VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra kybernetiky a biomedicínského inženýrství

Návrh a realizace sesterské aplikace pro RTLS urgentního příjmu Fakultní nemocnice Ostrava

Design and Implementation of RTLS Application for Nurses for Emergency

2018

Tran Hoang An

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra kybernetiky a biomedicínského inženýrství

Zadání diplomové práce

Student: **Bc. An Tran**

Studijní program: N2649 Elektrotechnika

Studijní obor: 3901T009 Biomedicínské inženýrství

Téma: Návrh a realizace sesterské aplikace pro RTLS urgentního příjmu Fakultní nemocnice Ostrava
Design and Implementation of RTLS Application for Nurses for Emergency

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem diplomové práce je vytvořit webovou aplikaci pro personál urgentního příjmu Fakultní nemocnice Ostrava, pomocí které bude možné ovládat RTLS na uživatelské úrovni.

Body zadání:
1. Seznámení se s prostředím urgentního příjmu.
2. Analýza, návrh a specifikace webové aplikace. K návrhu použijte např. jazyk UML.
3. Vytvoření back-end aplikace komunikující s databází Firebird. Zvolte vhodný nástroj a implementujte aplikaci (ASP.NET, C#, Java, Node Js, ...)
4. Vytvoření front-end uživatelské aplikace. Zvolte vhodný nástroj a implementujte aplikaci (ASP.NET, Angular, React, ...)
5. Nasazení aplikace na server a testovaní dle vytvořené specifikace.
6. Zhodnocení a závěr.

Seznam doporučené odborné literatury:

[1] MILES, Russ a Kim HAMILTON. *Learning UML 2.0*. Sebastopol, CA: O'Reilly Media, c2006. ISBN 978-0596009823.
[2] ALBAHARI, Joseph a Ben ALBAHARI. *C# 6.0 in a nutshell: the definitive reference*. Sixth edition. Sebastopol, CA: O'Reilly Media. ISBN 978-1491927069.
[3] BANKS, Alex a Eve PORCELLO. *Learning React*. Sebastopol, CA: O'Reilly Media, 2017. ISBN 978-1-4919-5462-1.
[4] BORRIE, Helen. *The Firebird book: a reference for database developers*. Berkley, CA: Apress Verlag, c2004. ISBN 978-1-59059-279-3.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jaromír Konečný, Ph.D.**

Konzultant diplomové práce: doc. Ing. Marek Penhaker, Ph.D.

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018

doc. Ing. Jiří Koziorek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 1. dubna 2018 .....

Tímto bych chtěl poděkovat vedoucímu diplomové práce panu Ing. Jaromírovi Konečnému, Ph.D. za jeho vedení a rady. Také děkuji naší studijní skupině, bez které bych magisterské studium nezvládl.

Rád bych poděkoval také své přítelkyni Marušce za její neutuchající podporu nejen během studia, rodině a mému psovi Beníškovi za podporu při tvorbě této diplomové práce, a to především vytvořením pohodlné atmosféry při práci.

Abstrakt

Tato diplomová práce se zabývá analýzou, návrhem a implementací webové aplikace na urgentním příjmu Fakultní nemocnice Ostrava. Zmíněná webová aplikace slouží jako měřící nástroj personálu urgentního příjmu pro správu RTLS systému. Práce popisuje vybrané webové klientské a serverové technologie. Detailněji je také popsán princip fungování technologií, které byly použity v praktické části diplomové práce. Další kapitolou je vlastní návrh řešení, kde je popsána analýza současné situace a popis řešení. Tyto kapitoly pojednávají o nynější situaci a identifikaci problému. Poté je popsán průběh tvorby softwarového řešení na základě zpracovaných analýz. Poslední části práce obsahují popis testování, nasazení do produkce, nástroje, které byly použity během vývoje a retrospektivní zhodnocení z pohledu autora. V závěru jsou poté konfrontovány výsledky této práce.

Klíčová slova: RTLS, C#, ReactJS, WebAPI, lokalizace

Abstract

This diploma thesis is about analysis, design and implementation of web application on Emergency of Faculty Hospital of Ostrava. The purpose of this web application is measuring instrument for personal hospital of RTLS system. This thesis describes choosen web client and server technologies. The technologies which were actually used in the practical part are described more detaily. Next part analysing the current situation and description of implemented solution. This part dedicated for analysing of the current situation and indentification of the issues. The last part of this thesis contains the test phase, deploy phase, the tools which where used and retrospective evaluation from the author's view. Follows by conclusion where the results are confronted.

Key Words: RTLS, C#, ReactJS, WebAPI, localization

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
CA	–	Certificate Authority
CSS	–	Cascade Style Sheet
DI	–	Domain Driven Design
DI	–	Dependency Injection
DNS	–	Domain Name Server
DOM	–	Document Object Model
DTD	–	Document Type Definition
DTO	–	Data Transfer Object
ECMA	–	European Computer Manufacturers Association
FNO	–	Fakultní nemocnice Ostrava
HTTP/S	–	HyperText Transfer Protocol / Secured
HQL	–	Hibernate Query Language
IDE	–	Integrated Development Environment
IIS	–	Internet Information Services
IMAP	–	Internet Message Access Protocol
JS	–	JavaScript
JSON	–	JavaScripObjectNotation
JSX	–	Java Serialization to XML
MVC	–	Model View Controller
NPM	–	Node Package Manager
OOP	–	Object Oriented Programming
ORM	–	Object Relational Mapping
POP3	–	Post Office Protocol
RSA	–	Rivest–Shamir–Adleman
RTLS	–	Real Time Location System
SMTP	–	Simple Mail Transfer Protocol
TCP	–	Transmission Control Protocol
UC	–	Use Case
UML	–	Unified Modeling Language
XML	–	eXtensive Markup Language
XSD	–	XML Schema Definition
WWW	–	World Wide Web

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V dnešní době automatizace a digitalizace je třeba využít tohoto trendu a převést do praxe metody pro zefektivnění práce v těch oblastech, kde to ještě před pár lety kvůli technickým bariérám nebylo možné. Jednou z oblastí, kterou lze moderními technologiemi zefektivnit je urgentní příjem nemocnic. Na toto oddělení, kde hrají i sekundy otázku života a smrti, je kladen vysoký nárok na monitorování a sledování ošetřovacího cyklu pacienta. Přínos digitalizace lokace a provedených ošetřovatelských výkonů v reálném čase je nezměrný. Například zpětným statistickým vyhodnocením lze posoudit, jaké fáze vyšetření bývají časově nejnáročnější. Vymezením kritických oblastí a jejich náprav lze zvrátit pomyslný jazýček vah na kladnou stranu při záchraně člověka. Jestliže jsou k dispozici data v reálném čase, nic nebrání tomu je logicky a přehledně zobrazit. Dalším přínosem je tedy nepochybně lepší orientace v současné situaci pro nově příchozí personál nebo při hromadných nehodách.

Cílem diplomové práce je vytvořit aplikaci pro personál urgentního příjmu Fakultní nemocnice Ostrava, pomocí které bude možné ovládat systém lokalizace v reálném čase (Real Time Location System dále jen RTLS) na uživatelské úrovni. Cílem je tedy v první řadě seznámit se s nynější situací na urgentním příjmu, pochopit a lokalizovat problémy, které trápí toto oddělení. Bylo potřeba se také poučit z dřívějšího pokusu o lokalizačním systému pacientů. Poté je cílem tyto poznatky přenést do užitkových a vývojových diagramů. Na základě těchto komplexních diagramů poté představit návrhy aplikace a paralelně sestavit backendové prostředí pro chod aplikace. Po schválení návrhu už zbývá jen podle něj naprogramovat skutečnou frontendovou část. V průběhu vývoje je důležité aktivně komunikovat se zaměstnanci urgentního příjmu a průběžně konzultovat případné nedostatky či zlepšení. Po odladění obou částí aplikace je třeba nasadit celkové řešení na server nemocnice a provést konečnou fázi testování v reálném provozu.

Celá tato diplomová práce se skládá ze čtyř kapitol. Diplomová práce začíná úvodní první kapitolou, která uvádí čtenáře do problematiky. Další kapitola podrobně popisuje technologie pro tvorbu webových aplikací. Popsané technologie byly vybrány na základě implementovaných technologií nebo z důvodu toho, že byly předchůdci implementovaných technologií a bylo potřeba vyzdvihnout jejich rozdíly. Další kapitola pak obsahuje vlastní návrh řešení, v této části práce je popsána analýza současné situace, postup vývoje a vysvětlení architektury aplikace. Poslední kapitola pak hodnotí celou práci, nastiňuje budoucí motivaci a uzavírá tuto diplomovou práci.

2 Technologie pro tvorbu webových aplikací

V této kapitole budou popsány technologie a koncepty, které byly použity v praktické části nebo mají úzkou spojitosti, bylo nutné je popsat pro vytvoření uceleného obrazu. Jsou také popsány přilehlé oblasti jako komunikace, typy dat či zabezpečení, které souvisí s touto problematikou.

2.1 Klient-Server

Jde o typ architektury, která odděluje síťově, topograficky a logicky část klientskou (dále jen klient) a serverovou (dále jen server). Některá logika či část programového postupu je vykonávána na klientské straně a ta druhá část na serveru. Klient překládá klientský požadavek tak, aby byl přijatelný serverem a čeká od něj odpověď, kterou překládá zpět tak, aby byla srozumitelná klientovi. Ten ji pak na obrazovce prezentuje uživateli. Klient je tedy aplikace na koncovém zařízení uživatele, která je nejčastěji reprezentována rozhraním (interfacem). Serverová část je naopak část, která je mimo uživatelské zařízení. Nejčastěji je reprezentována databází držící data, která jsou pak poskytována klientovi nebo naopak klientem posílána data pro ukládání. Na obrázku 1 je zobrazena architektura typu klient-server.[1]

Výhody

Největší výhoda této architektury je jasné oddělení aplikační logiky na dvě části. První částí je klientská strana, která je aktivní, posílá a přijímá žádosti na nebo ze severu a druhá část je serverová strana, která je pasivní, naslouchá všem požadavkům na síti, které se správně autorizují. Další výhoda je snadná údržba a rozšiřitelnost. Například je možné nahradit, opravit, modernizovat, přemístit server, aniž by to klienti poznali, nebo tím byli nějak ovlivněni. Další výhodou je bezpečnost. Tím, že jsou všechna data (i citlivá osobní data) bezpečně uložena na serverech, která jsou bezpečnější než většina klientů, je menší pravděpodobnost úspěšného úmyslného útoku či odcizení. Servery jsou také snadněji kontrolovatelné, co se týče přístupu a jeho sledování.

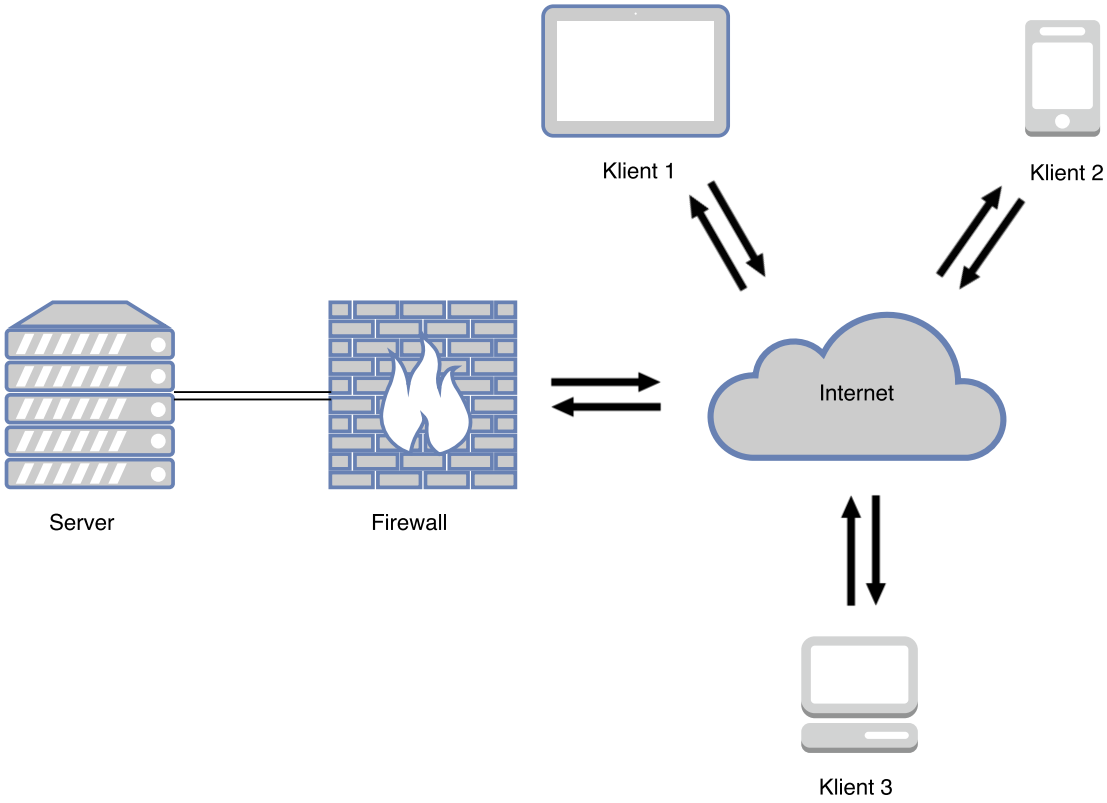
Nevýhody

Za nevýhodu se dá považovat nerobustnost[1]. Pokud dojde k výpadku serveru, veškeré žádosti klientů jsou neobslouženy a vráceny s chybovým kódem 5xx, což jsou chybové kódy serveru (většinou jde o 503 Služba nedostupná (Service unavailable)). Mezi další nevýhody řadíme přetěžování sítě. Počet požadavků klientů na server může přesáhnout mez, kterou server zvládne obsloužit. Může dojít ke krachu serveru, který vede k první nevýhodě (viz. výše).

Výše zmíněné nevýhody jsou již ale překonány. Nerobustnost řešení lze vyřešit například aplikováním serverových nodů s loadbalancerem. v případě, že by některý server vypadl, lo-

[1]Robustný – program naprogramovaný tak, že se pokusí za každou cenu běžet dál.

adbalancer přesměrovává veškeré požadavky na zbývající žijící nody. Tím je zajištěna robustnost řešení. Další překonanou nevýhodou, která byla popsána výše je přetěžování sítě. Toto lze obejít různými hardwarovými i softwarovými řešeními, čehož lze dosáhnout limitací požadavků v časovém intervalu.[2, 3]



Obrázek 1: Topologie sítě klient-server. Na levé straně je serverové prostředí, které může obsahovat více serverů nebo další technologie jako load-balancer, Domain Name Server (dále jen DNS) server, oddělené databázové stroje apod. Serverové prostředí je odděleno firewallem (s nebo bez různých restrikcí) pro bezpečnost před útoky. Příchozí požadavky ze sítě přicházejí přes internet od různých klientů. Klientem může být cokoliv, co dokáže vytvořit a odeslat správně požadavek. Ten je pak obsloužen serverem a vrácen zpět.

2.1.1 Způsob komunikace

Hypertextový komunikační protokol

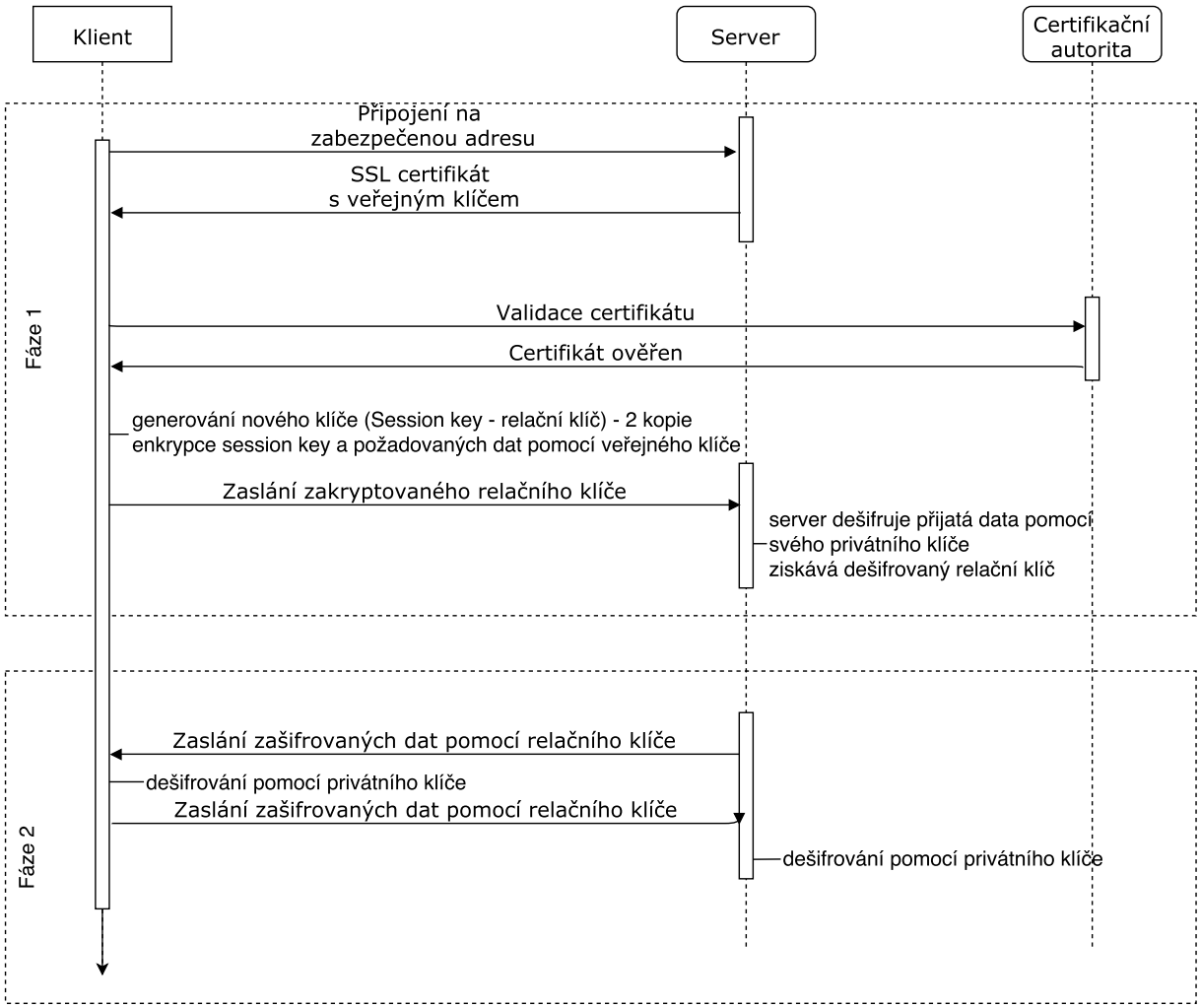
Hypertext Transfer Protocol (dále jen HTTP) je komunikační protokol, který patří k nejpoužívanějším způsobům komunikace dvou zařízení. Tento typ funguje způsobem request-response (požadavek-odpověď). Klient pošle serveru dotaz obsahující hlavičku a tělo dotazu. Hlavička obsahuje různé informace. Například označení požadovaného dokumentu, údaje o prohlížeči apod. Server přijme (pokud je validní) požadavek a poté odpoví pomocí několika řádků textu popi-

sujících výsledek dotazu (zda se dokument podařilo najít, jakého typu dokument je atd.), za kterými následují data (pokud byla požadována).

Tato komunikace se odehrává po síti, nejčastěji po internetu. HTTP protokol je ovšem nezašifrovaný, nezabezpečený a nezabezpečující integritu dat kanálu, který lze odposlouchávat, číst cizí informace zvenčí či podsouvat falešná nebo škodlivá data. Tento protokol obvykle používá TCP/80 port. Jak již bylo řečeno, HTTP komunikace je otevřená a přenášené informace mohou být po cestě mezi klientem a serverem odposlouchávány. Kdokoliv tedy může číst data, hesla a další citlivé údaje. Aby se tomuto odposlechu zamezilo, byl vyvinut protokol HTTPS (secured – zabezpečený).[4]

Základem zabezpečené komunikace je SSL certifikát, který zajišťuje šifrovanou komunikaci mezi serverem a klientem. Tento certifikát se skládá z veřejného klíče a privátního klíče. Zákaznický certifikát, který je vystavován certifikační autoritou (CA – certificate authority), je nainstalován na serveru a tím se také server prokazuje klientovi. Zprostředkující certifikát je oproti tomu nainstalován v prohlížeči a během komunikace se jím prokazuje důvěryhodnost certifikátu protistrany. Tento zabezpečený kanál většinou komunikuje na portu TCP/443.[5]

Na obrázku 2 je popsaný sekvenční diagram procesu HTTPS komunikace. Při první fázi je použit veřejný klíč pro vytvoření zašifrované komunikace. Tato fáze je pouze inicializační a probíhá jen jednou. Druhá fáze je pak použita při každé další komunikaci a je použit privátní klíč.



