Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Zpracování a vizualizace diagnostických hlášení z řídicího systému kolejového vozidla

Diplomová práce

Autor: Bc. Petr Stříteský
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Odborný konzultant: Ing. Stanislav Marek, AŽD Praha, s.r.o.

Hradec Králové květen 2021

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval/zpracovala samostatně a s použitím uvedené literatury.

V Hradci Králové dne 30.4.2021 Petr Stříteský

Poděkování:

Děkuji vedoucímu diplomové práce doc. Ing. Filipovi Malému, Ph.D. za metodické vedení práce a poskytnutí konzultací během jejího vzniku. Dále bych chtěl poděkovat odbornému konzultantovi, panu Ing. Stanislavovi Markovi ze společnosti AŽD Praha, s.r.o. taktéž za odborné vedení práce, nastínění základních problémů k řešení, připomínkování a testování a finální zhodnocení výstupů. Velké díky patří také kolektivu vývojového pracoviště VP01 AŽD Praha s.r.o. v čele s mým nadřízeným, Dr. Ing. Alešem Lieskovským, za vytvoření podmínek a prostoru pro studium při zaměstnání. Poděkování patří také společnosti MSV Elektronika, s.r.o., za poskytnutí dat nezbytných pro vznik práce. Dále bych rád poděkoval své rodině a blízkým, zejména pak také své přítelkyni Kateřině Neubauerové za nebetyčnou podporu, toleranci a poskytnutí příjemných podmínek pro vznik této práce.

Anotace

Cílem diplomové práce je uvést čtenáře do problematiky vývoje webových aplikací na platformě .NET a diagnostiky v oblasti mikroprocesorového řízení kolejových vozidel. Dále nastínit a vyvinout dle stanovených požadavků koncept webové aplikace, která dokáže začlenit diagnostická data do databáze a dále je analyzovat, třídit a vyhodnocovat v širším kontextu, popsat podrobně jednotlivé použité metody včetně základního historického kontextu a posléze na konkrétních příkladech demonstrovat jejich využití během vývoje aplikace. Vzniklá aplikace bude naplněna testovacími daty, na kterých bude demonstrována její funkce a proběhne její testování včetně všech použitých metod. Na základě toho dojde k interpretaci získaných dat a závěrečnému zhodnocení úspěšnosti a smysluplnosti takto nastíněného konceptu.

Annotation

Title: Processing and visualisation of diagnostic messages from a railway vehicle control system

The primary goal of this diploma thesis is to introduce the reader to the problematics of web application development on the .NET platform and diagnostics on the field of microprocessor control of railway vehicles. Furthermore, according to given requirements, the goal is to draft and develop the concept of a web application that will be capable of integrating diagnostic data to a database and perform specific operations such as analysis, sorting and evaluation of the data in a broader context. All the used methods will be described in detail, including the historical context and their use will be demonstrated specifically during the development of the application. The resulting application will be given a test data, which will demonstrate its functionality and test all the used methods. Based on this, the obtained data will be interpreted to evaluate the meaningfulness of using proposed concept in practice.

Obsah

1 Úvod ... 1
1.1 Cíl práce ... 2
1.2 Metodika zpracování ... 3
2 Přehled použitých metod ... 4
2.1 Programovací jazyk C# ... 4
2.1.1 Základní syntaxe ... 5
2.2 .NET technologie ... 6
2.2.1 Specifika a výhody ... 7
2.3 ASP.NET ... 8
2.3.1 Specifika a výhody ... 9
2.4 Razor Pages ... 10
2.5 HTML a CSHTML ... 11
2.6 CSS – kaskádové styly ... 11
2.6.1 Bootstrap ... 12
2.7 LINQ a Entity Framework ... 13
2.8 Microsoft Visual Studio ... 14
3 Poruchové logy kolejových vozidel ... 18
4 Vývoj a popis vnitřní logiky aplikace (back-end) ... 20
4.1 Požadavky a vstupy ... 20
4.2 Návrh ... 21
Tabulka chybových logů ... 21
Tabulka chyb ... 21
Tabulka rejstříků ... 22
Tabulka rejstříkových chyb ... 22
Tabulka řad (typů) vozidel ... 22
Tabulka jednotlivých vozidel ... 22
4.3 Databázové schéma ... 23
4.4 Souborová struktura ... 24
4.5 Oblast chybových logů ... 25

4.5.1 Chybové logy .his 25
4.5.2 Vytvoření modelů chybových logů 26
4.5.3 Vytvoření stránek pro zobrazení a práci s entitami....................27
4.5.4 Logika zpracování chybového logu 28
4.6 Oblast rejstříků chyb 31
4.6.1 Soubory rejstříků .csv 31
4.6.2 Vytvoření modelů rejstříků 32
4.6.3 Vytvoření stránek pro zobrazení a práci s entitami....................34
4.6.4 Logika zpracování rejstříku 35
4.7 Dekódování poruch 36
4.7.1 Vytvoření modelů řad a vozidel 37
4.7.2 Vytvoření stránek pro zobrazení a práci s entitami....................38
4.7.3 Vyhodnocení poruchy na základě spárování s rejstříkem....................40
4.7.4 Dekódování zdrojového bitu 42
4.7.5 Výsledné zobrazení detailu poruchy 44
4.8 Filtrování dat 44
4.9 Export dat 45
4.10 Statistika 47
5 Uživatelské rozhraní (front-end) 51
5.1 Implementace datového modelu do zobrazované stránky 51
5.1.1 Aplikační logika 52
5.1.2 Viditelná část stránky 53
5.2 Stylování 56
5.2.1 Sdílená část zobrazení 56
5.2.2 Charakteristická část zobrazení 62
6 Shrnutí výsledků 66
7 Závěry a doporučení 67
8 Seznam použité literatury 69
9 Přílohy 72

Seznam obrázků

Obr. 1 - změna vlastností prvku při najetí myší 12

Obr. 2 - vývoj indexu využívanosti IDE dle PYPL od roku 2005 do současnosti 14

Obr. 3 - uživatelské rozhraní MS Visual Studio 15

Obr. 4 - databázové schéma aplikace 23

Obr. 5 - schéma souborové struktury projektu 24

Obr. 6 - stránka s chybovými logy rozšířená o tlačítko "Nahrát log" 28

Obr. 7 - část stránky s formulářem pro nahrání souboru 28

Obr. 8 - chybová hláška v případě pokusu o nahrání špatného typu souboru 29

Obr. 9 - ukázka části původního .xlsx souboru s popisem poruch 32

Obr. 10 - náhled stránky prohlížení rejstříku 34

Obr. 11 - formulář pro zadání dat o vozidle do databáze 39

Obr. 12 - odkaz "Podrobnosti" v řádku dané poruchy 40

Obr. 13 - cíl odkazu na zobrazení detailu poruchy 40

Obr. 14 - výstup konzole databázového dotazu pro vyhledání definice poruchy 42

Obr. 15 - porucha "OMx" v rejstříku řady 753.6 42

Obr. 16 - zobrazení vyhodnocené poruchy 44

Obr. 17 - formulář pro vyhledávání v tabulce Fault 45

Obr. 18 - graf četnosti výskytu jednotlivých poruch u všech vozidel 48

Obr. 19 - záhlaví uživatelského rozhraní 57

Obr. 20 - Změna vlastností prvků při najetí myší 58

Obr. 21 - výsledný návrh uživatelského rozhraní aplikace 60

Obr. 22 - vykreslení indikátoru načítání 61

Obr. 23 - zdrojový kód uvítací stránky aplikace 62

Obr. 24 - stylování ovládacích tlačítek 62

Obr. 25 - výsledná podoba stránky Statistika 65

Seznam tabulek

Tabulka 1 - tabulka překódování zdrojového bitu 43

1 Úvod

V následující diplomové práci bude okrajově navázáno na autorovu práci bakalářskou, ve které se věnoval tématice automatizace řízení kolejových vozidel, konkrétně vývoji servisního zobrazení systému ATO over ETCS. Automatizace řídicích procesů vede k úspoře nákladů, zvýšení efektivity daných procesů a přispívá k eliminaci lidského faktoru v řízení, což pozitivně přispívá ke snížení míry nehodovosti. Nedílnou součástí informačních systémů v řízení však je také diagnostika řízeného systému a všech jeho částí, neboť správná funkce systému závisí na správné funkci všech jeho dílčích komponent, na správném měření sledovaných vstupních i výstupních hodnot a na schopnosti případné odchylky a chyby detekovat a adekvátním způsobem na ně zareagovat. Tato diagnostika chyb, konkrétně u kolejových vozidel s řídicími systémy společnosti MSV Elektronika, s.r.o., je hlavním tématem této práce, která se konkrétně zabývá orientačním vytvořením platformy pro diagnostiku chybových logů z takto řízených lokomotiv. Vzniklá platforma umožní ukládat chybové logy do databáze, dekódovat je, třídit a provádět nad nimi určené statistické operace včetně textového i grafického výstupu, například v podobě grafů. Z hlediska provedení se jedná o webovou aplikaci, která by měla uživateli nabídnout jednoduché a přehledné uživatelské rozhraní. Práce se skládá ze dvou hlavních celků – teoretické části, která je spíše popisná a analytická, a části praktické, která popisuje samotný vývoj, funkci aplikace a její testování. První část práce je věnována podrobnějšímu vhledu do problematiky řízení a automatizace v oblasti železniční dopravy, následně jsou podrobně popsány očekávané cíle, prostředky a nástroje k jejich dosažení. Dojde zde k analýze samotných chybových logů a metodice jejich vzniku, nebo například popisu programovacích jazyků, knihoven a softwaru, který byl při vývoji aplikace použit. Praktická část práce je zaměřena zejména na samotné programování aplikace, popis a zdůvodnění výběru použitých metod na konkrétních příkladech, je zde demonstrována a ověřena jejich funkce. Popsáno je zde také samotné uživatelské rozhraní včetně metodiky jeho struktury. Před dokončením vývoje bude aplikace poskytnuta odbornému konzultantovi, popřípadě jeho

prostřednictvím přímo firmě MSV Elektronika k posouzení, testování a připomínkování. Tato zpětná vazba poslouží jako základní podklad pro finální zhodnocení samotné práce, které spočívá v rozhodnutí, zda má smysl další vývoj nastíněného konceptu smysl. Pakliže nastíněný koncept uživatele osloví, bude mu nabídnuta možnost dalšího vývoje a přizpůsobení jeho konkrétním požadavkům. Vzhledem k jisté míře odbornosti některých popisů a postupů je v této diplomové práci kladen důraz na základní úvod do problematiky a srozumitelný popis každého řešeného problému. Přesto se ze strany čtenáře předpokládá základní znalost problematiky programování v jazyce C#, vývoje webů a webových aplikací a kolejové dopravy.

1.1 Cíl práce

Cílem práce, jak již bylo nastíněno, je vývoj konceptu platformy na bázi webové aplikace, která poslouží koncovému uživateli ke zpracování chybových logů z kolejových vozidel řízených systémem společnosti MSV Elektronika. Mezi dílčí požadavky na funkci a podobu aplikace patří:

- Zpracování hlavičky a obsahu textového logu z lokomotivy,
- Zatřídění obsahu do databáze a logického propojení s hlavičkou,
- Podrobné zpracování jednotlivých logů: Dekódování názvu, času, trvání, příčiny a doplňujících stavových informací,
- Vizualizace logů – tvorba filtrů dle času, hlášení, pohybu, tahu a kontextu,
- Dekódování doplňkových dat,
- Moderní, lehké a přehledné uživatelské rozhraní,
- Průzkum možností exportu dat a jeho implementace.

Výsledná aplikace by tak měla prokázat, že i velmi technicky charakteristická data v podobě chybových logů lokomotiv je možné pro uživatele přehledně a srozumitelně interpretovat. Předpokládá se, že vhodně interpretovaná data by mohla uživateli poskytnout důležité informace ohledně použitých zařízení a technologií a pomoci mu při optimalizaci a výběru nejvhodnějších řešení pro současně i v budoucnu instalované systémy a zařízení. Takovým výstupem by mohla

být kupříkladu informace, jaké jsou nejčastější příčiny konkrétní chyby, které dílčí části systému vykazují nejvyšší chybovost a za jakých okolností k chybám dochází. To by ve finálním důsledku mohlo pomoci nalézt řešení, jak vzniku těchto chyb zabránit. Takto vyvozené závěry by měly představovat základní výzkumné výstupy práce.

1.2 Metodika zpracování

Zpracování stanoveného tématu předpokládá pokročilou znalost použitých metod. Mezi ty patří například pokročilá znalost programovacího jazyka C#, ve kterém je programována veškerá aplikační logika. Vzhledem k tomu, že se však jedná o webovou aplikaci, je zapotřebí vycházet také ze základů tvorby webu, tedy zejména jazyka HTML a CSS, popřípadě Javascriptu. Důležitou součástí je také samotná databáze, ve které jsou data uložena a nad kterou se budou provádět veškeré dotazy. K tomu je zapotřebí znalost základů databází a jejich fungování. Bylo rozhodnuto, že výsledná aplikace bude vzhledem ke svému charakteru kombinujícím webové rozhraní a programovací jazyk C# vznikat na jádře platformy .NET Core verze 3. K programování a debuggingu bude použito vývojové prostředí Microsoft Visual Studio Professional 2019, které pro vývoj takových aplikací nativně poskytuje veškerou podporu. U veškerých použitých procesů a metod dojde k jejich základnímu popisu včetně historického kontextu a principu fungování. Ze zvolených metod a prostředků vychází také vybraná odborná literatura, která zahrnuje skripta, knihy, dokumentace a webové zdroje či návody týkající se daných programovacích jazyků a frameworku .NET. Tato literatura je volena s ohledem na to, aby dokázala s přesahem pokrýt jak teoretickou, tak praktickou část práce. Značná část odborné literatury tak poslouží také k získání znalostí pro vytvoření praktického výstupu práce. Před praktickou aplikací každé metody bude zprvu provedena její analýza, aby došlo k pochopení jednotlivých vnitřních jevů a souvislostí. Ta povede ke znalosti, zda je použití dané metody vhodné a jakým způsobem ji využít, aby práce byla co nejvíce efektivní a nedocházelo ke zbytečně složitým konstrukcím či situacím, kdy bude nutné se pro její nevhodnou aplikaci vracet o několik kroků zpět.

2 Přehled použitých metod

Níže dojde k postupnému zaměření na dílčí programovací jazyky, frameworky a software použitý při vývoji aplikace.

2.1 Programovací jazyk C#

Programovací jazyk C# je univerzální, objektově orientovaný programovací jazyk vyvinutý společností Microsoft pro platformu Windows jakožto součást .NET iniciativy. Jazyk je schválen Mezinárodní organizací pro normalizaci (ISO) Evropskou asociací výrobců počítačů (ECMA) a svojí syntaxí vychází z jazyka C, C++, díky objektové orientaci je však například velmi podobný jazyku Java. (1)

Historie jazyka sahá do roku 2002, jedná se tak o jeden z relativně mladších programovacích jazyků. Nejedná se však tak úplně o nový jazyk, spíše o jakési přizpůsobení standardního jazyka C aktuálním standardům a zejména světu objektově orientovaného programování. (2) O to se v historii snažil nejen formou různých rozšíření také jazyk C++, avšak s poněkud odlišným přístupem, který nemusel být každému programátorovi sympatický. Brzy jej tak předčil například jazyk Java, který se v žebříčku nejpopulárnějších jazyků udržel až do roku 2020. (3) Začátkem tisíciletí se tak Microsoft rozhodl pro vývoj zcela nového jazyka, který bude schopný jazykům typu Java konkurovat. První verze tak byla představena v již zmíněném roce 2002 spolu s novou verzí vývojového nástroje Microsoft Visual Studio. Od svého vydání se programovací jazyk nadále vyvíjí, stejně tak jako konkurenční Java. V prvotních fázích vývoje pochytil jazyk od Javy spoustu vlastností, později ale také Java převzala jistá specifika od C#.

„Snad nikomu neunikl spor Microsoft versus Sun, jehož důsledkem došlo více méně k zániku jazyka Java MS provenience. Visual J++, jak se verze z Redmondu jmenovala, měla na architekturu jazyka C# nemalý vliv díky svým kvalitám v mnohých oblastech. Přechod od Javy k C# byl pro Microsoft způsob jak urovnat vleklý spor a mít jazyk obdobných kvalit.“

(4 str. 77)

Během času se pak C# začal od Javy čím dál více vzdalovat a přestával se zároveň soustředit výhradně na objektovou orientaci. Ve druhé verzi, která byla vydána roku 2005, do jazyka například přibyly iterátory, ve třetí verzi vydané roku 2007 to pak byly výrazové stromy (expression trees), metody rozšíření, dotazy a lambda výrazy. Později přibyla například dynamická klíčová slova, která výrazně pomáhají programátorům při debuggingu, tedy řešení problémů. Od páté verze vydané roku 2012 se C# rozšířil o podporu asynchronního programování pro zvýšení pohodlí při spouštění dlouhých kódů. K dnešnímu dni je aktuální sedmá verze vydaná roku 2017. Od této verze je jazyk možné používat v kombinaci s .NET Core – open-source softwarovým frameworkem, který je nástupníkem .NET frameworku a je schopný kromě Windows běžet také na operačních systémech Linux a macOS. (2)

2.1.1 Základní syntaxe

Jazyk C# je poměrně intuitivní a obsahuje řadu vestavěných funkcí, které programátorovi usnadní práci. Je však třeba zmínit i to, že je díky tomu mnohem rozsáhlejší a je nutné počítat s vyšší paměťovou zátěží. C# například sice podporuje bitové operace, práce s pamětí na bitové úrovni je však pro programátora poměrně složitý úkol. I s takovými operacemi je však v případě zpracování některých dat možné se setkat a v konkrétním případě chybových logů lokomotiv s řídicím systémem MSV může nutnost čtení bitových polí vzniknout.

Se základním rozdílem oproti jazyku C se zpravidla setkáme již na začátku kódu na poli deklarací. Zatímco C hojně využívá hlavičkové soubory, které jsou implementovány pomocí direktivy `#include`, v C# je používána direktiva `using`. Základní datové typy vycházejí opět z C, některé však nesou odlišný název.

Například 8-bitový integer, v C označovaný jako *int8, int8_t* nebo *signed char*, nese v C# název byte, pro 16-bitový integer je pak namísto typu *int16* tento typ nazván *short*.

