Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Vývoj multiplatformní aplikace v jazyce Javascript

Bakalářská práce

Autor: Arťom Evsin
Studijní obor: Aplikovaná informatika

Vedoucí práce: Mgr. Daniela Ponce, Ph.D.

Hradec Králové leden 2018

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

vlastnoruční podpis

V Hradci Králové dne 26.4.2018 Arťom Evsin

Poděkování:

Rád bych poděkoval paní Mgr. Daniele Ponce, Ph.D. za cenné rady, věcné připomínky a vstřícnost při konzultacích a vypracování bakalářské práce.

Anotace

Bakalářská práce se zaměřuje na analýzu a porovnání nástrojů pro vytvoření multiplatformní aplikace v JavaScriptu. V práci jsou popsány vybrané knihovny a postupy, které lze využít k vytvoření takových aplikací. Pomocí těchto knihoven a postupů byla vytvořena jednoduchá aplikace a popsán postup při transformaci již existující aplikace. Tyto aplikace využívají architekturu klient-server a jejich klientské aplikace komunikují spolu se serverem pomocí REST API.

Annotation

Title: Multiplatform application development in Javascript language

This bachelor's thesis focuses on analysis and comparison of tools for development of multiplatform application in JavaScript. The work describes selected libraries and processes that can be used to develop such applications. These libraries and processes have been used to develop a simple application and describe process of transformation existing application into multiplatform. These applications use client-server architecture and their client applications use REST API to communicate with server.

Obsah

1 Úvod ... 1

2 Cíl práce ... 2

3 Metodika zpracování ... 3

4 Multiplatformní aplikace ... 4

4.1 Historie Javascriptu ... 4

4.2 Architektura aplikace ... 5

4.3 Serverová část ... 6

4.3.1 Node.js ... 6

4.3.2 Express ... 7

4.3.3 Sails.js ... 8

4.4 Webová aplikace ... 9

4.4.1 React.js ... 9

4.4.2 Vue.js ... 11

4.5 Desktopová aplikace ... 11

4.5.1 Electron ... 12

4.6 Mobilní aplikace ... 13

4.6.1 Cordova ... 13

4.6.2 React Native ... 13

5 Ukázka vývoje multiplatformní aplikace ... 15

5.1 Architektura ukázkové aplikace ... 15

5.2 Vytvoření serverové části ... 16

5.2.1 Příprava prostředí ... 16

5.2.2 Struktura aplikace ... 16

5.2.3 Úložiště ... 17

5.2.4 Model aplikace ... 17

5.2.5 Závislosti aplikace ... 19
5.3 Vytvoření klientské aplikace pro webový prohlížeč ... 20
5.3.1 Struktura aplikace React.js ... 20
5.3.2 Příprava prostředí ... 20
5.3.3 Komponenty ... 21
5.3.4 Propojení se serverovou části ... 22
5.3.5 Spuštění aplikace ... 22
5.4 Vytvoření desktopové aplikace ... 23
5.4.1 Příprava prostředí ... 23
5.4.2 Struktura aplikace ... 23
5.4.3 Sestavení aplikace ... 24
5.5 Vytvoření mobilní aplikace ... 24
5.5.1 Příprava prostředí ... 25
5.5.2 Struktura aplikace ... 25
5.5.3 Sestavení aplikace ... 26
6 Ukázka transformace na multiplatformní aplikaci ... 27
6.1 Popis aplikace ... 28
6.2 Analýza přechodu ... 28
6.3 Struktura serverové části aplikace ... 29
6.4 Struktura klientské aplikace ... 31
6.5 Odlišnosti aplikace pro různé platformy ... 31
6.6 Zabezpečení ... 33
6.7 Použité knihovny ... 33
7 Shrnutí výsledků ... 34
8 Závěry a doporučení ... 35
9 Seznam použité literatury ... 37

Seznam obrázků

Obr. 1 Typy aplikace 5
Obr. 2 Struktura Node.js........ 6

1 Úvod

Téma této bakalářské práce bylo zvoleno z důvodu vytvoření multiplatformní aplikace ve firmě, kde pracuji. Ačkoliv projekt nikdy nebyl zrealizován, zkoumal jsem možná řešení, a jedním z nich bylo vytvoření multiplatformní aplikace v JavaScriptu.

S rostoucí popularitou a rozšířením chytrých telefonů, tabletů a dalších „chytrých zařízení“ se zvyšuje i poptávka po softwaru určeném právě pro tuto platformu.

Zvyšuje se tlak na přístup k informacím 24 hodin denně, ať už jsme v práci, doma nebo na cestách. Musíme tak přizpůsobit zobrazení těchto informací na jednotlivých platformách – web, počítač, telefon.

Ještě před několika lety bylo běžné, že pro každou platformu se vyvíjela samostatná aplikace. Tento přístup však má svá úskalí v podobě časové i finanční náročnosti, špatné udržitelnosti kódu (implementace pro jednotlivé platformy se musí udržovat zvlášť), a v neposlední řadě porušení principu DRY (don't repeat yourself). Ačkoliv jednotlivé implementace aplikace pro jednotlivé platformy se v něčem liší (často i jenom v syntaxi), logika zůstává stejná.

Pravděpodobně i toto jsou důvody, které vedly k tak velké popularitě Javascriptu a jeho úspěšnému rozšíření na jiné platformy.

2 Cíl práce

Cílem této práce je optimalizovat výběr nástroje pro vývoj multiplatformní aplikace v JavaScriptu s ohledem na znovupoužitelnost a udržovatelnost kódu.

Zaměřil jsem se na aplikace, jejichž primárním cílem je zobrazovat data uložena na serveru a upravovat je. Jde tedy o hardwarově nenáročné aplikace, jejichž výpočetně náročné operace lze provádět na straně serveru. Je to dáno hlavně rozdílem v hardwarových možnostech zařízení, na nichž se aplikace bude používat. Hardware, na němž běží serverová část aplikace, se pak musí přizpůsobit nárokům nejen podle složitosti prováděných operací, ale i podle hustoty komunikace mezi serverem a klientskými aplikacemi.

Aplikace by měla být spustitelná v prostředí operačního systému Windows, Debian, Android a iOS. Také by měla být přístupna pomocí prohlížeče jako webová aplikace.

Co se týče nároku na software, tak všeobecně vzato, klientské aplikace lze spouštět na jakémkoli zařízení, které obsahuje prohlížeč s podporou spouštění JavaScriptu.

Pojmem platforma se v této práci označuje prostředí, ve kterém běží aplikace. Konkrétně se jedná o prohlížeč, v případě webové aplikace, případně operační systém v případě nativní aplikace, ať už se jedná o stolní počítač či telefon.

Obsahem této práce je teoretická část, kde zkoumám frameworky a knihovny, které lze použít pro vytvoření multiplatformní aplikace, a část praktická, která obsahuje ukázku procesu vytvoření takovéto jednoduché aplikace.

3 Metodika zpracování

V souladu s cílem této práce bude porovnáno několik JavaScriptových knihoven, které lze využít pro vývoj multiplatformní aplikace. Při porovnávání těchto knihoven bude kladen důraz na jejich znovupoužitelnost, stabilitu vývoje a komunitu kolem těchto knihoven.

S použitím těchto knihoven bude vytvořena jednoduchá multiplatformní aplikace, na níž se předvede postup vytvoření takové aplikace. Při samotném vývoji se bude maximalizovat množství sdíleného kódu mezi klientskými aplikacemi pro jednotlivé platformy. Dále bude popsán postup při transformaci již existující aplikace na multiplatformní.

Na těchto praktických ukázkách bude ověřeno, zda vybrané knihovny jsou optimální pro vytvoření multiplatformní aplikace.

4 Multiplatformní aplikace

Multiplatformní aplikace lze rozdělit na dva druhy. Prvním jsou aplikace, jejichž binární kód je spustitelný na různých zařízeních, popřípadě různých operačních systémech. Typickým příkladem těchto aplikací jsou aplikace v jazyce Java, které po zkompilování běží v prostředí JRE (Java Runtime Environment), které musí být nainstalováno na konečném zařízení. Jsou tedy spustitelné kdekoliv, kde může běžet toto prostředí. Tento typ je vhodný, pokud se nebude vyvíjet webová verze, která má omezení v podobě nativní podpory scriptovacích jazyků, která je zpravidla omezená na JavaScript.

Dalším typem multiplatformních aplikací jsou aplikace, které jsou uskupením několika implementací pro jednotlivé platformy. Pro každou platformu tedy existuje samostatná aplikace. Z pohledu funkčnosti však mezi těmito aplikacemi není rozdíl.

4.1 Historie Javascriptu

Za posledních několik let Javascript získává čím dál víc na popularitě. Jedním z posledních velkých milníků bylo vytvoření prostředí Node.js, které umožňuje vytvářet aplikace v Javascriptu na straně serveru.

Ačkoliv za celou historii Javascriptu byl několik pokusu o použití Javascriptu na serveru, například Netscape Enterprise Server již v roce 1995, žádné z implementací se nedostalo takového ohlasu a popularity jako právě Node.js. Za jeden z hlavních důvodů takové popularity se dá označit fakt, že Node.js implementuje událostmi řízenou architekturu (event-driven architecture), což umožňuje provádět neblokující vstupní/výstupní operace (non-blocking I/O). Díky tomuto přístupu je Node.js schopen zpracovávat mnohonásobně vyšší počet současných připojení (concurent connections).