Obrázek 2: Aktivitní diagram komunikace HTTPS. v první fázi je vyslán požadavek klienta na připojení zabezpečené stránky. Server poté vrací certifikát s veřejným klíčem, který je zpracován po odeslání certifikační autoritou. Klient poté přijímá odpověď (ta je kladná či zamítavá). Klient pak na základě veřejného klíče generuje dvě kopie relačního klíče. Tento klíč s daty (pokud jsou) je poté pomocí ověřeného veřejného klíče zakryptován a odeslán na server. Ten následně dešifruje požadovaná data pomocí svého privátního klíče. V případě úspěchu získává dešifrovaný relační klíč. Tímto krokem končí první fáze, kdy se navzájem ověřují certifikáty. V druhé fázi už probíhá ověřená komunikace. Po klientském požadavku server odesílá zpět požadovaná data zašifrovaná pomocí relačního klíče. Klient tato data dešifruje s privátním klíčem a získává tak dešifrované údaje. V opačném směru funguje tato fáze stejně. Klient zakryptuje data pomocí relačního klíče a server poté dešifruje data pomocí svého privátního klíče.

RSA šifrování

Tento typ šifrování se nejčastěji používá pro kryptování privátního klíče. Patří mezi asymetrické kryptografie, přičemž se od symetrické liší tím, že nemá pouze jeden klíč, pomocí kterého se přenášená zpráva šifruje a inverzním postupem dešifruje. Asymetrické šifry mají klíče dva. Soukromý a veřejný. Zašifrovaná zpráva je poslána přes nezabezpečenou síť. Obsah zůstane ale

utajen v situaci, kdy útočník nemá privátní klíč pro rozšifrování. Principem RSA šifrování je předpoklad, že rozložení velkého čísla na součin prvočísel je takřka nemožné. V procesu faktorizace (rozložení čísla na součin prvočísel), tedy $n = p.q$, není znám žádný algoritmus faktorizace, který by pracoval v polynomiálním čase[2].[6]

2.1.2 Formáty dat

Pro správný chod aplikací je nutné, aby mezi sebou nějakým způsobem komunikovali přes jakýkoliv komunikační standart (POP3, SMTP, IMAP, HTTP atd.). Existuje mnoho typů dat, ale v této práci si uvedeme dva nejmodernější a nejpoužívanější způsoby, jak formátovat data.

Rozšiřující značkovací jazyk

Extensible Markup Language (dále jen XML) je značkující jazyk, za jehož vznik a standardizaci vděčí konsorciu W3C[3], slouží pro serializaci dat. Jedná se o formát dokumentů, který předepisuje, jak zapisovat data společně s užitečnými daty. XML tagy nejsou předdefinované, lze si tedy definovat své vlastní. Tento jazyk a jeho kód ve své podstatě nic nevykonává. Pouze drží informaci, kterou přenáší. S XML lze vyměňovat data i mezi dvěma naprosto nesourodými systémy.

Tagy je možné předdefinovat v souboru dokument pro definici typů (Document Type Definition dále jen DTD), jestliže potřebujeme validovat na vstupu xml dokument s daty ve správném formátu. Lze automaticky kontrolovat, zda vytvářený či přijímaný dokument odpovídá definici, která je v DTD. Tento validační dokument ale není jediným způsobem, jak zkontrolovat správnost struktury XML dokumentu. Jelikož DTD neobsahuje možnost kontrolovat typy dat (čísla, řetězce znaků apod.), mnohem častěji se používají XML schémata nebo také definice XML schémy (XML Schema Definition (XSD)).

Další vlastností XML je, že v jednom dokumentu můžeme používat najednou nezávisle na sobě několik druhů tagů pomocí jmenných prostorů (namespaces). Z toho plyne, že v jednom XML dokumentu můžeme používat několik různých DTD nebo schémat bez konfliktů v pojmenování tagů.[7, 8]

Kódování XML dokumentu je vždy Unicode[4], kódování lze ale podle potřeb změnit. Aby byl XML dokument považován za správný musí splňovat alespoň následující body:

1. Elementy mohou být vnořeny, nemohou se ale překrývat. Znamená to, že každý tag musí být celý v jiném tagu. Ve výpisu 1 jsou v našem případě v tagu *produkt* vnořené tagy *nazev* a *cena*.

[2] Nicméně je očekáváno, že s příchodem opravdu výkonných kvantových počítačů je tento typ šifrování ohrožen.
[3] World Wide Web Consorcium je mezinárodní společnost, která vyvíjí webové standardy pro World Wide Web.
[4] Mezinárodní standard pro kódování znaků.

```
<cenik>
<produkt kategorie="polevky">
<nazev>Bramborova polevka</nazev>
<cena mena="CZK">25</cena>
</produkt>
</cenik>
```

Výpis 1: Ukázka kořenového elementu XML dokumentu.

2. Jména tagů v XML rozlišují malá a velká písmena, tzn. jsou case sensitive. XML tagy ve výpisu 2 jsou od sebe různé a nelze je proto mezi sebou zaměňovat.

```
<food>
</food>

<Food>
</Food>
```

Výpis 2: Malá písmena a velká písmena v XML

3. Parametr XML tagu se vepisuje do počátečního tagu, následuje rovnítko a poté je hodnota parametru ohraničena uvozovkami viz. výpis 3.

```
<day format="dd.mm.yyyy">20.02.2000</day>
```

Výpis 3: Parametr XML tagu

Má pouze jeden kořenový (root) element. Vy výpisu 1 je v našem případě root element *<cenik>*.[9]

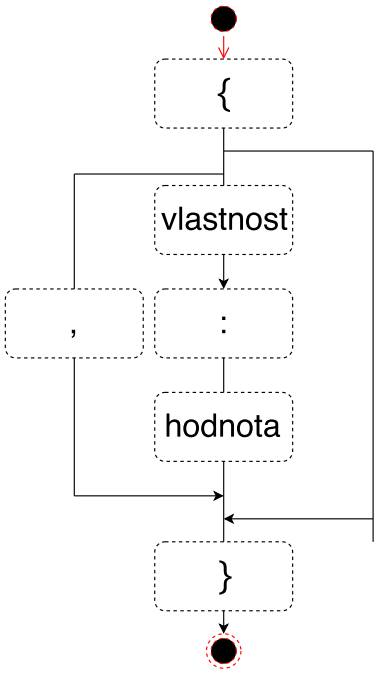
JavaScriptová objektová notace

Dalším druhem zápisu dat je formát JavaScript Object Notation (dále jen JSON). Byl navržen Douglasem Crockfordem a jeho specifikaci můžeme najít v RFC 4627[5]. V dnešní době patří mezi nejoblíbenější formáty na internetu. JSON vznikl v dobách, kdy se používal pro výměnu dat převážně formát XML. Tento způsob je opět nezávislý na platformě, která má za funkci ukládání a výměnu dat mezi různými systémy. I když tento formát obsahuje v názvu JavaScript, je tento textový způsob zápisu zcela obecný a lze použít v jakémkoliv programovacím nebo skriptovacím jazyce. JSON je jednoduše upravovatelný i čitelný pro člověka a je jednoduchý způsob pro uložení a analýzu dat.

[5] https://www.ietf.org/rfc/rfc4627.txt

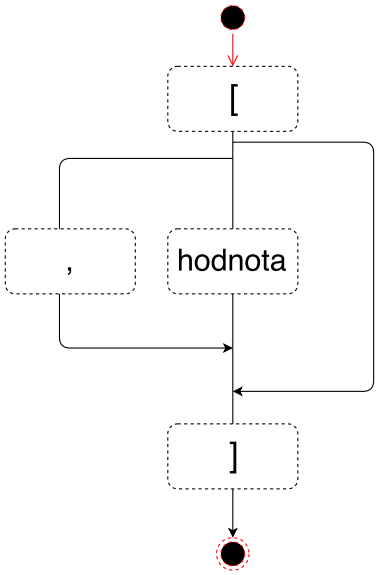
Hlavní charakteristikou JSONu je kolekce párů vlastnost a jeho hodnota. Základní datové struktury jsou realizovány následovně:

- **Objekt**
  Tento datový typ je v JSON formátu znázorněn způsobem neuspořádané množiny párů vlastnosti (property) a jeho hodnot (value). Objekt začíná levou složenou závorkou a ukončen je pravou složenou závorkou. Každá property je oddělena od své hodnoty dvojtečkou a jednotlivý pár property / value je oddělen čárkou. Na obrázku 3 je graficky znázorněna stavba syntaxe objektu v JSONu.



Obrázek 3: Zápis objektu v JSON formátu. Každý objekt začíná levou složenou závorkou. Poté následuje vlastnost oddělená od její hodnoty dvojtečkou. Každý další pár vlastnost-hodnota je poté oddělena čárkou. Výsledný objekt je poté uzavřen pravou složenou závorkou.

- **Pole**
  Pole je seřazenou kolekcí hodnot. Začíná znakem *[* (levá hranatá závorka) a končí znakem *]* (pravá hranatá závorka). Hodnoty jsou odděleny znakem, (čárka). Na obrázku 4 je graficky znázorněna stavba syntaxe pole v JSONu.



Obrázek 4: Zápis pole v JSON formátu. Každé JSON pole je ohraničené hranatými závorkami. Pole může obsahovat nula a víc hodnot, které jsou odděleny navzájem čárkou.

Výhoda JSONu oproti XML je ve velikosti přenášeného souboru. Jelikož je potřeba u XML otevírací a zároveň zavírací tag, s délkou a komplexnosti struktury dat se zvyšuje i velikost. Také knihovny pro parsování dat fungují rychleji s formátem JSON kvůli jednodušší syntaxi. Syntax JSONu totiž mnohem věrohodněji kopíruje objektový svět programovacích jazyků. Komplexnost XML přináší ale několik výhod. Mezi výhody patří schopnost lépe zpracovat například fotografie, grafy, zakódovanou hudbu, tedy zkrátka vše, co není primitivní datový typ.[10]

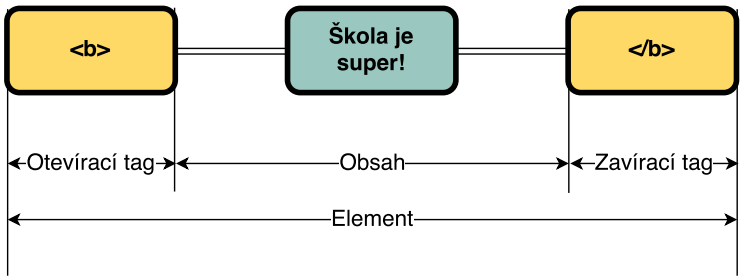
2.2 Webové technologie

Tyto aplikace jsou poskytovány uživatelům z webového serveru poskytovatele přes počítačovou síť internet nebo intranet. Tyto webové aplikace, jak název napovídá, fungují na jakémkoliv webovém prohlížeči, který je v dnešní době součástí takřka každého operačního systému. Tyto aplikace není nutné nějak instalovat na počítače uživatelů, aktualizovat či mazat. Tyto aplikace běží na standartním formátu HTML/XHTML, který je podporován na všech prohlížečích. Výhodou těchto aplikací je multiplatformnost a univerzálnost. Místo toho, aby byly psány programy zvlášť pro Windows, Linux nebo Mac OS X, je aplikace napsána v jednom provedení a je nabízena komukoliv, kdo přistoupí správně na webový server. Níže si popíšeme vybrané technologie pro tvorbu front–endových částí webové aplikace.

2.2.1 HTML – Hypertext Markup Language

HTML jazyk je standartní programovací jazyk pro tvorbu World Wide Web (WWW) stránek. Popisuje a definuje obsah webových stránek nebo aplikace. Dokument v jazyku HTML má přesně nadefinovanou strukturu a stejně jako u XML musí obsahovat kořenový element, hlavičku

elementu a tělo samotného dokumentu. Opět je zde výhoda multiplatformnosti a univerzálnosti (Vykreslování závisí na prohlížeči a ne na operačním systému). Společně s Cascading Style Sheets (CSS), který definuje vzhled a prezentaci, a JavaScriptem, který definuje funkcionalitu a chování, jsou často používány pro vývoj webových aplikací. HTML používá (stejně jako XML) tagy pro anotaci textu, obrázků a dalšího obsahu ve webovém prohlížeči. Tagy tohoto jazyka mohou buď být v páru např. <head> a </head>, přičemž první tag se nazývá otevírací tag a druhý tag se nazývá ukončovací, nebo se mohou vyskytovat jednotlivě (*<img>*). Zdrojové kódy HTML jsou uloženy v souboru s příponou *.html*. Tento soubor je pak načten webovým prohlížečem a přeloží (vyrendruje) kód do okna prohlížeče. Na obrázku 5 vidíme anatomii HTML elementu.



Obrázek 5: Grafické rozložení HTML elementu. Každý HTML tag je uvozen počátečním a koncovým tagem, přičemý koncový tag obsahuje lomítko před názvem elementu. Mezi HTML tagy je pak daný obsah.

Na výpisu 4 je znázorněn ukázkový program v HTML. Vše, co je obsaženo mezi tagem *<body>* a *</body>*, definuje obsah webové stránky. Zároveň vše, co je mezi tagem *<html>* a *</html>*, popisuje samotnou webovou stránku. Na příkladu můžeme vidět tag * * a * * vnořen v tagu *<body>*. Toto se nazývá vnoření tagů a vytváří se nám relace rodič-potomek, kde rodič je v našem případě element *body* a potomek je element *p*. Pro různé přizpůsobení tagů zde existují parametry (atributy) HTML tagu. Tento parametr se zapisuje stejně jako u XML (viz. výše) např. *<img src='image.jpg'>* nebo * *. Většina parametrů je nepovinná až na ty, které jsou povinné u některých tagů, například *img* má povinný atribut *src* a *alt* pro správné vykreslení obrázku na stránce.

```
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```

Výpis 4: Ukázka HTML dokumentu.

HTML vznikl v 90. letech minulého století a od té doby se v programovacím světě změnilo spoustu věcí. Nyní je používaná verze HTML 5, která vznikla v roce 2014. V dnešní době se všechny prvky, které jsou v interakci s uživatelem přesouvají do CSS jazyka. Přístup komunity programátorů se mění a cílem je vyvíjet lehce spravovatelné programy a oddělit od sebe funkcionalitou jednotlivé jazyky. Vizuální formatování s HTML tagy je již tedy zastaralé a nemělo by se používat. Představme si, že budeme muset změnit font celé webové stránky či aplikace. Pokud bychom postupovali s klasickým HTML, bylo by nutné přidat, změnit či odebrat hodnotu parametru na každém místě, kde se nachází font. S CSS jazykem toho jde dosáhnout jednoduše jednou změnou.[11][12]

2.2.2 CSS

CSS je jazyk, který, jak již bylo nastíněno výše, popisuje vzhled HTML, XHTML nebo XML a je nadstavbou těchto dokumentů. V dnešní době by se už mělo stylovat zásadně pomocí tohoto jazyka a ne v HTML. Hlavní úlohou CSS je tedy oddělení vzhledu od logického obsahu. S CSS styly lze přiřadit každému tagu v HTML určité vlastnosti (umístění, způsob zobrazení, vzhled atd.), a to dokonce i v závislosti na jakém rozlišení je HTML dokument vykreslován (obrazovka PC, tiskárna, mobilní telefon, zvukový výstup atd.). V CSS lze definovat i pravidla stylů. Styl definuje pravidla, jak prvek různě interpretovat. Takových pravidel může být spoustu. Nejvíce používané jsou např. barva, font, pozice, aj. Kaskádovost stylů je dána hierarchickým uspořádáním pravidel pro styly.

Způsob vytváření stylů je definován seznamem položek. Jednotlivou položkou může být pravidlo ve tvaru *selektor definice*. Selektor určuje, na který prvek se daný styl (pravidla) aplikuje. Definice pravidla, které popisuje konkrétní vlastnosti zvoleného selektoru, má tvar *vlastnost:hodnota vlastnosti*. Každé pravidlo je odděleno středníkem.[11][12]

Příklad definice stylu:

```
.registratePatientBorder {
    border-width: 0px 1px 0px 0px;
    border-style: solid;
    border-color: black;
    padding: 15px 15px 15px 15px;
}
```

Jelikož lze vkládat styly několika způsoby, tak se určování toho, který styl bude mít nejvyšší prioritu a použije se pro zobrazení elementu, řídí následujícími pravidly (číslování 1 je s nejvyšší prioritou a číslování 3 je s prioritou nejnižší):

1. Inline styly v HTML tagu
2. Externí a interní styly
3. Styly definované v prohlížeči jako základní

Implementace CSS do HTML

Jak již bylo řečeno výše, existují tři způsoby, jak vložit styly do HTML dokumentu. Tak, že je definován přímo v HTML tagu nebo pomocí externího souboru. Tyto metody lze libovolně kombinovat mezi sebou podle účelu.

Importování externího souboru lze docílit způsobem, že mezi HTML tagy *<head>* a *</head>* vepíšeme tag *<link>* či *<style>* přes který se importuje námi zvolený externí css soubor.

Příklad: <link rel="stylesheet" type="text/css" href="app.css"/>

Inline styl lze implementovat pomocí tagu <style> přímo uvnitř dokumentu. Tohoto způsobu využijeme pokud máme jednotný styl a vzhled celé webové aplikace a chceme u jedné části použít jiné pravidlo.

Příklad: Tento obor je super!

Importování interního vloženého stylu se docílí způsobem, že je daný selektor ohraničen *<style>* tagem.

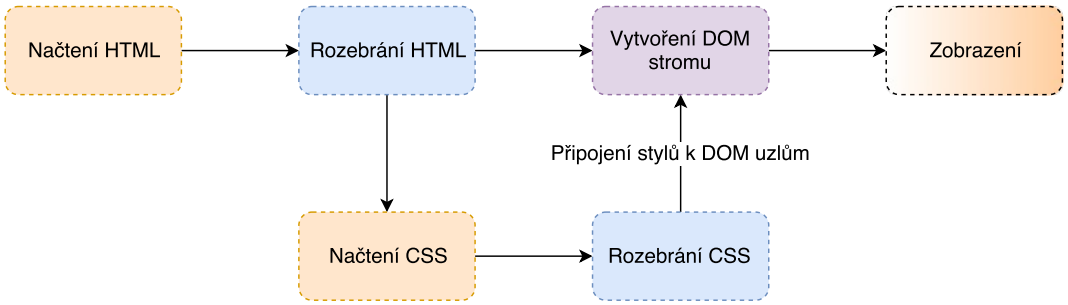
Příklad:

```
<style>
  p {
    color: green;
  }
</style>
```

Když webový prohlížeč zobrazuje dokument, musí v první řadě skombinovat obsah samotného dokumentu a jeho styly. Tento krok probíhá ve dvou fázích:

1. Konverze HTML a CSS do Document Object Model (dále jen DOM).
2. DOM je pak uložen v paměti operačního systému.
3. Prohlížeč následně zobrazí obsah DOMu.

Na obrázku 6 je znázorněno načtení HTML dokumentu s CSS styly.

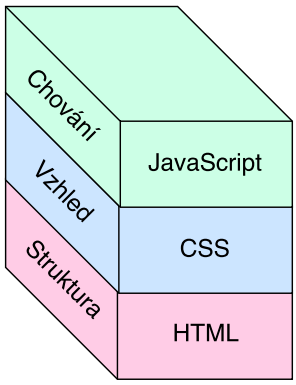


Obrázek 6: Načtení HTML a CSS prohlížečem. Po načtení HTML je prohlížečem tento dokument rozebrán přičemž zjistí lokaci CSS souborů. Styly jsou poté rozebrány a společně s HTML je vytvořen DOM, který je pak uložen v paměti operačního systému. Prohlížeč poté zobrazí obsah DOMu.

2.2.3 JavaScript

Javascript je multiplatformní, skriptovací a objektově orientovaný jazyk. Tento jazyk stejně jako HTML (kapitola 2.2.1) a CSS (kapitola 2.2.2) bývá implementovaný na straně klienta (ačkoliv open sourcový projekt Node.Js je JavaScriptový framework pro práci na straně serveru) např. webový prohlížeč. Byl vyvinut v roce 1995 Brendanem Eichem[6] ze společnosti Netscape, který byl jako poprvé nasazen ve stejnojmenném prohlížeči. Tento skriptovací jazyk pak byl následně implementován i v jiných prohlížečích konkurenčních firem. v roce 1997 byl jazyk standardizován a byla vytvořena specifikace ECMA-262, která byla podporována ve všech známých prohlížečích. Od té doby vzniklo 8 verzí ECMA scriptu (k roku 2017) díky kterému je JavaScript jeden z nejpoužívanějších jazyků v současnosti. Jelikož jde o interpretovaný (skriptovací) jazyk tak má oproti kompilovanému mnoho nevýhod a byl spousta vývojáři a komunitami odsouzený jako jen jednoduchý nástroj pro oživení stránek. Tento předsudek je ale snad v této době již překonán. Úspěch jazyka je zejména v jeho jednoduchosti použití. Na obrázku 7 je znázorněna struktura webové stránky, která implementuje všechny tři technologie. JavaScript tedy umožňuje vytvářet dynamické změny obsahu, kontrolování multimédií, animování obrázků a další akce, které umožňují interagovat s uživatelem.

[6]Programátor původem z USA, který vytvořil skriptovací jazyk Mocha. Poté byl přejmenován na LiveScript a ve stejný měsíc byl finálně změněn na JavaScript.



Obrázek 7: Struktura webové aplikace. Na tomto obrázku je znázorněna stavba a vrstva jednotlivých technologií. Nejspodnější vrstva HTML zabezpečuje strukturu a základní kostru webové stránky či aplikace. Následuje soubor stylu, který se stará o vzhled dané stránky. Nejvýše je položena vrstva s JavaScriptovým kódem, který má na starosti chování aplikace. Všechny tyto tři vrstvy dohromady dávají stavbu moderní webové stránky.

Syntaxe jazyka se podobá jazyku C, C# a Java a má rysy imperativního, strukturovaného, dynamického, funkcionálního a v neposlední řadě, jak již bylo řečeno, je to objektově orientovaný jazyk.

Imperativní a strukturovaný jazyk

Imperativní a strukturované programování je jedno ze základních programovacích paradigmat. Imperativní neboli procedurální programování popisuje úkon nebo výpočet pomocí posloupnosti příkazů a určuje přesný postup (algoritmus), jak danou úlohu řešit. v reálném světě je tohle například kuchařský recept či návod k sestavení nábytku z IKEY. Strukturované programování je další paradigma, které je založené na cyklech a větvení programu. Problém se rozloží na několik podproblémů a každý podproblém potom řeší určitá funkce s parametry.

Dynamický jazyk

Jazyky, které jsou dynamické fungují tak, že za běhu provádí většinu činnosti, kterou by kompilované jazyky již dělaly během doby kompilace. Datové typy jsou dynamicky určovány za běhu programu. Lze taky měnit objekty i celého objektového modelu za běhu a funkce eval umožňuje vykonávat nový kód v JavaScriptu předaný této funkci jako řetězec.

Funkcionální jazyk

Funkcionální paradigma je založeno na zápisu programu ve tvaru výrazu. Funkce lze předávat jako argument, vracet jako výsledek a přiřazovat do proměnných. Dalším znakem funkcionálního programování u JavaScriptu je zanořování funkcí, které umožňují definovat funkce v rámci jiné funkce. Dále uzávěry (closures) umožňují při invokaci zanořené funkce přístup k proměnných ze svého oboru platnosti (scope) i když k zavolání dojde úplně v mimo obor platnost (out of scope).

Objektově orientované programování – OOP

Toto paradima vychází od imperativního (procedurálního) programování. Tento model se skládá spíše z objektů než z akcí a dat než z logiky. Kvůli zapouzdření a abstrakci jsou metody (funkce) zapouzdřeny v objektech. Nutno podotknout ale, že v JavaScriptu se objekty nedefinují pomocí tříd (objektů), ale pomocí funkcí. Dědičnost se u tohoto jazyka realizuje trošku odlišným způsobem než v klasických kompilovaných jazycích. Dědičnost se obvykle realizuje pomocí vlastnosti *prototype* objektu, ve kterém je odkaz na objekt, pomocí kterého byl objekt vytvořen. Ukázka OOP je uvedena v příkladu 5.[13][14]

```
this.pozdrav = null;

function Zvire() {
  this.ozviSe = function() {
    console.log(this.pozdrav);
  }
}

var zvire = new Zvire();

function Pes() {
  this.pozdrav = "Haf Haf"
}

// nastavime do vlastnosti prototype instanci parent objektu se kterym podedi i~jeho
    metody
Pes.prototype = new Zvire();

// vytvorime instanci child tridy
var benny = new Pes();

benny.ozviSe(); // v konzoli bude "Haf Haf"
```

Výpis 5: Ukázka OOP v JavaScriptu

Při načítání webové stránky do prohlížeče je tedy použit kód (HTML, CSS a JavaScript) pro mechanismus vykreslení. Poté co je HTML a CSS sestaven a zobrazen je JavaScript spuštěn překladačem v prohlížeči. Tím je zajištěno, že struktura a styl stránky jsou již zavedeny, než se spustí JavaScript. Toto je chování, které je zamýšleno schválně – totiž velmi častým používáním jazyka JavaScript jsou dynamické úpravy kódu DOMu (HTML a CSS) pro aktualizaci uživatelského rozhraní prostřednictvím rozhraní API[7] Document Object Model (jak je uvedeno výše). Pokud by JavaScript byl načten dříve a pokusil se spustit, došlo by k chybám.

[7]Rozhraní (procedury, funkce, třídy apod.) pro programátory, které může využít.

Je důležité se nějakým způsobem chránit před potenciálním škodlivým kódem. Každé okno prohlížeče má svoje oddělené funkční prostředí (Execution environments) což znamená, že JavaScriptí kód, který běží v jednom okně nemůže ovlivňovat vedlejší stránku či jeho kód.

Implementace JavaScriptu do HTML

1. Inline JavaScript kód (nedoporučuje se)
2. Interní JavaScript kód
3. Externí Javascriptí kód

Inline JavaScript kód

Někdy se můžeme setkat s tímto výskytem JavaScriptu, který se nachází přímo v HTML tagu. Na příkladu 6 lze vidět, že HTML tag *<button>* má parametr *onClick*. Tento handler se spouští když je tlačítko spouštěno a volá JavaScript funkci nadefinovanou výše. Ačkoliv je tenhle kód plně funkční, neměl by se používat. Tenhle přístup nerespektuje oddělení logických vrstev viz. obrázek 7. v tomto příkladu se spojuje vrstva strukturální a behaviorální. Nedoporučuje se zanášet javascript snippety do HTML a také je tenhle způsob značně neefektivní. Pro správné chování tlačítka bychom museli definovat parametr *onClick* u každého tlačítka, který by byl v HTML dokumentu.

```
<button onClick="handleButtonClick()"></button>
```

Výpis 6: Ukázka inline JS kódu

Interní JavaScript kód

Dalším způsobem jakým můžeme použít JavaScript v HTML dokumentu je vepsáním JavaScriptí části mezi *<script>* a *</script>* HTML tagy viz. příklad 7.

```
<script>
   function handleButtonClick() {
      var p = document.createElement('p');
      p.textContent = 'Klikl jste na tlacitko!';
      document.body.appendChild(p);
   }

   var buttons = document.querySelectorAll('button');

   for (var i = 0; i < buttons.length ; i++) {
      buttons[i].addEventListener('click', handleButtonClick);
   }
</script>
```

Výpis 7: Ukázka interního JS kódu

Externí JavaScript kód

Posledním a doporučeným způsobem, jak importovat JS kód je následující příklad 8. Jako v případě interního JS kódu používáme tag *<script>* jen s tím rozdílem, že použijeme atribut *src* kde naimportujeme zvolený JS soubor s naším kódem. Tímhle máme přehledně oddělené logické části webové aplikace.[15]

```
<head>
   <script src="script.js"></script>
</head>
```

Výpis 8: Ukázka externího JS kódu

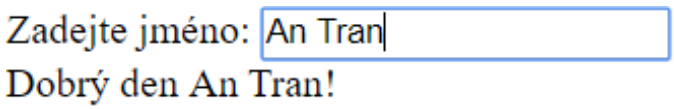
2.2.4 AngularJS

Nárůst obliby této technologie a javascriptu obecně přišel s nástupem HTML5. Tehdy už přestávala knihovna jQuery stačit z jednoho fundamentálního důvodu. Spojoval dohromady manipulaci s DOMem, zpracování události a aplikační logiku. Jelikož se s přibývajícím výpočetním výkonem a optimalizací na straně klienta se více a více logiky přesouvaly směrem ke klientovi bylo následkem, že se kód rozrůstal a nabýval na komplexnosti. Je nutné si uvědomit, že tehdy neexistoval klientský jazyk, který by ctil architekturu MVC (viz. kapitola 2.2.5), a proto se v roce 2009 poprvé objevil JavaScript framework AngularJS. Autorem je společnost Google a je jeden z klientských javascriptích nástrojů, který je meziplatformním řešením pro vytváření aplikací v klientských částí webové technologie. Je nutné se uvědomit, že před AngularJS nebyl způsob, jak vytvořit dynamické aplikace, kde se informace ve View neustále mění. AngularJS tedy rozšiřuje dokument HTML o spousty atributů a elementů, které v základu chybí. s postupem času se tento framework dynamicky měnil, zlepšoval a narůstal na oblibě. Mezi jeho přednosti je nutno

zmínit Two Way Data Binding, Dependency Injection, Testovatelnost, Directives. Jak již bývá zvykem, žádný framework není dokonalý a i tento má svoje mouchy. Paradoxně některé jeho přednosti jsou zároveň jeho největšími slabinami.

Dvoucestná synchronizace dat – Two Way Data Binding

Tento koncept řeší synchronizaci stavů mezi modelem a view. Na obrázku 8 a 9 a příkladu 9 je znázorněna jednoduchá aplikace, kde máme textové pole a pod ním vypisujeme obsah pole s přidaným textem. Ve chvíli kdy proběhne změna v textovém poli, se text přenese do modelu a poté se zapíše do části view. Tato automatizace je obsažena ve fundamentálních základech AngularJS a není proto potřeba se o nic starat.

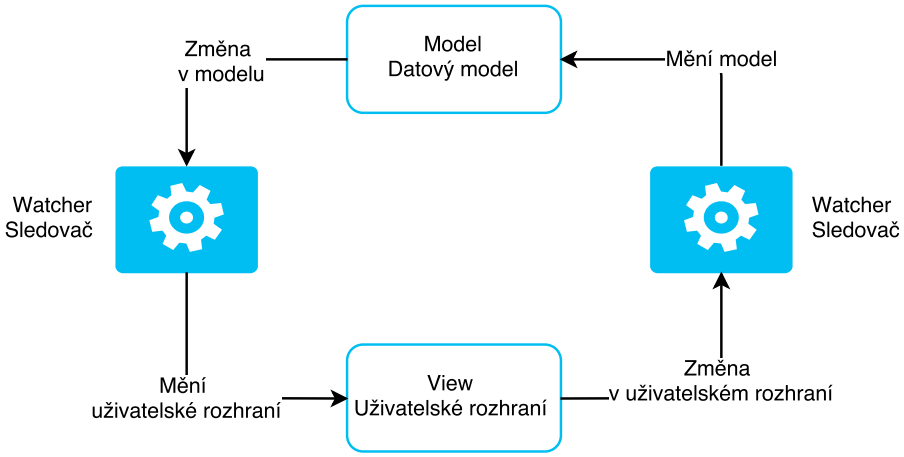


Obrázek 8: Ukázka dvoucestné synchronizace dat.

```
<div ng-app="MyApp" ng-controller="MyCtrl">
  <p> Zadejte jmeno: <input type="text" ng-model="name"><br>
  Dobry den <span ng-bind="name"></span>!</p>
</div>
```

Výpis 9: Zdrojový kód ukázky obrázku 9

Ve výpisu 8 vidíme v HTML tagu *input* přidanou AngularJS direktivu *ng-model*, který přesouvá do jeho modelu (v našem případě *name*) to co jsme napsali do textového pole. Další řádek obsahuje HTML tag *span* s direktivou *ng-bind*, který má za úkol vypsat všechna data, který obsahuje model *name*. Tímto jsme zajistili deklarativní způsob implementace což znamená funkční automatizovanou dvoucestnou synchronizaci dat.



Obrázek 9: Grafické znázornění dvoucestné synchronizace dat. Zde je graficky znázorněn, jak AngularJS zajišťuje modifikaci datového modelu a uživatelského rozhraní. Ve chvíli kdy proběhne změna v textovém poli, se text přenese do modelu a poté se zapíše do části view.

Další výhodou tohoto přístupu je že můžeme model vypsat na více místech view. Změnou dat v modelu se synchronizuje view na všech místech kde je namapovaný daný model. Tím odpadá povinnost sledovat, kde všude je daný výpis a synchronizace je spouštěna automaticky.

Jak již bylo řečeno, největší výhody AngularJS jsou zároveň i jeho nevýhody a jeden z důvodů proč byl nahrazen verzi 2.0, který samotnou AngularJS architekturu kompletně obměňuje. Největším problémem je náročnost tohoto přístupu. Na každou direktivu *ng-bind* jsou totiž zaregistrovány sledovače (watcher) a tedy s přibývající velikostí aplikace narůstá i počet watcherů, který sleduje a naslouchá každé změně. Tím narůstá náročnost aplikace a klesá její výkon. Dalším důvodem je, že ztrácíme plnou kontrolu nad elementy kdy se mají obnovovat a také u větších aplikací ztrácíme přehled odkud byla synchronizace spuštěna pokud potřebujeme zpětně zjišťovat příčinu chyby. Na obrázku 9 je znázorněno schéma, jak probíhá dvoucestná synchronizace dat.

Vkládání závislosti – Dependency Injection

Dependency Injection (dále DI) je programátorský přístup architektury, který řeší závislost mezi danými komponentami v aplikaci. Vkládá závislosti mezi komponentami (třídami) bez nutnosti referenci na sebe v době chodu programu. AngularJS framework obsahuje zabudovaný systém pro DI, který řeší tyto závislosti v celé aplikaci. v aplikaci proto pouze specifikujeme, které komponenty jsou používané jinými komponentami. AngularJS dokáže říct, která konkrétní

komponenta potřebuje tu danou vyhledáním specifických parametrů konstruktoru[8]. V ukázce 10 je deklarace kontroleru, kterému se při vytvoření předávají tři parametry, přes které jsou předávány závislosti. Jakmile systém vytváří novou instanci kontroleru, zkontroluje si jeho závislosti v deklaraci a všechny předá jako parametry. V kontroleru se poté pracuje s závislostmi, které se nazývají služby (services). Interní služby se prefixují znakem dolaru ($). Ve ukázce 10 používáme tři služby: *scope*, *routeParams* a *location*.

```
function ShowInfoPageCtrl($scope, $routeParams, $location) {
  //obsah controlleru
}
```

Výpis 10: Ukázka vkládání závislosti

Jelikož vkládání závislosti je zajištěno přes jméno parametru (AngularJS volá interně metodu *toString()*[9], čímž získá jména argumentů, které poté prohledává v celém listu existujících závislostí), dochází k problému kdy je zdrojový kód minifikován[10]. Poté co proběhne minifikace, vkládání závislosti přestává fungovat, protože již nenajde zminifikovaný parametr v seznamu závislostí. Programátoři AngularuJS přišli s dvěma řešeními pomocí kterých lze tenhle problém obejít. Oba způsoby jsou naznačený v ukázkách 11 a 12.

```
module.controller('ShowInfoPageCtrl', ['$scope', '$routeParams', '$location',
    ShowInfoPageCtrl]);
```

Výpis 11: Vložení závislostí v poli

```
ShowInfoPageCtrl.$inject = ['$scope', '$routeParams', '$location'];
```

Výpis 12: Vložení závislosti do vlasnosti *$inject*

Tento framework registruje všechny závislosti v globální hash-mapě což znamená následující nepříjemnost. Pokud je zaregistrována závislost pod stejným jménem dvakrát, novější registrace bez notifikace přepíše tu první. Tento nedostatek vede k nespočtu chybám v runtime aplikace a ke ztracenému času než je chyba nalezena.

Je nutné podotknout, že AngularJS jako framework byl popisován pro verzi 1.X. Vyšší verze (2.0 a výš) již tyto problémy překonaly ale došlo k fundamentálním změnám úplných základů architektury. Vyšší verze je tudíž nekompatibilní s verzi starší.[16, 17]

[8] Metoda, která se zavolá vždy když je vytvářena nová instance komponenty (třídy).
[9] Překonvertování daného zdroje na typ string.
[10] Proces odstranění všech nepotřebných znaků ze zdrojového kódu bez změny jeho funkčnosti.

2.2.5 ReactJS

ReactJS (nebo také React nebo React.js) je opensourcový[11] projekt, který zveřejnil v roce 2013 Facebook po interním užívání a odlazování. React přináší zásadní změnu v paradigmatu, který spojuje dohromady programování a HTML[12]. v této práci byl právě použit tento framework jako front-endová část aplikace, proto bude popsána detailněji s dalšími technologiemi, které se k Reactu pojí.

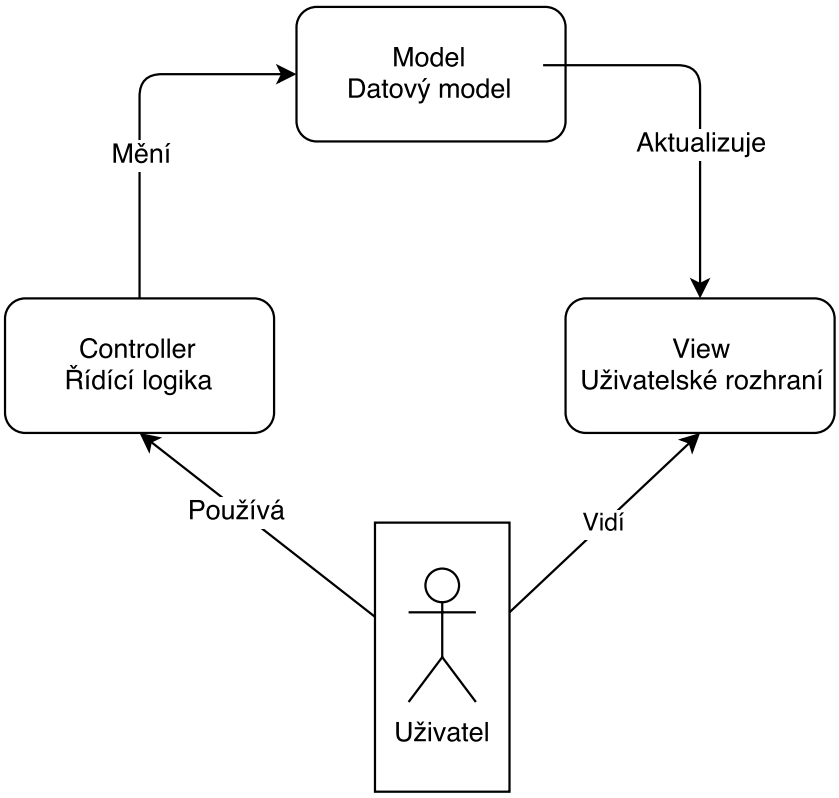
Jak již bylo řečeno, největší výhodou Reactu je, že se tvůrci soustředili na hlavní účel Javascriptích frameworků a to je View v MVC architektuře. Tvůrci se poučili s chyb předešlých frameworků a cílem bylo vyvinout prostředí, které je lehce udržovatelné, testovatelné a škálovatelné. React tedy dokáže upravovat DOM pouze deklarativním definováním HTML struktury pomocí JavaScriptích funkcí. Uvedeme tedy, jak má vypadat výsledný View na základě přichozích dat. React framework si poté složí svůj virtuální DOM, který pak optimalizovaně porovnává s vykresleným DOMem a vykresluje pouze rozdíly mezi virtuálním a skutečným DOMem. Tímto výrazně optimalizujeme rychlost klientské části aplikace.

Model View Controller

Pro kompletní obrázek o Reactu je potřeba popsat, co to vlastně MVC architektura je.

[11] Projekt, který lidé mohou modifikovat a sdílet, protože jejich zdrojové kódy js veřejně přístupné.

[12] Tato zásadní změna přinesla zezačátku Facebooku ohromnou vlnu kritiky, kdy přemýšlel nad stáhnutím tohoto frameworku.



Obrázek 10: MVC architektura.

Model-View-Controller (dále jen MVC) je jeden z nejpoužívanějších a nejobecnějších architektonických vzorů, který se uchytil zejména na webu, ačkoli původně vznikl na desktopových aplikacích. MVC můžeme rozdělit do tří základních komponent viz. obrázek 10.

- Model – data a související operace (tzv. business logika). Model neví, odkud data přišla a ani, jak budou výstupní data zformátována a vypsána.
- View – prezentace dat (např. uživatelské prostředí) obsahuje přímý odkaz na model, aby mohl jeho data prezentovat vnějšímu světu. View podobně jako Model vůbec neví, odkud mu data přišla, stará se jen o jejich zobrazení uživateli.
- Controller – řadič, který řídí tok událostí v programu (tzv. aplikační logika) obsahuje přímý odkaz na model, aby mohl modifikovat jeho data.

Motivací MVC je oddělení vrstev logiky, dat a výstupu. Řeší tedy staré programovací neduhy minulého století kdy se v jednom souboru či třídě střetávaly části, které řeší logické operace (mapování, databázové dotazy, volání cizích systémů či knihoven) a zároveň vykreslování výstupu. Čili je cílem oddělit tři logické části tak, aby je šlo upravovat samostatně a dopad změn byl na ostatní části co nejmenší.

Popišme si tedy MVC architekturu na něčem co všichni dobře známe a to stránky s pornografickým obsahem.

View je v našem případě to co jako uživatel vidíme a to jest série videí, které si můžeme prohlídnout, reklamy na zázračné prodloužení části těla či náruživé dámy z okolí.
Model jsou data, které jsou uchovávány za vrstvou View. v našem případě to jsou obrázky, videa, text a informace o herečkách atd.
Kontroler je nejhůře představitelnou částí na uvedeném příkladu. Projeví se ve chvíli, kdy uživatel interaguje s dynamickými komponentami ve View. Příkladem je kliknutí na tlačítko přehrát video. Po kliknutí je odstartována kaskáda operací, která má v tu chvíli na starost, aby se aktualizoval model, došlo k logickým operacím (kontaktování serveru pro data) a v poslední řadě překreslení (render) View. Controller je tedy jakousi ústřední výkonnou jednotkou, která se stará o celkové provázání funkčnosti aplikace.

Co je tedy ReactJS v MVC architektuře? React ve své základní podstatě je pouze View – uživatelské prostředí. Naším úkolem je tedy pouze dodat nová data do Reactích komponentů a o zbytek se postará vnitřní algoritmy frameworku.

Komponenty

Komponenty (components) jsou základním stavebním kamenem celého ReactJS. Tyto komponenty, které dokážou fungovat samostatně a jsou navzájem nezávislé, můžeme libovolně kombinovat mezi sebou a tvořit tak složitější velkopodnikové (enterprise) aplikace. Tato filozofie, která razí stavět složité komponenty z menších a znovupoužitelných komponent získala velikou oblibu a programatorskou základnu tohoto JS frameworku. Všechny komponenty jsou tvořeny popisem, jak by měl View vypadat. Popis je konstruován pomocí JSX (viz. níže) jazyka. Na příkladu 13 je popsána jednoduchá komponenta, která vykresluje element div obsahující obyčejný textový řetězec.