Nutno také zmínit, že C# již naplno pracuje s dynamickou alokací paměti. C# tak umožňuje ve funkcích použít například implicitně typovanou proměnnou *var*. O čištění paměti se stará namísto programátora garbage collector. (4) (5)

2.2 .NET technologie

Samotný název technologie .NET by mohl uživatele na první pohled mírně zmást. Toto označení však nemá mnoho společného s webovou doménou, jedná se spíše o náznak společnosti Microsoft, že tato platforma je moderní platformou s velkým potenciálem pro budoucí aplikace, která využívá rozdělení zpracování výpočetních operací mezi klienta a server. (6) Dalo by se říci, že cílem vzniku platformy .NET bylo vytvořit jednoduché, ale mocné nástroje pro programování ve Windows. Právě díky vývoji této platformy vznikl speciálně pro ni jazyk C#, který je od začátku navržen tak, aby s ní byl plně kompatibilní a implementoval naplno veškeré její funkce a možnosti. (6)

Spolu s rozvojem nejen objektově orientovaného programování, ale také požadavků na kompatibilitu a univerzalitu různých platforem začaly být tradiční programovací jazyky rozšiřovány o další a další funkce. To však často vedlo k tomu, že nemalé množství dodatečných modulů, které se svojí funkcí mnohdy vzájemně překrývaly, začalo narůstat do enormních rozměrů a tyto jazyky se svými rozšířenými variantami působily těžkopádně, neorganizovaně a složitě. Příkladem mohou být například modely MFC knihoven, které byly pro jazyky C, C++ a Visual Basic značně odlišné, respektive neměly prakticky nic společného. To vše vedlo ke snahám o zásadní změnu, kteréžto se staly hlavní příčinou přepracování dosavadního přístupu směrem k více koncepčnímu řešení. Cílem tedy bylo vytvořit jakési jednotné podhoubí, ze kterého bude možné při vstupu různých potřebných živin vytvořit různé organismy, které však budou sdílet své nejpevnější základy s ostatními. Příkladem takového jednotného řešení může být například koncept

standardizované klasifikace značení chyb při běhu aplikací. Framework .NET zavádí ve všech knihovnách značení chyb formou výjimek, stejně tak jako přináší určité možnosti jejich ošetření či zpracování. Oproti minulosti, kdy bylo takřka nemožné s chybou aplikace jakkoli pracovat a objev neočekávané či chybné hodnoty zpravidla vedl k pádu celého programu, má nyní programátor možnost se na vznik těchto chyb nějakým způsobem připravit a při jejich vyvolání zvoleným způsobem zareagovat. Díky kompatibilitě prostředků je také například možné výjimku vzniklou v komponentě psané v jednom jazyce zpracovat uvnitř komponenty psané v jazyce zcela jiném, neboť kód této výjimky je pro oba jazyky jednotný a předání její hodnoty je stejné, jako předání jakékoliv jiné proměnné. Nově vzniklá platforma však často kontroluje programátora ještě dříve, než k vyvolání výjimky vůbec dojde. Jádro frameworku umí například poznat, když se uživatel chystá provést nepovolenou manipulaci s pamětí, typy, nebo špatně ukazuje na index v poli. (4)

2.2.1 Specifika a výhody

Mezi základní výhody, které tato platforma přinesla, patří díky vlastnostem jazyka C# nativní podpora objektově orientovaného programování. Použití tohoto jazyka však v platformě .NET není nutností, ta podporuje také jazyky Visual Basic, J# nebo managed C++. Všechen takto napsaný kód se následně překládá do společného jazyka (Intermediate Language), což umožňuje veškerým komponentám psaným v různých jazycích naplno spolupracovat. Nově technologie přinesla také větší podporu dynamických webových stránek, a to zejména díky zdokonalení platformy ASP. Tím vznikla nová technologie ASP.NET, které je využito v naší webové aplikaci. Právě ASP.NET například umožňuje překlad jazyka C# na kód webových stránek či databázové operace. Pro datové operace obecně .NET podporuje také jazyk XML, aby bylo docíleno jednoduché možnosti importu a exportu dat z/do jiných platforem. Další změnou je přístup ke sdílení kódu mezi aplikacemi, kdy se .NET zbavuje použití .dll knihoven a nahrazuje je koncepcí sestavení zvanou assembly. Každé vzniklé sestavení disponuje funkcemi pro správu verzí nebo informacemi použitého zabezpečení, kdy je například možné určit, který uživatel nebo proces bude mít k jednotlivým funkcím v rámci tříd přístup. Platforma .NET je od svého vzniku plně

integrovaná do vývojového prostředí Microsoft Visual Studio, jehož v současnosti nejnovější verze 2019 je pro vývoj webové aplikace použita. V neposlední řadě je nutné také zmínit implementaci Language Integrated Query Framework (dále jen LINQ), který umožňuje uživateli prostřednictvím jazyka C# provádět dotazy vůči externím prvkům, jako jsou databáze nebo XML soubory. Tyto objekty prostřednictvím LINQ také mohou komunikovat s vývojovým prostředím Visual Studio, podporou dokončování kódu IntelliSense a ladicími nástroji. (6)

2.3 ASP.NET

Webové stránky v dnešní době již nejsou pouze jakousi interaktivní formou textových dokumentů, naopak se svojí podobou často blíží plnohodnotným aplikacím. Dokonce zejména některé mobilní aplikace již jsou spíše webovým prohlížečem, zobrazujícím pouze obsah své webové varianty uvnitř velmi minimalistického uživatelského rozhraní za účelem minimalizovat zátěž na tato mobilní zařízení. Většina dat je uložená na serverech a v aplikacích se pouze dočasně zobrazují, stejně tak jako velká část výpočetních úkonů již může probíhat přímo na serveru namísto koncového zařízení. Z hlediska formy prezentace obsahu jsou dnešní weby velmi živé, plné animací, zpětné vazby uživatelského vstupu, přehrávačů hudby či videa a dalších interaktivních prvků. Není to však ještě tak dávno, co se webové prohlížeče uživatele ptaly, zda mohou na stránce spustit daný skript či ovládací prvek, neboť jeho spuštění by mohlo znamenat pro uživatele bezpečnostní riziko či ztrátu výpočetního výkonu pro ostatní operace. Microsoft pro dynamické interaktivní weby vyvinul skriptovací platformu ASP, která však nenabízela mnoho možností, brzy přestala vyhovovat aktuálním požadavkům a především nebyla objektově orientovaná. (6)

ASP přineslo jistou revoluci v generování webových stránek a umožnilo uživateli vidět v čase se měnící prvky a hodnoty, přičemž výpočty s tím související běžely na straně serveru. Nejjednodušším příkladem může být třeba výpis aktuálního času, kdy se hodnota jednotlivých číslic každou sekundu mění, aniž by muselo být každou sekundu vyvoláno znovunačtení stránky. Toho bylo docíleno vložením skriptu ASP přímo do kódu HTML, který byl uvozován znaménky % uvnitř nám dobře známých

ostrých závorek. To však přispělo ke značné složitosti kódu v případě komplexnějších stránek, podobně jako v případě, kdy je CSS nebo Javascript kód přímo součástí daného HTML souboru. Takové řešení je v dnešním případě, kdy vzhled, logika a obsah stránek jsou tři zcela specifická odvětví, která vyžadují buď programátora s velmi diverzifikovaným oborem znalostí, nebo spolupráci více lidí, poměrně kostrbaté. Právě existence veškerého kódu uvnitř jednoho souboru se jevila jako veliký problém v případě, kdy by se na tvorbě webu podílelo více programátorů, přičemž jeden by měl na starost například právě aplikační logiku v ASP, zatímco jiný by se zabýval klasickou strukturou v HTML. Nejen že oba nemohli pracovat na jednom souboru současně (což by naopak některé dnešní verzovací a coworkingové softwary uměly řešit), ale vzájemným předáváním souboru k úpravám zpravidla vznikal tzv. špagetový kód. ASP skript navíc byl velmi vzdálený běžným programovacím jazykům, a proto bylo složité hledat programátory, kteří by skriptu dopodrobna rozuměli. Nesporným problémem tohoto řešení také bylo, že během aktualizace webu prakticky neexistoval způsob, jak jej během této akce udržet alespoň částečně „naživu“. ASP také sice nabízelo identifikaci klienta a založení pro klienta specifické relace, ta však mohla býti vždy pouze jedna a dnes běžná situace, kdy uživatel může mít v různých záložkách prohlížeče na stejném webu otevřeny různé formuláře tak byla zcela nemožná. I v tomto ohledu tak Microsoft přistoupil k razantnímu řešení celou architekturu přepracovat a na kořenech .NET frameworku tak vznikla nová technologie ASP.NET. (4)

2.3.1 Specifika a výhody

ASP.NET staví na veškerých výhodách, které plynou ze základní struktury frameworku .NET. Zajímavým faktem je, že i přes zmíněné nedostatky, které řešení klasického ASP dnes představuje, byla naplno zachována zpětná kompatibilita. Samotné nové rozhraní také přináší řadu zjednodušujících metod, jak od původního ASP pozvolna přejít k nové variantě. Díky integraci do .NET frameworku je možné využít technik ke tvorbě webových formulářů podobně, jako je tomu u tvorby klasických formulářových aplikací pro Windows. Nová verze ASP v .NET platformě

přináší výhody zejména v bezpečnosti, a to nejen co se týče ochrany přenášených dat, ale také v ukládání zadaných hodnot ještě před odesláním formuláře, které mohou zůstat zachovány i v případě obnovení relace. Velikou bolestí mnoha webových formulářů je situace, kdy se chceme kupříkladu vrátit k předchozímu kroku objednávky zboží, abychom upravili některé hodnoty, ale opětovným vyvoláním formuláře ztratíme veškerá dříve zadaná data a musíme jej tak vyplnit s čistým štítem znovu. Právě to již ASP.NET řeší pomocí cachování a kompilace Just-in-time. Stav relace je možné sdílet na několika serverech zároveň, které tak tvoří tzv. serverovou farmu. Rozšířené pak jsou autentifikační metody pro ověření totožnosti uživatele. Každá aplikace v ASP.NET je individuálně konfigurovatelná prostřednictvím XML souborů. (4)

2.4 Razor Pages

Razor je specifický druh programovací syntaxe v ASP.NET, který byl vydán roku 2011 spolu s vývojovým prostředím Visual Studio 2010. Jedná se prakticky o jakýsi modul umožňující snadno a rychle vytvářet jednoduché šablony webových stránek, ale také pomáhá stanovit strukturu jejich organizace, zejména co se týče systému souborů. Při jeho vzniku byl kladen důraz na splnění několika základních cílů, jako je kompaktní, stručný a plynulý design, snadná možnost osvojení si konceptu při základní znalosti HTML, čistou, rychlou a zábavnou syntaxi při psaní kódu, maximální provázanost s C#, Visual Studiem a IntelliSense a co je obzvláště důležité, možnost testovat vznikající webovou aplikaci lokálně, bez nutnosti existence webového serveru a domény. Razor přináší specifický druh jazyka pro generování webových stránek, zvaný C#html nebo cshtml, což je i přípona souborů definujících základní strukturu stránky. Jedná se o jakousi fúzi klasického HTML a jazyka C#, která při kompilaci vyústí vygenerováním finálního .html souboru. Na rozdíl od klasického ASP, kdy se bloky kódu nacházely uvnitř <% =%> tagů se v Razor označuje začátek kódu znakem @. Tento blok kódu nemusí být explicitně nijak uzavřen, neboť analyzátor obsahuje sémantické znalosti kódu C#, ze kterého automaticky vyčte, kde daný blok kódu končí. Tím je při programování ušetřeno nemalé množství stisků kláves a jedná se tak o projev snahy udělat tuto syntaxi,

respektive její použití co nejplynulejší. Taková syntaxe navíc dle vývojářů značně pomáhá zkomplikovat útoky typu XSS. (7)

2.5 HTML a CSHTML

HTML – HyperText Markup Language je základní značkovací jazyk pro psaní webových stránek. První verze vznikla již roku 1993 a jazyk je dále ve vývoji dodnes. Jeho nejvíce rozšířenou verzí je verze 4.01 standardizovaná roku 1999. Od roku 2012 se následně velmi rozšiřuje verze HTML5, rozšiřující jazyk o značné množství nových funkcí. HTML stanovuje základní strukturu webové stránky pomocí několika základních elementů, jako je hlavička, tělo či záhlaví stránky, jejichž obsah je definován pomocí tzv. tagů – značek určitého typu - odstavců, textů, obrázků a širší množiny dalších prvků a jejich vlastností. Jedním z důležitých prvků jsou hypertextové odkazy, které umožňují uživateli přejít na jinou stránku či prvek na stránce.

Vzhledem k tomu, že předmětem této práce je webová aplikace, pochopitelně se jazyku HTML nevyhneme, byť se jej dotkneme spíše okrajově, neboť základní část kódu je v rámci ASP.NET Razor Pages generována automaticky. Bez znalosti tohoto jazyka se však vývojář neobejde, neboť pro dosažení požadovaných funkcí a přizpůsobení vzhledu aplikace k obrazu svému je nutné automaticky vygenerovaný kód dále rozšiřovat a upravovat.

Jak bylo již dříve nastíněno, v rámci ASP.NET se pro definování front-endu při vývoji setkáváme se soubory typu .CSHTML. Ty představují jakousi variaci klasických .HTML souborů, do kterých je však možno vkládat specifické direktivy či bloky kódu v jazyce C#. Pomocí toho lze docílit automatizace výsledně vygenerovaného .html kódu, o což se při sestavení postará kompilátor.

2.6 CSS – kaskádové styly

Velmi běžnou součástí většiny webových stránek dnes jsou kaskádové styly. Samotný jazyk HTML sice nabízí jistou množinu možností, jak formátovat text, avšak tyto možnosti jsou pro dnešní potřeby formátování webových stránek značně

omezené, a to jak po funkční, tak estetické stránce. Weby psané čistě v HTML bez použití určitých nástaveb ve formě stylování či skriptů bychom v dnešní době našli pouze ve velmi malém množství. Důvodů je hned několik. Kaskádové styly (CSS – z anglického Cascading Style Sheets) nabízejí širokou řadu možností, jak jednoduše rozmístit prvky na stránce a přiřadit jim určité vlastnosti, jejichž implementace by v jazyce HTML byla velmi náročná, až téměř nemožná. Pomocí CSS neurčujeme pouze pozici, velikost, či barvu jednotlivých elementů, ať již vzhledem ke stránce samotné či k ostatním elementům v okolí, ale také určité chování, respektive změnu těchto vlastností v závislosti na akcích uživatele. Tím může být například změna barvy či velikosti elementu při najetí myší, zobrazení či skrytí elementu, animace a obecně automatizace stylovacích procesů. CSS je do velké míry integrované do jazyka HTML. Implementovat do HTML jej lze třemi různými způsoby – buď přímo v HTML definici daného elementu prostřednictvím atributu style, v hlavičce daného HTML souboru jako tzv. stylesheet – seznam stylů nebo v samostatném externím .css souboru, na který se HTML stránka odkazuje prostřednictvím tagu <link>. (8)

Obr. 1 - změna vlastností prvku při najetí myší
Zdroj: vlastní zpracování

2.6.1 Bootstrap

Bootstrap představuje jednu z nejrozšířenějších knihoven založených na CSS, HTML a JavaScriptu pro tvorbu rozhraní a formátování webů a webových aplikací. Bootstrap původně vznikl pro účely vývoje konzistentního rozhraní sociální sítě Twitter rukou Marka Otta a Jacoba Thortona, později byl pod licencí open-source zpřístupněn široké veřejnosti. Tento Framework si zakládá zejména na responzivním formátování stránky tak, aby byla smysluplně zobrazitelná na jakémkoli zařízení. Integrace tohoto nástroje dokáže usnadnit vývojářům značné množství práce s vytvářením šablon a rozvržení stránek díky široké škále

předdefinovaných elementů a mřížkovému systému. Bootstrap navíc vykazuje velmi vysokou míru kompatibility s různými webovými prohlížeči. (9)

Bootstrap je přirozenou součástí Razor Pages, jejichž základní rozhraní je na Bootstrapu založeno.

2.7 LINQ a Entity Framework

LINQ – Language Integrated Query představuje sadu technologií pro provádění operací nad datovými sklady, jako jsou SQL databáze nebo XML dokumenty v jazyce C#. Pro přístup k SQL databázi, čtení a zápisu dat a provádění dotazů nad nimi tak není nutné použít či znát přímo jazyk SQL, byť je jeho znalost pochopitelně výhodou. Náš databázový dotaz je tak možné napsat přímo v jazyce C#, přičemž o překlad do SQL se opět postará kompilátor. (10)

```
IEnumerable<int> orderingQuery =
from num in numbers
where num < 3 || num > 7
orderby num ascending
select num;
```

Ukázka kódu 1 - příklad dotazu nad databází prostřednictvím LINQ
Zdroj: (10)

Entity Framework Core představuje moderní open-source platformu pro objektově-relační mapování, která umožňuje .NET vývojářům efektivně pracovat s databázemi prostřednictvím objektů uvnitř svého projektu. Samotné dotazy i ukládání dat jsou tak přímo součástí jazyka C# a ve své syntaxi z něj přímo vyplývají. (11)

Dotaz nad databází tak může vypadat následovně:

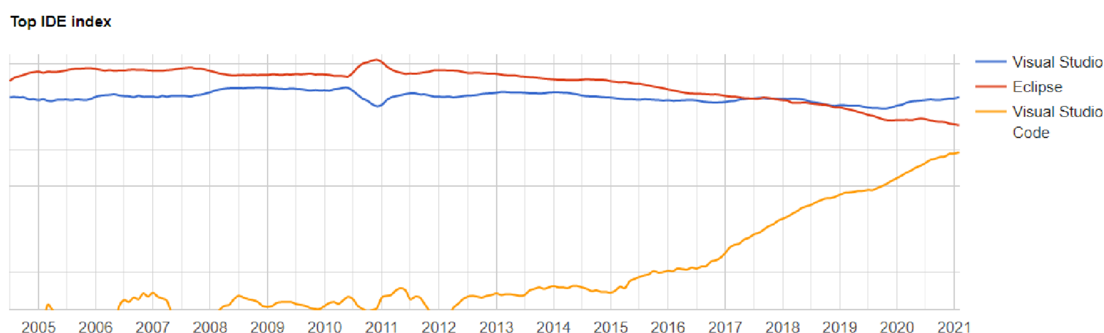
```
using (var db = new BloggingContext())
{
    var blogs = db.Blogs
        .Where(b => b.Rating > 3)
        .OrderBy(b => b.Url)
        .ToList();
}
```

Ukázka kódu 2 - příklad dotazu nad databází prostřednictvím EF Core
Zdroj: (11)

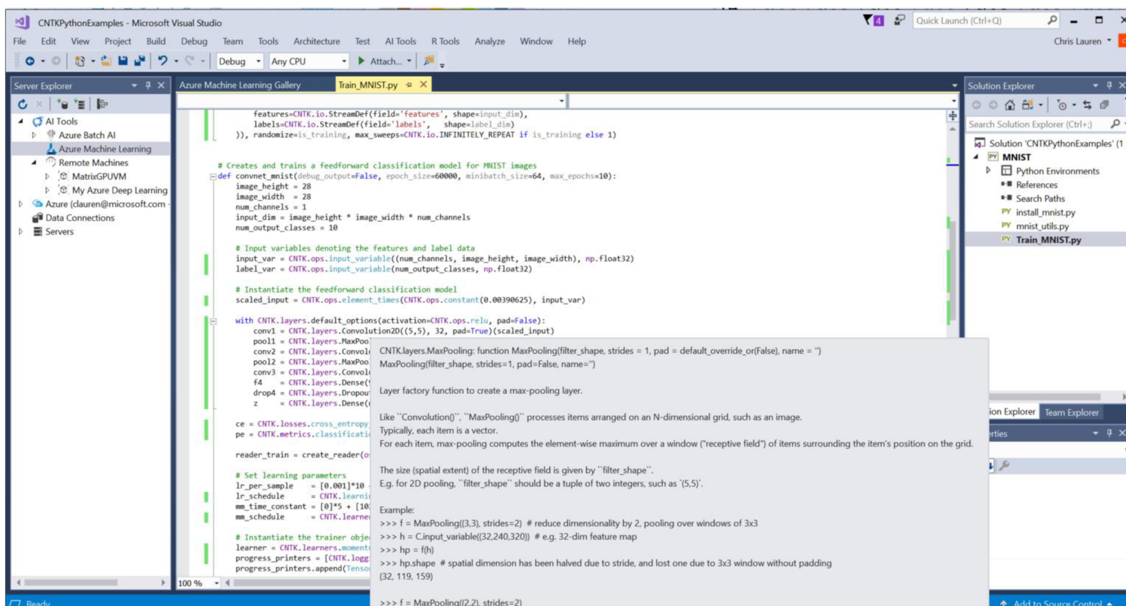
Na Entity Frameworku jsou založeny také Razor Pages a jeho použití v konkrétních případech budeme podrobně věnovat v rámci popisu vzniklé aplikace.