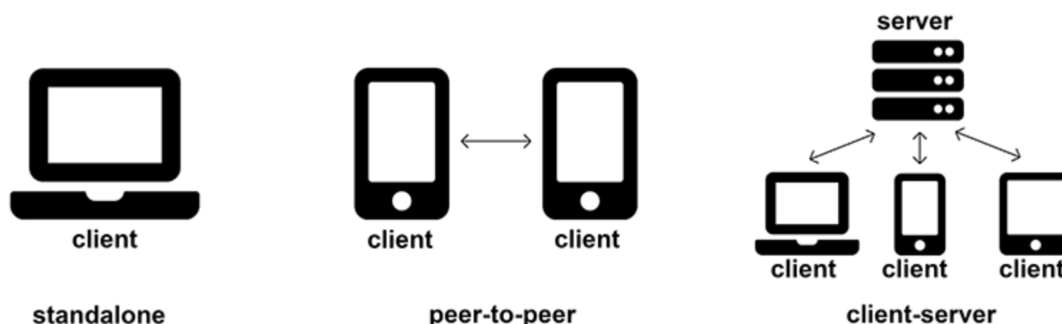
Za další významný milník lze považovat rok 2005, kdy Jesse James Garreta zavedl a popsal pojem AJAX [12], a konec srpna roku 2006, kdy byla vydána první stabilní verze javascriptové knihovny jQuery [13]. Jejich kombinace programátorům nesmírně zjednodušily vývoj frontendu a přispěly k zvýšení interaktivity webů.

4.2 Architektura aplikace

Multiplatformní aplikace lze z pohledu architektury rozdělit na 3 skupiny: standalone, klient-server a klient-klient (peer-to-peer). Standalone aplikace jsou takové aplikace, které sice mají implementace pro různé platformy, nicméně jsou na sobě nezávislé, tzn. nekomunikují mezi sebou. Klient-klient neboli peer-to-peer aplikace (také označované jako P2P) jsou takové aplikace, které jsou schopné komunikovat s jinými instancemi/instalacemi aplikace napřímo, například posíláním souboru z jednoho chytrého telefonu na jiný pomocí technologie Bluetooth.

Třetí skupinou jsou aplikace typu klient-server. Tyto aplikace jsou rozděleny na dvě části: klientskou aplikaci a serverovou službu. Klientská část je aplikace, se kterou pracuje uživatel – ať už je to webová stránka, mobilní nebo desktopová aplikace.

Serverová služba obvykle běží na vzdáleném serveru a obsluhuje požadavky, které přicházejí z klientských aplikací.



Obrázek 1 – Typy aplikace

Na obrázku 1 jsou znázorněny jednotlivé typy.

Tato práce se zabývá aplikacemi typu klient-server, a to hned z několika důvodů. V dnešní době se od moderních aplikací očekává, že přístup k datům, která pomocí nich ukládáme, budou přístupná online 24 hodin denně. Stejně tak se očekává, že přístup k těmto datům budeme mít bez ohledu na to, jakou platformu využijeme. To vše vede k potřebě tato data uchovávat centrálně, přičemž úložiště musí být přístupné online 24 hodině denně. Těmto požadavkům tedy nejlépe odpovídá architektura klient-server, kde se data budou ukládat na vzdáleném

serveru a klientská aplikace bude sloužit pouze jako nástroj pro modifikaci těchto dat.

4.3 Serverová část

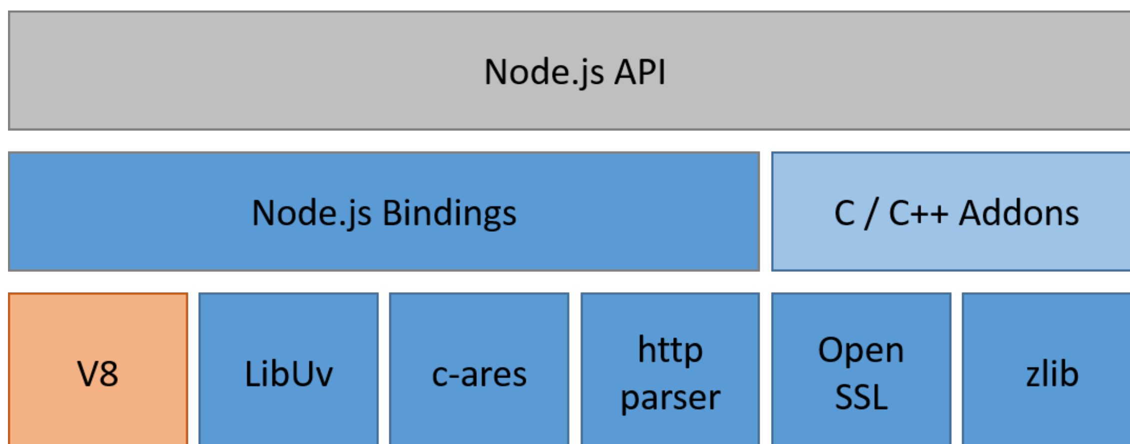
Díky použití architektury klient-server tak veškerý kód, který běží na straně serveru, je svým způsobem sdílený ve vztahu ke klientským aplikacím. Všechny klientské aplikace využívají stejný kód na serveru. Zároveň díky použití Javascriptu jak na straně serveru, tak i v klientské aplikaci lze sdílet i některé části kódu mezi serverovou části a klientskou aplikaci (typicky validace vstupních hodnot). Díky tomu se zjednodušuje údržba takového kódu, kdy validace se upravuje jen na jednom místě a nemusí se upravovat zvlášť pro backend a zvlášť pro frontend.

Pro účely této bakalářské práce bylo vzhledem k výběru jazyka vybráno běhové prostředí Node.js ve kterém běží javascriptový framework Express.

4.3.1 Node.js

Jak již zaznělo, Node.js je běhové prostředí pro spouštění javascriptových aplikací na straně serveru, které vyvinul v roce 2009 Ryan Dahl. Node.js spojuje Chrome V8, což je Javascriptový interpret, knihovnu libuv, která umožňuje asynchronní I/O operace a základní (core) knihovny Node.js, které zajišťují funkcionalitu na vyšší vrstvě.

Struktura komponent, ze kterých se skládá Node.js je zobrazena na následujícím obrázku:



Obrázek 2 – Struktura Node.js

Node.js používá událostmi-řízený, neblokující I/O model. Událostmi řízený model používá návrhový vzor Pozorovatel (Observer).

Ve chvíli, kdy se spouští aplikace, spolu s touto aplikací se spouští i událostní smyčka, ve které se zpracovává fronta události. Jakmile je vyvolána událost, vloží se její callback do fronty. Ve smyčce, která zpracovává jednotlivé události, dochází k tomu, že událost je přesunuta do zásobníku volání a posléze je zpracována. V případě I/O operací je taková operace delegována na knihovnu libuv, která po dokončení operace vloží událost zpět do fronty. Je nutné podotknout, že během provádění I/O operací, hlavní vlákno pokračuje ve zpracování události a nečeká na dokončení I/O operace.

Node.js prostřednictvím základních knihoven poskytuje velmi bohaté API pro práci s HTTP požadavky, práci se souborovým systémem, síťovou vrstvou, DNS aj.

Mimo těchto základních knihoven lze přidávat do Node.js také vlastní moduly. Pro správu těchto modulů slouží balíčkovací systém NPM. Dle údajů serveru Modulecounts [14] lze soudit, že v současnosti v systému NPM a tudíž pro Node.js existuje největší počet balíčku v porovnání s ostatními jazyky, a i vývoj počtu balíčků jednoznačně vykazuje stoupající tendenci.

4.3.2 Express

Tento framework byl vytvořen TJ Holowaychukem již v roce 2010 [15], tedy v době, kdy i samotná platforma Node.js podle údajů Google Trends [5] nebyla tolik známá. Později framework převzala společnost StrongLoop [16], která na bázi Express vytvořila vlastní framework. V roce 2015 pak společnost IBM koupila StrongLoop a je tak aktuálním vlastníkem frameworku [16].

Jak uvádí web projektu [6], Express je minimalistický framework vytvořený v Node.js. Tento framework poskytuje tenkou vrstvu pro zjednodušení práce především s obsluhou HTTP požadavků a šablonovacími systémy. Díky své jednoduchosti je součásti mnoha dalších frameworků.

Dá se říct, že Express vstupuje do životního cyklu aplikace hned po přijetí HTTP požadavku (zpracování samotného požadavku a vytvoření rozhraní pro

jednodušší přístup k datům požadavku a před odesláním odpovědi. V tomto kroku připravuje a vytváří odpověď.

Express hojně používá princip, kterému se v angličtině říká *middleware*. V souvislosti s frameworkem Express se jedná o přístup, kdy funkce dostane jako parametry objekt požadavku, objekt odpovědí a funkci, která je další v pořadí. Díky této vlastnosti lze snadno upravit odpověď vložením další funkce do tohoto seznamu. V praxi tak jednotlivé funkce obohacují či výslednou odpověď upravuji. Jedna funkce tak může provádět autorizaci a v případě neoprávněného přístupu již dále nevolat další funkce a rovnou odeslat odpověď například s kódem 403 a popisem chyby v těle odpovědi. Další funkce může v případě, že autorizace proběhla úspěšně, zaznamenat aktivitu uživatele do žurnálu události.