```
export default class Login extends React.Component {
  render() {
    return(
      <div>Ahoj svete!</div>
    );
  }
}
```

Výpis 13: Příklad React komponenty

Komponenta má pouze jednu povinnou funkci a to *render()*. v ní je popsáno, jak má daná komponenta ve výsledku vypadat v závislosti na tom jaké data jí budou popsána. Každá metoda *return()* musí vracet jeden kořenový (root) element (nejčastěji se používá element). Tento kořenový element již může obsahovat nespočet dalších elementů.

Java Serialization to XML (dále jen JSX)

V předchozí kapitole Komponenty bylo zmíněno JSX. Zápis JSX je velice podobný HTML (XML) a jde pouze o zjednodušení zápisu, který je poté čitelnější a srozumitelnější. Tento zápis není povinný, nicméně silně se doporučuje při práci s Reactem.

na

```
// JSX kod
<CustomJSX irony="true" color="blue">
  Biomedicina je super!
</CustomJSX>

// se kompiluje na
React.createElement(
  CustomJSX,
  {color: 'blue', irony: 'true'},
  'Biomedicina je super!'
)
```

Výpis 14: Ukázka JSX a jeho transkompilaci.

Jak vidíme z příkladu 14, JSX je pouze syntax sugar[13], který se poté kompiluje na funkcí Reactu. První část JSX určuje jméno Reactí komponenty. Velké písmeno názvu označuje, že značka JSX odkazuje na komponentu Reactu. Tyto značky se kompilují do přímého odkazu na danou proměnnou, takže pokud používáte JSX *<CustomJSX />*, *<CustomJSX />* musí být v rozsahu (scope).

Document Object Model

Document Object Model (dále jen DOM) je způsob, jak reprezentovat strukturované dokumenty skrze objekty. Jedná se o objektově orientovanou reprezentaci HTML nebo XML dokumentu. Také to je API, které umožňuje přístup či modifikaci obsahu. Pokud vztáhneme DOM k HTML tak HTML je text a DOM je jeho abstrakce v paměti prohlížeče. DOM je multiplatformní a umožňuje prezentaci a interakci s daty například v HTML nezávislou na jazyce. Proto operace, které lze provádět nad DOMem jako hledání uzlů, změna hodnot, odebrání uzlu apod. můžeme docílit pomocí JavaScriptu nebo CSS jelikož samotnou režií operací už má na starosti prohlížeč. Motivací Virtual DOMu (viz. níže) pro vývojáře Reactu je, že DOM nebyl nikdy navržen pro dynamické vykreslování UI[14].

Virtual DOM

Myšlenka Virtuálního DOMu kterou implementovali vývojáři a komunita do React frameworku je jedna ze zásadních novinek, která značně optimalizuje celou klientskou část aplikace.

[13] Nahrazení zápisu za jiný o stejné funkcionalitě za účelem zjednodušení zápisu či zlepšení čitelnosti.

[14] UI nebo-li User Interface je na první pohled viditelný objekt uživatelem. UI je zjednodušeně řečeno pouze vzhled aplikace.

Místo přímé interakce s DOMem, je vytvořena jeho abstraktní (virtuální) verze. Dále pracujeme pouze s touto odlehčenou abstraktní verzí. Můžeme jej libovolně měnit, přidávat, odebírat atd. a všechny změny pak najednou uložit do reálného DOMu. Další optimalizací je, že při ukládání do reálného DOMu, ukládáme pouze ty části, které se změnily. Poté pouze tyto změny překreslíme (re-render) a docílíme toho, že není změněna celá stránka ale pouze ty části, které potřebujeme.

Koncept, kterého zanechal tento framework je při každé změně dat přerendrovat celou stránku (nainicializovat všechny komponenty znova, zavěsit listenery atd.). Jelikož jsou komponenty Reactu předpisem, jak by měla ona komponenta vypadat, docházelo k problikavání View v průběhu překreslování. Nehledě na to, že překreslovat celou stránku kvůli změně jednoho textového políčka je značně náročné, trpí i samotný uživatelský komfort při problikávání stránky.

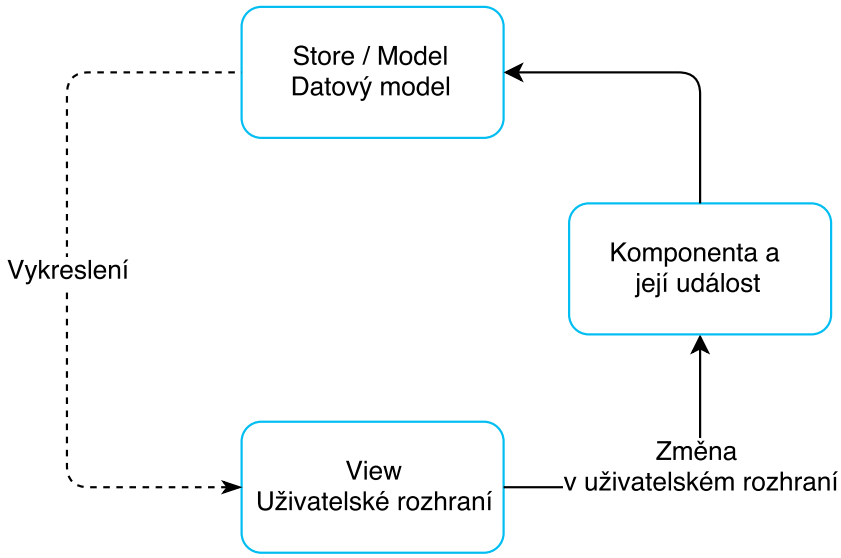
Existují dva způsoby, jak zjistit změnu dat:

- **Náročná kontrola (Dirty checking)** – pravidelně zjišťujte data a rekurzivně kontrolujte všechny hodnoty v datové struktuře.
- **Pozorování (Observable)** – sledování změny stavu. Pokud se nic nezměnilo, neděláme nic. Pokud se změní, víme přesně, co aktualizovat.

Jednosměrný tok dat – One way data flow

Narozdíl od AngularJS (viz. kapitola 2.2.4 a jeho dvoucestné synchronizaci dat (obrázek kapitola 9) se vydal React jiným směrem. Namísto pevného provázání modelu (Model) a uživatelského prostředí (View), změna v uživatelském prostředí neovlivňuje přímo store[15]. Často se model v Reactu označuje také jako single source of truth – jediný zdroj pravdy. To znamená, že pouze store dokáže měnit stav aplikace. Tímto docílíme snadného hledání chyb jelikož se vykreslování volá pouze tehdy pokud je změněn store. Na obrázku 11 je znázorněna graficky architektura toku dat.[18]

[15] V ReactJS se označuje model jako store.



Obrázek 11: Jednosměrný tok dat.

2.3 Technologie serverových částí webové aplikace

V této části diplomové práce budou popsány technologie serverových částí webové aplikace. Serverová část (backend) slouží nepřímo pro podporu front-endových služeb, obvykle tím, že je blíže k požadovanému zdroji dat (databáze) nebo má schopnost komunikovat s požadovanou databází. Zjednodušeně řečeno, jedná se o část aplikace, která se odehrává mimo klientské stanice. Jakmile data opustí klientské prostředí a doputuje na přístroj jiný, jedná se již o backend. Cílem serverové části je například uchovávat data, provádět nad nima libovolnou logiku, vracet požadována data na základě požadavku, výpočty apod. v následujících části si popíšeme technologii pro vývoj serverových částí aplikací, která byla použita v praktické části diplomové práce.

2.3.1 .NET Web API

.NET Web API[16] je framework od společnosti Microsoft pomocí kterého sestavujeme HTTP servisy, které dokážou obsluhovat příchozí požadavky jednoho či více klientů. .NET Web API je sada komponent, které zjednodušují programování HTTP. Výsledkem je, že webové rozhraní API je flexibilní a snadno se rozšiřuje. Jazykem pro vývoji na tomto serverovém frameworku je C#[17].

Domain Driven Design

Pojem Domain Driven Design (dále jen DDD) byl poprvé zaveden v roce 2004 Ericem Evansem v jeho stejnojmenné knize. Doména v tomto případě znamenám, že je to soubor společných požadavků, terminologie a funkce pro libovolný program, konstruovaný k vyřešení problému

[16] Application programming interface – rozhraní pro programování aplikací. Shrňuje sbírku procedur, funkcí, tříd či protokolů nějaké knihovny (ale třeba i jiného programu nebo jádra operačního systému), které lze využívat.
[17] Objektově založený jazyk od společnosti Microsoft.

v oblasti. Základem je, že návrh softwaru je silně ovlivněno oborem, pro který má být software vytvořen. DDD vyzdvihuje nutnost modelování objektů tak, aby tento model co možná nejvíce odpovídal reálné situaci. Snaží se tedy o to, aby program byl modelem reálného systému nebo procesu. Při implementaci podle DDD úzce spolupracujeme s odborníkem na doménu, který může vysvětlit, jak skutečný systém funguje. Pokud vyvíjíme informační systém v nemocnici, může být doménovým expertem vrchní sestra. Mezi programátorem a doménovým expertem se buduje všudypřítomný jazyk (ubiquitous language), což je jazyk, který popisuje koncept systému. Myšlenka je, že napsáno by mělo být to, co systém dělá způsobem, který odborník na doménu dokáže přečíst a ověřit, že je správný. v našem příkladu informačního systému bude všudypřítomný jazyk zahrnovat definici slov jako "pacient", "diagnóza", "léky"a tak dále. Koncepty popsané v UL (ubiquitous language) budou tvořit základ objektově orientovaného návrhu DDD poskytuje jasné pokyny o tom, jak by měly objekty interagovat. Tento koncept architektury definuje na vysoké úrovni abstraktnosti jednotlivé doménové modely:[19]

- **Entita** — jedná se o objekt s jednoznačným identifikátorem. v našem příkladě se jedná o pacienta a jeho identifikátorem je například rodné číslo. Tímto dokážeme rozlišit dvě entity od sebe. Tento základní stavební blok je datový objekt a jedná se o základní jednotkou informace. Pokud chceme v aplikační vrstvě pracovat s jakýmikoliv daty, která potom chceme ukládat do databáze, měníme instanci entity, nikdy neměníme data přímo. Entita definuje rozhranní, jaká data je možné uložit, změnit, získat.

- **Hodnotový objekt (Value object)** — reprezentuje informaci, která nemá identitu. Zajímá nás pouze jeho hodnota a ne konkrétně daná identita. Příkladem může být například již absorbovaná dávka rentgenového záření. Nezajímá nás konkrétní vyšetření ale pouze hodnota záření, kterou absorboval pacient.

- **Agregát (Aggregate)** – klastr objektů a hodnotových objektů s definovanými hranicemi kolem skupiny. Agregát zamezuje aby každý jednotlivý objekt (entita nebo hodnotový objekt) reagoval samostatně na externí podnět. Skupině objektů je přiřazen jednotlivá agregovaná kořenová položka. Nyní externí objekty již nemají přímý přístup ke každé jednotlivé entitě nebo hodnotovému objektu v agregátu, ale místo toho mají přístup pouze ke kořenovému prvku s jedním agregátem a používají jej k předávání pokynů skupině jako celku.

DDD také popisuje základní mechanismy:

- **Repozitář (Repository)** – při práci s entitami se očekává, že po restartu systému bude možné opět pracovat s daty, které entita obsahovala. Je potřeba tedy mapovat data na entitu. Tento proces se v relačních databází nazývá ORM (Object relational mapping). v DDD se o mapování dat z datového úložiště na objekty stará tedy repozitář. Ať již vytváří objekty z jakéhokoliv datového zdroje, vždy je prací repozitáře to, aby data převedlo

z konkrétního formátu na objekt se standardním rozhranním pro jejich zpracování – tedy na entitu. Repozitář je zodpovědný i za ukládání dat – tedy převod entity na data ve formátu datového zdroje a uložení těchto dat.

- **Služba (Service)** – při analýze domény kdy definujeme hlavní doménové objekty může nastat chvíle kdy některé části domény nejsou jednoduše mapovatelné na objekty. Během rozhovoru programátora a doménového experta vzniká všestranný jazyk, který popisuje základní principy domény, přičemž podstatná jména mohou prezentovat objekty a slovesa jednotlivá interakce mezi nima. Mohou ale nastat případy kdy slovesa nelze tak snadno namapovat na žádnou interakci mezi objekty. V tom případě je příhodné tuto funkci (interakci) definovat jako službu. Služby jsou vždy vystaveny jako rozhraní, nikoliv pro testovatelnost a podobně, ale pro vystavení množin soudržných operací. Zjednodušeně pokud musí existovat nějaká funkce důležitá pro doménu, ale nemůže být spojena s entitou nebo hodnotovým objektem, jedná se pravděpodobně o službu.

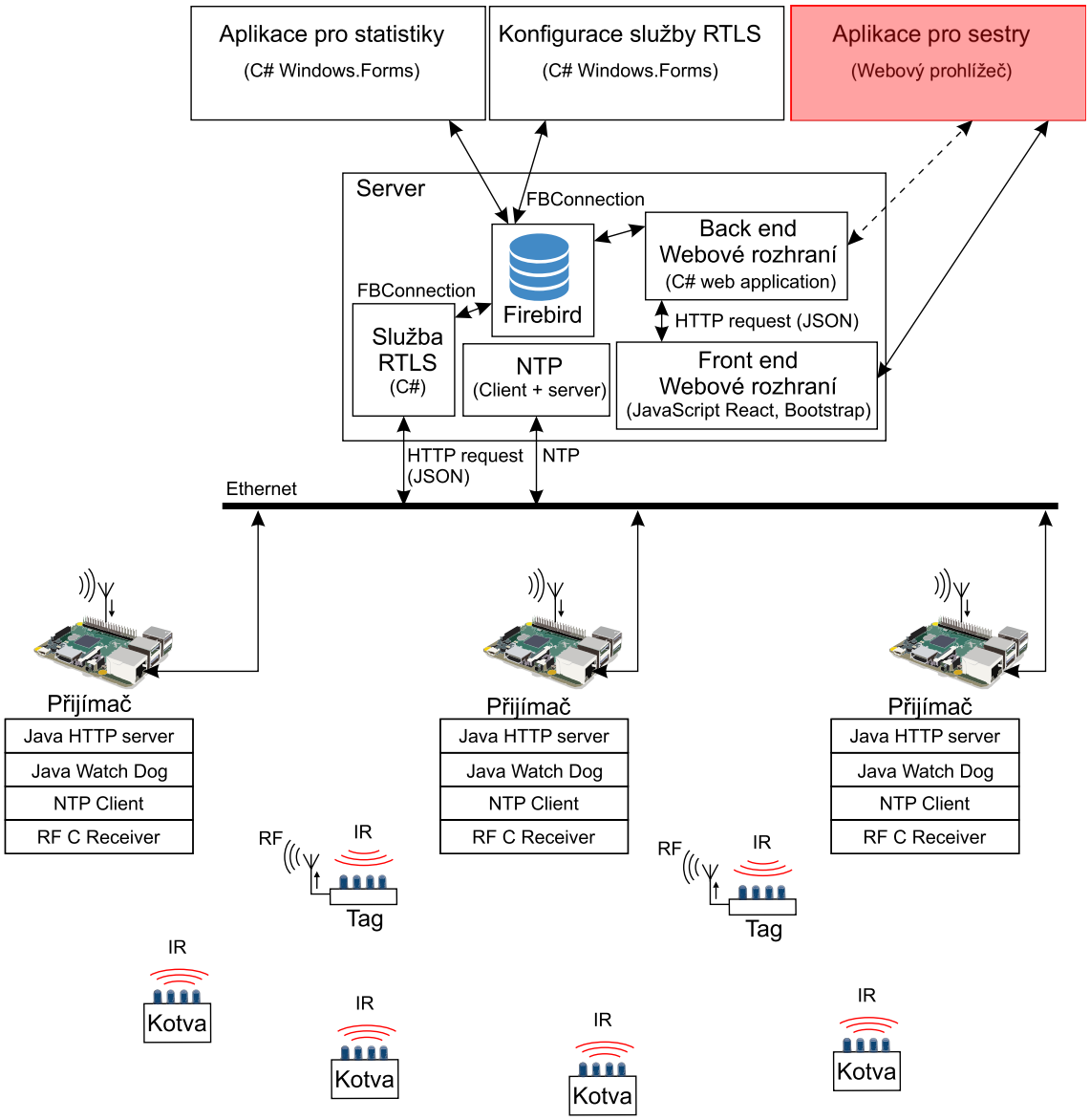
- **Fabrika (Factory)** – DDD navrhuje použití továrny, která zapouzdřuje logiku vytváření složitých objektů a agregátů. Zajišťuje, že klient nemá znalosti o vnitřním fungování objektů a jejich logiku.

3 Vlastní návrh řešení

V této kapitole budou popsány detaily, které jsou nutné pro pochopení praktického řešení této diplomové práce. Budou popsány jednotlivé architektury, UML diagramy, technická řešení, fáze testování, proces nasazení, sbírání zpětné vazby i retrospektivní zhodnocení.

3.1 Analýza současné situace

Jak již bylo v úvodu naznačeno (viz. kapitola 1) nedopadl předchozí pokus úspěšně. Toto řešení na míru bylo dodáno firmou Ekahau, která ovšem produkt zajišťující RTLS již na trh nedodává. Komunikace s dodavatelskou firmou neprobíhala na uspokojivé úrovni a firma nebyla schopná konzultovat vzniklé problémy v provozu. Jelikož šlo o black-box řešení, nešlo toto řešení modifikovat a případně opravit. Proto byl navržen lokační systém v reálném čase (Real Time Locating System - dále jen RTLS), který zastřešuje ucelené řešení. Z toho důvodu, že je každá část v plné režii, lze RTLS systém přímo naškalovat na požadavky zákazníka a optimalizovat v maximálně možném měřítku. V konečném důsledku byl personál urgentního příjmu zatížen větší administrativou a režií. Tato negativní zkušenost personálu pak hrála důležitou roli při definování požadavků a detailnější pochopení chodu na urgentním příjmu Fakultní nemocnice Ostrava. Tato diplomová práce zajišťuje část tohoto uceleného řešení a to měřící nástroj pro personál urgentního příjmu viz. obrázek 12.



Obrázek 12: RTLS schéma. Kotvy jsou v RTLS zařízení, které cyklicky vysílají do svého okolí informaci o sobě. Tyto kotvy jsou umístěny v každé místnosti minimálně jednou, vysílají pomocí infračerveného signálu informace o tom, ve které budově se kotva nachází, místnost ve které je kotva umístěna a roh místnosti. Předpokládá se maximálně 254 budov a 255 místností. Je nezbytné, aby na každé posteli pacienta byl přítomný tag. Lokalizační tagy jsou malá elektronická zařízení, která zajišťují lokalizaci v hlídaném prostoru. Tagy sbírají data z IR kotev a v případě potřeb vyšlou potřebná data. Data jsou vysílána v případě změně místnosti, zmáčknutí tlačítka na tagu nebo po uplynutí pěti minut po vysílání. Tato data jsou následně přijímána přijímači a webovým serverem, který tato data zpracuje a následně uloží do databáze. Tato centrální databáze pak obsahuje všechny události, které byly zaznamenány tagem. Toto úložiště dat pak využívá široká škála aplikací, konfiguračních nástrojů i budoucí statistické vyhodnocování. Obrázek a část textu převzata z [22].

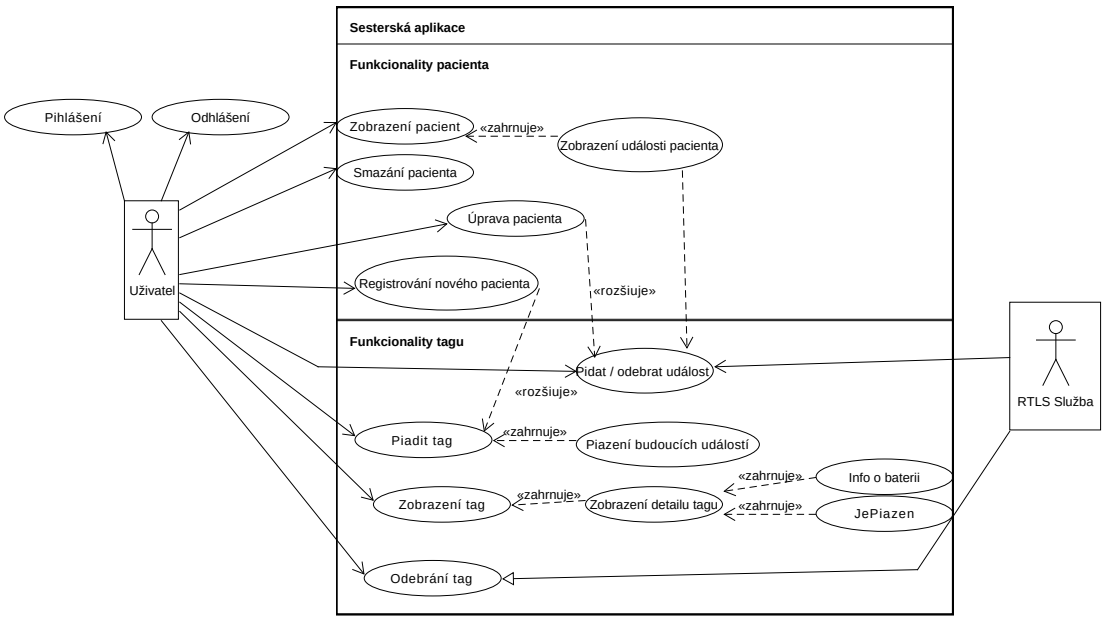
ekven

3.2 Popis řešení

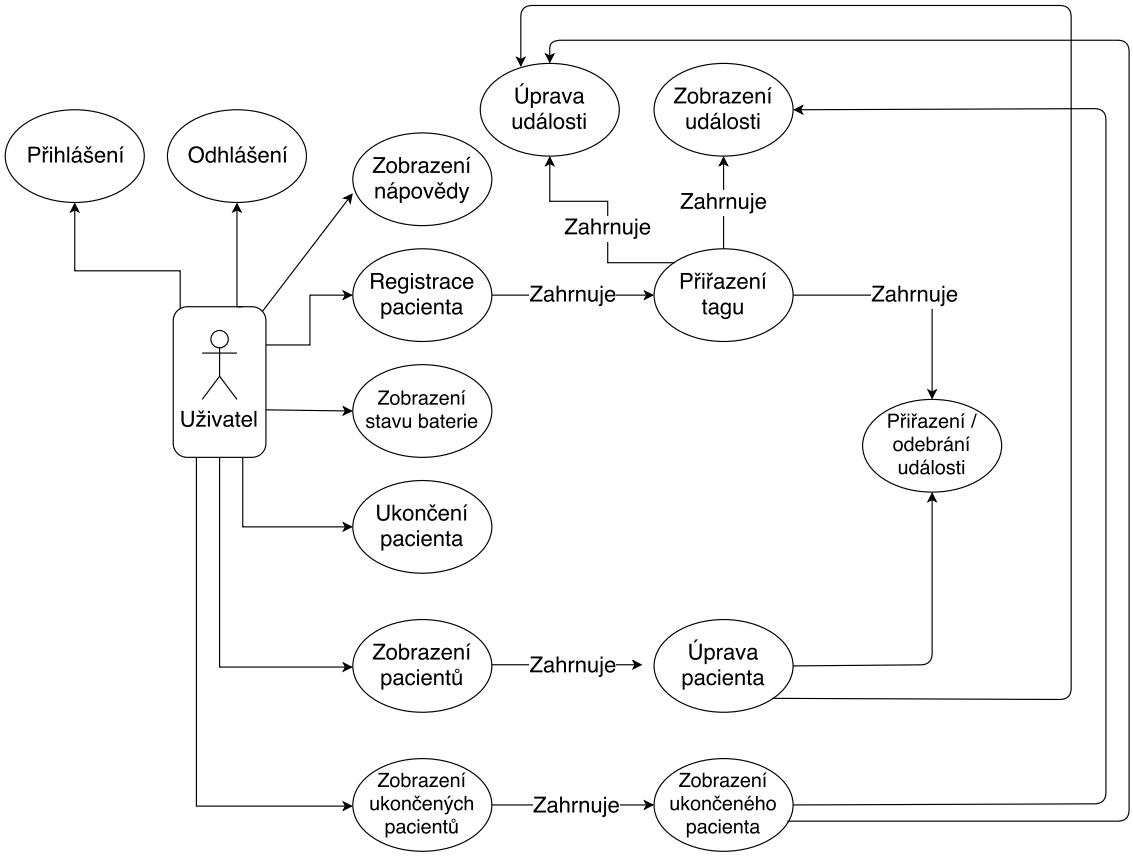
Praktická část řešení vznikala od června roku 2017. Počátečním krokem bylo seznámení se s doménou, doménovými experty a uživateli. Bylo potřeba sesbírat požadavky a informace o tom, jak to na urgentním příjmu FNO funguje. Tento krok bývá v procesu tvorby softwaru jeden z nejdůležitějších. Pokud nejsou dostatečně prodiskutovány a vymezeny požadavky, nemůže splňovat výsledný produkt očekávání zákazníka.

Unified Modeling Language diagramy

Unified Modeling Language (dále jen UML) je společný, sémanticky a syntakticky bohatý vizuální modelovací jazyk pro architekturu, návrh a implementaci komplexních softwarových systémů. Tento diagramový jazyk slouží pro rychlou orientaci, pochopení funkcionality a chování systému. V našem případě použijeme UML diagramy pro modelování funkcionalit (use case), které by měl systém umět. Komunikace byla pravidelná a z rozhovorů vyplynulo, že je třeba upravit use case diagram. Na obrázku 13 je znázorněn UML diagram funkcionalit systému na základě požadavků urgentního příjmu FNO po první návštěvě. Na obrázku 14 je poté znázorněn konečný diagram funkcionalit.[23]



Obrázek 13: Use case diagram po první schůzce.



Obrázek 14: Finální use case diagram.

Můžeme zpozorovat, že finální popis funkcionalit se od první verze značně liší. Některé aspekty byly zjednodušeny, některé zase naopak dodány. Po odsouhlasení use case diagramu lze pokračovat dále a to je návrh uživatelského rozhraní. Finální popis funkcionalit viz. níže.

- **Přihlášení do aplikace.**
- **Odhlášení z aplikace.**
- **Zobrazení nápovědy s popisem funkcionalit.**
- **Registrace pacienta.**
  Vytvoření nového měření, kde se dodá číslo karty, což je povinný údaj nutný pro vytvoření měření. Nepovinné údaje jsou jméno, prostřední jméno, příjmení, rodné číslo a datum narození.
- **Přiřazení tagu.**
  Tento krok je obsažen pouze v use casu registrace pacienta. Jedná se o úkon, kdy je jednotlivému pacientovi přiřazen tag.

- **Úprava události.**
  K této funkcionalitě se lze dostat od přiřazení tagu, úpravy pacienta či zobrazení ukončeného pacienta. Jde o úkon, kdy se upravují data události jako je čas či jeho název.
- **Zobrazení události.**
- **Zobrazení stavu baterie tagu.**
- **Ukončení pacienta (měření).**
- **Zobrazení pacientů.**
- **Úprava pacienta.**
  Jde o úpravu pacientových dat, práce s jeho událostmi nebo úprava jednotlivých událostí.
- **Zobrazení ukončených pacientů.**
- **Zobrazení ukončeného pacienta.**
- **Přiřazení / odebrání události.**
  Tento use case je o správě události aktivního pacienta (měření).

Návrh uživatelského rozhraní

Základní návrh (wireframe) byl bez režie ze strany klienta. Postup byl takový, že musel splnit všechny use casy (UC) z UML diagramu (viz. obrázek 14). Jak bylo uvedeno výše, UC se často měnil, doplňoval a redukoval, z toho vyplývá, že se poměrně hodně měnila i wireframe aplikace. První návrhy aplikace jsou dostupné viz. příloha 4.

Výběr technologického stacku[18]

Dalším krokem v praktické části bylo vybrat technologický stack technologií, který bude použit pro softwarové řešení pro správu pacientů a tagů. Bylo potřeba zohlednit již hotové vrstvy (databáze), zkušenosti autora s jednotlivými technologiemi, dlouhodobé trendy, podporu technologií, rychlost a výkonnost aplikace. Po zralé úvaze byl nakonec tech stack vybrán následovně:

- **Backend** – .NET Web Api2.3.1
- **Frontend** – ReactJS2.2.5
- **Databáze** (poskytnuta) – Firebird

[18] Výběr technologií ze kterých bude sestaven softwarový produkt

Technické řešení a popis frontendové části

V této podkapitole bude popsána detailněji frontendová část řešení. Jak bylo zmíněno výše, zvolenou technologií pro vývoj byl Javascriptový framework ReactJS.

Babel

Jelikož byla tato práce psána v EcmaScript 6 (ES6), který obsahuje funkcionality, které nejsou úplně všemi prohlížeči podporované, je potřeba překladače. Pro tyto účely byl vybrán Babel, který převádí Javascriptový zdrojový kód z vyšších verzí do verze nižší. Pokud by kód nebyl přeložen a tento kód by byl načten starší verzi prohlížeče, mohlo by se stát, že chování, vzhled nebo samotná aplikace bude nefunkční. Tento problém je nejvíce patrný u prohlížeče Internet Explorer, který ani doposud nebyl schopen podporovat dnes již základní Javascriptové konstrukce.[24]