2.8 Microsoft Visual Studio

Vývojové prostředí (anglicky Integrated Development Environment) je jedním ze základních, nikoli však nezbytných nástrojů každého programátora. Jedná se o samostatný program, který je určen k programování dalších aplikací a poskytuje k tomu uživateli určité množství nástrojů. Standardními součástmi dnes jsou editor zdrojového kódu, kompilátor a debugger. Pro vývoj aplikace v rámci této diplomové práce bylo zvoleno právě Microsoft Visual Studio 2019 ve verzi Professional, a to hned z několika důvodů. Primárně je to dlouhodobá osobní znalost a dobré zkušenosti autora s tímto prostředím, dále obrovské množství prostředků, které toto prostředí nabízí, podpora velkého množství programovacích jazyků a komponent a v neposlední řadě také vysoká míra individuální customizace. Právě toto osobní nastavení patří mezi určité charakteristiky každého programátora spolu s jeho autorským rukopisem. Byť takové věci, jako specifické nastavení zvýrazňování kódu nemá na funkci výsledného programu žádný vliv, pro programátora jde o velmi užitečnou pomůcku, která mu na čistě vizuální bázi pomáhá odlišit jednotlivé části kódu, druhy proměnných, definice od deklarací a další. Dle indexu PYPL se jedná v současnosti o nejvyužívanější vývojové prostředí na světě. (12)



Obr. 2 - vývoj indexu využívanosti IDE dle PYPL od roku 2005 do současnosti
Zdroj: (12)



Obr. 3 - uživatelské rozhraní MS Visual Studio
Zdroj: Microsoft Docs

Po dlouhou dobu byl vývoj aplikací pro Windows polem, na kterém mohli běžně operovat pouze zkušení znalci jazyků C a C++. Napsání běžné aplikace pro tento operační systém znamenalo probrat se složitým procesem registrace tříd a implementace knihoven pro samotnou integraci aplikace do operačního systému a jeho uživatelského rozhraní. Prvním nástrojem, který dramaticky změnil možnosti vývoje pro Windows, byla aplikace Visual Basic 1.0 vydaná roku 2001, která představovala jednoduché uživatelské rozhraní pro design windowsových aplikací, zejména co se týče vzhledu oken, formulářů a jejich ovládacích prvků. Namísto hledání šestnáctibitových identifikátorů jednotlivých ovládacích prvků v dokumentaci bylo nyní možné v návrháři vizuálně přetáhnout tento prvek ze sady nástrojů přímo do okna na jeho určenou polohu a dále pouze naprogramovat jeho funkci. Později, roku 1993, vyšel software Microsoft Visual C++ 1.0. Jednalo se prakticky o identický nástroj, avšak tentokrát, jak název napovídá, pro programovací jazyk C++. Obě aplikace se nadále vyvíjely a spolu s nimi přibývaly verze orientované na další programovací jazyky, například J++. Vždy se však jednalo o samostatná vývojová prostředí pro daný programovací jazyk. Velké množství různých aplikací, které však měly stejnou funkci a lišily se pouze v použitém kódu, vedlo k myšlence, že na celou problematiku by mohlo být pohlíženo z poněkud

odlišné dimenze, a zrodila se myšlenka všechny tyto programovací jazyky zahrnout pod společné vývojové prostředí. Tak roku 1997 vzniklo Microsoft Visual Studio 6.0, které sice nepřineslo žádné zásadní nové funkce, sjednotilo však všechny „Visual" aplikace verze 5.0 „pod společnou střechu". Později byla tato aplikace rozšířena o komponentu Visual InterDev, nástroj určený k navrhování webových stránek prostřednictvím ASP. (13)

Roku 2002 bylo spolu s vydáním první verze frameworku .NET zveřejněno Visual Studio .NET, které již poskytovalo jeho plnou integraci. O rok později vyšlo Visual Studio 2003, které opravovalo většinu problémů s první verzi .NET dedikovaného IDE. Uživatelské prostředí nyní již bylo velmi stabilní, přineslo vylepšený bezpečnostní model a podporu IPv6 protokolu, tvorbu mobilních aplikací a datové rozhraní pro přístup k ODBC a Oracle databázím. Díky své robustnosti a spolehlivosti je Visual Studio .NET 2003 stále některými programátory využíváno. (13)

Po dvou letech vyšla další nová verze, Visual Studio 2005 spolu s verzí .NET 2.0. To přineslo například lokální vývojový webový server pro webové aplikace, což uklidnilo zejména spoustu síťových administrátorů, kterým se nelíbilo používání lokálního IIS na firemních počítačích z bezpečnostních důvodů. V této verzi Microsoft také poprvé představil více edicí daného vývojového nástroje, a to Express Editions, která cílila zejména na studenty a „hobby" programátory a obsahovala plnou podporu všech zahrnutých programovacích jazyků, avšak chybělo zde široké množství podpůrných nástrojů. Tato verze byla dostupné zdarma. Druhou edicí byla verze Team System Editions, která se měla zařadit mezi profesionální, firemně využívané nástroje. Ta byla nabízena za 799 USD. (14) (14)

Visual Studio 2008 spolu s .NET frameworkem 3.5 přineslo vylepšení ve formě multiplatfomingu, kdy vývojář mohl pro každý projekt nastavit několik cílových platforem a kompilovat pro ně zároveň. Jazyky VB.NET a C# byly rozšířeny o podporu LINQ, XML a novou inicializační syntaxi. Zejména integrace LINQ byla jedním z faktorů, který skokově zvýšil popularitu tohoto IDE. Mezi další vylepšení patří například designer XAML rozložení. Tato verze již cílila zejména na vývoj

aplikací pro OS Windows Vista a webové aplikace s využitím Microsoft Silverlight. (13)

Následující verze Visual Studia se již zásadně nelišily od Visual Studia tak, jak jej známe dnes. Uživatelské rozhraní a rozvržení prvků je stabilní, liší se pouze zejména verze frameworků proti kterým je možné kompilovat. Samotná sada Visual Studia je velmi modulární a umožňuje uživateli nainstalovat takové prvky, které ke své práci potřebuje. Z na první pohled viditelných změn Microsoft v posledních verzích přidal také různé verze vzhledu uživatelského rozhraní, kdy doplnil dlouho očekávaný „black mode“, tedy tmavý motiv. Poslední verze navíc umožňuje kromě množiny přednastavených motivů vytvořit motivy své vlastní dle individuálních požadavků uživatele. Prostřednictvím instalace doplňků je také možné prostředí naplno propojit s některými verzovacími službami na platformě Git jako je Bitbucket nebo GitHub.

3 Poruchové logy kolejových vozidel

Historie poruchových logů kolejových logů v dnešní České Republice sahá k samotným počátkům implementace mikroprocesorového řízení ve vozidlech ještě za dob Československa. Okolo roku 1998 zadaly tehdejší Československé Dráhy (ČSD) vývoj nového motorového vozu s elektrickým přenosem výkonu ze spalovacího motoru na trakční motory. Zadání se ujala firma ČKD a vznikl tak zbrusu nový motorový vůz, kterému bylo přiděleno označení 843. První z prototypů byl roku 1995 prezentován na mezinárodním strojírenském veletrhu v Brně, kde získal ocenění zlatou medailí. O rok později začala spolu s přívěsnými vozy řady 043 a řídicími vozy řady 943 jejich sériová výroba. Již samotný záměr ovládat vůz kromě řízení z dedikovaného stanoviště také vzdáleně z řídicího vozu napovídá, že k ovládání byl zapotřebí moderní řídicí systém. Osazení motorového vozu mikroprocesorovým řízením umožnilo v té době nevídané funkce, které znamenaly zásadní revoluci v koncepci řízení kolejových vozidel na nový standard, který se u moderních kolejových vozidel udržuje doposud. Přechod na digitální řídicí systém znamenal například odstranění některých analogových ukazatelů na stanovišti strojvedoucího a jejich nahrazení displeji, které kromě zobrazování měřených hodnot umožnily např. také ovládání některých funkcí a především pokročilou diagnostiku. Vůz totiž disponoval navíc speciálním diagnostickým počítačem, který se neangažoval v řídicím procesu, ale jeho účelem byl zejména sběr dat z ostatních systémů. Tento počítač označovaný zkratkou DPV – Diagnostický počítač vozidla se kromě kontroly a vyhodnocování stavů z jednotlivých vstupů mohl v případě detekce chybového stavu postarat o jeho zalogování včetně údajů o čase a okolnostech jeho vzniku. Takové chybové hlášení pak bylo možné zpětně zobrazit na displeji strojvedoucího, či exportovat do výstupního souboru. Poruchová hlášení vznikala na základě převzetí již hotových poruchových signalizací od ostatních systémů, jako např. řídicího počítače trakce, čidel dveří apod. či vlastní programovou logikou. V závislosti na charakteru takového hlášení pak vzniklé hlášení mohlo být okamžitě zobrazeno nebo pouze uloženo. Představme si poruchové hlášení, které je snadno pochopitelné: „Přehřátí chladicí kapaliny“. Pokud se takové hlášení vyskytlo, vypsalo se na displeji strojvedoucího, a ten na toto

oznámení adekvátně zareagoval, vše proběhlo v pořádku. Dotyčný strojvedoucí posléze tuto informaci zaznamenal do knihy oprav a při pravidelné servisní prohlídce vozidla již pracovníci dílny věděli, že mají hledat problém v chladicím okruhu vozidla. Velmi brzy se ovšem osvědčilo, že v případě, kdy měli pracovníci dílny možnost do logu historie poruch nahlédnout a zjistit, zda například dochází k této poruše opakovaně, bylo možné z tohoto zjištění vyvodit širší závěry. Obecně platí, že čím více informací může obsluha či dílna z daného hlášení zjistit, tím lépe. Na konci minulého století však byly možnosti výpočetní techniky výrazně nižší a množství uchovaných dat tak bylo oproti dnešnímu stavu značně omezené. I tak dokázali autoři tohoto řídicího systému vytvořit možnosti, které byly po mnoho let nepřekonány při vývoji dalších vozidel s mikroprocesorovým řízením.

Mezi základní ukládané údaje patřilo např.:

- Výskyt, tj. datum, čas a doba trvání poruchového hlášení,
- Zdroj hlášení (jedno hlášení mohlo být sdruženo z více důvodů či zdrojů, které bylo možné později rozlišit,
- Kontext hlášení, který popisoval základní stav vozidla, jako např. která z kabin byla v době trvání události obsazená, zda bylo aktivní brzdění apod.,
- Aktuální rychlost vozidla (která je pro diagnostiku poruch velmi zásadní informací),
- Doplňková data.

Záznam historie poruch vozidla je tedy chronologicky uspořádaný záznam poruchových hlášení, která byla systémem vyhodnocena. Je velmi zajímavé, k jakým výsledkům vede, pokud se u nové technologie naplno využívají její současné možnosti namísto pouhé náhrady morálně zastaralého řešení.

4 Vývoj a popis vnitřní logiky aplikace (back-end)

V následující kapitole bude popsán vývoj back-endu, tedy pro uživatele neviditelné části stránek. Jedná se o jakési pozadí stránky, na kterém se odehrává veškerá aplikační logika, jako jsou výpočty, práce s databází a další. Tato část stránky, byť je pro uživatele neviditelná, však musí být s viditelnou částí pevně propojená, neboť musí být schopná pružně reagovat na uživatelský vstup.

4.1 Požadavky a vstupy

Jak již bylo nastíněno v úvodu, cílem práce je sestavení webové aplikace pro zpracování chybových hlášení lokomotiv s řídicími a diagnostickými systémy od společnosti MSV Elektronika. Tato chybová hlášení, specifické logy poruch stažené z řídicích počítačů vozidel ve formátu .his jsou základním vstupem této aplikace. Další důležitou vstupní součástí jsou takzvané rejstříky chyb, které obsahují podrobné definice jednotlivých poruch a informace k jejich správnému vyhodnocení. Tyto rejstříky se mohou lišit jak pro různé řady vozidel, tak pro různé verze řídicích a diagnostických softwarů a daný kód poruchy tak může mít u odlišných vozidel mírně odlišný význam. Vzhledem k tomu, že u výrobce tyto rejstříky existovaly doposud pouze ve formě souborů aplikace Microsoft Excel (.xlsx), bylo rozhodnuto, že pro potřeby aplikace budou z těchto .xlsx souborů vygenerovány soubory .csv s předem definovanou strukturou, které si dále po jejich nahrání již aplikace potřebným způsobem zpracuje. Po nahrání dat o poruchách a rejstříků pro jejich vyhodnocení do aplikace dojde k jejich uložení do databáze. Aplikace následně přiřadí dané poruchy danému vozidlu, které při vyvolání požadavku na dekódování chyby spáruje s požadovaným rejstříkem a dle něj ji vyhodnotí. Uživatel bude mít následně možnost tato surová i interpretovaná data sledovat, třídit a filtrovat dle stanovených kritérií. Součástí bude také statistická vizuální interpretace dat prostřednictvím grafů. To vše má být umístěno v moderním, lehkém a přehledném uživatelském rozhraní.

4.2 Návrh

Před samotným vývojem aplikace bylo nastíněno jakési orientační schéma struktury aplikace, které pomůže formovat jak databázi, tak samotné uživatelské rozhraní. Toto schéma bylo následně po celou dobu vývoje dodržováno a doznalo jen minimálních změn. Schéma sestává zejména z následujícího:

Tabulka chybových logů

Tabulka chybových logů obsahuje základní informace z hlavičky daného logového souboru. Ta obsahuje zejména:

- číslo (identifikátor) daného vozidla,
- datum a čas stažení chybového logu,
- celkovou ujetou dráhu ke dni a času stažení,
- verze jednotlivých řídicích a diagnostických počítačů.

Tabulka chyb

Tabulka chyb následně sestává z jednotlivých záznamů o chybách vztažených k danému chybovému logu. Mezi základní obsažené informace patří:

- zkratka dané poruchy,
- datum a čas vzniku,
- trvání dané poruchy,
- zdrojový bit,
- rychlost v době vzniku,
- poměrný tah vozidla v době vzniku,
- číslo poruchy,
- kontext,
- diagnostické bajty,
- ujetá dráha v době vzniku.

Tabulka rejstříků

Tabulka rejstříků obsahuje seznam jednotlivých rejstříků pro jednotlivá vozidla a verze softwaru řídicího a diagnostického počítače. Patří sem:

- řada (typ) vozidla,
- verze rejstříku,
- software řídicího počítače,
- datum nahrání rejstříku

Tabulka rejstříkových chyb

Tabulka rejstříkových chyb sestává z množiny všech známých chyb daného vozidla a dané verze softwaru, včetně jejich detailního popisu, respektive dat k podrobnému vyhodnocení dané poruchy. Patří sem zejména:

- číslo a identifikátor dané poruchy,
- zkratka,
- celý název poruchy,
- popis pro vyhodnocení zdrojového bitu,
- klasifikace poruchy,
- data pro zobrazení poruchy (na displeji strojvedoucího).

Tabulka řad (typů) vozidel

Tabulka obsahuje zejména řadu vozidla vztaženou k danému rejstříku, dle kterého se mají chyby pro dané vozidlo vyhodnotit.

Tabulka jednotlivých vozidel

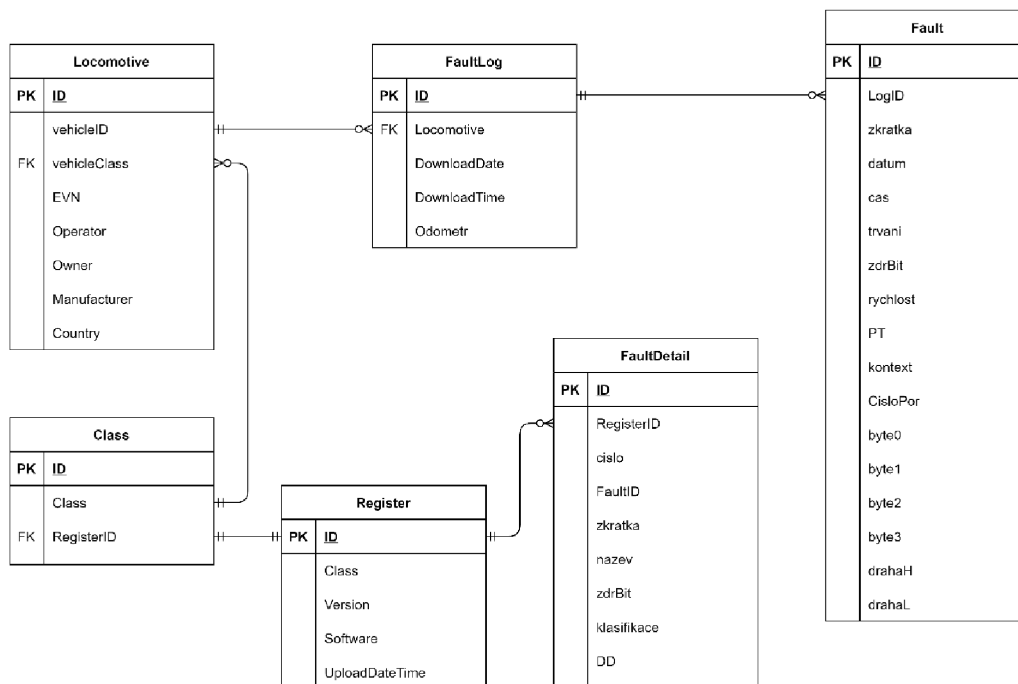
Jedná se o tabulku obsahující informace o jednotlivých vozidlech. Kromě přiřazení vozidla dané řadě obsahuje také doplňující informace, které poslouží zejména pro statistické účely. Patří sem:

- číslo (identifikátor) daného vozidla,

- řada vozidla,
- provozovatel,
- majitel,
- výrobce,
- stát, ve kterém je vozidlo zaregistrováno.

4.3 Databázové schéma

Dle nastíněného návrhu struktury aplikace bylo vytvořeno databázové schéma obsahující jednotlivé datové tabulky a jejich strukturu:



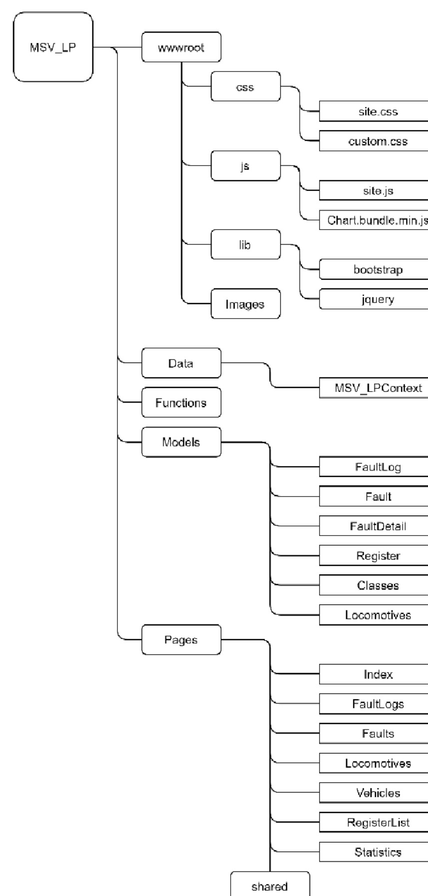
Obr. 4 - databázové schéma aplikace
Zdroj: vlastní zpracování

Databázové schéma ve vysokém rozlišení je přílohou této práce.

Je možné si všimnout, že pro detailní vyhodnocení dané poruchy je nutné projít všechny tabulky dané databáze. Důvodům nutnosti použití tohoto řešení se budu věnovat v následující části práce: 4.7 - Dekódování poruch.

4.4 Souborová struktura

V souladu s nastíněnými schématy a zvyklostmi pro ASP.NET Razor Pages byla sestavena také souborová struktura projektu. Ta sestává zejména z adresáře Data, kde se nachází samotná databázová struktura, adresáře Models obsahujícího jednotlivé datové modely, adresáře Pages obsahující front-end i back-end jednotlivých stránek nebo adresáře Functions pro případné externí knihovny funkcí. Nelze zapomenout ani na kmenový adresář webu wwwroot obsahující například soubory s CSS šablonami, knihovny Bootstrap nebo obrázky. Důležitost dodržování jistých konvencí ohledně souborové struktury v ASP.NET Razor Pages vyjma přehlednosti spočívá zejména v tom, že navigace na webu probíhá formou odkazů na jednotlivé stránky v rámci složky Pages. Díky integraci Entity Frameworku můžeme základní strukturu jednotlivých stránek ve Visual Studiu generovat automaticky – včetně front-endového souboru .cshtml a back-endového skriptu v C#. Provázání front-endu a back-endu pak funguje pomocí direktiv v hlavičce .cshtml souboru. Metody předání dat mezi skriptem a viditelnou částí stránky si podrobně nastíníme v další části práce. Pro aplikace pracující s databází je zde navíc dostupná funkce CRUD (create, read, update, delete), která vygeneruje ke každé stránce navíc také stránky pro vytvoření, čtení, úpravu nebo odstranění jednotlivých záznamů. Každá ze stránek pracuje s přiřazeným datovým modelem ze složky Models. Za zmínku stojí také složka shared obsahující soubor _Layout.cshtml. V tomto souboru je definována struktura společná pro všechny stránky v projektu – například navigační lišta nebo záhlaví stránek.