Na principu middleware řeší Express i směrování (anglicky *routing*) požadavků jednotlivým částem aplikace pro jejich následné zpracování. V zásadě se jedná o již zmíněné middleware, s tím rozdílem, že jsou vázané na objekt směrovače (routeru). Spolu s možností využití regulárních výrazů při definování směrování poskytuje framework velmi silný a zároveň jednoduchý nástroj pro zpracování požadavků. Po rozpoznání a vyhledání odpovídajícího směrování se provede volání funkce definované pro toto směrování. Uvnitř této funkce se zpracují data požadavku a funkce vrátí upravenou odpověď. Tato odpověď se buď vrátí zpět klientovi nebo se předá dalšímu odpovídajícímu směrování.

Další významnou části frameworku je možnost integrace šablonovacího systému. Ten umožňuje upravit odpověď tak, že předá předem nadefinované proměnné do šablony, která se odešle zpět jako odpověď.

4.3.3 Sails.js

Sails.js je Javascriptový framework, který byl vytvořen v roce 2012 Mikem McNeilem [17]. Sails.js vznikl jako spojení frameworku Express a knihovny pro web-sockety Socket.io. Díky tomu se Sails.js stal velmi mocným nástrojem, který poskytuje routování, sockety, práci s databázi, REST API a další funkcčnosti, bez nutnosti instalace dalších modulů.

Instalace Sails.js probíhá stejně jako instalace jiného balíčku pomocí balíčkovacího systému NPM.

```
npm -g install sails
```

Ukázka 1 – Instalace Sails.js

Po instalaci stačí již jenom vytvořit projekt:

```
sails new todo-app-project
```

Ukázka 2 – Vytvoření nového projektu v Sails.js

Sails.js umožňuje velmi jednoduše vygenerovat entitu pro použití pomocí REST API. Následujícím příkazem Sails.js vygeneruje dva soubory – model, pro nadefinování struktury entity a controller, ve kterém se definují akce (Action), které se vztahují k dané entitě.

```
sails generate api todoItem
```

Ukázka 3 – Vygenerování REST API entity v Sails.js

Nyní je hotová základní implementace serverové části pro jednoduchou ToDo aplikaci. Stačilo k tomu nadefinovat strukturu entity TodoItem a o zbytek se postará framework. To znamená vytvoření potřebných akcí pro REST API, ukládání do persistentního úložiště a další související činnosti.

V tuto chvíli na portu, který je nastaven v konfiguraci, běží aplikace a na URL /todoItem je dostupný výpis všech úkolů, které jsou uloženy v aplikaci. Pomocí HTTP metod (GET, POST, PUT, DELETE) se tak dají upravovat data, která jsou na serveru uložena.

4.4 Webová aplikace

Javascript byl v roce 1995 napsán Brendanem Eichem pro vytváření dynamických webů [18]. S postupem času se v tomto jazyce začaly vytvářet malé knihovny i velké frameworky, které usnadňovaly vývoj. Dokonce se začaly objevovat i různé nadstavby (například Dart od Google [19]), jejichž výsledný kód se kompiloval do Javascriptu.

4.4.1 React.js

S jednou takovou nadstavbou přišla i společnost Facebook, která v roce 2013 uvolnila React jako open-source projekt, který již přes dva roky vyvíjela, a v tu dobu nasadila na dva své největší projekty – Facebook a Instagram [10].

React přichází s novým přístupem k vývoji celého frontendu. Používá systém komponent: téměř vše je v Reactu komponenta. Komponentou tak může být například celá aplikace (App), která se skládá z dalších dílčích komponent, například Header, Menu, Content, Footer. Každá z těchto komponent může být tvořena dalšími komponentami nebo být samostatně vykreslitelná, tj. nese v sobě logiku toho, jak se bude vykreslovat na základě dat, která získá.

Tato data komponenty dostávají/získávají ze stavu aplikace. Toto je další významná vlastnost Reactu: data jsou řízena centrálně, tj. existuje jediný zdroj dat (source of truth). Komponenty mění stav a reagují na změny stavu. Ve chvíli, kdy se změní stav aplikace, React sestaví a vypočítá nejdříve virtuální DOM (Document Object Model) a tento virtuální DOM následně promítne do skutečného DOMu.

Virtuální DOM je abstraktní zjednodušená verze skutečného DOMu. Místo modifikace skutečného DOMu lze upravovat virtuální DOM a následně tyto změny uložit do skutečného DOMu. Díky tomu, že při ukládání se nejdříve vypočítají rozdíly a následně se aplikují co nejmenší možné změny, změna DOMu je pak mnohonásobně rychlejší, než kdyby se upravoval skutečný DOM.

Možnost implementace Reactu do již existující aplikace je velmi jednoduchá právě díky využiti komponentového systému. Lze tak nejdříve převést do Reactu jenom určitou část aplikace, například Kalendář. Zbytek aplikace může zůstat beze změny..

Vedlejším vlivem použití Reactu je i to, že kvůli tomu, že rozvržení a logika zobrazení jednotlivých komponent jsou uchovávány v samotných komponentách, tak jediné, co potřebuje předávat zvenku, jsou data. Nejsnazší cesta, jak tato data získat, je předat je hned při načtení aplikace. Nicméně vzhledem k tomu, že se data obvykle dynamicky mění, lze (a je to nejrozšířenější metoda) je získávat asynchronně pomocí AJAXu v nějaké strukturované podobě. Pro přenos dat se pak nejčastěji používá formát JSON (Javascript Object Notation).

Pro zjednodušení zápisu a přehlednost přichází , React s novým rozšířením syntaxe pro Javascript – JSX. Zápis v JSX se velmi podobá zápisu v HTML a i sami vývojáři Reactu doporučuji jeho použití pro elementy popisující to, jak má vypadat uživatelské rozhraní.

4.4.2 Vue.js

Framework Vue.js je velmi podobný frameworku React. Z porovnání, které je dostupné přímo na webu projektu [7], se lze dozvědět, že oba frameworky pracují s virtuálním DOMem a používají komponenty. Odlišnosti však jsou především v zápisu.

Na rozdíl od Reactu, kde se komponenty zapisují v syntaxi JSX a který se následně převádí do JavaScriptu, Vue.js se zapisuje přímo do HTML souboru. Podle mého názoru je díky tomu křivka učení mnohem strmější.

Vue.JS používá systém znovupoužitelných komponent. Na rozdíl od Reactu se však zapisují přímo do HTML a myslím si, že v tomto ohledu má mnohem čitelnější strom komponent než v Reactu, kde se každá komponenta definuje zvlášť a není tak vidět celý strom komponent.

4.5 Desktopová aplikace

I přestože se spousta aplikaci, včetně těch komplexnějších (například aplikace Microsoft Office), v dnešní době převádí (případně duplikuje) na web, desktopové aplikace jsou stále hojně využívané.

Jedním z důvodů je potřeba pracovat offline, mít přístup k prostředkům počítače, jako je úložiště nebo tiskárna, nebo i jednoduchost spuštění: stačí poklepat ikonu na ploše místo otevírání prohlížeče a zadávaní adresy aplikace.

Je nutno říci, že Javascript není průkopníkem v tomto přístupu, a již existují řešení, která umožňují spouštět webové aplikace jako nativní na desktopu, nicméně se tak dobře neuchytily. Jako jedním z mnoha lze uvést Bowline pro Ruby [20].

Jak jsem již uvedl na začátku kapitoly, velkou konkurenci pro desktopové aplikace začínají být webové jednostránkové aplikace (Single Page Application – SPA). Jejich výhody jsou zřejmé – není nutné cokoliv instalovat k sobě do počítače a obvykle jsou přístupné z jakéhokoliv počítače 24 hodin denně, 365 dnů v roce. Díky tomu, že aplikace není nainstalovaná lokálně, ale je přístupná jako webová stránka, se zjednodušuje proces aktualizace takovýchto systému, kdy většinou postačí jednoduché obnovení stránky, aby se načetla nová verze aplikace.

V kombinaci s cloudovým úložištěm, kde data jsou uložena na vzdáleném serveru, dochází i k velmi zajímavé integraci desktopové a webové jednostránkové

aplikaci. Uložená data pak lze upravovat jak z webové aplikace, tak i z desktopové, případně i z jiných aplikací a zařízení. Vhodným příkladem je Microsoft Office 365. Tato aplikace je dostupná v desktopové i v webové verzi, která sice disponuje méně funkcemi, ale postačí pro jednoduchou úpravu dokumentů. Pokud je potřeba využívat pokročilejší funkce, například makra, musí se použít desktopová verze.

Dalším příkladem je aplikace Spotify, jejíž desktopová verze na rozdíl od webové umožňuje ukládat hudbu pro poslech i bez připojení k Internetu. Avšak oproti Microsoft Office je její desktopová aplikace pouze zabalená webová verze, která má rozšířené funkce.

4.5.1 Electron

Nejpopulárnější javascriptový framework pro vývoj desktopových aplikací je dnes Electron. Konkuruje feameworku NW.js, který ale přesto, že byl vytvořen dříve (o 1,5 roku), má nižší počet „hvězdiček" na serveru GitHub, což se dá považovat za relevantní kritérium.