```
patients.map(patient => console.log(patient.firstName));
```

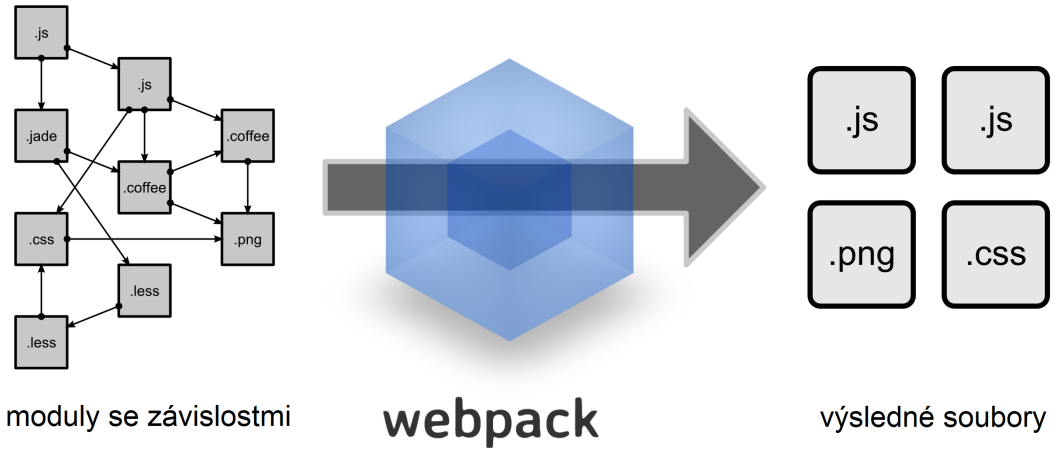
Výpis 15: ES6 kód využívající šipkového zápisu anonymní funkce.

```
patients.map(function (patient) {
   return console.log(patient.firstName);
});
```

Výpis 16: ES6 kód po přeložení pomocí Babelu.

Webpack

Webpack je nástroj, který sdružuje velké množství zdrojových kódů do jediného souboru-bundle. Jelikož místo mnoha drobných JavaScriptových souborů je výsledný soubor jen jeden, je výsledná režie prohlížeče jednodušší a rychlejší.[21]



Obrázek 15: Funkcionalita Webpacku. Převzato z [21].

Redux

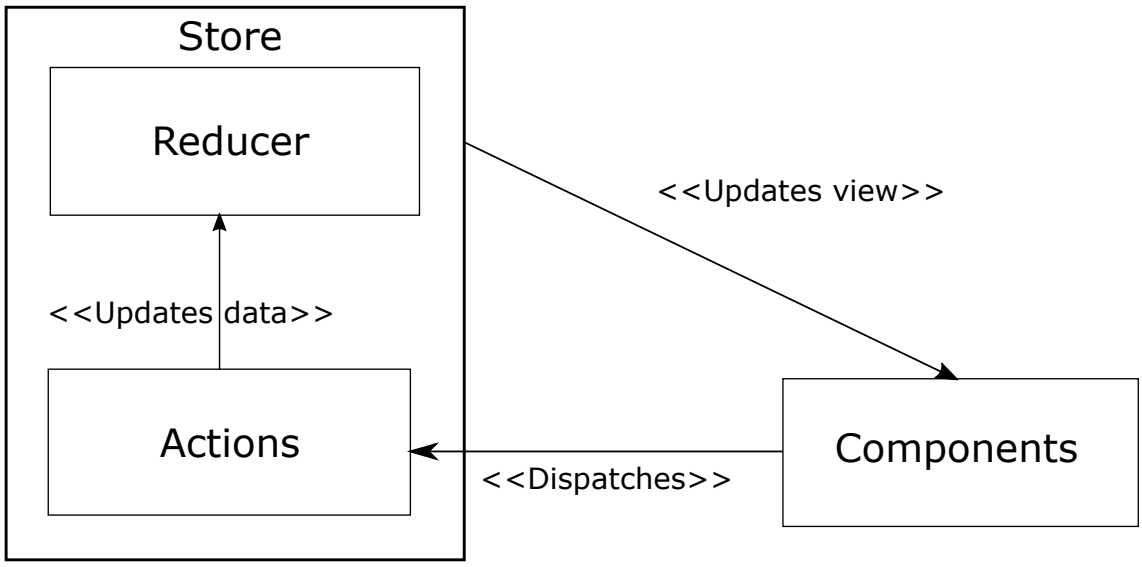
Redux je knihovna, která se inspirovala knihovnou Flux od společnosti Facebook a byla vytvořena v roce 2015 Danem Abramovem. Redux může usnadňovat správu stavu (state) aplikace. Jinými slovy, pomáhá řídit tok dat, která jsou zobrazována, a reaguje na akce uživatele.
Základní principy Reduxu jsou:

1. Store je single-source-of-truth. (viz. 2.2.5)
   Store celé aplikace je uložen v objektovém stromu v jednom uložišti. Stav všech komponent aplikace závisí právě na onom objektovém stromu.

2. Store je pouze v módu čtení. Nelze měnit store přímo z View vrstvy.

V Reduxu jsou pro práci se storem uzpůsobeny *akce*, které se vysílají (dále jen dispatch). Tyto akce po dispatchnutí volají speciální funkce tzv. *reduktory* (dále jen reducer), které následně modifikují store. Reducer si můžeme představit jako předdefinovaný objekt, který uchovává veškerá data části aplikace a také operace, jak s těmito daty pracovat.

Pro představu například *PatientsReducer* obsahuje veškeré informace o daných pacientech, jako je jejich jméno, příjmení, prostřední jméno, rodné číslo, číslo karty, datum narození apod. Dále obsahuje operace pro práci s těmito daty, jako je například *ADD_PATIENT*. Tyto operace mají určité požadavky a omezení a její konkrétní příklad bude dále popsán v kapitole (viz. reducer operace). Tento proces zobrazuje obrázek číslo 16.



Obrázek 16: Redux diagram

Jak na obrázku vidíme, komponenty *dispatchují akce*, které odchytí reducer a aktualizuje data. Tato nová data jsou následně promítnuta zpět na komponenty, které se podle potřeby překreslí.

Reducer operace

Pokud chceme v aplikaci změnit stav operace, musíme použít Redux. Redux obsahuje reducery, které se skládají z dat a operací, která modifikují data.

Data mohou být velmi jednoduchý objekt, jako je například následující výpis kódu.

```
const patientsInitialState = {
  patients : []
}
```

Výpis 17: Ukázka inicializačních dat v reduceru pacienta.

Operace pak reprezentuje jedna funkce, která přijímá dva argumenty. První argument je aktuální stav aplikace a druhý je akce, která chce tento stav měnit. Na typu této akce se pak rozhodne, která konkrétní operace se vykoná. Příklad této funkce lze vidět na následujícím výpisu kódu.

```
const PatientsReducer = (state = patientsInitialState, action) => {
  switch(action.type) {
    case 'ADD_PATIENT':

      return {
```

```
            ...state,
            patients: [...state.patients, action.payload]
        }

    case 'FETCH_ACTIVE_PATIENTS':
        state.patients = action.payload
        return Object.assign({}, state);
    default:
        return state;
  }
}
```

Výpis 18: Reducer funkce

Každý jednotlivý příkaz *case* reprezentuje jeden typ akce, kterou lze s daty vykonat. Akce kromě svého typu nese ještě *payload*, což jsou nová data. Je nutné si dát pozor, aby žádná akce nemodifikovala stávající stav, ale vracela vždy kompletně nový objekt (immutable). O toto se v našem případě stará metoda *Object.assign({}, state, newData)*, kde první parametr značí, že se má vytvořit úplně nový objekt.

React Bootstrap

React Boobstrap je open-source frontendový framework pro rychlejší a jednodušší vývoj webových aplikací. Obsahuje šablony návrhů založené na HTML a CSS pro typografii, formuláře, tlačítka, tabulky, navigaci, modální okna a mnoho dalších. Využitím tohoto frameworku lze zajistit správně reagující (responsive) aplikaci ve všech rozlišeních. React Bostrapobstrap je verze Bootstrapu, která byla přepsána do Reactu. Každá vlastnost je zde reprezentována React komponentou, kterou je potřeba nejdříve naimportovat. Je využívána napříč celou aplikací od jednoduchých tlačítek po složitější komponenty, jako jsou například modální okna. Příklad použití vidíme na výpisu 19.