Obr. 5 - schéma souborové struktury projektu
zdroj: vlastní zpracování

```
<li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-page="/Index">Domů</a>
</li>
```

Ukázka kódu 3 - hypertextový odkaz na Razor stránku v navigačním menu
Zdroj: vlastní zpracování

4.5 Oblast chybových logů

První oblastí, na kterou jsem se při vývoji zaměřil, byla oblast chybových logů. Na základě ukázek několika souborů chybových logů .his byly sestaveny patřičné datové modely a stránky pro práci s nimi.

4.5.1 Chybové logy .his

Základním zdrojem dat jsou chybové logy z řídicích počítačů vozidel, které obsahují hlavičku a výčet jednotlivých chybových hlášení. Každý řádek odpovídá jednomu chybovému hlášení, přičemž dílčí informace daného hlášení jsou v řádku odděleny tabulátorem, popřípadě několika mezerami. Jeden takový soubor může obsahovat až tisíce jednotlivých chybových hlášení.

```
*** 753.605 ** 18-01-03, 09:19:25 ** Ujeto    21197.903 km   ***
** CRV 1.00, PVZa0.00, RTR 1.00, DPV 1.01, PVZd0.05, POR 0.00 **
** --- 0.00, --- 0.00, --- 0.00, KAM 1.36, --- 0.00, --- 0.00 **
***                                                          ***

KON : 01-03  09:09:45    578s [40]   0    +0 <42> #084 255   0   0   0  226376
INSP: 01-03  09:07:14     58s [10]   0    +0 <42> #165   0   0   0   0  226376
****: 01-03  09:07:13   Trva  [01]   0    +0 <42> #000   1  67 116  79  226376
INSP: 01-03  08:04:24   Trva  [10]   0    +0 <42> #165   0   0   0   0  226376
****: 01-03  08:04:23   Trva  [01]   0    +0 <42> #000   1  67 116  79  226376
!SRP: 01-03  06:42:37   Trva  [02]   0    +0 <CC> #040 208   0   8 136  226144
StuS: 01-03  06:32:59    147s [02]   0    +0 <42> #023  21  21  21  21  225832
tV<5: 01-03  06:32:59   5.12s [01]   0    +0 <42> #024  21  21  21  21  225832
tV<5: 01-03  06:32:26   0.32s [01]   0    +0 <42> #024  21  21  21  21  225832
```

Ukázka kódu 4 - část souboru chybového logu .his
Zdroj: MSV Elektronika

Na ukázce výše je možné vidět hlavičku chybového logu a několik jeho prvních řádků s popisem chyb. První řádek hlavičky obsahuje číslo lokomotivy, datum a čas stažení a celkovou ujetou dráhu vozidla v době stažení. Další řádky pak obsahují verze jednotlivých řídicích a diagnostických nástrojů. Na šestém řádku již začíná výpis jednotlivých poruch, obsahující (zleva): zkratku poruchy, datum a čas vzniku, trvání, zdrojový bit, rychlost, poměrný tah, kontext, ID poruchy, diagnostické bajty a ujetou

dráhu v době vzniku poruchy. Na základě obsahu tohoto souboru byly vytvořeny modely pro chybové logy a dílčí poruchy.

4.5.2 Vytvoření modelů chybových logů

Struktura datového modelu pro chybové logy přímo vyplývá z hlavičky každého chybového logu. V programovém modelu se nachází jako třída FaultLog s následující strukturou:

```
public class FaultLog
    {
        [Key]
        [Display(Name = "ID Logu")]
        public int ID { get; set; }
        [Display(Name = "Vozidlo")]
        public string Locomotive { get; set; }
        [Display(Name = "Datum stažení")]
        [DataType(DataType.Date)]
        public DateTime DownloadDate { get; set; }
        [Display(Name = "Čas stažení")]
        [DataType(DataType.Time)]
        public DateTime DownloadTime { get; set; }
        [Display(Name = "Odometr")]
        public float Odometr { get; set; }
    }
```

Ukázka kódu 5 - definice třídy FaultLog

Tato definice se v zásadě neliší od standardní definice třídy v jazyce C#, je však možné si všimnout zvláštních atributů ve hranatých závorkách nad jednotlivými prvky. Jedná se o datové anotace v rámci EF Core. Například atribut [Key] u proměnné ID určí, že se jedná o primární klíč, který je jedinečný a autoinkrementační. Atribut [Display(Name)] naopak přiřazuje dané proměnné popis, který se má v rámci stránek u dané proměnné zobrazit. Atributy [DataType] naopak pomáhají překladači přiřadit datům správný formát.

Jak již bylo uvedeno, každý chybový log obsahuje množinu poruch, které jsou jeho součástí. Jinými slovy, každá hlavička odpovídající entitě FaultLog obsahuje množství dalších entit, které popisují jednotlivé zaznamenané poruchy. Pro ty byla vytvořena třída Fault, která opět vychází ze struktury popisu jednotlivých poruch v logu. Třída má následující strukturu:

```
public class Fault
    {
        [Key]
        [Display(Name = "ID poruchy")]
        public int ID { get; set; }
        [Display(Name = "ID logu")]
        public int LogID { get; set; }
        public string zkratka { get; set; }

        [DataType(DataType.Date)]
        [Display(Name = "datum vzniku")]
        public DateTime datum { get; set; }
        [Display(Name = "čas")]
        public string cas { get; set; }
        [Display(Name = "trvání")]
        public string trvani { get; set; }
        [Display(Name = "zdrojový bit")]
        public string zdrBit { get; set; }
        [Display(Name = "rychlost")]
        public int rychlost { get; set; }
        [Display(Name = "PT")]
        public string PT { get; set; }
        public string kontext { get; set; }
        [Display(Name = "číslo poruchy")]
        public string CisloPor { get; set;}
        public ushort byte0 { get; set; }
        public ushort byte1 { get; set; }
        public ushort byte2 { get; set; }
        public ushort byte3 { get; set; }
        public int drahaH { get; set; }
        public int drahaL { get; set; }
    }
```

Ukázka kódu 6 - definice třídy Fault
Zdroj: vlastní zpracování

4.5.3 Vytvoření stránek pro zobrazení a práci s entitami

V momentě, kdy dojde k nadefinování datové struktury, se kterou je potřeba pracovat, je vytvoření základních stránek pro práci s nimi velmi jednoduché. Pomocí funkce CRUD tak byly vývojovým prostředím automaticky vygenerovány stránky FaultLogs a Faults, každá z nich obsahující základní index se seznamem všech záznamů dané třídy a stránky pro přidání nového záznamu, detailního zobrazení daného záznamu, jeho úpravu či odstranění. Tím velmi rychle a snadno došlo k získání jakéhosi základu, se kterým je možné dále pracovat, avšak pro plnění námi požadované funkce to zdaleka nestačí. U chybových logů se například nepočítá s tím, že by je uživatel ručně vytvářel a zadával jejich obsah. Stránka Create tak v tomto případě postrádá smysl. Zdrojem dat zde jsou přímo chybové logy, proto bylo

zapotřebí vytvořit stránku, ve které proběhne jejich nahrání. Za tímto účelem byla stránka Create odstraněna, zatímco byla vytvořena nová stránka Upload. Do stránky Index (v kontextu chybových logů) pak bylo doplněno tlačítko „Nahrát log“ odkazující právě na stránku s formulářem pro nahrání souboru.

Záznamy chybových logů

Nahrát log

Hledej lokomotivu: Dle čísla: Všechny Dle data: Všechny Filtruj

ID Logu	Vozidlo	Datum stažení	Čas stažení	Odometr	
1	753.604	03.01.2018	14:29	24178,14	Upravit | Podrobnosti | Odstranit
2	753.605	03.01.2018	9:19	21197,902	Upravit | Podrobnosti | Odstranit
3	753.605	04.06.2019	12:04	95152,19	Upravit | Podrobnosti | Odstranit
4	753.605	26.06.2019	8:59	95859,04	Upravit | Podrobnosti | Odstranit
5	753.606	10.11.2017	9:14	1111,06	Upravit | Podrobnosti | Odstranit

Obr. 6 - stránka s chybovými logy rozšířená o tlačítko "Nahrát log"
Zdroj: vlastní zpracování

Nahrajte chybový log .his

Soubor:

Vybrat soubor Soubor nevybrán

Nahrát

Obr. 7 - část stránky s formulářem pro nahrání souboru
Zdroj: vlastní zpracování

4.5.4 Logika zpracování chybového logu

Nyní, když byl vytvořen funkční formulář pro nahrání souboru, je zapotřebí přesunout se k back-endu: Vytvoříme skript, který postupně projde nahrávaný soubor a zatřídí jednotlivá data do databáze. Předtím je však zapotřebí věnovat pozornost také jisté míře kontroly nahrávaných souborů. Zcela primární a triviální metodou je kontrola typu souboru, který je nahráván.

Je považováno za zbytečné podrobně procházet veškerý obsah souboru Upload.cshtml.cs, je však vhodné zaměřit se alespoň na ty nejdůležitější části kódu. Kontrola typu souboru je provedena pomocí pole řetězců, obsahujícího jednotlivé povolené typy souborů, v našem případě pouze „.his“.

```
private readonly string[] _permittedExtensions = { ".his" };
```

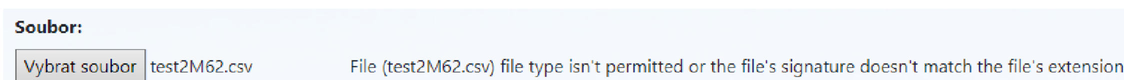
Ukázka kódu 7 - pole řetězců s povolenými příponami

Tento seznam povolených typů souborů je posléze předán systémové třídě FileHelpers, která se již postará o srovnání přípony nahrávaných souborů s povolenými.

```
var formFileContent = await
FileHelpers.ProcessFormFile<BufferedSingleFileUploadDb>(FileUpload.FormFile,
ModelState, _permittedExtensions);
```

Ukázka kódu 8 - kontrola správnosti přípony souboru

V případě, že kontrola proběhne neúspěšně, tedy nedojde k nalezení shody mezi příponou nahrávaného souboru a povolenými příponami, akce vyvolá návrat ke stránce s žádostí, aby uživatel vybral vhodný soubor.



Obr. 8 - chybová hláška v případě pokusu o nahrání špatného typu souboru
Zdroj: vlastní zpracování

Za předpokladu, že kontrola proběhne úspěšně, dostáváme se k samotnému čtení obsahu souboru. Nejprve je potřeba přečíst hlavičku:

```
string result = System.Text.Encoding.UTF8.GetString(formFileContent);
            StringReader reader = new StringReader(result);
            string header = reader.ReadLine();
            string Locomotive = header.Substring(4, 7);
            string D_D = header.Substring(15, 8);
            string D_T = header.Substring(25, 8);
            string dateYear = D_D.Substring(0, 3);
            DateTime DownloadDate = DateTime.ParseExact(D_D, "yy-MM-dd", null);//log download date
            DateTime DownloadTime = DateTime.ParseExact(D_T, "HH:mm:ss", null);//
            float Odo = float.Parse(header.Substring(46, 9), CultureInfo.InvariantCulture.NumberFormat);
```

Ukázka kódu 9 - čtení hlavičky chybového logu

Čtení je prováděno čtením obsahu daného souboru řádek po řádku. Obsah každého řádku je následně nahrán do řetězce. Jednotlivá data z řádku jsou čteny pomocí

metody Substring(), která umožní ze vstupního řetězce získat podřetězec definovaný startovací pozicí a délkou. Je vhodné věnovat pozornost deklaraci proměnné „Odo" vyjadřující ujetou dráhu (odometrii) vozidla, respektive parametrům metody Parse(), která převede numerickou část znakového řetězce na číslo. Parametr CultureInfo v tomto případě slouží k tomu, aby tečka byla rozeznána jako desetinná čárka pro převod na proměnnou typu float. Takto získaná data posléze stačí k vytvoření a naplnění databázové entity FaultLog:

```
/* creation of the FaultLog */
            var file = new FaultLog
            {
                Locomotive = Locomotive,
                DownloadDate = DownloadDate,
                DownloadTime = DownloadTime,
                Odometr = Odo
            };

            _context.FaultLog.Add(file);
            await _context.SaveChangesAsync();
```

Ukázka kódu 10 - vytvoření entity FaultLog a její naplnění daty

Následně, po přeskočení prázdných řádků, je možné začít číst jednotlivé poruchy obsažené v logovém souboru. Prochází se jednotlivé řádky, dokud nedojde na konec souboru, přičemž data jsou opětovně získávána pomocí podřetězců na konkrétních pozicích. Pochopitelně, data by bylo možné získat z programátorského hlediska jednodušší cestou pomocí metody Split(), vzhledem však k tomu, že se jedná o soubor s pevným formátem a s ohledem k možnému vysokému množství záznamů byla opětovně zvolena výpočetně jednodušší metoda Substring(). Takto je posléze možné vytvořit entity Fault, které jsou naplněny daty a přiřazeny k dané entitě FaultLog:

```
string Entry;
            /* creation of the Fault entry */
            while ((Entry = reader.ReadLine()) != null) //read till the end
            {
                string date = Entry.Substring(6, 5);
                string dateComplete = string.Concat(dateYear, date);
                var faultEntry = new Fault
                {
                    LogID = file.ID,
                    zkratka = Entry.Substring(0, 4),
                    datum = DateTime.ParseExact(dateComplete, "yy-MM-dd", null),
                    cas = Entry.Substring(13, 8),
                    trvani = Entry.Substring(24, 5),
                    zdrBit = Entry.Substring(30, 4),
                    rychlost = int.Parse(Entry.Substring(35, 3)),
```

```
                PT = Entry.Substring(40, 4),
                kontext = Entry.Substring(45, 4),
                CisloPor = Entry.Substring(50, 4),
                byte0 = ushort.Parse(Entry.Substring(55, 3)),
                byte1 = ushort.Parse(Entry.Substring(59, 3)),
                byte2 = ushort.Parse(Entry.Substring(63, 3)),
                byte3 = ushort.Parse(Entry.Substring(67, 3)),
                drahaH = int.Parse(Entry.Substring(71, 7)),
                drahaL = int.Parse(Entry.Substring(79, 7))
            };
            _context.Fault.Add(faultEntry);
            await _context.SaveChangesAsync();
        }
```

Ukázka kódu 11 - čtení jednotlivých poruch uvnitř logu a vytvoření entity Fault

Tímto způsobem je možné plnit databázi logů a chyb daty. Primární klíč tabulky logů – identifikátor logu slouží zároveň jako cizí klíč pro tabulku chyb, kde se pomocí proměnné LogID přiřazují jednotlivé poruchy jednotlivým logům.

4.6 *Oblast rejstříků chyb*

4.6.1 Soubory rejstříků .csv

Další důležitou oblastí z hlediska vstupních dat pro vyhodnocování jednotlivých poruch jsou tzv. rejstříky chyb. Ty měl výrobce k dispozici ve formě tabulkových .xlsx souborů, které však nejsou pro zpracování externí aplikací zcela vhodné. Bylo proto přistoupeno k metodě exportu dat z těchto souborů do formátu .csv, tedy textového souboru s oddělovači. Ukázku zdrojového .xlsx souboru je možné vidět na Obr. 5. Exportem jsme získali soubor .csv s následující strukturou:

```
Seznam poruch;;;;;;;
VOZIDLO:;;2M62;! Poruchy mající ve sloupci ID zkratku místo čísla se na displeji zobr
azí jako Rxxx, kde xxx je číslo poruchy (zkratky se ukazují po stisku F2) !;;;;
VER:;;2.00;Platí pro SW: 17L21;;;;
čis.;ID;zkr.;název;zdrojový bit;;klasifikace;DD
000;4000;****;*** DPV zapnuto ***;;;LOG;ZAP
001;8000;POZ;Požár %1;"[elektrického rozváděče, v kabině, ve strojovně, spalovacího m
otoru; -, -, -, -]";;PORUCHA ČERVENÁ;0
005;9979;POZv;Požár ve vlaku;;;PORUCHA ČERVENÁ;0
006;8016;SMYK;Smyk;;;STAV MODRÁ;FCR
007;R007;R007;REZERVA R007;;;LOG;0
008;8502;TSR2;Nesoulad tlaků vzduchu za brzdovými rozváděči;;;STAV MODRÁ;0
009;8512;TSB2;Nesoulad tlaků vzduchu v brzdových válcích;;;STAV MODRÁ;0
010;8528;PDDB;Překročení doby brzdění doplňkovou brzdou;;;STAV MODRÁ;0
011;8020;SMYT;Trvalý smyk;"[1, 2, 3, 4; -, -, -, -]";;STAV MODRÁ;FCR
013;0017;nMax;Zásah přeotáčkové ochrany %1 spalovacího motoru;;;PORUCHA ČERVENÁ;DSL
014;0016;nMin;Zásah podotáčkové ochrany %1 spalovacího motoru;;;PORUCHA ČERVENÁ;DSL
015;0112;TST;Překročení doby startu %1 spalovacího motoru;;;PORUCHA ČERVENÁ;DSL
```

Ukázka kódu 12 - část vyexportovaného souboru .csv s popisem poruch

Seznam poruch						
VOZIDLO:	2M62	! Poruchy mající ve sloupci ID zkratku místo čísla se na displeji zobrazí jako Rxxx, kde xxx je číslo poruchy (zkratky se ukazují po stisku F2) !				
VER:	2.00	Platí pro SW: 17L21				
čis.	ID	zkr.	název	zdrojový bit	klasifikace	DD
000	4000	****	*** DPV zapnuto ***		LOG	ZAP
001	8000	POZ	Požár %1	[elektrického rozváděče, v kabině, ve strojovně, spalovacího motoru; -, -, -, -]	PORUCHA ČERVENÁ	0
002	R002	R002	REZERVA R002		LOG	0
003	R003	R003	REZERVA R003		LOG	0
004	R004	R004	REZERVA R004		LOG	0
005	9979	POZv	Požár ve vlaku		PORUCHA ČERVENÁ	0
006	8016	SMYK	Smyk		STAV MODRÁ	FCR
007	R007	R007	REZERVA R007		LOG	0
008	8502	TSR2	Nesoulad tlaků vzduchu za brzdovými rozváděči		STAV MODRÁ	0
009	8512	TSB2	Nesoulad tlaků vzduchu v brzdových válcích		STAV MODRÁ	0
010	8528	PDDB	Překročení doby brzdění doplňkovou brzdou		STAV MODRÁ	0
011	8020	SMYT	Trvalý smyk	[1, 2, 3, 4; -, -, -, -]	STAV MODRÁ	FCR
012	8680	bNS	Chybná manipulace s přepínačem návěstního světla	[předním, zadním, -, -; -, -, -, -]	STAV MODRÁ	
013	0017	nMax	Zásah přeotáčkové ochrany %1 spalovacího motoru		PORUCHA ČERVENÁ	DSL
014	0016	nMin	Zásah podotáčkové ochrany %1 spalovacího motoru		PORUCHA ČERVENÁ	DSL
015	0112	TST	Překročení doby startu %1 spalovacího motoru		PORUCHA ČERVENÁ	DSL
016	8033	TCEK	Čekání mezi starty		STAV MODRÁ	DSL
017	0208	STYJ	Porucha trakčního obvodu		PORUCHA ČERVENÁ	0
018	0240	STYB	Porucha brzdového obvodu		PORUCHA ČERVENÁ	0
019	8041	OKAS	Vnucený okamžitý stop spalovacího motoru		LOG	DSL
020	6000	KSR	Otevřené dveře elektrického rozváděče	[rozvaděče, prostor EDB, -, -; -, -, -, -]	PORUCHA ČERVENÁ	KON
021	0113	UCAs	Řídicí systém %1 spalovacího motoru bez napájení		PORUCHA ČERVENÁ	0
022	0036	TVOn	Vysoká teplota hlavního chladicího okruhu %1 spalovacího motoru		PORUCHA ŽLUTÁ	DSL
023	R023	R023	REZERVA R023		LOG	
024	0096	HVOd	Nízká hladina chladicí kapaliny %1 spalovacího motoru		PORUCHA MODRÁ	DSL
025	0464	IZ1	Nízký izolační stav trakčního obvodu %1		PORUCHA ŽLUTÁ	0
026	0466	IZ2	Nízký izolační stav obvodu buzení trakčního generátoru		PORUCHA ŽLUTÁ	0
027	8032	StuS	Vnucený studený start		LOG	DSL
028	0032	tV<	Nízká teplota chladicí kapaliny %1 spalovacího motoru		STAV MODRÁ	DSL
029	5088	kDPV	Porucha komunikace řídicího systému		PORUCHA MODRÁ	0
030	2560	kHYD	Porucha komunikace s blokem řízení hydraulického obvodu		PORUCHA ČERVENÁ	0

Obr. 9 - ukázka části původního .xlsx souboru s popisem poruch
Zdroj: MSV Elektronika

Důležitou částí vyexportovaného seznamu poruch je hlavička obsahující informaci o řadě (typu) vozidla a verzi použitého diagnostického softwaru. Tyto informace budou vhodné zejména při dekódování jednotlivých logů poruch, aby bylo zřejmé, pro jakou řadu daný rejstřík platí, a tedy dle jakého rejstříku se má daná porucha dekódovat.

