoneNumberViewModel


Services::
ISmsSender
-_smsSender
Controller
Controllers::
ManageController
Controller
Api::
RidesController
-_smsSender
Services::
AuthMessageSender
-_emailSender
-_emailSender
Controller
Controllers::
AccountController
-_emailSender
Services::
IEmailSender