```
import { Button } from "react-bootstrap";

export default class Login extends React.Component {

  render() {
    return (
      <div className="Login">
        <Button bsSize="large" type="submit">
          Login
        </Button>
      </div>
    );
  }
```

```
}
```

Výpis 19: Ukázka React Bootstrap

Technické řešení a popis backendové části

V této podkapitole bude popsáno technické řešení serverové části aplikace. Tato část byla implementována jako C# Web API (viz. 2.3.1) využívající architekturu S#harp, ORM systém NHibernate a databázi Firebird. Tyto technologie budou popsány níže.

S#harp architektura

Architektura, která byla použita v backendové části aplikace v této diplomové práci je S#harp. Jedná se o architektonický základ pro rychlé vytváření udržitelných webových aplikací, který využívá .NET MVC s NHibernate[19] jako ORM. Staví na designu Domain Driven Design (viz. výše). Architektura jasně definuje následující vrstvy, přičemž nižší vrstva by neměla vědět nic o té vrstvě vyšší.

Vrstvy jsou následující:

- **Doména (Domain)**
  V této nejnižší vrstvě jsou obsažené doménové objekty. Entity, hodnotové objekty, vazby mezi objekty či doménové servisy sídlí v této vrstvě. Doménová vrstva by měla být striktně nezávislá na jakýkolich vyšších vrstvách. Tato vrstva, která obsahuje doménovou logiku by tedy neměla vědět nic o tom, jaký typ ORM je použit, v jakém formátu data přicházejí či jaká technologie je použita pro uživatelské rozhraní.

- **Infrastruktura (Infrastructure)**
  Vrstva infrastruktury nastavuje zdroje dat třetích stran. V této vrstvě se například definuje způsob připojení (connection string) na databáze, volání do cizích API nebo načítání ze souboru. Tato vrstva umožňuje komunikovat s externími systémy přijímáním, ukládáním a poskytováním dat na vyžádání. V této vrstvě je logika, která řeší integrování našeho systému s externími systémy. Tato vrstva je závislá na doménové vrstvě a zároveň je silně nezávislá na vrstvách vyšších.

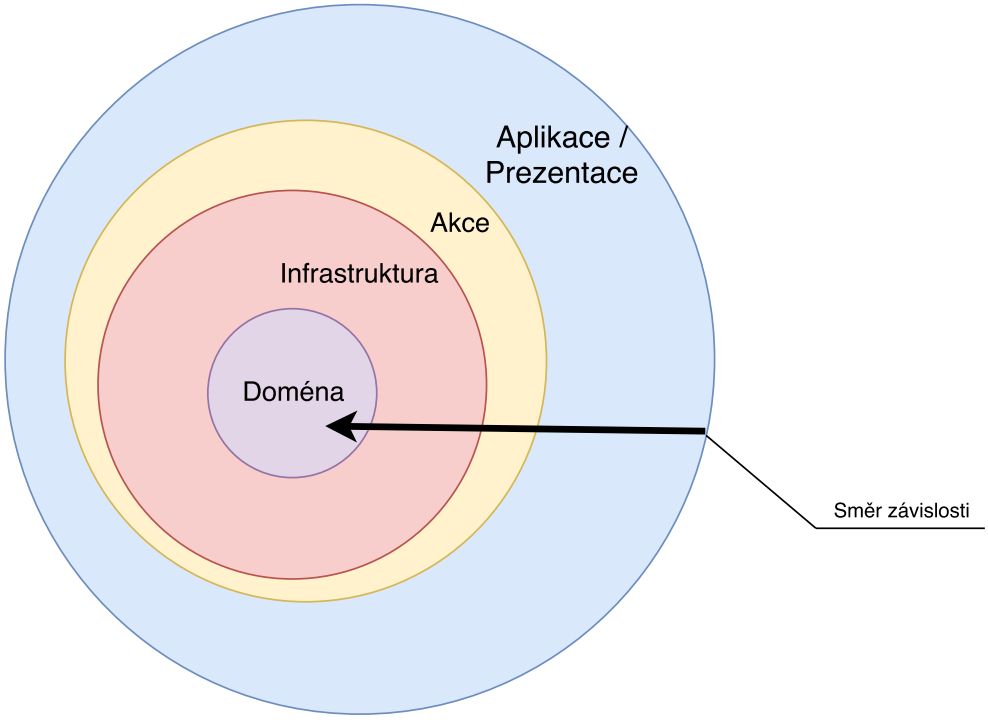
- **Akce (Tasks)**
  Vrstva akcí nebo také vrstva aplikačních služeb. Její úlohou je propojení jakékoli nepodnikatelské logiky z různých služeb třetích stran. Data, která jsou získána ve vrstvě infrastruktury jsou poslána vyšší vrstvě, kde dojde ke konkrétní úpravě výsledků na Data Transfer Object (DTO). Jedná se o objekt, který vychází z doménové vrstvy. Narozdíl ale od entity nebo hodnotového objektu obsahuje navíc nebo postrádá některé vlastnosti

[19] Open source projekt, který mapuje například z relačních databází na objektově relační v .NET frameworku.

a hodnoty. Jedná se tedy o pomocný objekt, který se používá k zapouzdření dat a odesílání z jednoho subsystému aplikace do jiného. Hlavní výhodou je, že snižuje množství dat, které je třeba odesílat mezi aplikacemi po síti. Představme si databázi pacientů, kde naše entita obsahuje ohromné množství vlastností (jméno, datum narození, rodné číslo, bydliště, obvodní lékař, poslední hospitalizace apod.) a je vyvíjen backend pro statistické vyhodnocení vytíženosti jednotlivých ordinací. V tomto případě nás nezajímá konkrétní údaje pacienta a DTO bude tedy mnohem menší objekt než entita. V této vrstvě by měly být nadefinované také jednotlivé překladače (mappers) mezi DTO a doménovým objektem. Tento krok je zásadní pro škálovatelnost a robustnost backendu. Mějme DTO objekt, který nám poslal klient, který obsahuje neúplné informace nebo data ve špatném formátu (např. chybějící číslo karty u pacienta). Překládač tedy přeloží DTO objekt na entitu a pokud chybí vlasnosti bez kterých například neuložíme do databáze, můžeme automaticky vrátit chybu. Tímto se zamezí zbytečné režií pro vrstvu infrastruktury. Tato vrstva je závislá na doméně a infrastruktuře a zároveň neví nic o vrstvách vyšších.

- **Aplikace / Prezentace (Application / Presentation)**
  Tato nejvyšší vrstva obsahuje nejčastěji uživatelské rozhraní (pokud backend nějaké má), rozhraní pro správu nebo v případě diplomové práce řadiče (controller). Jelikož je frontend zpracován jinou technologií a hlavně je na straně klienta, rozhraní pro adminy není třeba a v této vrstvě jsou tedy jen řadiče, které plní úlohu vstupních bodů do backendového systému. Tyto řadiče zpracovávají příchozí požadavky a odesílají odpověď zpět volajícímu. V této vrstvě se také mohou nacházet konfigurační soubory specifické pro aplikaci (číslo portu, cesty k certifikátům, přístupové údaje apod.). Tato vrstva je plně závislá na všech spodních.

Cílem architektury (viz. obrázek 17) je tedy oddělit jednotlivé vrstvy softwaru pro rozšířitelnost, škálovatelnost a testovatelnost. Motivací je také rozbít aplikaci do funkčních bloků, které mají logickou závislost. Se správně rozvrženou architekturou tedy nemusíme předělávat celý software, pokud budeme migrovat například databázi. Stačí jen změnit vrstvu infrastruktury. Pokud nejsme spokojeni s dosavadním rozhraním třetí strany, přepisujeme tedy pouze kód ve vrstvě akcí. V reálném světě doména (obraz reality) zůstává pořád stejná v nezávislosti na tom, jaké změny jsou na naší straně. Software by měl plně reflektovat požadavky domény a pokud se doména změní, je třeba zavést změny ve všech vrstvách.[20]



Obrázek 17: Rozvrstvení aplikace podle S#harp architektury a ukázka závislosti.

NHibernate

NHibernate je knihovna pro objektové modelování relačních databází (Object Relational Mapping dále jen ORM) určená pro použití v .NETu. Tyto ORM vrstvy jsou zde z toho důvodu, že neexistuje snadné spojení mezi objektovým a databázovým světem. v objektovém světě se vše točí kolem objektů, které drží informace o datech. v databázovém světě se data ukládají do tabulek a jednotlivé objekty jsou reprezentovány řádky a jeho vlastnosti pak sloupcemi. V tomto řešení byla nad touto NHibernate knihovnou ještě naimplementována knihovna FluentNHibernate, která eliminuje úmorné ruční mapování pomocí rozsáhlých XML dokumentů (viz. 2.1). Na základě nadefinovaných konvencí pro FluentNHibernate dokáže tato nadstavba nad ORM vrstvou nabídnout časově a kódově úspornou alternativu ke standartnímu mapování v NHibernate. Konvence můžeme psát v C# jazyce, což zlepšuje čitelnost, debugování a refaktorovatelnost. Dalším důvodem je, že kompilátor nám nehlídá obsah XML dokumentu a snadno tedy může docházet k typografickým chybám, kdy se zamění názvy proměnných nebo vlasností objektů. Na výpisu 20 vidíme příklad použití mapování.[25]

```
public class PatientOverride : IAutoMappingOverride<Patient>
{
   public void Override(AutoMapping<Patient> mapping)
   {
      mapping.Cache.ReadOnly().Region("LongTermReadWrite");
      mapping.Id(x => x.Id).GeneratedBy.Sequence("GEN_PATIENT_ID");
```

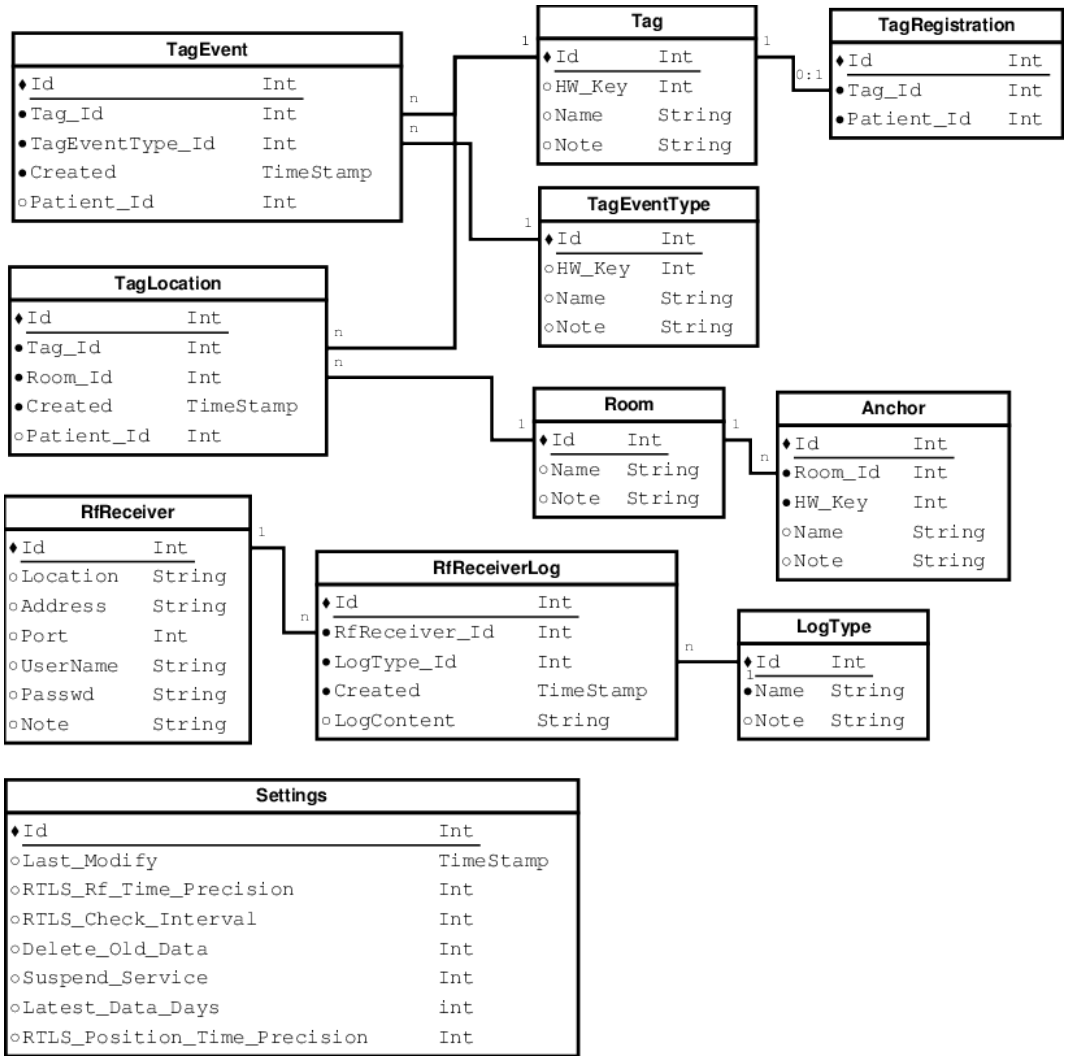
```
        mapping.HasOne(patient => patient.Tag).Not.LazyLoad().Cascade.All();
    }
}
```

Výpis 20: Ukázka mapování pomocí FluentNHibernate

Další funkcí NHibernatu je poskytnutí metod pro vykonávání dotazů nad databází. Generuje dotazy v Hibernate Query Language (dále jen HQL), což je dotazovací jazyk na nižší úrovni. Pomocí nastavení dialektu dokáže tedy přeložit různé syntaxe (MS SQL, MySQL, Oracle, PostgreSQL, SQLite, Firebird) dotazovacích jazyků do jednoho univerzálního HQL jazyka. NHibernate také zajišťuje správu a mapování dědičností a vazeb do databázové světa ve formě primárních a cizích klíčů. Dále, jak je vidět na výpisu, 20 využívá NHibernate i cachování dat. Caching je proces, kdy se data ukládají do dočasného místa v paměti, která jsou rychle dostupná. Caching se obnovuje, pokud jsou data změněná.

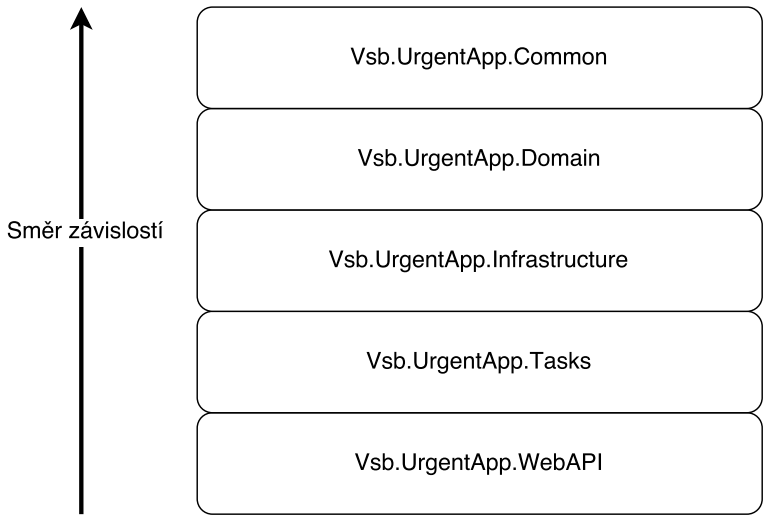
Firebird

Firebird je open-source multiplatformní relační databáze, která vznikla z InterBase databáze v roce 2006. Jelikož návrh databáze vzniknl dřív, byla tato databáze společně se schématem již poskytnuta. Pro účely tohoto řešení byly navíc vytvořeny tabulky *Patient* a *User* společně s generátory pro tvorbu ID. Databázové schéma je na obrázku 18.



Obrázek 18: Schéma RTLS databáze.

WebAPI rozčleněno podle S#harp architektury znázorněno na obrázku 19



Obrázek 19: Implementace backendu podle S#harp architektury.

Na obrázku 20 je znázorněn sekvenční diagram vytvoření pacienta. Tento diagram je pouze zjednodušený obrázek těch nejzásadnějších operací, které se dějí mezi danými participanty.

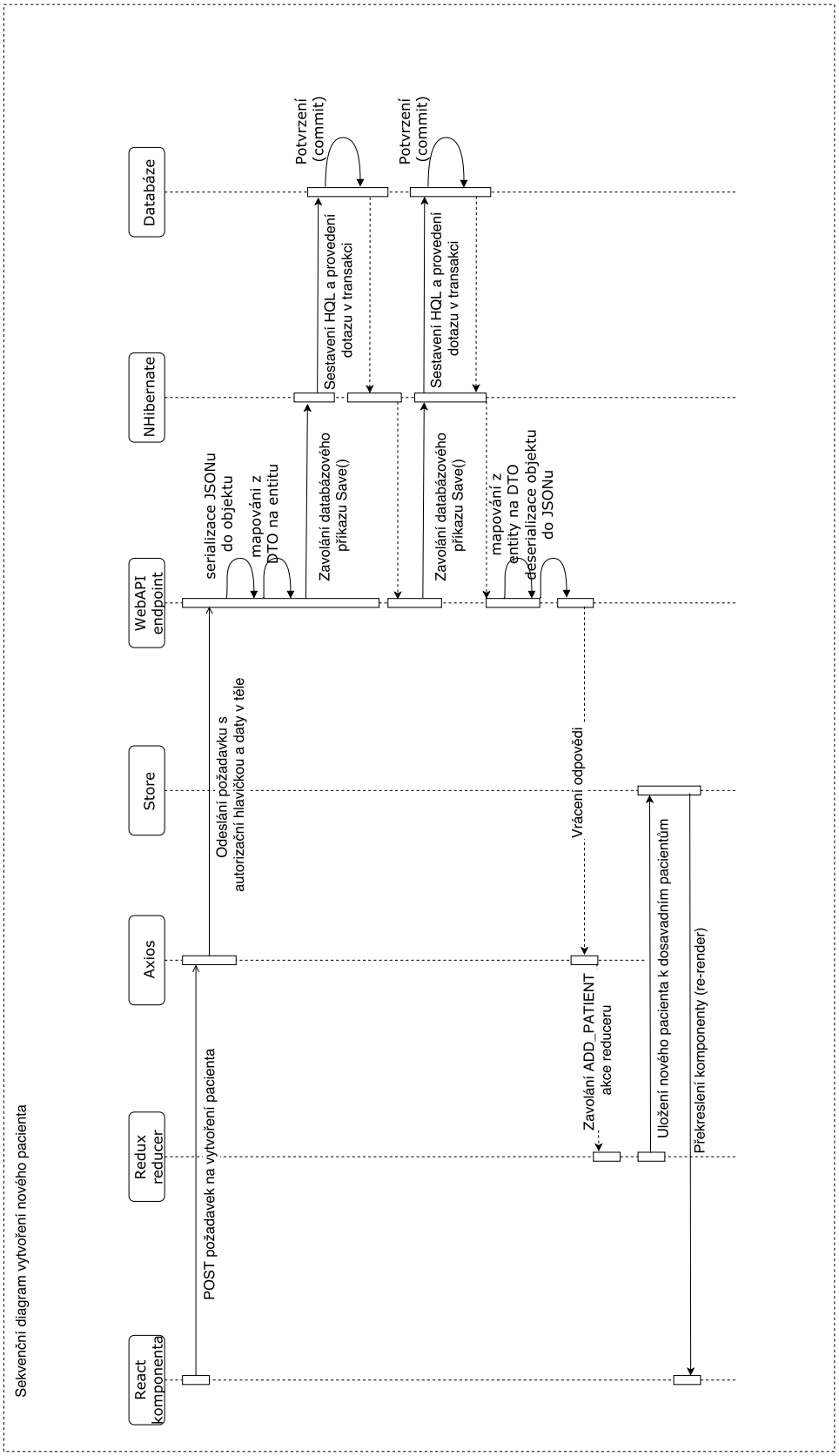
Kompilace

Tato kapitola bude pojednávat o způsobu sestavení všech částí programu nutných pro úspěšné spuštění aplikace. V prvé řadě je nutné nainstalovat následující nástroje:

- .NET kompilátor (obsažen ve Visual Studio)
- NodeJS + NPM (doporučená verze 6.10.1)

Řešení se dělí na složku *backend*, která obsahuje zdrojové kódy z backendové části a *frontend*, která zase obsahuje zdrojové kódy frontendové části. Pro zkompilování souborů backendové části nám pouze stačí vývojové prostředí Visual Studio (viz. 3.4), které obsahuje integrovaný .NET kompilátor. Tento nástroj nám poté zakompiluje soubory (přípona *.cs*) do dynamických linkovaných knihoven (Dynamic Link Library dále jen DLL). Tyto soubory se nachází v cestě *backend\Vsb.UrgentApp.UI\Vsb.UrgentApp.API\bin*. Backend obsahuje ještě jeden projekt, který obsluhuje požadavky pro získání html souboru (který obsahuje styly a zabundlovaný JS soubot viz. 2.2.1 a 2.2.5). Proces kompilace je stejný jako u API projektu a DLL soubory se nachází v *backend\Vsb.UrgentApp.UI\Vsb.UrgentApp.UI\bin*. V případě, že nejsou staženy závislosti knihoven třetích stran, je potřeba přes Nugget Manager obnovit tyto knihovny v případě, že selže automatický proces inicializace Visual Studia.

Sestavení frontendu je trochu složitější. Zaprvé je nutno v adresáři *frontend* spustit příkaz *npm install*, kterým stáhneme všechny externí zavislosti, které jsou nadefinovány v souboru *package.json*. V tomto souboru je také závislost na Webpack, který se tímto příkazem nainstaluje



Obrázek 20: Sekvenční diagram vytvoření pacienta.

také, ale je jej potřeba nainstalovat také globálně. Globální instalace je potřeba, aby byl příkaz *webpack* dostupný v příkazové řádce, aby mohl být spouštěn pomocí skriptů. Této globální instalace docílíme spuštěním příkazu *npm install webpack@3.8.1 --global*. Dále- ve stejném adresáři je potřeba spustit příkaz *npm run build*, který pomocí webpacku sestaví výsledný JS soubor a zároveň s pomocí Babelu převede na verzi, které rozumí i starší webové prohlížeče. Tento výsledný JS soubor poté vloží do *backend\Vsb.UrgentApp.UI\Vsb.UrgentApp.UI\built* , kde je narefencován v HTML souboru.

3.3 Testování, nasazení a zpětná vazba

Testování

Testování softwaru je proces vedoucí k informování zúčastněných stran o kvalitě softwarového produktu nebo testované služby. Provádění testů může také poskytnout objektivní a nezávislý pohled na software. Testovací procesy zahrnují provádění programu nebo aplikace s úmyslem najít chyby softwaru a ověřit, zda je softwarový produkt vhodný pro použití. Testují se klasické použití, ale použití, která se málokdy stanou (corner cases).

Testování backendové části probíhalo pomocí unitových testů (dále unit testing). Unit testing je testovací metoda, pomocí níž jsou zkoušeny jednotlivé bloky zdrojového programu. Testování unitů je zajištěno pomocí frameworku NUnit, který podobně jako JUnit v Java světě, zajišťuje prostředí a runnery pro testy. Ukázka unitového testu je na výpisu. 21 V tomto případě je unit testu dáno na vstup DTO s daty, které potom vstupuje do metody *Create*, přičemž návratová hodnota je uložený objekt v databázi. Ten nesmí být *null*.