4.6.2 Vytvoření modelů rejstříků

Stejně jako tomu bylo v případě chybových logů, je nutné před samotnou definicí jejich zobrazení vytvořit modely. Základním modelem rejstříku je třída Register. Ta má následující strukturu:

```
public class Register
    {
        [Key]
        public int ID { get; set; }

        [Display(Name = "Vozidlo")]
        public string Locomotive { get; set; }

        [Display(Name = "Verze")]
        public float Version { get; set; }

        public string Software { get; set; }

        [DataType(DataType.DateTime)]
        [Display(Name = "Nahráno")]
        public DateTime UploadDateTime { get; set; }

    }
```

Ukázka kódu 13 - struktura třídy Register

Definice a funkce jednotlivých atributů byly popsány již v přechozí kapitole a proto se nepovažuje za nutné je podrobněji definovat znovu. Třída je tvořena převážně informacemi vyplývajícími z hlavičky vstupního .csv souboru, dále rozšířená o automaticky získaný čas a datum nahrání pro případné využití pro statistické účely. Podobně jako v případě chybových logů, také k jednotlivým hlavičkám registrů se posléze vážou jednotlivé položky chyb s daty pro jejich dekódování. Ty jsou definovány ve třídě FaultDetail:

```
public class FaultDetail
    {
        [Key]
        public int ID { get; set; }
        [Display(Name = "ID registru")]
        public int Register_ID { get; set; }
        [Display(Name = "Číslo")]
        public uint cislo { get; set; }
        [Display(Name = "ID poruchy")]
        public string Fault_ID { get; set; }
        [Display(Name = "Zkratka")]
        public string zkratka { get; set; }
        [Display(Name = "Název")]
        public string nazev { get; set; }
        [Display(Name = "Zdrojový bit")]
        public string zdrBit { get; set; }
        [Display(Name = "Klasifikace")]
        public string klasifikace { get; set; }
        [Display(Name = "DD")]
        public string DD { get; set; }
    }
```

Ukázka kódu 14 - Struktura třídy FaultDetail

Zásadní vztažnou informací je zde proměnná „zkratka“, která bude posléze použita pro spárování dané chyby s položkou v rejstříku, dle které se má dekódovat. K tomu však podrobně později v kapitole 4.7.3.

4.6.3 Vytvoření stránek pro zobrazení a práci s entitami

Obdobně jako v předchozím případě je nyní možné v projektu vytvořit patřičné stránky pro práci s entitami chybových rejstříků. Prostřednictvím CRUD tak byla vytvořena stránka RegisterList. Na rozdíl od předchozí situace však je již samostatná hlavní stránka se seznamem všech položek rejstříku zbytná. Jednotlivé definice chyb v rámci rejstříku je možné vyvolat pomocí podřízené stránky Details. Náhled této stránky lze vidět na Obr. 10. Stejně jako v předchozím případě je nutné vytvořit také stránku pro nahrání daného .csv souboru s rejstříkem.

Prohlížení rejstříku

Lokomotiva 2M62 verze 2 SW 17L21

ID	1
Vozidlo	2M62
Verze	2
Nahráno	18.04.2021 3:21:17

ID	ID registru	Číslo	ID poruchy	Zkratka	Název	Zdrojový bit	Klasifikace	DD
1	1	0	4000	****	*** DPV zapnuto ***	,	LOG	ZAP
2	1	1	8000	POZ	Požár %1	"[elektrického rozváděče, v kabině, ve strojovně, spalovacího motoru, -, -, -, -]"		PORUCHA ČERVENÁ
3	1	2	R002	R002	REZERVA R002	,	LOG	0
4	1	3	R003	R003	REZERVA R003	,	LOG	0
5	1	4	R004	R004	REZERVA R004	,	LOG	0
6	1	5	9979	POZv	Požár ve vlaku	,	PORUCHA ČERVENÁ	0
7	1	6	8016	SMYK	Smyk	,	STAV MODRÁ	FCR
8	1	7	R007	R007	REZERVA R007	,	LOG	0
9	1	8	8502	TSR2	Nesoulad tlaků vzduchu za brzdovými rozváděči	,	STAV MODRÁ	0
10	1	9	8512	TSB2	Nesoulad tlaků vzduchu v brzdových válcích	,	STAV MODRÁ	0
11	1	10	8528	PDDB	Překročení doby brzdění doplňkovou brzdou	,	STAV MODRÁ	0
12	1	11	8020	SMYT	Trvalý smyk	"[1, 2, 3, 4, -, -, -, -]"		STAV MODRÁ
13	1	12	8680	bNS	Chybná manipulace s přepínačem návěstního světla	"[předním, zadním, -, -, -, -, -, -]"		STAV MODRÁ

Obr. 10 - náhled stránky prohlížení rejstříku
Zdroj: vlastní zpracování

4.6.4 Logika zpracování rejstříku

Pro nahrání rejstříku opětovně dojde k vytvoření dedikované stránky, u které je stanovena logika kontroly souboru při nahrání a dále dochází ke zpracování rejstříku následujícím způsobem:

```
string result = System.Text.Encoding.UTF8.GetString(formFileContent2);
            StringReader reader = new StringReader(result);
            reader.ReadLine(); //skip first line
            string h = reader.ReadLine(); //second line
            string[] h1 = h.Split(';');
            string Locomotive = h1[2];
            h = reader.ReadLine(); //third line
            string[] h2 = h.Split(';');
            float Version = float.Parse(h2[2], CultureInfo.InvariantCulture.NumberFormat);
            string Software = h2[3].Substring(h2[3].Length - 5);
            DateTime UploadDateTime = DateTime.Now;
var NewRegister = new Register
            {
                Locomotive = Locomotive,
                Version = Version,
                Software = Software,
                UploadDateTime = UploadDateTime
            };

            _context.Register.Add(NewRegister);
            await _context.SaveChangesAsync();
```

Ukázka kódu 15 - zpracování hlavičky souboru registru

```
reader.ReadLine(); // skip 4th line

            string line;
            while ((line = reader.ReadLine()) != null) //read till the end of file
            {
                string[] Entry = line.Split(';');
                var faultDetailsEntry = new FaultDetail
                {
                    Register_ID = NewRegister.ID,
                    cislo = uint.Parse(Entry[0]),
                    Fault_ID = Entry[1],
                    zkratka = Entry[2],
                    nazev = Entry[3],
                    zdrBit = Entry[4] + "," + Entry[5],
                    klasifikace = Entry[6],
                    DD = Entry[7]
                };
                _context.FaultDetail.Add(faultDetailsEntry);
                await _context.SaveChangesAsync();
            }
```

Ukázka kódu 16 - zpracování položek registru poruch

Je možné si všimnout, že tentokrát je při čtení jednotlivých dat z řádku použita funkce Split(). Tato funkce rozdělí řetězec dle požadovaného znaku nebo množství znaků, a získané podřetězce následně uloží do pole. Tuto funkci lze použít například z důvodu, že chybové rejstříky zpravidla čítají řádově nižší stovky položek než chybové logy a není proto třeba brát takové ohledy na efektivitu výpočetních

operací. Její použití je však vzhledem k faktu, že Microsoft Excel nedokáže exportovat do souboru .csv s použitím tabelace pro zarovnání hodnot nutností. Řešení formou odkazů na konkrétní pozici v poli by vzhledem k proměnné délce řádků nebylo bez nutnosti další modifikace souboru možné.

Nastíněným způsobem tak aplikace dokáže plnit databázové entity Register a FaultDetail nahranými daty ze souborů .csv. Takto získaná data jsou nezbytná pro dekódování poruch z chybových logů.

4.7 Dekódování poruch

Při bližším zkoumání obsahu rejstříků a chybových logů je možné si povšimnout společného pole „zkratka“ v obou tabulkách. Jako pravděpodobně nejjednodušší řešení propojení tabulek by se mohlo jevit přímé použití pole „zkratka“ jakožto cizího klíče tabulky Faults a primárního klíče tabulky FaultDetails. Situace je však reálně o něco složitější. Existuje totiž možnost, že jedna zkratka může mít u různých vozidel či různých verzí softwaru různý nebo mírně odlišný význam. Stejně tak mohou nastat situace, kdy různá vozidla disponují značně rozdílným seznamem možných poruch. Příkladem může být elektrická lokomotiva, která pochopitelně nebude disponovat poruchami spojenými se spalovacím motorem, u lokomotivy nezávislé trakce se naopak nepředpokládá například měření napětí v trakčním vedení. Vlivem toho je k problému přistupováno tak, že každý typ vozidla má svůj vlastní seznam možných poruch a tedy vlastní odpovídající rejstřík. V praxi mohou navíc nastat i situace, kdy různá vozidla stejného typu mohou mít různé verze řídicího a diagnostického softwaru, pokud se například jedná o jinou výrobní sérii stejného typu vozidla pro jiného zákazníka, které mohou opět disponovat odlišnými seznamy poruch, a tudíž budou vyžadovat specifický rejstřík. S touto možností však aplikace pro potřeby této diplomové práce prozatím nepočítá a možná řešení budou implementována během dalšího vývoje.

Dalším charakteristickým aspektem je fakt, že zatímco chybové logy jsou vztaženy ke konkrétním vozidlům, rejstříky chyb jsou vztaženy pouze k řadě (typu) vozidla a použitému softwaru (Rejstřík se vztahuje např. k lokomotivní řadě 753.6, zatímco

poruchový log k lokomotivě 753.604, což je jedna konkrétní lokomotiva typu 753.6). Aby bylo možné tato data propojit a efektivně sledovat ve vyšším kontextu, situace si vyžádala nutnost jednak existence tabulky řad (typů) vozidel, kterým přísluší jednotlivé rejstříky pro dekódování poruch, a dále tabulky konkrétních vozidel, která přiřadí jednotlivá vozidla dané řadě. Existence druhé ze zmíněných tabulek má navíc smysl i z jiného důvodu, a to sice možnosti jejího rozšíření o doplňující data jako je provozovatel, majitel, výrobce či země registrace daného vozidla, s jejichž pomocí získáme rozšiřující možnosti filtrování dat.

4.7.1 Vytvoření modelů řad a vozidel

Model řad nazvaný VehClass, jak již bylo zmíněno, obsahuje množinu řad vozidel ve vztahu ke konkrétnímu rejstříku. Jeho struktura je následující:

```
public class VehClass
    {
        [Key]
        [Display(Name = "ID řady")]
        public int ID { get; set; }
        [Display(Name = "Označení řady")]
        public string Class { get; set; }
        [Display(Name = "ID rejstříku")]
        public int RegisterID { get; set; }
    }
```

Ukázka kódu 17 - definice třídy VehClass

Model jednotlivých vozidel Loco je o něco rozsáhlejší. Obsahuje identifikátor konkrétního vozidla, řadu, které dané vozidlo přísluší, evropské registrační číslo vozidla, název provozovatele (dopravce nebo aktuální nájemce, který vozidlo provozuje), majitele, výrobce a stát, ve kterém je dané vozidlo zaregistrováno:

```
public class Loco
    {
        [Key]
        [Display(Name = "ID")]
        public int ID { get; set; }
        [Display(Name = "Číslo")]
        public string VehID { get; set; }
        [Display(Name = "Řada")]
        public string VehClass { get; set; } //FK
        public string EVN { get; set; }
        [Display(Name = "Provozovatel")]
        public string Operator { get; set; }
        [Display(Name = "Majitel")]
        public string Owner { get; set; }
        [Display(Name = "Výrobce")]
        public string Manufacturer { get; set; }
        [Display(Name = "Stát")]
        public string Country { get; set; }
    }
```

Ukázka kódu 18 - definice třídy Loco

Na rozdíl od předchozích tabulek nejsou tomto případě k dispozici strojová data, ze kterých by bylo možné čerpat data pro naplnění databáze. Zadání správných údajů pro korektní spárování dat tak záleží na uživateli. Zatímco doplňující data o vozidle (provozovatel, majitel a další) nejsou pro spárování podstatná, poskytují možnosti filtrování výsledků například dle země, ve které je vozidlo zaregistrováno. Naopak zadání správných informací o identifikačním čísle a řadě (typu) vozidla je zásadní, neboť bez něj by nedošlo ke správnému spárování vozidla s rejstříkem pro vyhodnocení poruch.

4.7.2 Vytvoření stránek pro zobrazení a práci s entitami

Vytvoření základní struktury stránek je opět docíleno pomocí CRUD. Na rozdíl od předchozích situací tentokrát je nutné využít také stránku Create pro zadání dat. Stránka obsahuje formulář, který nabízí uživateli zadání dat pro každou položku tabulky. Náhled stránky Create pro seznam vozidel můžeme vidět na Obr. 11.

Vytvořit nový záznam

Vozidlo

Číslo

Řada

EVN

Provozovatel

Majitel

Výrobce

Stát

Create

Zpět na seznam

Obr. 11 - formulář pro zadání dat o vozidle do databáze
Zdroj: vlastní zpracování

4.7.3 Vyhodnocení poruchy na základě spárování s rejstříkem

Nyní dojde k nastínění, jak reálně probíhá z hlediska aplikační logiky samotné vyhodnocení dané poruchy. Detaily dané poruchy lze vyvolat v daném řádku seznamu poruch kliknutím na odkaz „Podrobnosti“.

Obr. 12 - odkaz "Podrobnosti" v řádku dané poruchy
zdroj: vlastní zpracování

Použití daného odkazu vede k zobrazení stránky Details. Doplňujícím parametrem samotného odkazu je identifikátor dané poruchy. Daný identifikátor si převezme aplikační logika na straně back-endu stránky s detaily poruchy pro vyhledání vhodného rejstříku pro její dekódování.

Obr. 13 - cíl odkazu na zobrazení detailu poruchy
zdroj: vlastní zpracování

Samotné dekódování probíhá následovně: Zprvu proběhne kontrola, zda daný identifikátor je platný (nachází se v rejstříku poruch). Pokud ano, je volána funkce GetDetail, jejímž argumentem je samotná struktura Fault obsahující data dané poruchy.

```
public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }
            Fault = await _context.Fault.FirstOrDefaultAsync(m => m.ID == id);
            if (Fault == null)
            {
                Result = "CHYBA! ID poruchy nenalezeno.";
                return NotFound();
            }
            DisplayDetail = GetDetail(Fault);
            if (DisplayDetail == null)
            {
                return Page();
            }
            return Page();
        }
```

Ukázka kódu 19 - zpracování identifikátoru poruchy

Funkce GetDetail() využívá samostatně definovaného modelu DisplayDetail obsahujícího veškeré proměnné nezbytné pro zobrazení detailu poruchy. Funkce zprvu na základě ID dedikovaného logu, který je součástí struktury Fault vyhledá v rejstříku logů odpovídající log. Následně vyhledá v tabulce vozidel vozidlo takové, které odpovídá identifikátoru vozidla z daného chybového logu. Pakliže je vozidlo nalezeno, pokračuje vyhledávání v tabulce registrů, kde hledá odpovídající řadu vozidla. Pakliže dojde v registru k nalezení shody, prohledává nadále tabulku FaultDetail, ve které dojde k vyhledání seznamu chyb odpovídajícímu danému vozidlu. Z této množiny následně vyhledá odpovídající zkratku dané poruchy. Jedná se tak o rozsáhlý databázový dotaz nad několika tabulkami. Pro jeho složitost bylo k diagnostice využito konzole. Samotný kód spojení tabulek prostřednictvím EF Core a výstup konzole při hledání popisu dané poruchy je následující:

```
var Log = _context.FaultLog.Find(Fault.LogID); //Najde log, kam chyba patří
                var Locomotive = Log.Locomotive;
                Console.WriteLine("Hledám lokomotivu: " + Locomotive);
                var LocomotiveClass = _context.Loco.Where(veh => veh.VehID == Locomotive).Select(veh =>
veh.VehClass).SingleOrDefault();
                if (LocomotiveClass != null)
                {
                    Console.WriteLine("Lokomotiva " + Locomotive + " nalezena. Řada: " + LocomotiveClass);
                }
                else
                {
                    Result = "CHYBA! Lokomotiva " + Locomotive + "v databázi vozidel nenalezena.";
                    Console.WriteLine("CHYBA! Lokomotiva " + Locomotive + "v databázi vozidel nenalezena.");
                    return null;
                }
                var RegToUse = _context.Register.Where(reg => reg.Locomotive == LocomotiveClass).Select(reg =>
reg).SingleOrDefault();
                if (RegToUse != null)
                {
                    Console.WriteLine("Rejstřík pro řadu " + LocomotiveClass + " nalezen. Verze: " + RegToUse.Version + " SW: "
+ RegToUse.Software);
                }
                else
                {
                    Result = "CHYBA! Rejstřík chyb pro řadu " + LocomotiveClass +  " nenalezen.";
                    Console.WriteLine("CHYBA! Rejstřík chyb pro řadu " + LocomotiveClass + " nenalezen.");
                    return null;
                }
                var FdToUse = _context.FaultDetail.Where(fd => fd.Register_ID == RegToUse.ID).Select(fd => fd);
                if (FdToUse == null)
                {
                    Result = "CHYBA!! ID registru řady nesouhlasí s ID registru chyby.";
                    Console.WriteLine("CHYBA!! ID registru řady nesouhlasí s ID registru chyby.");
                    return null;
                }
                else
                {
                    Console.WriteLine("Nalezeno: " + FdToUse.Count().ToString() + " záznamů");
                }
                var DesToUse = FdToUse.Where(des => des.zkratka == Fault.zkratka).Select(des => des).SingleOrDefault();
                if (DesToUse != null)
                {
                    Console.WriteLine("Chyba vyhodnocena jako " + DesToUse.nazev + ".");

                }
                else
                {
                    Result = "CHYBA! Odpovídající záznam pro chybu " + Fault.zkratka + " v rejstříku nenalezen.";
                    Console.WriteLine("CHYBA! Odpovídající záznam pro chybu " + Fault.zkratka + " v rejstříku nenalezen.");
                    return null;
                }
```

Ukázka kódu 20 - Rozsáhlý databázový dotaz pro dekódování poruchy

```
Hledám lokomotivu: 753.604
Lokomotiva 753.604 nalezena. Řada: 753.6
Rejstřík pro řadu 753.6 nalezen. Verze: 2 SW: 17L21
Nalezeno: 326 záznamů
Chyba vyhodnocena jako Odpojení trakčního motoru %1 přepínačem.
```

Obr. 14 - výstup konzole databázového dotazu pro vyhledání definice poruchy
zdroj: vlastní zpracování

Samotné vyhledání odpovídajícího záznamu však z hlediska získání všech podrobností o poruše nemusí být dostačující. Některé poruchy totiž obsahují ukazatel tzv. zdrojového bitu. Pomocí zdrojového bitu, je-li definován, je možné získat další doplňující informace o poruše, je však nutné jej prvně vyhodnotit. Jak je ostatně možné vidět také z výstupu konzole na Obr. 14, v případě dané poruchy bylo zjištěno, že se jedná o odpojení trakčního motoru přepínačem, jedna informace zde však chybí. Znaménko % následované číslem uživateli intuitivně napovídá, že na jeho místo je možné dosadit určitou hodnotu. V tomto případě tato hodnota říká, o jaký trakční motor se konkrétně jedná.