Electron je knihovna, která umožňuje vývoj desktopových aplikací s použitím HTML, JavaScriptu a CSS. Výslednou aplikaci pak lze zkompilovat jako nativní aplikaci pro Windows, Linux nebo Mac.

Základem Electronu je Chromium (prohlížeč od Google s otevřeným kódem), Node.js a knihovny pro podporu nativních funkcí jednotlivých platforem. Electron je kombinuje do stejného kontextu, což umožňuje volat jak metody Node.js API, tak i metody Chromium API.

Obecně Electron funguje tak, že zabalí již hotovou javascriptovou aplikaci do balíčku spolu s prohlížečem. Při otevření aplikace se nejdříve otevře prohlížeč s otevřenou stránkou, kterou je vlastně webová aplikace.

Electron rozlišuje dva typy procesů: hlavní (main), ve kterém je spuštěn prohlížeč, a vykreslovací (renderer) procesy, které vykreslují jednotlivá okna aplikace. Jednotlivá okna se přitom dají přirovnat k jednotlivým záložkám v prohlížeči. Při zavření okna dochází i k ukončení jeho vykreslovacího procesu.

4.6 Mobilní aplikace

Vzhledem k tomu, že jednotlivé platformy mohou obsahovat části kódu, který je specifický pro tu konkrétní platformu, je dobré pro jednotlivé platformy vytvářet vlastní balíčky (např. todo-app-osx, todo-app-win, apod.), které jako závislost mají hlavní aplikaci; v případě aplikace v této práci je to balíček todo-app-web. Vlastní balíčky poslouží jako obálka, která rozšiřuje funkčnost aplikace.

4.6.1 Cordova

Cordova je platforma pro vývoj nativních mobilních aplikací v JS, HTML a CSS pro platformy jako jsou Android, IOS, Windows Phone a některé další.

Cordova vznikla koncem roku 2011, kdy společnost Adobe se rozhodla pro uvolnění PhoneGap jako open source [9]. Nyní je Cordova vyvíjená pod záštitou organizace The Apache Software Foundation [9].

Platforma ve výchozím stavu neposkytuje rozhraní pro zprostředkování funkcionality jednotlivých platforem. Pro jejich použití však lze nainstalovat moduly pro jednotlivé zdroje. Jednotlivé moduly jsou knihovny, které jsou vyvíjeny zvlášť, což umožňuje mnohem flexibilnější správu jednotlivých modulů. Například v případě, že vyjde nová verze modulu pro kameru telefonu, stačí aktualizovat jednu knihovnu místo aktualizace celé platformy.

Repozitář modulů pro Cordovu aktuálně obsahuje téměř 2000 modulů [9].

4.6.2 React Native

React Native vznikl během interního hackatonu Facebooku již v roce 2013 [21]. Dva roky na to (začátkem roku 2015) byl představen na konferenci React.js Conf a krátce poté Facebook na F8 oznámil [21] jeho uvolnění jako open-source na serveru GitHub.

Přesto že je React Native poměrně nový framework, jeho vývoj je velmi dynamický. Po dvou letech vývoje jeho repozitář na GitHubu obsahuje více než 9000 commitů, 148 releasů a v počtu hvězdiček je na celkovém 14. místě.

React Native se oproti Cordově zásadně liší. Cordova zabalí webovou stránku do nativní aplikace, která tuto stránku zobrazuje. React Native používá vlastní jazyk, který se podobá JSX. Výsledný kód se pak kompiluje přímo do nativního jazyka

platformy, pro kterou se aplikace kompiluje. Díky tomuto přístupu lze v aplikaci používat přímo nativní jazyk pro jednotlivé platformy. Tato možnost se hodí především v případech, kdy je kladen velký důraz na rychlost aplikace.

Aktuálně React Native podporuje platformu Android a iOS. Nicméně existují knihovny, které umožňují použití React Native například na platformě Windows, na webu a dalších platformách.

5 Ukázka vývoje multiplatformní aplikace

V této kapitole popisuji vývoj multiplatformní aplikace – úkolníčku. Jedná se o jednoduchou aplikaci, jejíž hlavní úlohou je ukládat záznamy (úkoly) zadané uživatelem a měnit jejich stav.

Z implementačního hlediska jde o jednu aplikaci rozdělenou na čtyři části: jednu serverovou a tři klientské. Na serverové části je použit framework Express.js, na webové React.js, pro zabalení webové aplikace do nativní desktopové aplikace je zvolen Electron a pro zabalení do mobilní aplikace – Cordova.

5.1 Architektura ukázkové aplikace

Ukázková aplikace bude obsahovat 4 části: serverovou část a 3 klientské aplikace (webovou, mobilní a desktopovou).

Serverová část slouží především jako centrální úložiště, pomocí kterého jednotlivé klientské aplikace komunikují mezi sebou. Má tedy i další úlohu, a to synchronizaci dat napříč všemi zařízeními. Na serveru běží prostředí Node.js, které bude pohánět hlavní aplikaci, která používá javascriptový framework Express.js. Serverová aplikace bude komunikovat s klientskými aplikacemi pomocí REST API architektury.

Klientská aplikace pro web je vytvořena pomocí knihovny pro vývoj frontendu React. Pomocí frameworku Cordova je webová aplikace zabalená do nativní aplikace pro jednotlivé mobilní operační systémy jako Android, iOS a Microsoft. Cordova pracuje na principu, že existující webovou aplikaci zabalí do nativní aplikace, a ta pomocí WebView vykresluje webovou aplikaci uvnitř té nativní. Zároveň poskytuje obecné rozhraní pro využití většiny funkcí mobilních telefonů, které poskytuji jednotlivé platformy. Pro účely této práce je vytvořená pouze aplikace pro Android.

Framework Electron pracuje na podobném principu jako Cordova. Existující webovou aplikaci zabalí do nativní aplikace pro Linux, macOS či Windows, která používá Chromium a Node.js pro běh aplikace. Stejně jako Cordova, poskytuje rozhraní pro komunikaci s operačním systémem.

5.2 Vytvoření serverové části

Serverová část je společná pro všechny klientské aplikace a je hlavním zdrojem, který zpracovává data z klientských aplikací. Je to dáno především tím, že platformy, na nichž běží klientské aplikace, mohou být velmi odlišné z pohledu hardwaru.

Klientské aplikace komunikují se serverovou části pomocí definovaného rozhraní a komunikace probíhá s použitím architektury REST API.

Díky tomuto způsobu komunikace lze kdykoliv nahradit implementaci serverové části za jinou, která bude poskytovat stejné komunikační rozhraní.

5.2.1 Příprava prostředí

Pro ukázkovou aplikaci jsem zvolil server s operačním systémem Debian, na kterém je nainstalován framework Node.js. Samotná instalace je velmi jednoduchá díky balíčkovacímu systému apt-get. Pro instalaci je tedy nutno spustit příkaz:

```
apt-get install nodejs
```

Ukázka 4 – Instalace Node.js

Pro instalací balíčkovacího systému pro Node.js pak stačí spustit příkaz:

```
apt-get install npm
```

Ukázka5 – Instalace balíčkovacího systému NPM

Vzhledem k tomu, že Node.js má vestavenou podporu pro komunikaci pomocí protokolu HTTP i HTTPS, není potřeba žádný další software, pomocí kterého bude aplikace komunikovat ven.

Implementace těchto protokolů je v Node.js řešená velmi nízkoúrovňově, a při vývoji webových aplikací je práce s nimi velmi nepohodlná. Z tohoto důvodu jsem se rozhodl použít framework Express.js, který v mnohém zjednodušuje práci s obsluhou HTTP požadavků i odpovědí, a to například při parsování požadavků a sestavení odpovědí (včetně práce s Cookie), routování požadavků a poskytuje rozhraní pro šablonovací systémy.

5.2.2 Struktura aplikace

Klientské aplikace komunikují s serverovou pomocí REST API. Aplikace obsluhuje pět požadavků:

- Přidat (/api/v1/add) – požadavek typu POST
- Upravit (/api/v1/edit/{id}) – požadavek typu PUT
- Smazat (/api/v1/delete/{id}) – požadavek typu GET
- Seznam (/api/v1/list) – požadavek typu GET
- Detail (/api/v1/detail/{id}) – požadavek typu GET

Vzhledem k malé velikosti aplikace budou jednotlivé požadavky definovány přímo v hlavním souboru aplikace (app.js). Zpracování jednoho požadavku pak vypadá zhruba takto:

```
app.get('/detail/:id', function (req, res,
next) {
    const id = req.params.id;
    let task = tasks.filter(function (item)
{
        return item.id === id;
    }).shift();

    res.send({result: 'OK', data:
task.toJson()});
    return next();
});
```

Ukázka 6 – Zpracování požadavku s použitím Express.js

Toto volání zaregistruje pro požadavek typu GET na adrese /detail/:id (první parametr) obslužnou funkci, která se předává jako druhý parametr. Obslužná funkce dostane tři argumenty: req (požadavek – request), res (odpověď – response) a next (další). Parametry req a res jsou objekty, které obsahují data požadavku a odpovědi. Parametr next je funkce, která se má zavolat jako další v pořadí.