```
[Test]
public void CreatePatientWithoutTag_PatientDto_success()
{
   PatientDto patient = new PatientDto
   {
      Card_Id = 135464,
      SocialSecurityNumber = "13556/0018",
      FirstName = "Olafson",
      MiddleName = "Lameson",
      LastName = "Lamason",
      BirthDate = new DateTime(1950, 05, 05).ToString(),
      Tag = null,
      Deleted = null
   };

   var result = patientTasks.Create(patient, false);

   Assert.IsNotNull(result);
```

```
}
```

Výpis 21: Ukázka unit testů.

Implementovány byly i integrační testy, které zajišťují správnou funkci třetích stran (v našem případě inicializace připojení k databázi). Přes integrační testy bylo i pro snazší testování jiných částí naimplementováno vyresetování databáze do původního stavu a naplnění testovacích hodnot. Ve výpisu 22 je znázorněno testování připojení k databázi a ve výpisu 23 je ukázáno vyresetování databáze a naplnění testovacími hodnotami

```
[Test]
public void ConnectToFBDb_void_success()
{
   try
   {
      Infrastructure.Db.FireBirdConnection.InitializeFirebird();
   }
   catch (Exception ex)
   {
      Assert.Fail("Expected no exception, but got: " + ex.Message);
   }
}
```

Výpis 22: Ukázka integračního testu.

```
[Test]
public void ResetDb_void_success()
{
   ConnectToFBDb_void_success();
   ResetGenerator_void_success();
   ResetDatabaseData_void_success();
   FillData_void_success();
}
```

Výpis 23: Vrácení databáze do původního stavu.

Testování frontendové části probíhalo manuálně procházením use casů a corner casů. Existují však nepřeberné množství automatizovaných nástrojů jako například Mocha.js, cypress.io nebo Selenium. Testování v době vývoje backendu probíhalo pomocí aplikace Postman- (viz. kapitola 3.4) produkt od firmy Google, který dokáže sestavovat a posílat HTTP požadavky. Následně testování probíhalo ve spolupráci s testerkou Monikou Borovou. Na základě šablony pro oznámení chyby (bug report template) byly nedostatky zanalyzovány a opraveny. Příklad bug reportu je na obrázku 21.

Category	Label	Value
Bug ID	ID number	1
	Name	Úprava čísla karty
	Reporter	Monika Borová
	Submit Date	29.01.2008
Bug overview	Summary	
	URL	v1/patients/edit
	App version	1.0.9
Environment	Platform	Google Chrome
	Operating System	Windows 10
	Browser	Google Chrome
Bug details	Steps to reproduce	1. Vyberu již existujícího pacienta. 2. Změním číslo karty na 788895643213fr 3. Dojde sice k upozornění, že číslo karty musí být číslo, ale po stisku tlačítka Změnit dojde k uložení a uzavření dialogu.
	Expected result	Tlačítko změnit by mělo být disabled.
	Actual result	Tlačítko změnit by mělo není disabled.
	Description	
Notes	Notes	
Developer Actions	Přidán ternární operator pro vyhodnocení stringu v čísle karty.	

Obrázek 21: Oznámení chyby.

Nasazení

Server pro nasazení testovacího prostředí byl poskytnut autorem. Jelikož jde o Windows Server 2012, proběhlo vypublikování do internetu s pomocí Internet Information Services (dále jen IIS), což je služba pro obsluhu příchozích požadavků na server. Výpis aplikací v IIS je zobrazen na obrázku 22. Aplikační pool *SesterskaAplikace* je pool, který na dotázání vrací HTML soubor, který spouští celý frontend aplikace. Aplikační pool *UrgentAppApi* je pool pod kterým běží server, který naslouchá požadavky na dotyčném portu. Je nutné daným aplikačním poolům

nastavit cestu, kde mají dotyčné soubory nutné pro chod aplikace hledat. Kořenová cesta je defaultně *C:\inetpub\wwwroot*, kde je obsažena složka se jménem daného poolu. Také je potřeba nastavit adresu či IP adresu s portem[20].

```
PS C:\Users\trannann> Import-Module WebAdministration
PS C:\Users\trannann> Get-ChildItem -Path IIS:\AppPools