4.7.4 Dekódování zdrojového bitu

Zdrojový bit sestává ze dvou číslic X a Y v šestnáctkové soustavě, každá z nich tedy může nabývat hodnot od 0 do F. Pomocí hodnoty těchto dvou číslic známých z podrobností chyby v tabulce Fault lze zjistit, jaké bity uvedené v odpovídajícím záznamu rejstříku byly příčinou dané poruchy pomocí tzv. tabulky překódování (Tabulka 1). V případě výše uvedeného příkladu poruchy se zkratkou „OMx“ viditelné také na Obr. 12 u lokomotivy 753.604 odpovídající opojení trakčního motoru má zdrojový bit hodnotu [10]. Pokud dojde k vyhledání dané poruchy v rejstříku pro řadu 753.6, je možné vidět, že pole zdrojových bitů obsahuje hodnoty 1, 2, 3, 4 pro číslici X, zatímco pole pro číslice Y je prázdné (Obr. 15). Dle tabulky překódování tak lze určit, že se jedná o odpojení trakčního motoru č. 4.

646	2	319	8627	OMx	Odpojení trakčního motoru %1 přepínačem	"[1, 2, 3, 4, -, -, -, -]"

Obr. 15 - porucha "OMx" v rejstříku řady 753.6
zdroj: vlastní zpracování

číslice	X				Y			
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0								
1				*				*
2			*				*	
3			*	*			*	*
4		*				*		
5		*		*		*		*
6		*	*			*	*	
7		*	*	*		*	*	*
8	*				*			
9	*			*	*			*
A	*		*		*		*	
B	*		*	*	*		*	*
C	*	*			*	*		
D	*	*		*	*	*		*
E	*	*	*		*	*	*	
F	*	*	*	*	*	*	*	*

Tabulka 1 - tabulka překódování zdrojového bitu
zdroj: MSV Elektronika

V rámci aplikace bylo vyhodnocení zdrojového bitu dle tabulky překódování pomocí příkazu switch např. pro číslici X provedeno následujícím způsobem:

```
string pomX = null;
string pomY = null;
var X = Fault.zdrBit.Substring(1, 1);
var Y = Fault.zdrBit.Substring(2, 1);
switch (X)
{
    case "0": pomX = null; break;
    case "1": pomX = pomZdrBit[3]; break;
    case "2": pomX = pomZdrBit[2]; break;
    case "3": pomX = pomZdrBit[3] + " a " + pomZdrBit[2]; break;
    case "4": pomX = pomZdrBit[1]; break;
    case "5": pomX = pomZdrBit[1] + " a " + pomZdrBit[3]; break;
    case "6": pomX = pomZdrBit[1] + " a " + pomZdrBit[2]; break;
    case "7": pomX = pomZdrBit[1] + " a " + pomZdrBit[2] + " a " + pomZdrBit[3]; break;
    case "8": pomX = pomZdrBit[0]; break;
    case "9": pomX = pomZdrBit[0] + " a " + pomZdrBit[3]; break;
    case "A": pomX = pomZdrBit[0] + " a " + pomZdrBit[2]; break;
    case "B": pomX = pomZdrBit[0] + " a " + pomZdrBit[2] + " a " + pomZdrBit[3]; break;
    case "C": pomX = pomZdrBit[0] + " a " + pomZdrBit[1]; break;
    case "D": pomX = pomZdrBit[0] + " a " + pomZdrBit[1] + " a " + pomZdrBit[3]; break;
    case "E": pomX = pomZdrBit[0] + " a " + pomZdrBit[1] + " a " + pomZdrBit[2]; break;
    case "F": pomX = pomZdrBit[0] + " a " + pomZdrBit[1] + " a " + pomZdrBit[2] + " a " + pomZdrBit[3]; break;
}
```

Ukázka kódu 21 - převedení tabulky překódování pro číslici X na strojový kód

4.7.5 Výsledné zobrazení detailu poruchy

Po úspěšném spárování všech tabulek a vyhodnocení zdrojového bitu jsou podrobnosti dané poruchy v aplikaci zobrazeny přehledně v následující tabulce:

Odpojení trakčního motoru 4 přepínačem				
vozidlo	datum	čas	trvání	ujeto
753.604	02.01.2018	07:33:29	Trva	307160
rychlost	poměrný tah	dráha	klasifikace	
0 km/h	+0	307160		

Obr. 16 - zobrazení vyhodnocené poruchy
zdroj: vlastní zpracování

4.8 Filtrování dat

V následující části dojde k zaměření se na aplikační logiku jedné z dalších hlavních funkcí aplikace, a to sice metod filtrování dat dle stanovených kritérií. Možnost filtrování dat poslouží uživateli zejména pro statistické účely, například sledování výskytu poruch daného druhu. Funkce filtrování je využito také pro zobrazení určité podmnožiny všech záznamů konkrétního druhu, například pro zobrazení poruch vztahujících se k jednomu konkrétnímu logu. Díky tomu zobrazení obsahu logu nevyžaduje existenci samostatné stránky, namísto toho je otevřen seznam poruch, obsahující však pouze poruchy obsažené v určeném logu. Právě na seznamu poruch dojde k demonstraci metody filtrování dat v ASP.NET Razor Pages a jejich provedení na konkrétních příkladech.

Metody filtrování dat na rozdíl od metod pro pouhé zobrazení dat vyžadují četnější spolupráci mezi viditelnou částí stránky a kódem v pozadí. Součástí seznamů, které nabízejí filtrování se tak stávají formuláře, které po načtení stránky umožní uživateli nastavit určitá vyhledávací kritéria (Obr. 17). Těmito kritérii může být například text, jehož obsah lze vyhledávat v rámci konkrétní proměnné, ale také výběr ze seznamu známých hodnot dané proměnné. Poté, co uživatel nastaví vyhledávací kritéria, je tento dotaz na serveru zpracován zdrojovým kódem stránky a po jeho

vyhodnocení jsou uživateli opětovně zobrazena tentokrát již vyfiltrovaná data. Zdrojový kód stránky tak není volán pouze během jejího načtení.

V rámci ASP.NET probíhá vyhledávání standardně prostřednictvím LINQ. Data, u kterých je požadována možnost vyhledávání se definují pomocí rozhraní IQueryable. Jedná se prakticky o proměnnou s definovaným typem dat, jejíž hodnotou je právě dotaz nad databází. Například při třídění poruch (tabulka entit typu Fault) dle čísla logu, identifikátoru vozidla, druhu chyby nebo data výskytu je lze definovat následovně:

```
IQueryable<int> LogQuery = from l in _context.FaultLog
                                orderby l.ID
                                select l.ID;

IQueryable<string> LocoQuery = from m in _context.FaultLog
                                orderby m.Locomotive
                                select m.Locomotive;

IQueryable<string> FaultQuery = from o in _context.Fault
                                orderby o.zkratka
                                select o.zkratka;

IQueryable<string> DateQuery = from n in _context.Fault
                                orderby n.datum.ToString().Substring(0, 10)
                                select n.datum.ToString().Substring(0,10);
```

Ukázka kódu 22 - dotazy nad databází pro filtraci dat

Obr. 17 - formulář pro vyhledávání v tabulce Fault

4.9 Export dat

Jedním z bodů zadání je také průzkum možností exportu dat a jeho implementace. Export dat proniká částečně za hranice základní sestavy ASP.NET Razor Pages. Za tímto účelem byl proto do sestavy přidán doplněk CsvHelper, který poskytuje možnosti exportu dat do formátu .csv. Provedení exportu dat je z hlediska aplikační logiky poměrně jednoduché. Do logiky filtru byla přidána instance doplňku CsvHelper, která při zpracování databázového dotazu, jehož výsledek je uživateli zobrazen, uloží daná data také do paměti. Pakliže si uživatel přeje takto získaná data také lokálně uložit na svůj počítač, kliknutím na tlačítko „Export“, které bylo do

stránky s chybami doplněno, dojde k zapsání obsahu paměti do souboru .csv, který je aplikací vygenerován a stažen do počítače uživatele. Implementace záznamu dat do paměti vypadá následovně:

```
var writeFile = new StreamWriter(stream, leaveOpen: true);
            var csv = new CsvWriter(writeFile, CultureInfo.InvariantCulture, true);
            csv.WriteHeader<ExportModel>();
            foreach (var Log in FaultLog)
            {
                foreach (var Fault in Fault.Where(f => f.LogID == Log.ID))
                {
                    var Entry = new ExportModel
                    {
                        Vehicle = Log.Locomotive,
                        zkratka = Fault.zkratka,
                        Date = Fault.datum.ToString(),
                        Time = Fault.cas.ToString(),
                        trvani = Fault.trvani,
                        zdrBit = Fault.zdrBit,
                        rychlost = Fault.rychlost,
                        PT = Fault.PT,
                        kontext = Fault.kontext,
                        cislo = Fault.CisloPor,
                        b0 = Fault.byte0,
                        b1 = Fault.byte1,
                        b2 = Fault.byte2,
                        b3 = Fault.byte3,
                        drahaH = Fault.drahaH,
                        drahaL = Fault.drahaL
                    };
                    csv.WriteRecord<ExportModel>(Entry);
                }
            }
            await csv.FlushAsync();
            await writeFile.FlushAsync();
            await stream.FlushAsync();
```

Ukázka kódu 23 - zaznamenání výstupu filtru do paměti

Funkce, která je následně volána při stisknutí tlačítka Export, je popsána na ukázce kódu níže:

```
public FileStreamResult OnPostExport()
        {
            Stream stream2 = new MemoryStream(StreamContents);
            Console.WriteLine("Export initiated.");
            var d = stream2.Length;
            Console.WriteLine(d);
            return new FileStreamResult(stream, "text/csv") { FileDownloadName =
            "export.csv" };
        }
```

Ukázka kódu 24 - funkce pro stažení výstupního .csv souboru

4.10 Statistika

Další požadovanou funkcí vyjma textové interpretace požadovaných dat je také jejich interpretace vizuální. Tím je myšlena zejména možnost vykreslení grafů. V této podkapitole budou nastíněny metody, jak lze získaná data vyobrazit graficky.

Vzhledem k tomu, že Razor Pages přímo nedisponují grafickými nástroji, kterých by bylo možné pro vykreslení grafů využít, bylo nutné provést analýzu dostupných metod, které se budou jevit pro řešení daného problému vhodné. Po rozsáhlém průzkumu a výběru z nabízených možností bylo rozhodnuto využít open-source nástroje Chart.js. Jedná se o knihovnu napsanou v jazyce JavaScript, která je určená pro vizualizaci dat a nabízí širokou škálu typů grafů. Zveřejněna byla roku 2013 na platformě GitHub, v rámci které se dnes jedná se o druhý nejpopulárnější nástroj pro tvorbu grafů ve webových aplikacích. (15)

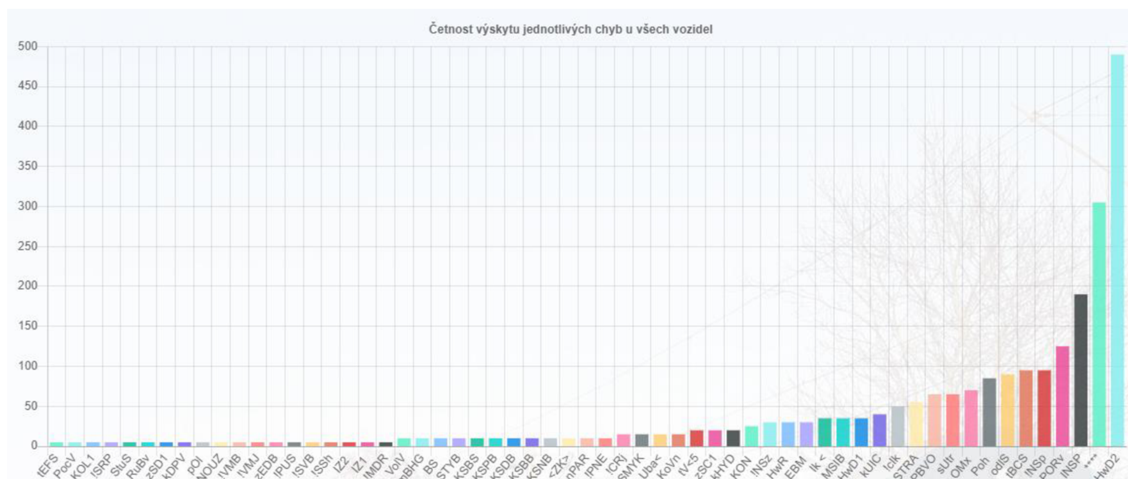
Byť bylo původním záměrem se při vývoji aplikace JavaScriptu vyhnout, neboť s tímto jazykem se autor necítil zcela obeznámený a sebejistý, aby jej v širším měřítku využíval. Avizovaná lehkost a jednoduchost syntaxe i jeho implementace jej přesvědčila o vhodnosti tohoto nástroje.

Samotná instalace spočívá v zahrnutí zdrojového souboru nástroje do adresáře s javascripty ve kmenové složce webu a jeho inicializace v rámci konfiguračního souboru aplikace. Skripty pro vykreslení samotných grafů byly následně zahrnuty přímo do zdrojového souboru cílové stránky.

Za účelem vizuální interpretace obsažených dat byla vytvořena samostatná stránka Statistics. Zdrojový kód stránky během jejího načtení provádí dotazy nad databází pro získání dat interpretovaných v grafech. Data jsou následně předána skriptům grafů, které se postarají o jejich vykreslení. Pro demonstraci vizuální interpretace dat bylo zvoleno použití následujících grafů:

- Četnost výskytu jednotlivých poruch u všech vozidel,
- četnost výskytu všech poruch u jednotlivých vozidel,
- četnost výskytu všech poruch u jednotlivých řad.

Množství a podoba dalších zobrazených grafů včetně možnosti selekce zobrazovaných dat může být později přizpůsobena konkrétním požadavkům společnosti MSV Elektronika. Jeden z vykreslených grafů zobrazující četnost jednotlivých poruch u všech vozidel je vyobrazen na Obr. 18.



Obr. 18 - graf četnosti výskytu jednotlivých poruch u všech vozidel
zdroj: vlastní zpracování

JavaScript pro vykreslení uvedeného grafu vypadá následovně:

```
Chart.defaults.global.legend.display = false;
    var myInvoices;
    var MyData;
    var MyLabels;

    getChartData();

    function showChart() {
        console.log(MyData);
        console.log(MyLabels);

        const colorScheme = […]

        let popCanvasName = document.getElementById("invChart");
        let barChartName = new Chart(popCanvasName, {
            type: 'bar',
            options: {
                title: {
                    display: true,
                    font: {
                        size: 32
                    },
                    text: 'Četnost výskytu jednotlivých chyb u všech vozidel'
                },
                responsive: false,
                scales: {
                    yAxes: [{
                        ticks: {
                            beginAtZero: true
                        }
                    }]
                }
            },
```

```
            data: {
                labels: MyLabels,
                datasets: [{
                    label: 'Invoice data',
                    data: MyData,
                    backgroundColor: colorScheme
                }]
            }
        });
    }


    function getChartData() {
        return fetch('./Statistics?handler=FaultOccurenceChartData',
            {
                method: 'get',
                headers: {
                    'Content-Type': 'application/json;charset=UTF-8'
                }
            })
            .then(function (response) {
                if (response.ok) {
                    return response.text();
                } else {
                    throw Error('Response Not OK');
                }
            })
            .then(function (text) {
                try {
                    return JSON.parse(text);
                } catch (err) {
                    throw Error('Method Not Found');
                }
            })
            .then(function (responseJSON) {
                myInvoices = responseJSON;
                console.log(myInvoices);
                MyData = myInvoices.amountList;
                MyLabels = myInvoices.categoryList;
                //console.log(MyData);
                showChart();
            })
    }
```

Ukázka kódu 25 - JavaScript pro vykreslení grafu pomocí Chart.js

Data pro zobrazení v grafu jsou back-endem získána následujícím způsobem:

```
public async Task<JsonResult> OnGetFaultOccurenceChartData()
        {
            var invoiceChart = new CategoryChartModel();
            invoiceChart.AmountList = new List<uint>();
            invoiceChart.CategoryList = new List<string>();
            var FaultLog = _context.FaultLog.Select(p => p);
            var Fault = _context.Fault.Select(r => r.zkratka);
            List<string> Values_all = new List<string>();
            List<string> Values_specific = new List<string>();
            uint counter1 = 0;
            uint counter2 = 0;
            foreach (var entry in FaultLog)
            {
                foreach (var value in Fault)
                {
                    Values_all.Add(value);
                    if (Values_specific.Contains(value))
                    {
                        counter1++;
                    }
                    else
                    {
```

```
                Values_specific.Add(value);
                counter2++;
            }
        }
    }
    foreach (var value in Values_specific)
    {
        Res_FO currentResult = new Res_FO { Name = value, Count = 0 };
        foreach (var v in Values_all)
        {
            if (v == value)
            {
                currentResult.Count++;
            }
        }
        Res_FaultOccurence.Add(currentResult);
    }
    IEnumerable<Res_FO> query = Res_FaultOccurence.OrderBy(p => p.Count);
    foreach (var inv in query)
    {
        invoiceChart.AmountList.Add(inv.Count);
        invoiceChart.CategoryList.Add(inv.Name);
    }
    return new JsonResult(invoiceChart);
}
```

Ukázka kódu 26 - databázový dotaz pro vykreslení grafu

5 Uživatelské rozhraní (front-end)

Doposud byla věnována pozornost zejména té části aplikace, která je pro uživatele neviditelná - kódům, které stojí za zpracováním a tříděním zobrazovaných dat, stanovení jejich struktur a sestavení databáze. Nyní nastal čas věnovat se pro změnu té části, která je na první pohled viditelná a způsobům, jak jsou tyto dvě neodlučitelné části propojené.

Jedním z požadavků na podobu výsledné aplikace je lehké, moderní a přehledné uživatelské rozhraní. Pokud bychom měli na toto rozhraní vskutku minimalistické požadavky, Razor Pages, jejichž viditelná část je založená na Bootstrapu, nabízejí opravdu velmi čisté základní uživatelské rozhraní, které by nenáročnému vývojáři umožnilo se viditelnou částí stránek příliš nezabývat. Naše požadavky na uživatelské rozhraní jsou přeci jen o něco náročnější a specifické a bylo nutné se tak věnovat jeho úpravě směrem k designovému, ale stále čistému řešení. Důraz je kladen na přehlednost, ale také vizuální atraktivitu zobrazení.

V následující části tak bude popsáno zejména stylování jednotlivých stránek, použitých metod, technik a důvodům jejich užití. Ještě předtím je však vhodné se blíže zmínit o způsobech, kterými jsou mezi zobrazovanou a výpočetní částí stránek předávána data.

5.1 Implementace datového modelu do zobrazované stránky

Soubory .cshtml, jak již bylo nastíněno, umožňují vkládat do HTML kódu pomocí direktivy „@“ různé parametry, funkce, proměnné nebo i celé bloky kódu. Právě toho je využito v případě, chceme-li ve stránce zobrazit nějaká data nebo naopak nějaká data ze stránky předat aplikační logice k vyhodnocení. Typickým příkladem prvního případu je předání výstupu databázového dotazu, příkladem druhé situace jsou pak například parametry filtru, dle kterého se mají data z databáze vytřídit. V Razor Pages je zpravidla součástí každé složky dané stránky jeden soubor definice stránky typu .cshtml a jeden soubor aplikační logiky s dvojitou příponou .cshtml.cs. Použití této direktivy si můžeme demonstrovat například na stránce Faults. Složka „Faults“

obsahuje soubory hlavní stránky „Index.cshtml“ a „Index.cshtml.cs“. Podobně je tomu i u ostatních podstránek typu CRUD – například Details.cshtml a Details.cshtml.cs. Již z předchozích kapitol víme, že stránka Faults slouží k zobrazení určité množiny poruch, obsahuje také však formulář pro nastavení parametrů filtru.

5.1.1 Aplikační logika

Pro lepší demonstraci je vhodné začít ještě jedním krátkým nahlédnutím do souboru s aplikační logikou Index.cshtml.cs, na které již dříve byla demonstrována logika provádění filtrace dat.