5.2.3 Úložiště

Ukázková aplikace používá in-memory úložiště, které není perzistentní. Tento typ úložiště je zvolen především pro zjednodušení aplikace, jelikož hlavním záměrem této aplikace je vytvoření aplikace pro spuštění na různých platformách.

5.2.4 Model aplikace

Model aplikace je tvořen pouze jednou entitou – Task. Tato entita má několik atributů:

- ID: unikátní identifikátor úkolu
- Title: název úkolu
- Desctription: popis úkolu
- Created: časové razítko, kdy byl úkol vytvořen
- Done: časové razítko, kdy úkol byl splněn

```
exports = module.exports = {};
let moment = require('moment');
const guid = require('./guid.js');

class Task {
    constructor(title, description) {
        this.id = guid.guid();
        this.title = title;
        this.description = description;
        this.createdAt = moment();
        this.doneAt = null;
    }

    markAsDone() {
        this.doneAt = moment();
    }

    toJson() {
        return {
            id: this.id,
            title: this.title,
            description: this.description,
            createdAt:
this.createdAt.format(),
            doneAt: this.doneAt !== null ?
this.doneAt.format() : null
        };
    }
}
exports.Task = Task;
```

Ukázka 7 – Model entity Task

Pro generování ID se používá funkce pro generování náhodného čísla „guid()“, která vytváří řetězec, který se podobá GUID v4.

```
var exports = module.exports = {};
exports.guid = function () {
    function s4() {
        return Math.floor((1 +
Math.random()) *
0x10000).toString(16).substring(1);
    }
    return s4() + s4() + '-' + s4() + '-' +
s4() + '-' + s4() + '-' + s4() + s4() +
s4();
}
```

Ukázka 8 – Funkce pro generování GUID v4

5.2.5 Závislosti aplikace

Jak již částečně vyplývá z kódu, serverová část je závislá na několika externích knihovnách.

Knihovna *express* je hlavní knihovnou frameworku Express.js, který zjednodušuje práci s obsluhou HTTP požadavků.

Knihovna *body-parser* je také součásti frameworku Express.js a je v přímé závislosti na jeho hlavní knihovně. Jeho úlohou je parsování obsahu příchozích HTTP požadavků. V praxi to znamená, že zpracuje požadavek a převede jeho obsah do lépe zpracovatelné podoby. Takto zmodifikovaný obsah je pak přístupný v proměnné *req.body*.

Knihovna *moment* poskytuje abstrakci pro práci s datem časem.

Všechny závislosti jsou definovány v souboru *package.json* v sekci *dependencies*. V tomto souboru je zároveň definován název aplikace, verze a připravený příkaz pro spuštění aplikace. Díky tomuto příkazu lze snadno spustit aplikaci pomocí *npm start*.

```
{
  "name": "todoapp",
  "version": "0.0.1",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "~4.15.2",
    "body-parser": "~1.17.1",
    "moment": "latest"
  }
}
```

Ukázka 9 - Soubor package.json

5.3 Vytvoření klientské aplikace pro webový prohlížeč

Webová klientská aplikace je vytvořena pomocí knihovny React.js, která umožňuje vytvářet aplikace s použitím komponentového systému. Webová aplikace je z pohledu architektury MVC tvořena především pohledovou části (View) a tenkou vrstvou řadičů (Controller), které jsou zodpovědné za komunikaci se serverou částí.

Zvláštnosti webové aplikace je i to, že právě ona slouží jako základ pro mobilní a desktopovou aplikaci.

5.3.1 Struktura aplikace React.js

Klientská aplikace pro webové prohlížeče používá framework React.js. Celá aplikace je pouze zobrazovací vrstva, která zobrazuje data získaná přímo ze serveru. Jako jediná z klientských aplikaci nemá modelovou vrstvu. Aplikace nepracuje offline, což vychází ze samotné povahy webových aplikaci, kdy je nutné připojení k internetu, aby se aplikace vůbec načetla. V případě výpadku připojení je uživatel o této skutečnosti informován a po celou tuto dobu není možné provádět žádné změny.

5.3.2 Příprava prostředí

Prostředí pro vývoj aplikace vyžaduje stejně jako serverová část instalaci Node.js a balíčkovacího systému NPM. Ukázková aplikace používá několik knihoven, které jsou vyžadovány jako závislosti v hlavním souboru *package.json*.

Primárními závislostmi jsou knihovny React a ReactDOM. To jsou základní knihovny, které tvoří zobrazovací vrstvu. Patří k nim i knihovna Axios, která zajišťuje komunikaci se serverovou části pomocí technologie AJAX.

Pro přidání podpory JSX syntaxe se používá knihovna Babel, respektive její rozšíření pro React. Babel je kompilátorem JavaScriptu, který kompiluje různé verze JavaScriptu a různé nadstavbové jazyky do čistého JavaScriptu (dle specifikace ECMAScript 5). Výsledkem kompilace je kód v JavaScriptu, který je podporován ve většině prohlížečů a běhových prostředí.

Další podpůrnou knihovnou je Webpack, s jehož pomocí lze řídit automatické sestavování (build) aplikace. V ukázkové aplikaci spočívá úloha této knihovny v zkompilování JSX syntaxe do ECMAScript 2015 a přesun výsledného souboru spolu se souborem *style.css* do složky *public*, která je přístupná pro prohlížeče.

Pro běh této aplikace na produkčním serveru tedy postačí soubor index.html a složka public.

5.3.3 Komponenty

Aplikace je složena z několika komponent:

- App – hlavní komponenta, která zanořuje do sebe všechny ostatní
- List – komponenta pro zobrazení seznamu úkolů
- Row – komponenta, která zobrazuje detail úkolu při zobrazení v seznamu úkolů
- Detail – zobrazuje kompletní informace o úkolu při jeho rozkliknutí
- ExtendedFormDetail – zobrazuje formulář pro vytvoření úkolu. Kromě pole Název obsahuje i pole Popis.

Hlavní komponenta obsahuje veškerá data aplikace – její stav. Stav komponenty může měnit pouze sama komponenta, díky čemuž lze předejít nežádoucím vedlejším efektům. Pokud jiná komponenta potřebuje tento stav ovlivnit, je nutno předat odpovídající zpětné volání (callback) z hlavní komponenty až k této komponentě. Díky tomuto principu vnořená komponenta nemůže ovlivnit nadřazenou komponentu jinak než přes definované callbacky.

Komponenta *List* slouží jen jako kontejner pro jednotlivé řádky výpisu. Řádky výpisu jsou předávány z hlavní komponenty. Komponenta je zároveň tunelem pro funkce *handleDelete()* a *handleDetail()*, které nepoužívá přímo, ale jen je předává komponentě *Row*.

Row komponenta vykresluje název úkolu a tlačítka pro zobrazení detailu a smazání jednotlivých úkolů.

Komponenta Detail zobrazuje kompletní informace o úkolu, tj. název, popis, datum vytvoření, datum označení úkolu jako hotového, případně tlačítko pro označení úkolu jako hotového.

Komponenta ExtendedFormDetail se zobrazuje při kliknutí na tlačítko ‚++' v hlavním okně aplikace. Tato komponenta obsahuje pole pro vložení Názvu úkolu a pole pro zadávání popisu úkolu. Do pole Název se automaticky nastavuje hodnota zadaná v hlavním okně aplikace pro zjednodušení vytváření úkolu.

5.3.4 Propojení se serverovou části

Propojení se serverovou části řeší knihovnou Axios. Komunikaci se serverem lze řešit i pomocí Fetch API, nicméně tato funkčnost je poměrně nová a není podporována napříč všemi prohlížeči. Co se týče podpory ze strany prohlížečů, je Axios na tom mnohem lépe.

Axios podporuje běžně používané metody požadavků (request methods), umožňuje paralelní zasílání několika požadavků najednou a jejich zpracování až po obdržení všech odpovědí a další funkce, které pro jednoduchost projektu v něm nebyly použity.

5.3.5 Spuštění aplikace

Zkompilovaná aplikace se spouští jako obyčejná webová stránka otevřením souboru index.html v prohlížeči.

Pro zkompilování webové aplikace je nutno použít prostředí s nainstalovaným frameworkem Node.js, ve kterém se nainstalují potřebné závislosti aplikace příkazem *npm install* (musí se spouštět z adresáře *web*) a zkompilovat aplikaci příkazem:

```
./node_modules/.bin/webpack -p
```

Ukázka 10 – Kompilace aplikace

Výsledkem zkompilování budou dva nové soubory *src/client/public/bundle.js* a *src/client/public/style.css*. Pro běh aplikace je potřeba soubor *src/client/index.html* a celá složka *src/client/public*.

5.4 Vytvoření desktopové aplikace

Desktopová aplikace je vytvořena zabalením webové aplikace do nativní desktopové aplikace pomocí nástroje Electron. Ve výsledku je taková aplikace tvořena z prohlížeče Chrome a dalších podpůrných knihoven, které poskytují rozhraní pro přístup k funkcím operačního systému přímo z aplikace.

5.4.1 Příprava prostředí

Pro vytvoření desktopové aplikace je použita knihovna Electron, která zabalí existující aplikaci do nativní aplikace pro desktopy.