Name                    State        Applications
----                    -----        ------------
.NET v2.0               Started
.NET v2.0 Classic       Started
.NET v4.5               Started
.NET v4.5 Classic       Started
Classic .NET AppPool    Started
DefaultAppPool          Started      Default Web Site
SesterskaAplikace       Started      SesterskaAplikace
UrgentAppApi            Started      UrgentAppApi
```

Obrázek 22: Aplikační pooly IIS.

3.4 Nástroje

- Visual Studio 2015
  Integrované vývojové prostředí (Integrated Development Environment dále jen IDE), které bylo využito v této práci pro vývoj backendu v C# od firmy Microsoft.

- Visual Studio Code
  Open source IDE, které bylo využito v této práci pro vývoj frontendové části v HTML, CSS a ReactJS.

- Node Packager Module
  Neboli NPM je správce a registr balíčků (knihoven) . Jedná se o online úložiště JavaScript knihoven, které si může kdokoliv stáhnout a použít ve svých projektech. Jedná se také o verzovací nástroj, který hlídá závislosti a podporované verze mezi danými závislostmi.

- Nugget
  Nugget je podobně jako NPM nástroj, který je dodáván jako rozšíření pro Visual Studio IDE. Slouží také jako online úložiště knihoven a registr, který hlídá závislosti mezi danými knihovnami. Je určen výhradně pro vývojové prostředí aplikací firmy Microsoft.

- Postman
  Postman je aplikace firmy Google, která funguje s HTTP API rozhraními. Jde o aplikaci, která snadno sestavuje požadavky a čitelně prezentuje odpovědi ze serveru.

[20] Daný port je potřeba také otevřít ve Windows Firewallu jako inbound rule.

- Umlet
  Freeware aplikace pomocí které lze vytvářet jednoduché UML diagramy. Nicméně byla nedostačující co se týče uživatelského pohodlí a nabízených grafických komponent.

- Draw.io
  Bohatý freeware nástroj pro vytváření všech typů UML diagramů. Obsahuje také nepřeberné množství hotových grafických komponent (firewall, databázový server atd.) a obsahuje velké množství modifikací. Existuje jako webová i stand-alone aplikace.

4 Závěr

Cílem této práce bylo vytvořit aplikaci pro personál urgentního příjmu Fakultní nemocnice Ostrava, pomocí které by bylo možné ovládat RTLS na uživatelské úrovni. Cílem bylo tedy v první řadě seznámit se s nynější situací na urgentním příjmu. Pochopit a lokalizovat problémy, které trápily toto oddělení. Poté bylo potřeba tyto poznátky přenést do užitkových a vývojových diagramů. Na základě těchto komplexních diagramů dále představit návrhy uživatelského prostředí aplikace a paralelně přitom sestavovat backendovou část pro chod aplikace. Po schválení všemi zúčastněnými lidmi zbývalo naprogramovat podle návrhu skutečnou frontendovou část. V průběhu vývoje aktivně probíhala komunikace s koncovými uživateli produktu, kterými jsou personál urgentního příjmu Fakultní nemocnice Ostrava. Aktivně se komunikovalo z důvodu zlepšení, problémů či funkcionality. Po důkladném testování bylo řešení nasazeno v produkčním prostředí, kde poté proběhlo testování v reálném provozu.

Práci na tomto projektu, kde byla zahrnuta široká škála dovedností lidí, byla velice obohacující. Bylo mi umožněno navrhnout řešení od úplného začátku (green field software development), přičemž toto řešení je poté používáno v oblasti, kde je důraz na maximální nechybovost. Tato práce obsahuje počáteční analýzu, návrh uživatelského rozhraní, vývoj obou částí aplikace, testování a nakonec nasazení. Budoucí motivací je dále statistické zpracování výsledků a určení rizikových míst (bottleneck), kde dochází ke zbrždění vyšetřovacího cyklu pacienta. Tento statistický výstup poté povede k analýze a zlepšení těchto nedostatků, které zefektivní celé oddělení a tím i zvýší kvalitu vyšetření.

První kapitola ze čtyř pojednává o úvodu, motivaci a cílech této diplomové práce. Pro praktickou část bylo potřeba zanalyzovat dostupné technologie pro tvorbu webových aplikací, čemuž se věnuje kapitola 2, kde je i mimojiné probrána architektura řešení, klady a zápory technologií a způsob komunikace mezi frontendem a backendem. Předposlední kapitola pak pojednává o vlastním návrhu řešení. V této části práce byla popsána analýza situace před dodáním softwaru, postup vývoje a detailní vysvětlení architektury aplikace. Poslední kapitola pak zhodnocuje celou práci a nastiňuje budoucí motivaci, která pak uzavírá tuto diplomovou práci.

Použitá literatura

[1] ORFALI, Robert., Dan. HARKEY a Jeri. EDWARDS. *Client/server survival guide.* 3rd ed. New York: John Wiley, c1999. ISBN 0471316156.

[2] BERSON, Alex. *Client-server architecture.* New York: McGraw-Hill, c1992. ISBN 0070050767.

[3] RENAUD, Paul E. *Introduction to client/server systems: a practical guide for systems professionals.* 2nd ed. New York: Wiley Computer Pub., c1996. ISBN 0471133337.

[4] GOURLEY, David. a Brian. TOTTY. *HTTP: the definitive guide.* Sebastopol, CA: O'Reilly, 2002. ISBN 1565925092.

[5] SHIFLETT, Chris. *HTTP developer's handbook.* Indianapolis, Ind.: Sams, c2003. ISBN 0672324547.

[6] STEVE BURNETT AND STEPHEN PAINE. *RSA Security's official guide to cryptography.* New York: Osborne/McGraw-Hill, 2001. ISBN 0072192259.

[7] ERIK T. RAY. *Learning XML.* 2nd ed. Beijing: O'Reilly, 2003. ISBN 1449378870.

[8] ELLIOTTE RUSTY HAROLD & W. SCOTT MEANS. *XML in a nutshell.* 3rd ed. Sebastopol, CA: O'Reilly, 2004. ISBN 1449379044.

[9] PITTS-MOULTIS, Natanya *XML black book.* 2nd ed. Scottsdale, AZ: Coriolis Group Books, c2001. ISBN 1576107833.

[10] BASSETT, Lindsay. *Introduction to Javascript Object Notation: a to-the-point guide to JSON.* Sebastopol, CA: O'Reilly, 2015. ISBN 1491929480.

[11] DUCKETT, Jon. *HTML & CSS: design and build websites.* Indianapolis, IN: Wiley, c2011. ISBN 1118008189.

[12] JENNIFER NIEDERST ROBBINS. *Learning web design: a beginner's guide to HTML, CSS, Javascript, and web graphics.* 4th ed. Sebastopol, CA: O'Reilly, 2012. ISBN 1449319270.

[13] DUCKETT, Jon, Gilles RUPPERT a Jack MOORE *JavaScript & jQuery: interactive front-end web development.* Indianapolis, IN: Wiley, 2014. ISBN 1118531647.

[14] CROCKFORD, Douglas. *JavaScript: the good parts.* Sebastopol: OReilly, 2008. ISBN 0596517742.

[15] FLANAGAN, David. *JavaScript: the definitive guide.* 6th ed. Sebastopol, CA: O'Reilly, 2011. ISBN 0596805527.

[16] SESHADRI, Shyam a Brad GREEN. *AngularJS up and running.* Sebastopol, California: O'Reilly, 2014. ISBN 1491901942.

[17] FREEMAN, Adam. *Pro AngularJS.* New York, NY: Apress, 2014. ISBN 1430264489.

[18] [online]. [cit. 2018-01-30]. Dostupné z: https://reactjs.org/

[19] EVANS, Eric. *Domain-driven design: tackling complexity in the heart of software.* Boston: Addison-Wesley, c2004. ISBN 0321125215.

[20] [online]. [cit. 2018-01-30]. Dostupné z: http://sharp-architecture.readthedocs.io/en/latest/

[21] [online]. [cit. 2018-01-30]. Dostupné z: https://github.com/webpack/docs/wiki/what-is-webpack

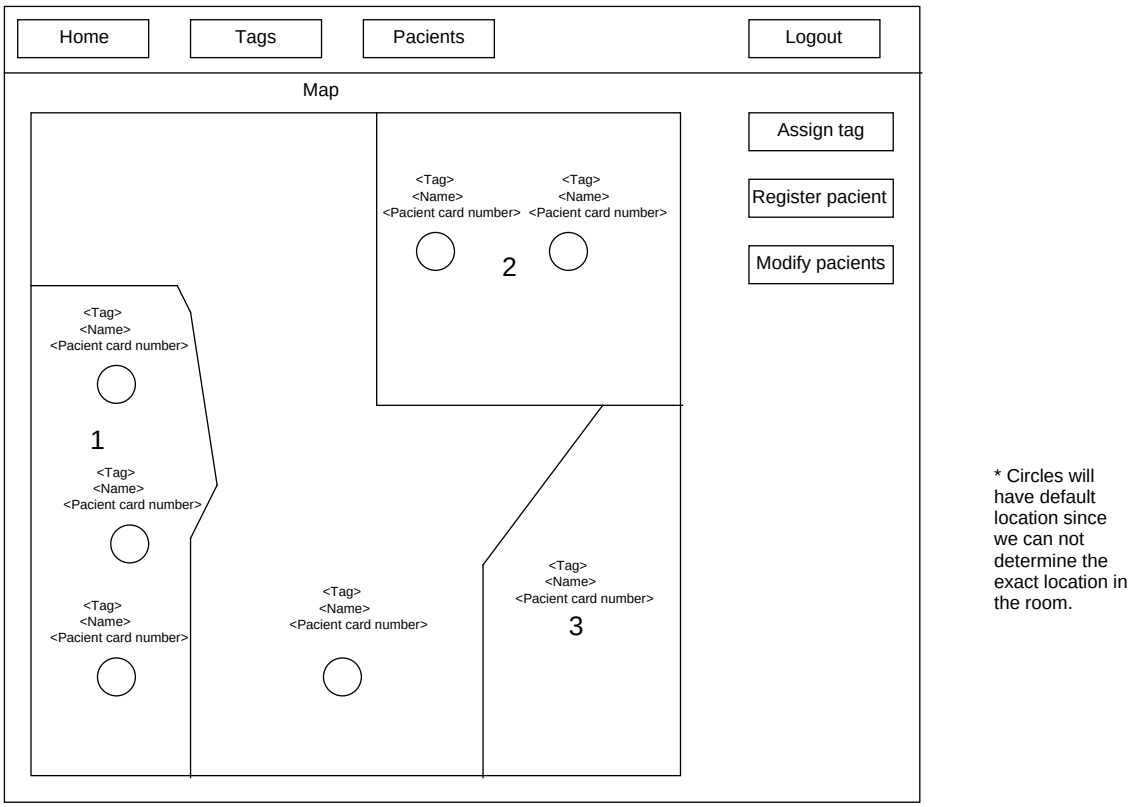
[22] Dokumentace RTLS: Technický popis systému RTLS [online]. Ostrava, 2017 [cit. 2018-02-03]. Dokumentace. VŠB-TUO. Vedoucí práce Ing. Jaromír Konečný, PhD.

[23] UML 2.0 in a Nutshell. 2nd ed. Sebastopol: O'Reilly Media, 2009. ISBN 0596552319.

[24] SANDEEP K. P. *Quick ES2015 Scripting Using Babel.js: Learn ES6 important features quickly* CreateSpace Independent Publishing Platform, 2016. ISBN 1532783868.

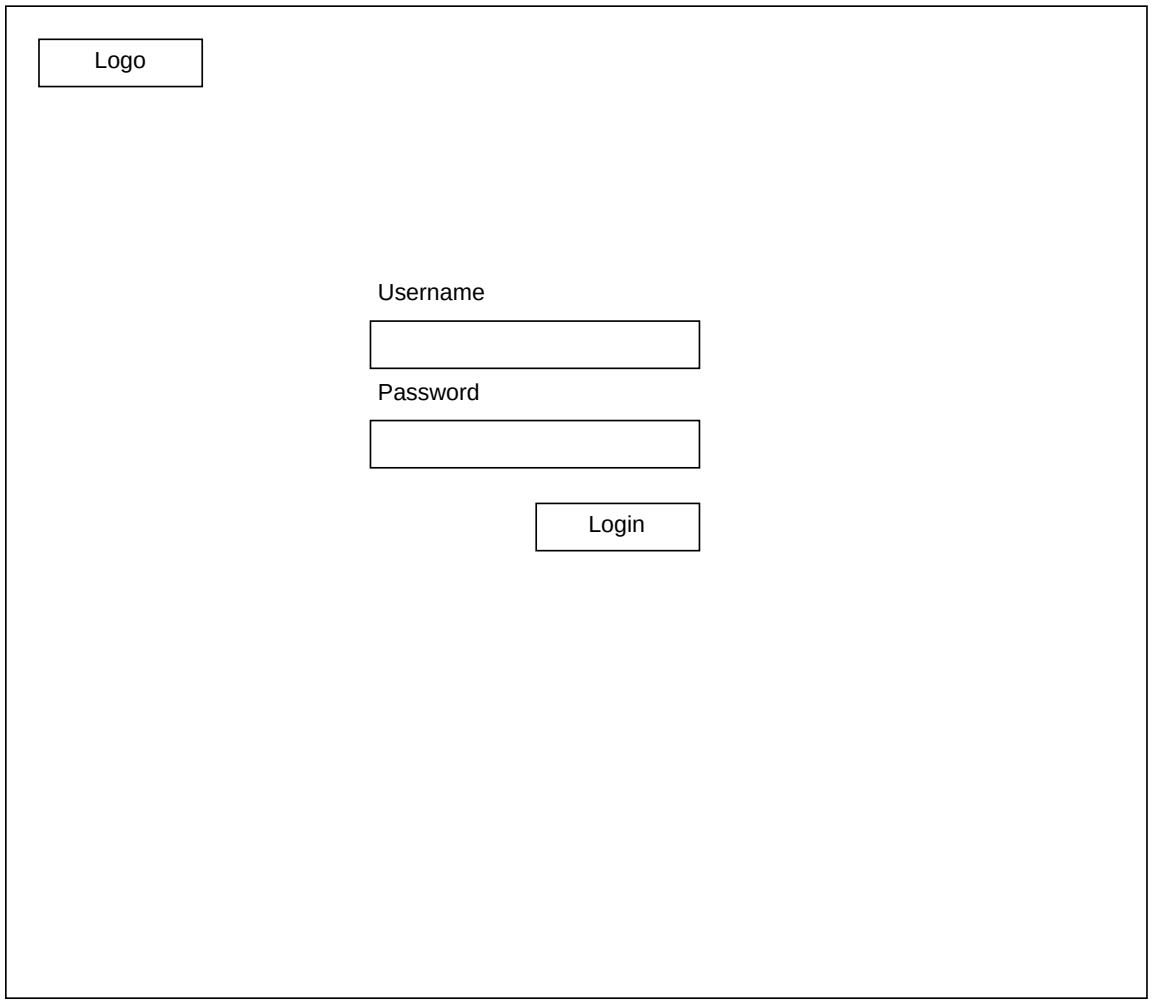
[25] KUATE, Pierre Henri. *NHibernate in action.* Greenwich: Manning, c2009. ISBN 1932394923.

A Grafický návrh uživatelského rozhraní 1.



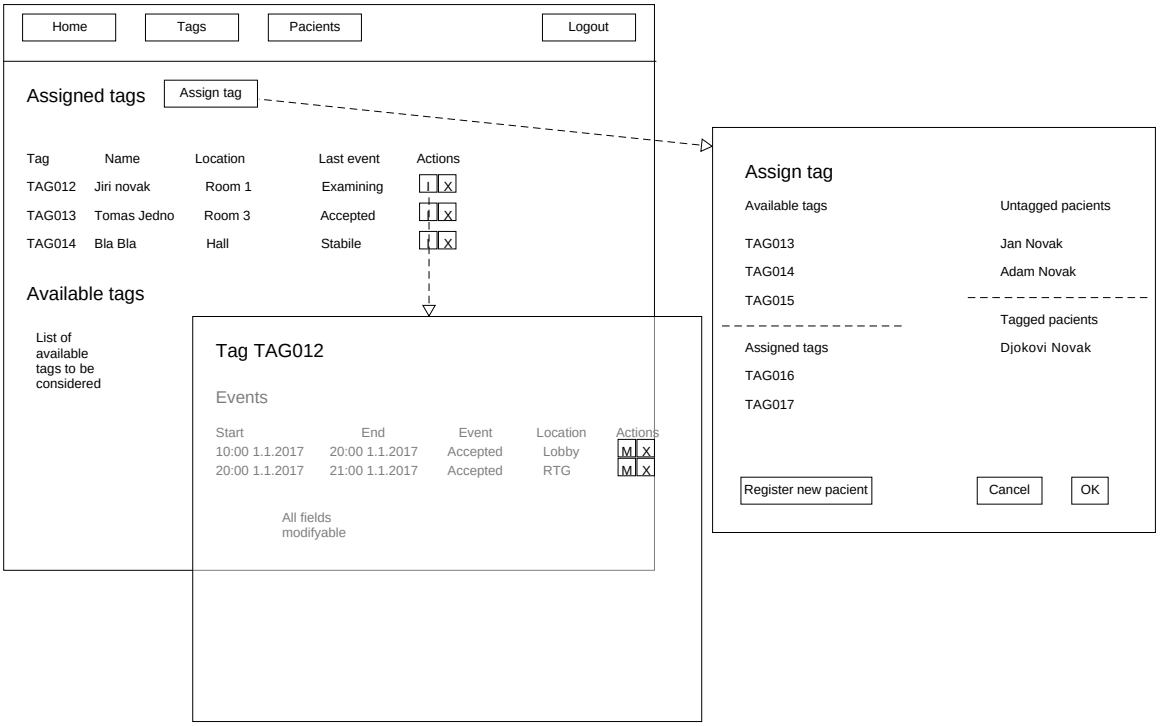
Obrázek 23: Grafický návrh uživatelského rozhraní 1.

B Grafický návrh uživatelského rozhraní 2.



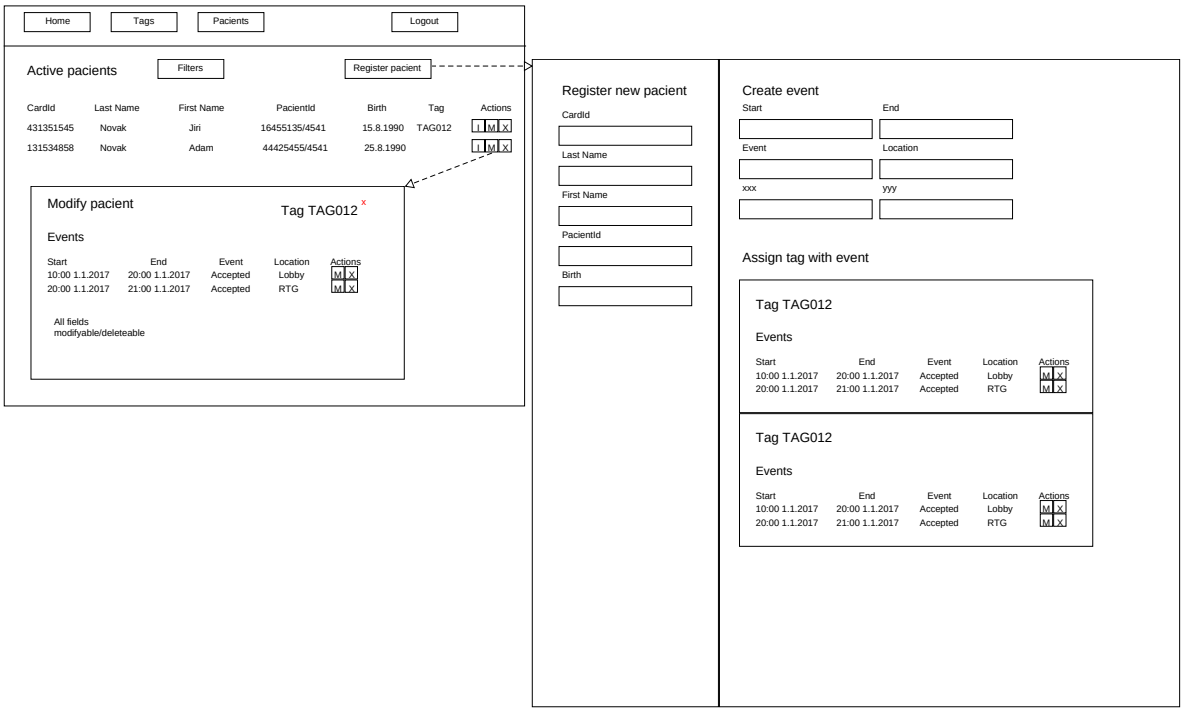
Obrázek 24: Grafický návrh uživatelského rozhraní 2.

C Grafický návrh uživatelského rozhraní 3.



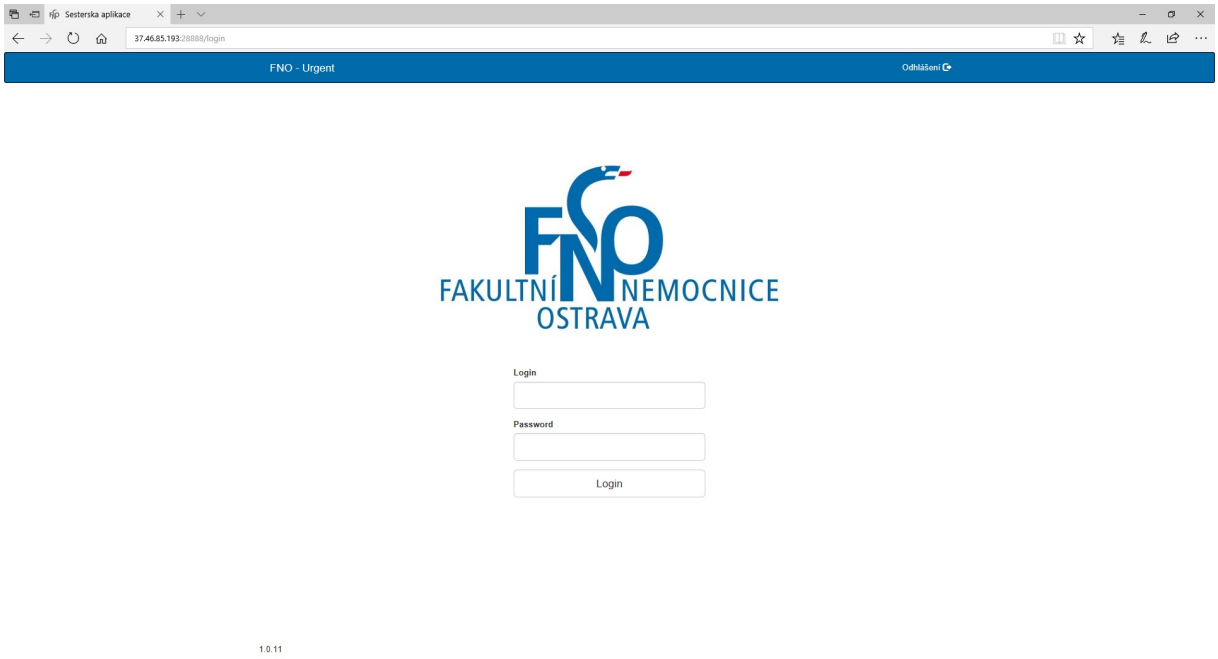
Obrázek 25: Grafický návrh uživatelského rozhraní 3.

D Grafický návrh uživatelského rozhraní 4.



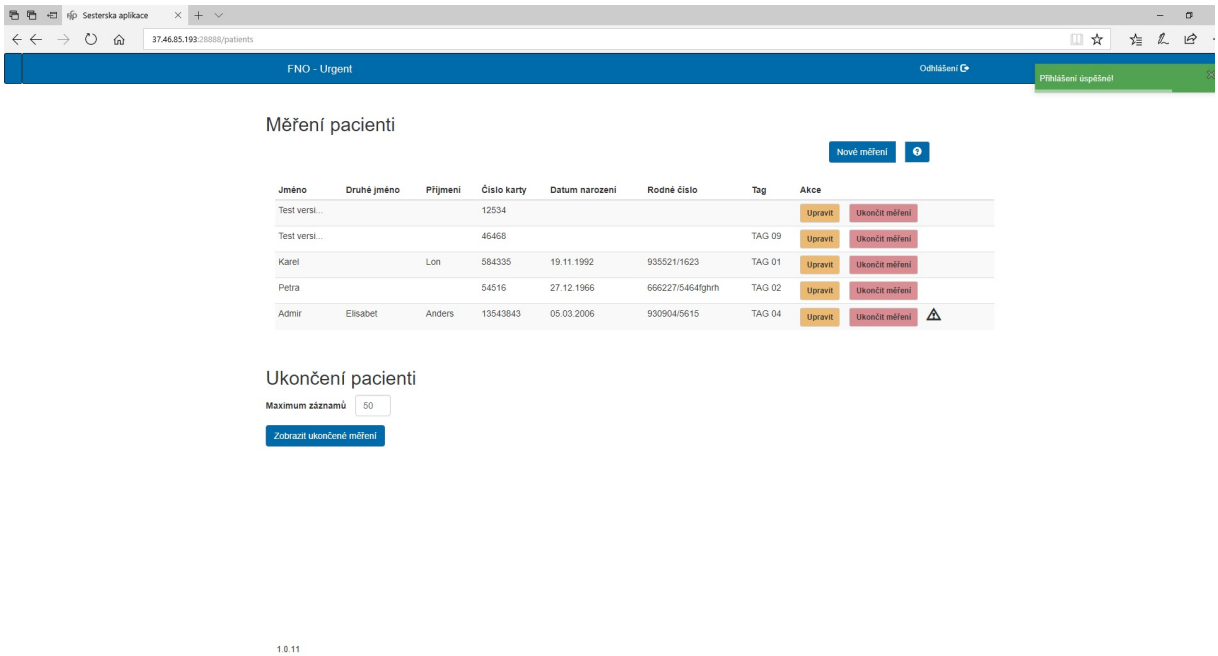
Obrázek 26: Grafický návrh uživatelského rozhraní 4.

E Přihlašovací okno aplikace.



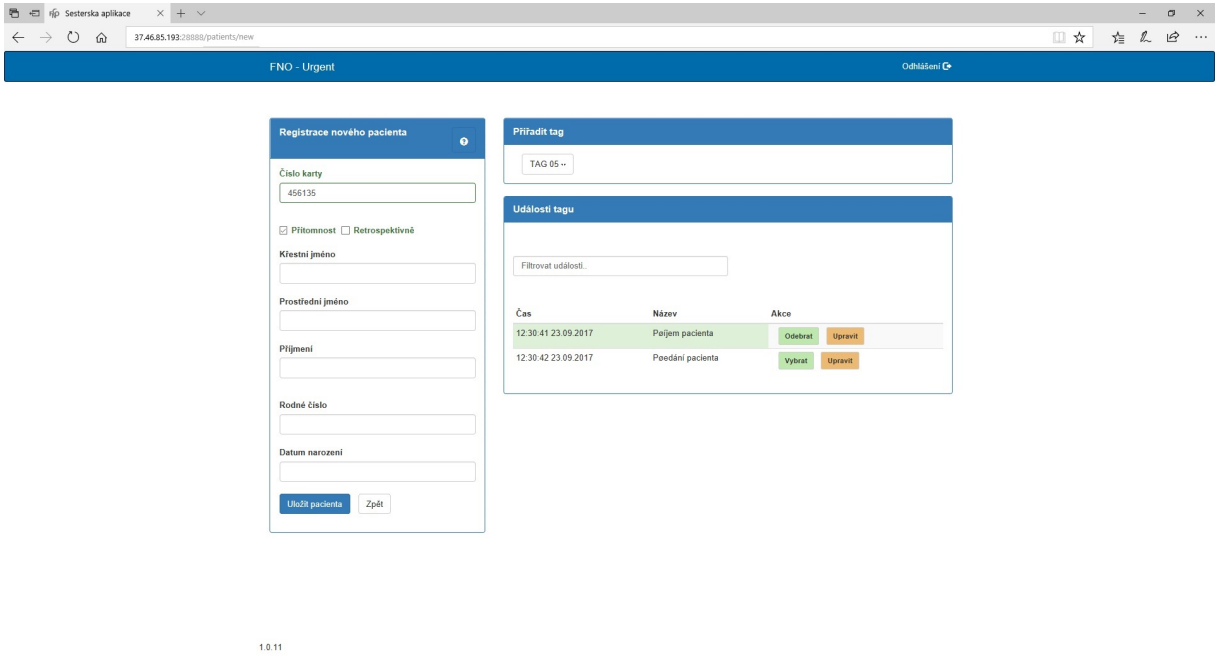
Obrázek 27: Přihlašovací okno aplikace.

F Hlavní okno aplikace.



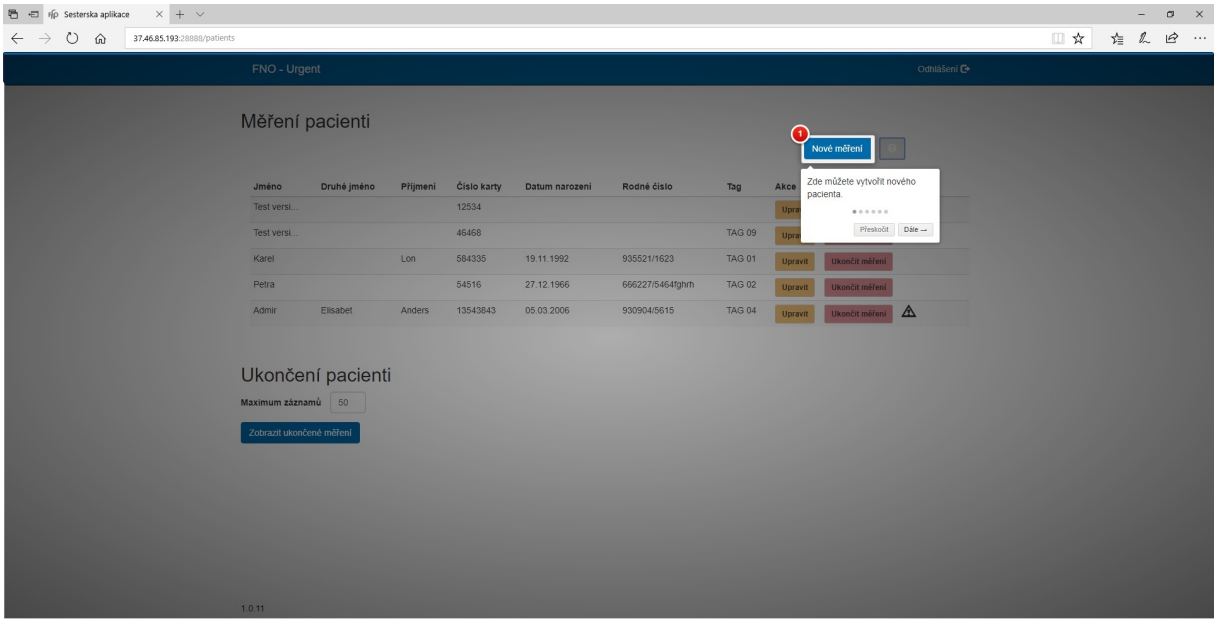
Obrázek 28: Hlavní okno aplikace.

G Okno registrace nového pacienta aplikace.



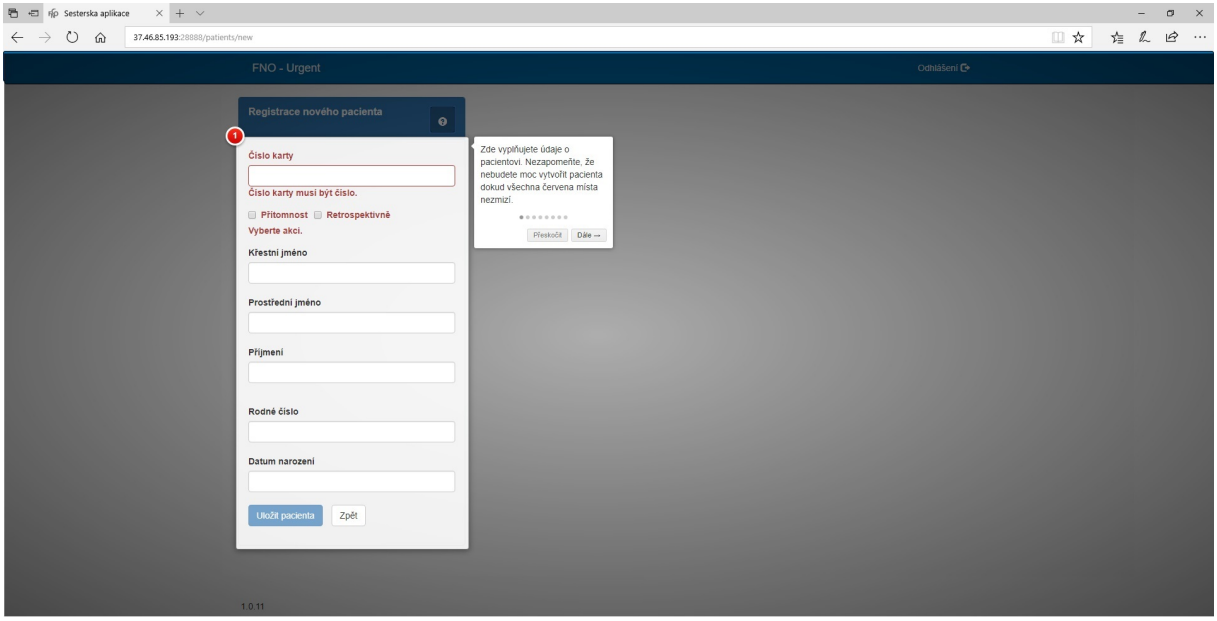
Obrázek 29: Okno registrace nového pacienta aplikace.

H Nápověda v hlavním okně aplikace.



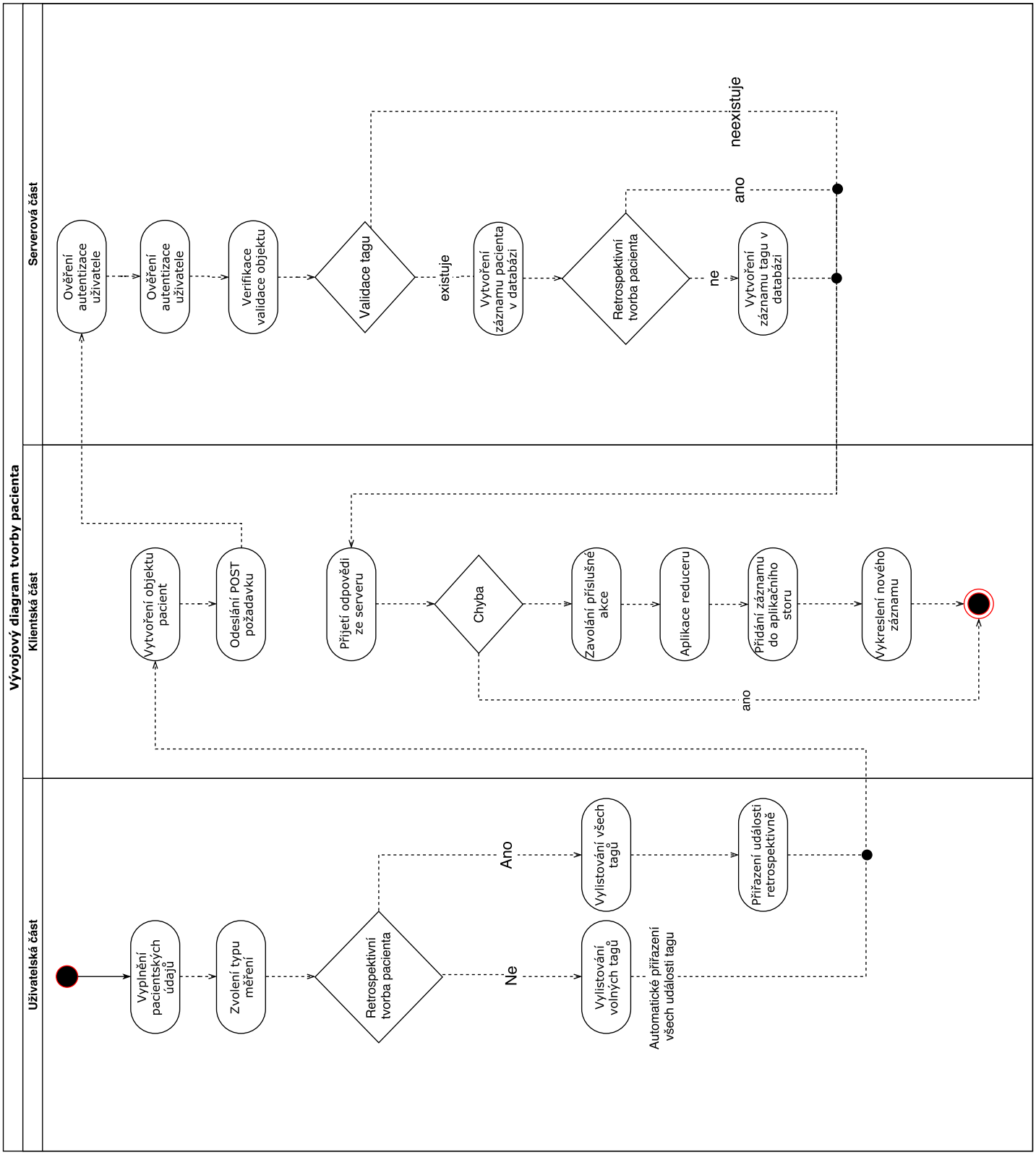
Obrázek 30: Nápověda v hlavním okně aplikace.

I Nápověda v okně registrace nového měření.



Obrázek 31: Nápověda v okně registrace nového měření.

J Diagram tvorby pacienta.



Obrázek 32: Diagram tvorby pacienta.