Již v samotném záhlaví souboru si lze všimnout direktivy „using“, pomocí které je souboru určeno, k jakým názvoslovím má mít přístup. Mezi ně patří například množina datových modelů.

```
using MSV_LP.Data;
using MSV_LP.Models;
```

Ukázka kódu 27 - použití direktivy using pro import modelu

Následně je definováno názvosloví Faults, které je součástí struktury Pages, tedy struktury všech stránek, které jsou součástí popisovaného kontextu.

```
namespace MSV_LP.Pages.Faults
```

Ukázka kódu 28 - definice názvosloví Faults

Následně je vytvořen lokální kontext, který se pro potřebu naší stránky používá a následně dojde k vytvoření modelu dané stránky nazvaného IndexModel, který se naplní daty z kontextu.

```
private readonly MSV_LP.Data.MSV_LPContext _context;
public IndexModel(MSV_LP.Data.MSV_LPContext context)
    {
        _context = context;
    }
```

Ukázka kódu 29 - vytvoření datového modelu stránky

Dále si je možné všimnout definic typu IList, tedy jakéhosi seznamu dat. Jak je z definic patrné, jedná se o seznamy entit typu Fault a FaultLog, tedy o seznamy

chybových logů a jednotlivých poruch. Právě tyto seznamy jsou stránce při načtení předávány k zobrazení.

```
public IList<Fault> Fault { get; set; }
public IList<FaultLog> FaultLog { get; set; }
```

Ukázka kódu 30 - definice datových seznamů

Dále lze v souboru zpozorovat definice několika proměnných, které mají pomocí parametru SupportsGet = true nastaveno, že mohou přijímat hodnoty pomocí metody Get. To napovídá, že se jedná právě o ty proměnné, kterým mohou být předány hodnoty z formuláře pro nastavení filtru na zobrazené stránce.

```
[BindProperty(SupportsGet = true)]
public string SearchString { get; set; }

[BindProperty(SupportsGet = true)]
public int LogID { get; set; }
```

Ukázka kódu 31 - definice proměnných filtru

Poslední proměnné, které stojí v tomto případě za pozornost, jsou definice a deklarace proměnných typu SelectList. Ty také souvisí s formulářem filtru. Jedná se například o rozevírací seznam všech vozidel, která jsou v databázi k dispozici. Tento seznam je naplněn daty při načítání stránky, aby uživatel mohl z tohoto seznamu vybrat již známé vozidlo, pro které chce zobrazit výsledky.

```
public SelectList locoList { get; set; }
locoList = new SelectList(await LocoQuery.Distinct().ToListAsync());
```

Ukázka kódu 32 - definice rozevíracích seznamů

V závěru kódu jsou proměnné seznamu hodnot k zobrazení naplněny daty s výsledky odpovídající vyhledávaným kritériím filtru:

```
FaultLog = await FaultLogs.ToListAsync();
Fault = await Faults.ToListAsync();
```

Ukázka kódu 33 - naplnění seznamů daty dle kritérií

5.1.2 Viditelná část stránky

Nyní se již lze přesunout ke kódu samotné stránky, tedy k souboru Index.cshtml. V jeho hlavičce je možné vidět způsob provázání stránky s dříve zmíněným IndexModel-em pomocí direktivy „@“. Stránka tak nyní má přístup ke všem proměnným definovaným uvnitř tohoto modelu stránky.

```
@page
@model MSV_LP.Pages.Faults.IndexModel
```

Ukázka kódu 34 - provázání stránky s datovým modelem

Následující řádek kódu předává vnitřní logice překladače název dané stránky, který se má zobrazit v prohlížeči. Jinými slovy určuje, co má být obsahem HTML tagu <title> při generování čistého HTML.

```
@{
    ViewData["Title"] = "Index";
}
```

Ukázka kódu 35 - nastavení názvu stránky

Následuje definice formuláře pro nastavení filtru. Zde si lze všimnout způsobu provázání s datovým modelem:

```
<form>
    <p>
        Vyhledávání: <input type="text" asp-for="SearchString" />
        Dle logu:
        <select asp-for="LogID" asp-items="Model.LogList">
            <option value="">Všechny</option>
        </select>
        Dle vozidla:
        <select asp-for="LocomotiveNo" asp-items="Model.locoList">
            <option value="">Všechny</option>
        </select>
        Dle poruchy:
        <select asp-for="FaultSearchString" asp-items="Model.faultList">
            <option value="">Všechny</option>
        </select>
        Datum:
        <select asp-for="DateNo" asp-items="Model.dateList">
            <option value="">Vše</option>
        </select>
        <input type="submit" value="Filtruj" />
    </p>
</form>
```

Ukázka kódu 36 - definice formuláře nastavení filtru

Další a pravděpodobně nejdůležitější částí stránky pak je definice samotné tabulky pro zobrazení dat. V hlavičce tabulky stojí za pozornost direktiva @Html.DisplayNameFor(). Pokud si vzpomínáte, v kapitole 4.5.2: Vytvoření modelů chybových logů došlo k popisu, jak pomocí atributu [Display(Name = "…") přiřadit určité položce tabulky název, který se má uživateli zobrazit namísto názvu proměnné. Právě direktiva DisplayNameFor se na tento název odkazuje. Namísto

názvu proměnné „CisloPor“ tak v hlavičce tabulky lze vidět například popis „číslo poruchy“.

```
<table class="table">
    <thead>
        <tr>
            ...
            <th>
                @Html.DisplayNameFor(model => model.FaultLog[0].Locomotive)
            </th>
            ...
            <th>
                @Html.DisplayNameFor(model => model.Fault[0].CisloPor)
            </th>
            ...
            </tr>
        </thead>
        ...
```

Ukázka kódu 37 - definice hlavičky tabulky

Následuje pak již definice těla tabulky pro zobrazovaná data. V tomto případě pochopitelně není třeba znát popis zobrazovaných dat, ale jejich hodnoty, proto se namísto direktivy DisplayNameFor používá direktiva @Html.DisplayFor(). Vzhledem k tomu, že zobrazovaných dat je neurčité množství, je možné v .cshtml použít příkaz @foreach k vypsání všech požadovaných hodnot. Ukázka takové konstrukce vypadá v našem případě následovně:

```
<tbody>
       @foreach (var item1 in Model.FaultLog)
       {
           @foreach (var item2 in Model.Fault)
           {
               <tr>
                   <td>
                       @Html.DisplayFor(modelItem => item1.ID)
                   </td>
                   <td>
                       @Html.DisplayFor(modelItem => item1.Locomotive)
                   </td>
                   <td>
                       @Html.DisplayFor(modelItem => item2.ID)
                   </td>
                   <td>
                       @Html.DisplayFor(modelItem => item2.zkratka)
                   </td>
                   <td>
                       @Html.DisplayFor(modelItem => item2.datum)
                   </td>
                     ...
                 </tr>
           }
       }
</tbody>
```

Ukázka kódu 38 - definice těla tabulky

Dále zbývá zmínit konstrukce pro odkaz na podrobnosti o dané poruše. Ten se taktéž pochopitelně generuje pro každý řádek tabulky a jako jeho součást je předáván identifikátor dané poruchy, která se má vyhodnotit.

```
<td>
    <a asp-page="./Details" asp-route-id="@item2.ID">Podrobnosti</a>
</td>
```

Ukázka kódu 39 - definice odkazu na podrobnosti poruchy

5.2 Stylování

Nastal čas se nyní již zabývat samotným grafickým návrhem vzhledu aplikace. Každá ze stránek je dělena na tři základní části – záhlaví, tělo a zápatí stránky. Dále je možné rozdělit stylování stránek na část, která je společně sdílená všemi stránkami a na část, která je specifická pro jednotlivé stránky. Stylování stránky vychází kromě knihovny Bootstrap ze dvou speciálních .css souborů, a to sice site.css obsahující základní styly specifické pro Razor Pages a custom.css obsahující vlastní specifické styly definované autorem. Výsledné zobrazení uživatelského rozhraní je možné vidět na Obr. 21.

5.2.1 Sdílená část zobrazení

Sdílená část návrhu stránek je definována souborem _Layout.cshtml nacházejícím se ve složce Shared. Kód z tohoto souboru je posléze při kompilaci doplněn o kód jednotlivých dílčích stránek. Nachází se zde odkazy na všechny tabulky stylů (.css soubory), knihovnu Bootstrap, JavaScripty a další. Dále zde je možné spatřit definici záhlaví stránek, které je v rámci celé aplikace neměnné. Podobně je tomu také se zápatím. Jedinou částí, která zde není pevně definovaná, je samotné tělo stránky. To je pochopitelné, neboť jeho obsah se odvíjí od definice konkrétních stránek. Tělo je zde tedy definováno uvnitř elementu třídy „container" pouze funkcí @RenderBody(), která se postará o import zobrazovaných elementů dle konkrétního souboru stránky.

Záhlaví

Záhlaví stránky je v HTML uvozováno tagem <header>. Obsahem záhlaví je v levé části logo společnosti MSV Elektronika, v jeho pravé části se pak nachází navigační menu s odkazem na jednotlivé dílčí stránky. Základní stylování záhlaví vychází z předdefinovaných stylů v knihovně Bootstrap a není tak nutné se jím dopodrobna zabývat. Jeho styl byl pouze lehce pozměněn přidáním stínu, které zápatí vrhá na zbytek stránky pro jeho zvýraznění a vytvoření prostorového dojmu. Částí, která však je odlišná a specifická, je pak navigační menu. To bylo autorem upraveno směrem k vlastnímu vzhledu.

Obr. 19 - záhlaví uživatelského rozhraní
Zdroj: vlastní zpracování

Z hlediska stylování bylo záhlaví za účelem vytvoření stínu zahrnuto do třídy .box-shadow, která je v souboru site.css definována následovně:

```
.box-shadow {
  box-shadow: 0 .25rem .75rem rgba(0, 0, 0, .05);
}
```

Za účelem stylování položek nabídky bylo v souboru custom.css předefinováno několik základních tříd knihovny Bootstrap:

```
.navbar-expand-sm .navbar-nav .nav-link {
    padding-top: 0;
    padding-bottom: 0;
    padding-left: 3rem;
    padding-right: 0.5rem;
    transition: 0.3s;
}
.navbar-nav .nav-item {
    font-weight: bold;
    border-right: 2px solid black;
}
```

Ukázka kódu 40 - CSS stylování záhlaví aplikace

Tím bylo docíleno například doplnění designového oddělovače jednotlivých položek nabídky prostřednictvím CSS parametru border-right. Dalším zamýšleným prvkem byla změna formátování odkazu v případě najetí myší. Ten v tomto případě změní barvu písma na modrou barvu loga MSV Elektronika. Tím bylo docíleno zvýšení

dojmu odezvy stránek na uživatelský vstup. V rámci CSS se specifické formátování při najetí myší definuje doplněním parametru :hover za název dané třídy:

```
.navbar-nav .nav-item :hover {
        color: blue;
        border-right: 2px solid #0055A5;
    }

    .navbar-nav .nav-item a.text-dark:focus, a.text-dark:hover {
        color: #0055A5 !important;
    }
```

Ukázka kódu 41 – změna CSS stylování v případě najetí myší

Obr. 20 - Změna vlastností prvků při najetí myší
zdroj: vlastní zpracování

Další zajímavou funkcí CSS, která stojí za zmínku, je možnost definování odlišného stylování například pro první či poslední prvek seznamu. Vlivem nepříliš náročných úprav přednastavené šablony tak bylo docíleno zajímavého a moderního stylování záhlaví stránky.

Tělo

Tělo stránky, zabírající největší plochu okna je místem, ve kterém se zobrazují veškeré důležité informace. Vzhledem k tomu, že jeho obsah vychází z definic jednotlivých stránek, bylo se v případě těla stránky třeba zabývat pouze pozadím a jeho velikostí. Velikost těla stránky byla omezena tak, aby v její spodní části zbyl dostatek místa pro zápatí. Jeho výška se tak rovná celkové výšce okna, zmenšené o výšku záhlaví a zápatí. Je důležité zmínit, že výška těla stránky však nelimituje výšku zobrazovaného obsahu uvnitř. Pakliže je celková výška obsahu těla vyšší, než výška těla stránky, což je v případě výpisu vysokého množství chybových logů předpokládanou situací, lze se uvnitř pole pohybovat prostřednictvím vertikálního posuvníku. Toho bylo docíleno nastavením parametru overflow-y na hodnotu scroll. Záhlaví a zápatí tak při tom zůstane na stejném místě a vždy bude viditelnou součástí okna. Obdobně je tomu v případě, kdy by šířka zobrazovacího okna byla nižší, než je šířka zobrazovaného obsahu. V takovém případě dojde k zobrazení posuvníku

horizontálního. Pro pozadí byl použit obrázek s železniční tématikou, který je uvnitř okna responsivně roztáhnut tak, aby vždy okno vyplňoval. Na obrázek však byl použit efekt vyblednutí s nízkou mírou průhlednosti, aby nedocházelo k vizuálnímu rušení se zobrazovanými daty. Stylování těla stránky je obsaženo v souboru custom.css a má následující podobu:

```
.tbody {
    height: calc(100vh - 60px - 68px);
    overflow-y: scroll;
    background-image: url(/Images/961.jpg);
    background-position: center;
    background-size: cover;
    background-color: rgba(255,255,255,0.9);
    background-blend-mode: overlay;
}
```

Ukázka kódu 42 - CSS stylování těla aplikace

Zápatí

Zápatí stránky obsahuje informace o autorovi aplikace a období jejího vývoje spolu s odkazem na stránku s autorskými právy. Pro vizuální odlišení od zbytku aplikace má jeho pozadí tmavě šedou barvu. Pro budoucí využití je zvažováno, že by zápatí mohlo být využíváno také například k zobrazení legendy k zobrazovaným datům nebo grafům. Stylování zápatí je obsaženo v souboru site.css a má následující podobu:

```
.footer {
    position: absolute;
    bottom: 0;
    width: 100%;
    white-space: nowrap;
    line-height: 60px;
    background-color: rgba(0,0,0,0.85);
}
```

Ukázka kódu 43 - CSS stylování zápatí aplikace

Obr. 21 - výsledný návrh uživatelského rozhraní aplikace
zdroj: vlastní zpracování

Indikátor načítání

Vzhledem k faktu, že provádění některých dotazů nad databází obzvláště v případě diagnostiky poruch může s ohledem na množství dat v databázi hypoteticky trvat i několik desítek sekund a Razor Pages přímo nedisponují vestavěnou indikací práce na pozadí, byly zkoumány možnosti, jak takového zobrazení docílit svépomocí. Požadovaného výsledku bylo dosaženo zejména prostřednictvím HTML a CSS s minimálním využitím JavaScriptu. Indikátor načítání byl vytvořen jako samostatná HTML stránka obsahující pouze indikátor načítání, která se v případě aktivních výpočtů na pozadí vykreslí překryvně v oblasti těla stránky. Ve středu této stránky se nachází animovaný CSS indikátor - několik rotujících kruhů, který dává najevo, že

aplikace zpracovává daný požadavek. Provedení na straně CSS je součástí souboru custom.css uvnitř třídy .loading. O zobrazení daného prvku se stará jednoduchý JavaScript:

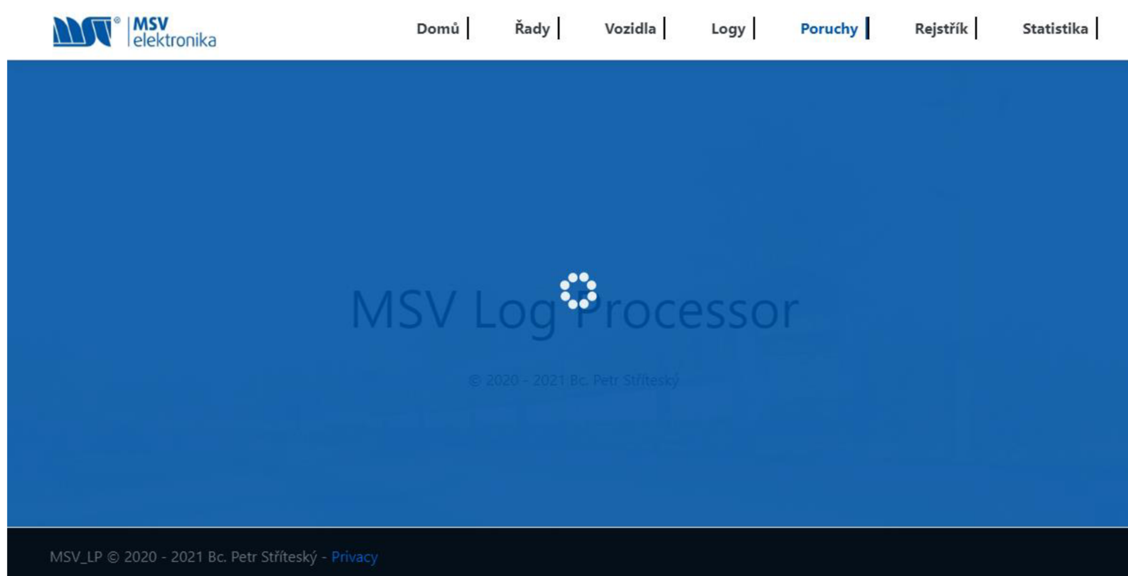
```
function displayBusyIndicator() {
    $('.loading').show();
}

$(window).on('beforeunload', function () {
    displayBusyIndicator();
});

$(document).on('submit', 'form', function () {
    displayBusyIndicator();
});
```

Ukázka kódu 44 - JavaScriptové funkce pro zobrazení indikátoru načítání

Indikátor načítání překryje tělo stránky modrou barvou společnosti MSV Elektronika, zatímco se spustí zobrazení kulatého indikátoru a animace rotace. Výsledek daného řešení je možné vidět na Obr. 22.



Obr. 22 - vykreslení indikátoru načítání
zdroj: vlastní zpracování

5.2.2 Charakteristická část zobrazení

Hlavní stránka

Hlavní, uvítací stránka aplikace obsahuje velkým písmem název aplikace a níže menším písmem opět jméno autora a období vývoje aplikace. Její zdrojový kód je proto velmi jednoduchý:

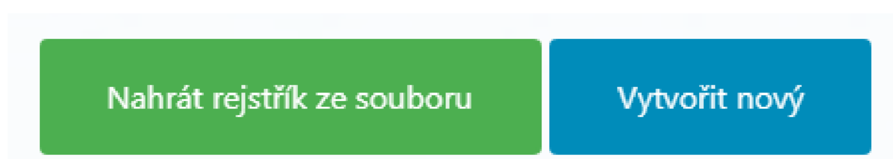
```
<div class="text-center" style="top: 20vh;">
    <h1 class="display-4" style="margin-top: 30vh; font-weight: normal; color: black;">MSV Log Processor</h1>
    <p>&copy; 2020 - 2021 Bc. Petr Stříteský</p>
</div>
```

Obr. 23 - zdrojový kód uvítací stránky aplikace

V případě budoucího vývoje se však předpokládá, že bude tato stránka rozšířena o formulář pro přihlášení uživatele do aplikace.

Stránky se seznamy

Mezi tzv. stránky se seznamy patří stránky Řady, Vozidla, Logy, Poruchy a Rejstřík. Vzhledem k tomu, že jejich stylování má drtivou většinu rysů společných, považuje se za nadbytečné popisovat jej pro každou ze stránek jednotlivě. Mezi hlavní společné rysy těchto stránek patří zobrazení ovládacích prvků a tabulky s danými daty. Způsoby vykreslení dat a filtrů byly popsány již v předchozích kapitolách, nyní tak dojde k omezení popisu pouze na stylování jako takové. Pro přidávání či nahrávání nových záznamů byla v CSS vytvořena specifická výrazná tlačítka, která se nacházejí nad seznamem dat. Jejich podobu je možné vidět na Obr. 24.