Elektron je dostupný jako knihovna v balíčkovacím systému NPM a tedy pro instalaci knihovny stačí použít příkaz:

```
npm install electron --save-dev
```

Ukázka 11 – Instalace knihovny Electron

Dalším krokem je vytvoření souboru *package.json*, ve kterém se definuje název, verze a vstupní soubor aplikace. V tomto souboru se také definují parametry pro knihovnu *electron-builder*, a to pro které operační systémy se má vytvořit spustitelná aplikace.

5.4.2 Struktura aplikace

Soubor *main.js* je podle definice v *package.json* hlavním vstupním souborem celé aplikace. Právě v tomto souboru se vytváří hlavní okno aplikace. Okna aplikace se vytváří vytvořením instance třídy *BrowserWindow*, do kterého lze načíst obsah souboru *index.html* webové aplikace. V nativním okně operačního systému se tak otevře okno prohlížeče a zobrazí se webová stránka.

Webovou aplikaci proto zkopírujeme do složky *app* a v *main.js* nadefinujeme, že se do hlavního okna aplikace načte soubor *app/index.html*.

Pro použití nativních notifikací operačního systému nutno upravit soubor *platform-specific/events.js*. Pro zobrazení notifikací je použito rozhraní HTML5 Notification API. Místo zavolání metody *alert()* se vytvoří nová instance třídy *Notification*, která okamžitě zobrazí nativní notifikaci.

HTML5 Notification API lze použít i v případě webové aplikace. V tomto případě jsou však různé způsoby zobrazení notifikací použity záměrně, abych poukázal na to, jak se dá vypořádat s rozšířením funkcionality pro jednotlivé platformy.

5.4.3 Sestavení aplikace

Pro sestavení aplikace se použije knihovna *electron-builder*, která se postará o zkompilování aplikace. Ukázková aplikace bude zabalena pro spuštění v operačním systému Windows. Do *package.json* je tedy nutno přidat sekci *build*:

```
"build": {
  "appId": "todoapp",
  "win": {
    "target": "portable"
  },
  "linux": {
    "target": "AppImage"
  }
},
```

Ukázka 12 – Sekce build souboru package.json

Po spuštění příkazu *node_modules\.bin\electron-builder* se ve složce *dist* vytvoří spustitelný soubor aplikace.

5.5 Vytvoření mobilní aplikace

Mobilní aplikace stejně jako desktopová je webovou aplikací zabalenou do nativní aplikace pro mobilní telefony. Pracuje na stejném principu s tím rozdílem, že místo prohlížeče používá komponentu WebViews.

5.5.1 Příprava prostředí

Pro vytvoření mobilní aplikace je použit framework Cordova. Ačkoliv ukázková aplikace je vytvořena pro mobilní telefony s operačním systémem Android, Cordova podporuje i další platformy.

Samotný framework je dostupný jako balíček v balíčkovacím systému NPM a lze jej nainstalovat pomocí příkazu *npm install cordova*.

Další nezbytnou součástí je prostředí Java Development Kit. Prostředí lze stáhnout na oficiálních stránkách Oracle.

K sestavení aplikace je také nutno nainstalovat Android SDK, který je dostupný buď jako součást Android Studio nebo samostatně. Vzhledem k tomu, že pro účely této ukázkové práce je celé Android Studio zbytečné, stačí nainstalovat Android SDK Tools, který poskytuje nutnou funkcionalitu pro sestavení aplikace. Pro správné fungování nástroje je nutno přijmout licenční podmínky. Toho lze dosáhnout spuštěním příkazu */tools/bin/sdkmanager --licenses* ve složce, kde je nainstalován Android SDK Tools. Pro spuštění je také nutno stáhnout pomocí programu sdkmanager nástroj pro emulaci zařízení, který umožní testování aplikace pro Android na PC bez nutnosti instalovat aplikaci do mobilního telefonu, a obraz samotné platformy Android. Instalace emulátoru se spouští příkazem */tools/bin/sdkmanager emulator*. A instalace obrazu platformy */tools/bin/sdkmanager "system-images; android-23; google_apis; x86"*.

Pro sestavení aplikace Cordova používá automatizační nástroj Gradle, který je také nutno nainstalovat.

Posledním krokem je vytvoření virtuálního zařízení, které emuluje mobilní telefon s operačním systémem Android. Toto zařízení bude mít název *test_device* a bude používat dříve stažený obraz *system-images;android-23;google_apis;x86*.

5.5.2 Struktura aplikace

Podobně jako u aplikace pro desktopy, struktura zůstává stejná. Stačí jen nakopírovat aplikaci do složky *www* v projektu.

5.5.3 Sestavení aplikace

Pro spuštění aplikace pro testování lze využít emulátoru zařízení s operačním systémem Android. Emulátor se spuštěnou aplikaci lze spustit příkazem *cordova run android*. Aplikaci tak lze otestovat bez nutnosti ji zveřejňovat v Google Play nebo připojování mobilního telefonu s OS Android.

Pro sestavení aplikace se pak použije velmi podobný příkaz *cordova build android*, který vytvoří soubor s názvem *app-debug.apk* ve složce *\platforms\android\app\build\outputs\apk\debug*. Tento soubor lze nainstalovat buď přímo do telefonu s operačním systémem Android nebo digitálně podepsat a nahrát na Google Play.

6 Ukázka transformace na multiplatformní aplikaci

V předchozí kapitole jsem popsal, jak lze vytvořit novou aplikaci, kterou lze použít na více platformách. Často se však stává, že není možné zahodit existující aplikaci a vytvořit novou, multiplatformní, na zelené louce. Jeden z důvodů je například ten, že při kompletním přepsání aplikace často dochází k tomu, že čas, který lze věnovat vytváření nových funkcí aplikace či jejich vylepšení, se tráví jejich znovunapsáním a často to končí tak, že ty funkce nejsou tak odladěné a fungují jinak, než je uživatel zvyklý. Pravděpodobně nejznámějším příkladem takového nepovedeného přepisu je Netscape Navigator [22].

Existující aplikaci lze z pohledu její transformace na multiplatformní zařadit do jedné z několika skupin. První skupinu bych pojmenoval jako Nativní aplikace, které nevyužívají architektury klient-server. Příkladem takové aplikace může být jednoduchý pokladní systém. Takový systém pak lze provozovat jak na desktopu, tak i na tabletu. V tomto případě se nejčastěji bude jednat o kompletní přepis aplikace. Záleží však na konkrétních případech. V případě, že aplikaci bude nutno provozovat pouze na desktopu a na tabletu, pak by volba nejspíše padla na nativní aplikaci v Java, která bude spustitelná jak na desktopu, tak i v tabletu. Pokud by však bylo požadováno i spuštění v prohlížeči, musela by se tato aplikace upravit na aplikaci typu klient-server a v tomto případě bych rozhodně sáhl po JavaScriptu.

Další skupinou jsou aplikace, které sice jsou vytvořeny jen pro jednu platformu, ale jsou navržené jako aplikace typu klient-server. V tomto případě hodně záleží na způsobu komunikace mezi klientskou aplikaci a serverem. Limitujícím faktorem může být například proprietární protokol, který znemožňuje jednoduše vyměnit klientskou aplikaci, nebo například i chybějící specifikace komunikačního rozhraní na straně serveru. V opačném případě lze vytvořit nové klientské aplikace, které mohou buď nahradit existující klientskou aplikaci nebo je doplnit tak, že se aplikace vytvoří jen pro ty platformy, pro něž aplikace chybí.

Do třetí skupiny řadím webové aplikace, které jsou ze své podstaty aplikacemi typu klient-server. Na aplikační úrovni se dají rozdělit dle typu generování výsledného kódu na dva typy – generování na straně klienta a generování na straně serveru. Při generování výsledného kódu na straně serveru

dochází k tomu, že výsledný kód se vrátí již jako součást odpovědí (obvykle jako HTML kód). Tudíž o tom, jaká bude podoba odpovědí se rozhoduje již během zpracování požadavku od prohlížeče. Aplikace s generováním výsledného kódu na straně klienta fungují zcela odlišně. Při prvním otevření se spustí aplikace, která odpovídá za generování výsledného kódu (tedy i za to, jak bude stránka vypadat) a na základě vstupních dat, ať už výchozích nebo od uživatele, generuje výsledný kód. Při další komunikaci se serverem obvykle očekává jen čistá data, která zahrne podle potřeby do výsledného kódu . Ve druhém případě se jedná o aplikaci, která s největší pravděpodobnosti je již připravena pro zabalení do nativní aplikaci jiných platforem. V prvním případě bude nutno existující aplikaci upravit tak, aby bylo možné komunikovat se serverem pomocí nějakého definovaného rozhraní a následně vytvořit klientskou aplikaci, která může, ale nemusí vycházet z původního kódu.

6.1 Popis aplikace

Jedná se o aplikaci CERSTOR, jejíž úkolem je automatizované zpracování atestů k hutním materiálům od různých dodavatelů a zasílání těchto atestů zákazníkům na základě informací obsažených v atestech.

Aplikace je napsána v jazyce PHP s použitím architektury MVC. Z pohledu typu aplikace, jak popisuji výše, ji lze zařadit do třetí skupiny, konkrétně jde o aplikaci s generováním výsledného kódu na straně serveru.

6.2 Analýza přechodu

Aplikace již na několika místech používá technologii AJAX. V některých případech odpověď serveru obsahuje čistá strukturovaná data, někdy celý kus šablony, který se má vykreslit. Částečně tak aplikace používá nějaké komunikační rozhraní, které však není sjednoceno.

Pro zachování kompatibility s existujícím stavem aplikace a zajištění plynulého přechodu na použití multiplatformní aplikace bude nutno vytvořit nové komunikační rozhraní, které bude fungovat zároveň s již existujícím. Díky použití architektury MVC se toto rozšíření dá vyřešit pouhým přidáním další sady řadičů, které budou použity pouze pro tuto komunikaci.

Vzhledem k tomu, že řadiče částečně obsahuji i logiku aplikace, která nespadá pod jejich odpovědnost, bude nutno tyto části zrefaktorovat a vyčlenit do samostatných služeb, aby nedošlo k duplikování kódu a porušení principu DRY.

Pro zajištění spuštění obou verzi aplikace bude nutno také vytvořit nový vstupní bod aplikace určený pro načtení multiplatformní verze. Toto bude zajištěno vytvořením dalšího řadiče, který bude vracet inicializační část multiplatformní verze – tedy načtení základního HTML kódu s načtením JavaScriptové klientské aplikace.

6.3 Struktura serverové části aplikace

Aplikace má z pohledu uživatele několik případu užití:

- Ruční nahrání atestu
- Vyhledání atestu
- Prohlížení atestu
- Odeslání atestu emailem
- Úprava údajů o atestu
- Stažení atestu
- Správa emailové schránky
- Uložení atestu z emailu

Pro každý z těchto případů užití je nutno vytvořit nejméně jeden vstupní bod aplikace, který bude definován jednotným identifikátorem zdroje (URI) a parametry, které tento vstupní bod vyžaduje.

Vstupní bod aplikace pro vyhledávání atestu bude dostupný na URI *`/api/v1/certificate-search`* a vyhledávací parametry se budou předávat metodou POST. Struktura předávaných vyhledávacích parametrů bude vypadat následovně:

```
{
 "$schema": "http://json-schema.org/draft-
06/schema#",
 "title": "SearchParameters",
 "type": "object",
 "properties": {
  "criteria": {
   "title": "Vyhledávací kritéria",
   "type": "object",
   "properties": {
    "id": {
     "description": "identifikátor atestu",
     "type": "integer"
    },
    "date": {
     "description": "datum atestu (yyyy-mm-
dd)",
     "type": "string"
    },
    "manufacturer": {
     "description": "výrobce",
     "type": "string"
    },
    "heatnumber": {
     "description": "tavba",
     "type": "string"
    },
    "grade": {
     "description": "jakost",
     "type": "string"
    }
   }
  },
  "page": {
   "description": "stránka pro zobrazení",
   "type": "integer"
  },
  "limit": {
   "description": "počet záznamů na
stránce",
   "type": "integer"
  }
 }
}
```

Ukázka 13 – Struktura požadavku pro vyhledání atestu

Výsledný požadavek pro vyhledání atestu od výrobce Marcegaglia s tavbou AG15633, který vypíše prvních 30 výsledků by mohl vypadat takto:

```
{
 "criteria": {
  "manufacturer": "Marcegaglia",
  "heatnumber": "AG15633"
 },
 "page": 1,
 "limit": 30
}
```
Ukázka 14 – požadavek na vyhledání atestu

Zvláštním typem případu užití je ruční nahrání atestu. Vzhledem k tomu, že spolu s metadaty atestu je nutno nahrát soubor, musí se tento požadavek rozdělit na dva. Je to dáno samotným protokolem HTTP, který neumožňuje použití násobných hlaviček Content-Type.

První požadavek odešle na server soubor s nastaveným Content-Type na multipart/form-data. Na serveru se soubor uloží a v odpovědi vrátí vygenerovaný token, který jednoznačně identifikuje nahraný soubor.

Klientská aplikace následně odešle další požadavek, který kromě metadat atestu bude obsahovat i token, který se vrátil v odpovědi po nahrání souboru.

6.4 Struktura klientské aplikace

Strom základních komponent klientské aplikace:

```
<App>
  <Header>
       <Menu/>
  </Header>
  <Content/>
  <Footer/>
</App>
```
Ukázka 15 – Strom základních komponent

Obsah komponenty Content se mění dle aktuálního případu užití, který lze aktivovat pomocí navigace v komponentě Menu.

6.5 Odlišnosti aplikace pro různé platformy

Z návrhu aplikace pro jednotlivé platformy obvykle vyplývají odlišnosti aplikace pro jednotlivé platformy, s kterými je nutno počítat při implementaci systému.

V případě aplikace CERSTOR se jedná o několik odlišnosti pro různé platformy. Všechny tři platformy budou používat vlastní notifikace. Desktopová aplikace bude umět navíc nahrávat atesty přímo ze skeneru bez nutnosti mezikroku, kdy uživatel musí dokument nejdříve oskenovat a pak jej nahrát. Společně s mobilní aplikací bude desktopová také umět ukládat vybrané atesty pro offline prohlížení. V offline režimu se na těchto platformách bude zobrazovat pouze seznam atestů dostupných ke stažení a bude dostupná možnost nahrávání atestů do aplikace pro jejích následné nahrání na centrální server, jakmile bude aplikace opět online.

Kvůli těmto odlišnostem je nutno aplikaci rozdělit na dvě části – společná část a část, která se u jednotlivých platforem liší, případně přebývá či chybí. Pro přehlednost se tyto dvě části vyčlení do samostatných složek – *common* a *specific*. Obsah složky *common* bude pro všechny platformy stejný, kdežto obsah složky specific se bude lišit.

Vzhledem k tomu, že každá aplikace pro jednotlivé platformy bude řešit notifikace jinak – konkrétně za pomocí nativních notifikací platforem – dává smysl vytvořit v aplikaci vlastní službu, která bude přeposílat notifikace jednotlivým implementacím. Tato služba načte soubor *notification-sender.js* ze složky *specific* a přepošle nově vytvořené instanci třídy, která implementuje rozhraní *NotificationSender*. Nutno podotknout, že čistý JavaScript neobsahuje možnost definování rozhraní, proto slovním spojením „implementuje rozhraní" je myšlena implementace fiktivního rozhraní, které definuje metody této třídy.

Další odlišnosti pro desktopovou aplikaci je nahrávání atestu přímo ze skeneru prostřednictvím knihovny Dynamic Web TWAIN. Tato odlišnost se řeší podobně jako notifikace a to tak, že pro vytvoření komponenty s formulářem pro přidávání atestu se použije speciální třída, která bude zodpovědná za vytvoření této komponenty a na základě toho, zda ve složce *specific* existuje soubor *certificate-add-form-factory.js*, tento soubor načte a přidá do formuláře potřebnou funkčnost.

Pro úplnost zmíním fakt, že podobná funkčnost s možností použití fotoaparátu na mobilním zařízení je již implementována v prohlížeči a tedy při

výběru souboru pro nahrání, nabízí možnost výběru nově pořízené fotografie. Odpadá tak nutnost tuto funkčnost znovu implementovat.

Možnost prohlížení atestů v režimu offline se musí implementovat na dvou místech. První je komponenta pro zobrazování tlačítek akcí nad atestem. V případě, že se bude jednat o desktopovou nebo mobilní aplikaci, bude nutno rozšířit komponentu o nové tlačítko, kterým se atest stáhne do offline úložiště a označí ho v aplikaci jako přístupný v režimu offline. Zde je nutno podotknout fakt, že tato akce probíhá pouze na straně klientské aplikace a seznam uložených atestů musí existovat lokálně v zařízení. Další komponenta, kterou je nutno rozšířit, je komponenta Content, která rozhoduje o tom, která část aplikace se má aktuálně vykreslit. Ještě před tímto rozhodnutím se zkontroluje, zda existuje soubor `content-manager.js` a pokud ano, provede kód v tomto souboru. Tento kód zkontroluje, zda je zařízení online, a pokud není, zobrazí seznam atestů s možnosti jejich stažení.

6.6 Zabezpečení

Zabezpečení aplikace se nemění a zajišťuje se na straně serveru s použitím Sessions a tokenu předávaném v HTTP hlavičkách Cookies.

6.7 Použité knihovny

Pro vytvoření této aplikace bude použita knihovna Vue.js pro webovou aplikaci, knihovna Electron pro vytvoření desktopové aplikace a knihovna Cordova pro vytvoření mobilní aplikace.

Na rozdíl od předchozí aplikace, je pro tento případ vybrána pro vytvoření webové aplikace knihovna Vue.js, a to především kvůli tomu, že ji lze nasadit a postupně zavádět do již existujícího kódu. Lze tedy vzít originální HTML kód, který je použit pro generování výsledného kódu na straně serveru, a rozšířit ho s použitím direktiv Vue.js.

Knihovny Electron a Cordova jsou zvoleny především díky své popularitě a rozsáhlé komunitě a z nich vyplývající velké znalostní bázi.

7 Shrnutí výsledků

V první části této práce bylo popsáno několik knihoven, z nichž některé jsou použity pro ukázkovou aplikaci. Jsou popsány jejich výhody a nevýhody a proč byly zvoleny pro ukázkovou aplikaci.