Obr. 24 - stylování ovládacích tlačítek
zdroj: vlastní zpracování

Tato tlačítka jsou v tabulce stylů definována následovně:

```
.button { /* BLUE */
    background-color: #008CBA;
    border: 2px solid #008CBA;
    color: white;
    padding: 15px 32px;
    text-align: center;
    text-decoration: none;
    display: inline-block;
    font-size: 16px;
    transition-duration: 0.4s;
    border-radius: 4px;
    cursor: pointer;
}

    .button:hover {
        background-color: white; /* Green */
        color: #008CBA;
        border: 2px solid #008CBA;
    }
```

Ukázka kódu 45 - stylování ovládacích tlačítek

Podobně jako u položek nabídky je u nich standardem změna vlastností při najetí myší. Nejvýraznější položkou těchto stránek je však oblast dat, ve které se nachází tabulka s uvedenými záznamy. Pro lepší vizuální rozlišení zobrazovaných řad byl pozměněn styl tabulky tak, aby se každý sudý řádek podbarvil tmavším pozadím. Toho je docíleno využitím další vlastnosti CSS, a to specifickému stylování pro každý lichý či sudý prvek třídy. Toho je docíleno pomocí parametru :nth-child(even). Jak z názvu vyplývá, je možné stejný parametr použít pro libovolný index prvku dané třídy.

```
.table tr:nth-child(even) {
    background-color: rgba(0,0,0,0.05);
    background-blend-mode: overlay;
}
```

Ukázka kódu 46 - odlišné stylování pro každý sudý prvek třídy

Stránky s grafickým obsahem

Stránkami s grafickým obsahem se myslí stránky určené primárně pro zobrazení grafů a jiných grafických elementů. Ukázku takové stránky v našem případě reprezentuje stránka Statistika. Vzhledem k tomu, že grafické vlastnosti grafů jsou definovány v rámci skriptů samotného grafu, jejichž podoba byla nastíněna

v přechozí kapitole, definice stránek tak obsahují pouze elementy jednotlivých grafů.

```
<div class="chart-container">
    <div class="row">
        <canvas id="invChart" style="width: 100%; height: 500px;"></canvas>
    </div>
    <div class="row">
        <div class="col">
            <canvas id="locoChart" style="width: 100%; height: 500px;"></canvas>
        </div>
        <div class="col">
            <canvas id="classChart" style="width: 100%; height: 500px;"></canvas>
        </div>
    </div>
</div>
```

Ukázka kódu 47 - definice oblasti grafů

Na poli grafů v případě stránky statistika si však můžeme všimnout další vlastnosti knihovny Bootstrap. Tou je možnost efektivního rozvržení stránky do jednotlivých řádků a sloupců. Jak název napovídá, třída *row* odpovídá řádkům, zatímco třída *col* odpovídá sloupcům. Při bližším prozkoumání výše uvedeného kódu si můžeme povšimnout, že první z řádků obsahuje pouze jeden sloupec, bude se v něm tak nacházet pouze jeden graf zaujímající celou šířku řádku. Následující řádek však obsahuje sloupce dva, budou se v něm tak nacházet dva grafy vedle sebe. Výsledná podoba stránky statistika je vyobrazena na Obr. 25.

MSV elektronika

Domů | Řady | Vozidla | Logy | Poruchy | Rejstřík | Statistika |

Statistika

Četnost výskytu jednotlivých chyb u všech vozidel

Četnost výskytu všech chyb u jednotlivých vozidel

753.604 753.605 753.606

Četnost výskytu všech chyb u jednotlivých řad

753.6 2M62

MSV_LP © 2020 - 2021 Bc. Petr Stříteský - Privacy

Obr. 25 - výsledná podoba stránky Statistika
zdroj: vlastní zpracování

6 Shrnutí výsledků

V rámci práce byla vyvinuta webová aplikace dle stanovených požadavků. Samotný vývoj sestával z analýzy možných metod pro docílení požadovaného výsledku, podrobného studia těchto metod z dostupné literatury či dokumentací a následné implementace daných řešení včetně testování jejich funkce. Část použitých metod mi byla před samotným vznikem práce neznámá a bylo tak nutné si jejich použití osvojit. Jednotlivé použité metody byly podrobně popsány a na konkrétních příkladech demonstrována jejich funkce a implementace. Pro účely ověření funkce byla aplikace naplněna testovacími daty zejména z celkem pěti chybových logů ze tří lokomotiv řady 753.6. Na základě těchto dat bylo například zjištěno, že nejčastěji se vyskytující poruchou je Porucha komunikace diagnostického počítače 2. Ta indikuje závady v komunikaci s externě připojenými moduly či zařízeními, v tomto případě s řídicí jednotkou spalovacího motoru Catterpillar. Nutno dodat, že metody třídění a interpretování dat jsou pouze návrhovými koncepty, které mohou být následně využity pro přizpůsobení konkrétním požadavkům uživatele. Díky osvojení si metod třídění a grafické interpretace dat následně tato úprava konkrétním požadavkům uživatele díky nabytým znalostem a zkušenostem nebude zvláště náročná. Nejpřínosnější výstup práce tak byl shledán v osvojení si těchto metod a nabytí drahocenných dovedností, které značně rozšířily autorovy obzory v dané problematice, a veškerý další vývoj této aplikace, projeví-li o ni potenciální uživatelé zájem, popřípadě vývoj zcela jiné webové aplikace pracující s databázemi tak nebude představovat závažnější problém. Podařilo se úspěšně ovládnout práci s databázemi prostřednictvím webového rozhraní, osvojit si nástroj LINQ a EF Core, které umožnily psát nad databází dotazy bez nutnosti kombinace různých programovacích jazyků. Oprášeny byly také doposud spíše základní znalosti pro tvorbu webu, tedy práce s HTML, CSS a frameworkem Bootstrap a došlo také, byť v omezené míře, na použití jazyka JavaScript, který byl autorovi před vznikem práce velikou neznámou. Po ověření správné funkce aplikační logiky byla aplikace obalena jednoduchým, moderním a přehledným uživatelským rozhraním. Estetické vnímání daného uživatelského rozhraní je však věcí čistě subjektivní, a proto nezbývá než doufat, že bude veřejností, respektive potenciálními uživateli přijato.

7 Závěry a doporučení

Práce během svého vzniku prošla všemi nastíněnými fázemi. Ve své teoretické části byly popsány veškeré použité metody včetně historického kontextu jejich vzniku. Důraz byl kladen na důkladné prozkoumání problematiky před její aplikací. To zahrnovalo zejména studium, analýzu, návrh schémat či testování na menších, samostatných prvcích před jejich použitím v cílové aplikaci. V následující praktické části pak tyto metody byly implementovány a s jejich pomocí vznikl koncept funkční webové aplikace schopné provádět předpokládané úkony. Praktická aplikace užitých metod byla demonstrována zejména na ukázkách programovacího kódu, jejich výstupy pak byly naopak prezentovány na obrázcích. Za uvedenými řádky programovacího kódu však pochopitelně stojí široká škála neúspěšných pokusů, které vedly eliminací chyb k jejich postupnému zdokonalování až po zcela funkční kód. To je však v oblasti vývojářství informačních technologií poměrně běžný postup, za který snad v dobré víře není nutné se příliš stydět. Ne nadarmo se říká, že kód, který se zkompiluje a zdánlivě funguje na první pokus, v sobě s velikou pravděpodobností skrývá zásadní chybu. Připouští se však, že některé z použitých metod zde znamenaly pro autora zcela novou zkušenost a jejich implementace tak nemusí být tou nejefektivnější či z programátorského hlediska nejčistší. Vzhledem však k faktu, že existuje veliký předpoklad dalšího rozvoje ve znalostech dané problematiky, bude pravděpodobně možné se s odstupem času k projektu stavět více kriticky. Nutno také dodat, že pro pokročilejší testování aplikace bude zapotřebí ji naplnit značně širší množinou dat, která reálně může čítat i stovky logů ze stovky různých vozidel. Právě z toho důvodu nemohla být provedena rozsáhlejší diagnostika časové náročnosti prováděných úkonů, neboť její provádění pouze na testovací množině by bylo neprůkazné. Nyní nezbývá, než předat nastíněný projekt potenciálnímu uživateli ke zhodnocení, zda v jeho existenci spatří případný potenciál pro další rozvoj, a pokud ano, bude mu postoupen k připomínkování, ze kterého vyvstane nutnost se věnovat dalšímu vývoji a přizpůsobení aplikace jeho požadavkům. Předpokládá se tak, že zásadních změn doznají například možnosti filtrování dat či jejich grafického vyobrazení, neboť ty budou muset být upraveny

dle toho, které výstupy se mu budou jevit jako z informačního hlediska nejhodnotnější.

Obecně tak zpracování a výstupy této práce jsou autorem vnímány jako obrovská množina nových znalostí a zkušeností přinášejících nové možnosti uplatnění v dalších etapách profesního i studijního života. Lze také považovat za veskrze pozitivní, že se práce týká velmi specifické praktické problematiky s určitou mírou pravděpodobnosti, že může být nadále využita k praktickým účelům a jejím výstupem tak nemusí být pouze splnění určité povinnosti pro zakončení studia.

8 Seznam použité literatury

1. **GeeksForGeeks.** C# Programming Language. *GeeksForGeeks.* [Online] 20. Srpen 2020. https://www.geeksforgeeks.org/csharp-programming-language/.

2. **FRIEDMAN, Janice.** When Was C# Created? A Brief History. *C# Station.* [Online] Březen 18, 2020. https://csharp-station.com/when-was-c-sharp-created-a-brief-history/.

3. **TIOBE Software BV.** TIOBE Index. [Online] Leden 2021. https://www.tiobe.com/tiobe-index/.

4. **KAČMÁR, Dalibor.** *Programujeme .NET aplikace ve Visual Studiu .NET.* Praha : Computer Press, 2001. ISBN 80-7226-569-5.

5. **Team at Tutorials Point.** C# - Basic Syntax. *Tutorialspoint.* [Online] https://www.tutorialspoint.com/csharp/csharp_basic_syntax.htm.

6. **NAGEL, Christian, a další.** *C# 2008 - Programujeme profeonálně.* [překl.] David a kol. DIRGA. Brno : Computer Press, 2009. ISBN 978-80-251-2401-71-2.

7. **GUTHRIE, Scott.** Introducing "Razor" – a new view engine for ASP.NET. *ScottGu's Blog.* [Online] Microsoft, 3. Červenec 2010. https://weblogs.asp.net/scottgu/introducing-razor.

8. **JANOVSKÝ, Dušan.** CSS styly - úvod. *Jak psát web.* [Online] https://www.jakpsatweb.cz/css/css-uvod.html.

9. **Bootstrap team.** About. *GetBootstrap.com.* [Online] https://getbootstrap.com/docs/4.1/about/overview/.

10. **Microsoft.** Language Integrated Query (LINQ). *Microsoft Docs.* [Online] 2. Únor 2017. https://docs.microsoft.com/cs-cz/dotnet/csharp/programming-guide/concepts/linq/.

11. —. Entity Framework Core. *Microsoft Docs.* [Online] 20. Září 2020. https://docs.microsoft.com/cs-cz/ef/core/.

12. **CARBONNELLE, Pierre.** Top IDE index. *PYPL Index.* [Online] https://pypl.github.io/IDE.html.

13. **NOVAK, István, a další.** *Visual Studio 2010 and .NET 4 Six-in-One: Visual Studio, .NET, ASP.NET, VB.NET, C#, and F#.* místo neznámé : Wrox, 2010. ISBN: 978-0-470-49948-1.

14. **Microsoft.** Microsoft Details Pricing and Licensing for Visual Studio 2005 and Simplifies Microsoft Developer Network Subscriptions. *Microsoft.* [Online] Microsoft, 21. Únor 2005. https://news.microsoft.com/2005/03/21/microsoft-details-pricing-and-licensing-for-visual-studio-2005-and-simplifies-microsoft-developer-network-subscriptions/.

15. **Borovikov, Ruslan.** Top 10 JavaScript Charting Libraries for Every Data Visualization Need. *Hackernoon.com.* [Online] 27. Duben 2019. https://hackernoon.com/10-javascript-charting-libraries-data-visualization-b77523d23372.

16. **VIRIUS, Miroslav.** *Od C++ k C#.* České Budějovice : Kopp, 2002. ISBN 80-7232-176-5.

17. —. *Programování pro .NET.* Praha : ČVUT v Praze, Fakulta jaderná a fyzikálně inženýrská, 2011. ISBN 978-80-01-04866-4.

18. **ANDERSON, Rick a NOWAK, Ryan.** Introduction to Razor Pages in ASP.NET Core. *Microsoft.* [Online] 2. Prosinec 2020. https://docs.microsoft.com/en-us/aspnet/core/razor-pages/.

19. **JANOVSKÝ, Dušan a Yuhů.** CSS - kaskádové styly. *Jak psát web.* [Online] https://www.jakpsatweb.cz/css/.

20. **ČÁPKA, David.** Kompletní kurz CSS frameworku Bootstrap - Online kurz. *ITnetwork.cz.* [Online] https://www.itnetwork.cz/html-css/bootstrap/kurz.

21. **HERCEG, Tomáš.** Úvod do jazyka SQL. *DotNetPortal.* [Online] 3. Září 2007. https://www.dotnetportal.cz/clanek/50/Uvod-do-jazyka-SQL.

22. **Microsoft.** *Microsoft C# Language Specifications.* místo neznámé : Microsoft Press, 2001. ISBN 0-7356-1448-2.

23. **W3Schools.** Online web tutorials - Learn HTML, Learn CSS, Learn Bootstrap, Learn Javascript, Learn SQL, Learn C#. *w3schools.com.* [Online] Refsnes Data. https://www.w3schools.com.

24. **VIRIUS, Miroslav.** *Programování v C#: od základů k profesionálnímu použití.* [editor] Petr SOMOGYI. Myslíme v... Praha : Grada Publishing, a.s., 2020. ISBN 978-80-271-1216-6.

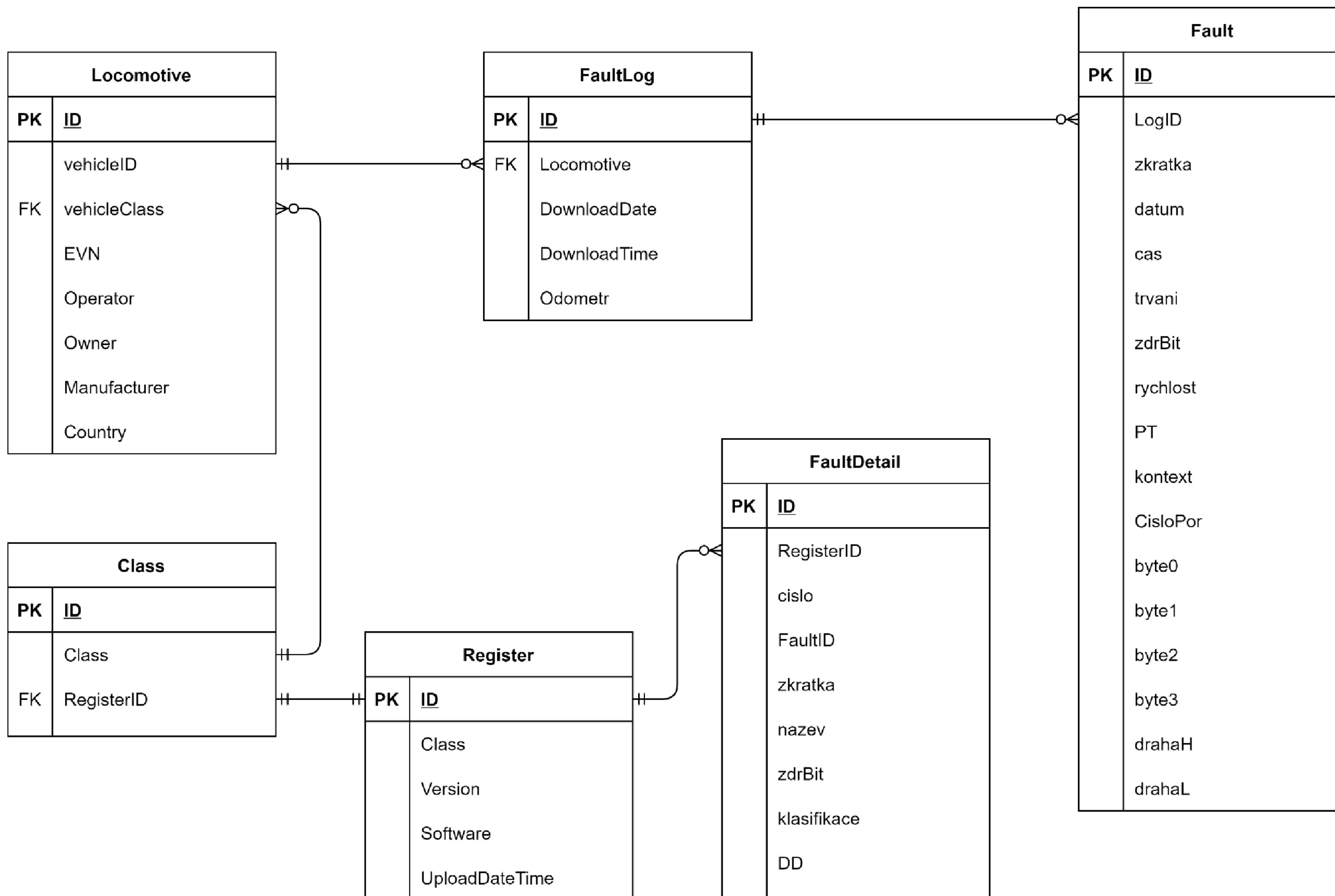
Zdroje použitých dat

1. **MSV Elektronika s.r.o.**, Poštovní 662, 742 13 Studénka

9 Přílohy

1. Databázové schéma aplikace MSV Log Processor

Příloha č. 1



Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor:	**Petr Stříteský**
Studium:	I1900761
Studijní program:	N1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název diplomové práce:	**Zpracování a vizualizace diagnostických hlášení z řídicího systému kolejového vozidla**
Název diplomové práce AJ:	Processing and visualisation of diagnostic messages from a railway vehicle control system

Cíl, metody, literatura, předpoklady:

Na základě stanovených požadavků níže sestavit SW ve formě webové aplikace, která bude zpracovávat, dekódovat, vizualizovat a filtrovat data z chybových hlážení kolejového vozidla dle požadavku uživatele.
Požadavky:

- Zpracování textového logu z lokomotivy - hlavičky i obsahu
- Zatřídění obsahu do databáze a propojení s hlavičkou
- Podrobné zpracování jednotlivých logů: Dekódování názvu, času, trvání, příčiny a doplňujících stavových informací
- Vizualizace logů - tvorba filtrů dle času, hlášení, pohybu, tahu a kontextu (minimálně)
- Dekódování doplňkových dat
- Moderní, lehké a přehledné uživatelské rozhraní
- Průzkum možností exportu dat a jeho implementace

Doc. Ing. M. Virius, CSc.: Od C++ k C#. Kopp, České Budějovice 2002. ISBN 80-7232-176-5 Doc. Ing. Miroslav Virius, CSc.:Programování pro .NET, ČVUT v Praze, Fakulta jaderná a fyzikálně inženýrská, 2011, ISBN 978-80-01-04866-4 C. Nagel, B. Evjen, J. Glynn, K. Watson, M. Skinner: Professional C# 2008.. Wiley Publishing (Wrox) 2008. ISBN 978-0-470-19137-8. (případně Český překlad C# 2005 -- Programujeme profesionálně, Computer press, Brno 2006. ISBN 80-251-1181-4.) D. Kačmář: Programujeme .NET aplikace ve Visual Studiu .NET. Computer Press 2001. ISBN 80-7226-569-5 Microsoft C# Language Specifications. Microsoft Press 2001. ISBN 0-7356-1448-2 Rick Anderson a Ryan Nowak: Introduction to Razor Pages in ASP.NET Core, Microsoft, 2.12.2020, dostupné online na https://docs.microsoft.com/en-us/aspnet/core/razor-pages/ Yuhů, Dušan Janovský: CSS - kaskádové styly, Jak psát web, dostupné online na https://www.jakpsatweb.cz/css/
David Čápka: Kompletní kurz CSS frameworku Bootstrap - Online kurz, ITnetwork.cz, dostupné online na https://www.itnetwork.cz/html-css/bootstrap/kurz W3schools online web tutorials - Learn HTML, Learn CSS, Learn Bootstrap, Learn Javascript, Learn SQL, Learn C#. w3schools.com, dostupné online na https://www.w3schools.com Tomáš Herceg: Úvod do jazyka SQL. DotNetPortal, 3.9.2007, dostupné online na https://www.dotnetportal.cz/clanek/50/Uvod-do-jazyka-SQL

Garantující pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Ing. Filip Malý, Ph.D.
Datum zadání závěrečné práce:	14.1.2018