V druhé části je popsán postup při vývoji jednoduché aplikace pro evidování úkolů. Aplikace je vytvořena pro několik platforem – jako webová aplikace pro použití pomocí prohlížeče, jako nativní aplikace pro operační systém Windows a jako nativní aplikace pro operační systém Android. Zároveň byla vytvořena serverová část, která komunikuje se všemi klientskými aplikacemi.

Při vytvoření klientských aplikaci se kladl důraz na znovupoužitelnost kódu. Ve výsledných aplikacích tak lze pozorovat velkou část kódu, která je sdílená mezi webovou, desktopovou a mobilní aplikací. Lze říci, že není sdílený jen ten kód, který je specifický pro jednotlivé platformy a vyplývá z odlišnosti funkčnosti pro jednotlivé platformy.

8 Závěry a doporučení

Jak již bylo poznamenáno v úvodu, poznatky a zkoumání v této práci vychází z analýzy možných způsobů vytvoření multiplatformní aplikace pro mého zaměstnavatele. Při analýze možných řešení jsem jednotlivé frameworky zkoumal spíše rámcově bez podrobného zkoumání funkcionality. Jedním z hlavních požadavků bylo najít co nejefektivnější (z hlediska vývoje) způsob, jak multiplatformní aplikace vytvářet.

Při úvodní analýze možných řešení, byl JavaScript jediným kandidátem, který bylo možné použít pro všechny požadované platformy – web, desktop i mobil. Při bližším seznámení a na základě zjištěných informací a subjektivních názorů jsem přišel k závěru, že tento způsob má svá omezení.

Způsob popsaný v této práci se hodí především pro aplikace, které nejsou výpočetně náročné (pokud se jedná o klientské aplikace), případně i pro ty, u nichž lze tyto náročné výpočty přesunout na stranu serveru. Velmi dobře se pro to hodí architektura MVC, která je rozdělená mezi serverovou část a klientskou aplikaci. Model aplikace se nachází na serverové části a jen některé jeho části (typicky validace) jsou duplikovány v klientské aplikaci pro snížení množství požadavků na server a zvýšení rychlosti interakce uživatele s aplikaci. Vrstva řadičů (Controller) se pak nachází jak na straně serveru, tak i na straně aplikace a zajišťují vzájemnou komunikaci. Vrstva pohledová je pak zcela jiná na straně serveru, kde server vrací jen čistá data, a na straně aplikace, která tyto data zpracuje a zobrazí je ve výsledné podobě uživateli.

Již při analýze multiplatformních aplikací založených na architektuře klient-server je nutno myslet na to, jak se aplikace vypořádá s výpadkem spojení se serverem. S tímto pak je nutno počítat i v následujících fázích vývoje, při návrhu i implementaci. Tuto funkčnost lze vnímat jako další odlišnost platformy od jiných.

I přestože všechny klientské aplikace sdílí většinu kódu, jsou tam i části, které jsou specifické pro jednotlivé platformy. Tyto části nutno udržovat zvlášť, čímž se zvyšuje i náročnost takového projektu. Na druhou stranu, pokud porovnám vytvoření všech klientských aplikací jako nativních, tak kvůli použití různých programovacích jazyků, které s sebou nesou další komplikace v podobě například

synchronizace funkčnosti pro jednotlivé platformy, lze konstatovat, že vývoj multiplatformních aplikaci v JavaScriptu je násobně rychlejší a efektivnější.

Nicméně tento přístup výrazně ztrácí, pokud se jedná o výpočetně náročné aplikace, kde nativní aplikace jsou podstatně rychlejší, nebo je nutný přístup k funkcím, které knihovny pro vytváření aplikaci pro jednotlivé platformy neposkytují.

Za hlavní výhodu aplikací, které používají stejný jazyk jak na straně serveru tak i na straně klienta, bych označil především možnost sdílení kódu mezi těmito částmi. Například použití stejného kódu pro validaci uživatelského vstupu, kdy stejná data se validují jak na frontendu, pro poskytnutí uživateli co nejrychlejší odezvy v případě chyby, tak na straně serveru, pro zajištění konzistence dat bez ohledu na použitého klienta.

V dnešní době bych však preferoval kombinaci JavaScriptu na frontendu a klasického objektového jazyku na backendu, například Java. JavaScript se skvěle hodí na frontend díky snadnému vytváření interaktivních aplikací, které na rozdíl od klasických webových stránek poskytují mnohem plynulejší práci s jejich obsahem. JavaScript mi nepřijde jako dobrá volba především kvůli rychlosti, jakou se vyvíjí. To, co platilo před rokem, již dnes nemusí platit, stejně tak to, co platí dnes nemusí platit za rok. Toto přináší velkou nejistotu v tom, že za nějakou dobu bude nutno vynakládat mnohem více času pro údržbu takové aplikace, případně by se musela celá přepsat.

9 Seznam použité literatury

[1] MARTIN, Robert C. Clean code: a handbook of agile software craftsmanship. Upper Saddle River, NJ: Prentice Hall, c2009. ISBN 9780132350884.

[2] CHLAPEK, Dušan, Václav ŘEPA a Iva STANOVSKÁ. Analýza a návrh informačních systémů. Praha: Oeconomica, 2011. ISBN 978-80-2451-782-7.

[3] ELLIOTT, Eric. Programming JavaScript applications. Sebastopol: O'Reilly, 2014. ISBN 1491950293.

[4] HOWARD, Daniel. Node.js for PHP developers. Sebastopol, CA: O'Reilly, 2013. ISBN 978-1449333607.

[5] Google Trends. Trendy Google [online]. Google LLC, [cit. 2018-02-05]. Dostupné z: https://trends.google.com/trends/explore?date=all&q=node.js

[6] Express.js. Node.js web application framework [online]. The ExpressJS Organization, [cit. 2018-02-05]. Dostupné z: https://expressjs.com/

[7] Vue.js [online]. Evan You, [cit. 2018-02-05]. Dostupné z: https://vuejs.org

[8] Webová dokumentace MDN [online]. Mozilla, [cit. 2018-02-08]. Dostupné z: https://developer.mozilla.org/

[9] Apache Cordova [online]. The Apache Software Foundation, [cit. 2018-02-08]. Dostupné z: https://cordova.apache.org/

[10] React - A JavaScript library for building user interfaces [online]. Facebook Inc, [cit. 2018-02-02]. Dostupné z: https://reactjs.org/

[11] Electron | Tvořte multiplatformní desktopové aplikace pomocí jazyku Javascript, HTML a CSS [online]. Electron, [cit. 2018-02-02]. Dostupné z: https://electronjs.org

[12] Ajax: A New Approach to Web Applications | Adaptive Path [online]. Adaptive Path, [cit. 2018-02-02]. Dostupné z http://adaptivepath.org/ideas/ajax-new-approach-web-applications/

[13] History | jQuery Foundation [online]. The jQuery Foundation, [cit. 2018-02-02]. Dostupné z https://jquery.org/history/

[14] Modulecounts [online]. Erik DeBill, [cit. 2018-02-02]. Dostupné z http://www.modulecounts.com/

[15] BROWN, Ethan. Web development with Node and Express. Sebastopol, CA: O'Reilly, 2014. ISBN 978-1491949306.

[16] The Unbelievable History of the Express JavaScript Framework [online]. Dor Tzur, [cit. 2018-02-02]. Dostupné z https://thefullstack.xyz/history-express-javascript-framework/

[17] Sails.js | Realtime MVC Framework for Node.js [online]. The Sails Company, [cit. 2018-02-02]. Dostupné z https://sailsjs.com

[18] A brief history of JavaScript [online]. Ben Aston [cit. 2018-02-05]. Dostupné z https://medium.com/@benastontweet/lesson-1a-the-history-of-javascript-8c1ce3bffb17

[19] Dart programming language [online]. Google LLC, [cit. 2018-02-05]. Dostupné z https://www.dartlang.org/

[20] Shoes! [online]. Team Shoes, [cit. 2018-02-05]. Dostupné z http://shoesrb.com

[21] React Native [online]. Facebook Inc, [cit. 2018-02-02]. Dostupné z http://facebook.github.io/react-native

[22] Things You Should Never Do, Part I [online]. Joel Spolsky, [cit. 2018-02-02]. Dostupné z https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2017/2018

Studijní program: Aplikovaná informatika
Forma: Kombinovaná
Obor/komb.: Aplikovaná informatika (ai3-k)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Evsin Arťom	Chelčického 1242/2, Hradec Králové - Pražské Předměstí	I14493

TÉMA ČESKY:

Vývoj multiplatformní aplikace v jazyce Javascript

TÉMA ANGLICKY:

Multiplatform application development in Javascript language

VEDOUCÍ PRÁCE:

Mgr. Daniela Ponce, Ph.D. - KIT

ZÁSADY PRO VYPRACOVÁNÍ:

Cílem práce je optimalizovat výběr nástroje pro vývoj multiplatformní aplikace v JavaScriptu s ohledem na znovupoužitelnost a udržovatelnost kódu.

SEZNAM DOPORUČENÉ LITERATURY:

Podpis studenta: .... **Datum:** ....

Podpis vedoucího práce: .... **Datum:** ....

(c) IS/STAG , Portál - Podklad kvalifikační práce , I14493 , 26.04.2018 02:02