VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2020

Jan Bartoň

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

OPTIMALIZACE SPOUŠTĚNÍ SINGLE PAGE APLIKACE

OPTIMIZATION OF SINGLE-PAGE APPLICATION STARTUP

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — Jan Bartoň

AUTHOR

VEDOUCÍ PRÁCE — Ing. Petr Ilgner

SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Telekomunikační a informační systémy**

Ústav telekomunikací

Student: Jan Bartoň ***ID:*** 195268
Ročník: 3 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Optimalizace spouštění single page aplikace

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s problematikou výkonu při zpracování webových stránek a interpretace Javascriptu. Analyzujte problémy aplikace Kentico Kontent (zástupce single-page Javascript aplikace) a představte optimalizační techniky pro zrychlení webové aplikace, jako je rozdělení aplikace do menších celků a separátní optimalizace Javascript balíčků pro starší a moderní prohlížeče. Navrhněte a implementujte vybrané optimalizační techniky v aplikaci Kentico Kontent. Změřte a vyhodnoťte efekt implementovaných úprav.

DOPORUČENÁ LITERATURA:

[1] MORGAN, Joe. Simplifing JavaScript: writing modern JavaScript with ES5. ES6, and beyond. Raleigh, 2018, xiv, 260 str. ISBN 978-1-68050-288-6.

[2] SCOTT Emmit. SPA Design and Architecture: Understanding Single Page Web Applications. Manning Publications, 2015, 275 str. ISBN 978-1617292439.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 8.6.2020

Vedoucí práce: Ing. Petr Ilgner
Konzultant: Richard Juchelka, Kentico software s.r.o.

prof. Ing. Jiří Mišurec, CSc.
předseda rady studijního programu

UPOZORNĚNÍ:
Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Tato práce se zabývá optimalizací spouštění single page JavaScript aplikace Kentico Kontent. Soustředí se zejména na rozdělení JavaScript knihoven a balíčků do samostatných souborů podle různých kritérií. Provedené úpravy v aplikaci jsou členěny do několika scénářů. Pro každý ze scénářů bylo provedeno šest druhů měření, díky kterým bylo zjištěno, jaký dopad mají provedené změny na rychlost načítání aplikace a množství přenesených dat ze serveru ke klientovi. Výsledkem práce je návrh úprav, díky kterým se aplikace spouští rychleji, protože je ze serveru přenesen menší objem dat a aplikace zvládne využít efektivněji mezipaměť prohlížeče.

KLÍČOVÁ SLOVA

aplikace, JavaScript, Kentico, optimalizace, React.js, webpack

ABSTRACT

This thesis concerns with optimalisation of launching the single page JavaScript application Kentico Kontent. It especially focuses on dividing JavaScript libraries and packages into independent files according to certain criteria. These adjustments to the application are divided into several scenarios. For every scenario, six types of measurements have been carried out, due to which the impact of the change on the speed of loading the application and the amount of transferred data from server to client was found. The result of this thesis is a proposal of adjustments which lead to a faster launch of the application, due to a smaller amount of data being transferred from the server, therefore the application can use the browser cache more effectively.

KEYWORDS

application, JavaScript, Kentico, optimalisation, React.js, webpack

BARTOŇ, Jan. *Optimalizace spouštění single page aplikace*. Brno, 2020, 67 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Petr Ilgner

Vysázeno pomocí balíčku `thesis` verze 3.05; `http://latex.feec.vutbr.cz`

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Optimalizace spouštění single page aplikace“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Děkuji panu Ing. Petru Ilgnerovi za odborné vedení mé práce. Dále děkuji panu Bc. Richardu Juchelkovi za jeho odborné rady, ochotu a trpělivost při konzultacích mé bakalářské práce. V neposlední řadě bych chtěl také poděkovat své rodině, přátelům a všem, kteří mě při studiu podporovali.

Obsah

Úvod **12**

1 JavaScript **13**

1.1 ECMAScript . . . 13

1.2 Technical Committee 39 . . . 15

1.2.1 TC39 proces . . . 15

1.3 JavaScript engine . . . 17

1.3.1 Interpretace a kompilace . . . 17

1.3.2 Just-in-time kompilace . . . 18

1.4 Single page aplikace . . . 20

1.4.1 React.js . . . 20

1.5 webpack . . . 22

1.5.1 Základní koncepty . . . 22

2 Systém pro správu obsahu **24**

2.1 Headless CMS . . . 24

2.2 Kentico Kontent – zástupce headless CMS . . . 24

2.2.1 Uživatelské rozhraní aplikace . . . 25

2.3 Infrastruktura aplikace Kentico Kontent . . . 27

2.4 Počáteční nastavení aplikace . . . 29

2.4.1 webpack . . . 29

2.4.2 HTTP komprese . . . 32

2.4.3 HTTP caching . . . 32

2.4.4 Nedostatky počátečního stavu . . . 33

3 Měření a aplikované optimalizace **35**

3.1 Měření jednotlivých scénářů . . . 35

3.2 Původní stav aplikace . . . 37

3.2.1 Výsledky měření . . . 38

3.3 Rozdělení knihoven a vlastního kódu do samostatných souborů . . . 39

3.3.1 Provedené změny . . . 39

3.3.2 Výsledky měření . . . 40

3.4 Samostatný soubor pro každý balíček třetí strany . . . 41

3.4.1 Provedené změny . . . 41

3.4.2 Výsledky měření . . . 43

3.5 Rozdělení balíčků třetích stran i vlastního kódu do několika samostatných souborů . . . 44

3.5.1 Provedené změny 44
3.5.2 Výsledky měření 45
3.6 Optimalizace předchozího scénáře a asynchronní načítání větších částí aplikace 46
3.6.1 Provedené změny 46
3.6.2 Výsledky měření 49

Závěr **50**

Literatura **52**

Seznam symbolů, veličin a zkratek **55**

Seznam příloh **56**

A Přílohy **57**
A.1 Veškeré naměřené hodnoty 57
A.1.1 Výchozí stav aplikace 57
A.1.2 Rozdělení knihoven a vlastního kódu do samostatných souborů 60
A.1.3 Samostatný soubor pro každý balíček třetí strany 62
A.1.4 Rozdělení balíčků třetích stran i vlastního kódu do několika samostatných souborů 64
A.1.5 Optimalizace předchozího scénáře a asynchronní načítání větších částí aplikace 66

Seznam obrázků

2.1 Headless CMS princip 25
2.2 Vytváření obsahu v Content Item editoru. 26
2.3 Vytváření modelu v Content Model editoru. 27
2.4 Zjednodušený model infrastruktury Kentico Kontent 28
2.5 Komunikace klienta se serverem, kdy je využito místního úložiště, odkud může být načtena část příloh nutných pro spuštění aplikace. . 34
3.1 Chrome DevTools – jednotlivé JavaScript požadavky při původním stavu aplikace. 37
3.2 Chrome DevTools - seznam JavaScript příloh obsahující knihovny a balíčky. Optimalizovaný poměr počtu souborů k jejich velikosti. . . . 47

Seznam tabulek

3.1 Naměřené hodnoty pro počáteční stav aplikace. 38
3.2 Naměřené hodnoty pro scénář, ve kterém jsou přílohy rozděleny na dva soubory. 40
3.3 Naměřené hodnoty pro scénář obsahující jeden soubor s vlastním kódem aplikace a jedním souborem pro každou knihovnu či balíček. . . 43
3.4 Naměřené hodnoty pro rozdělené knihovny a balíčky třetích stan i vlastní kód aplikace. 45
3.5 Naměřené hodnoty pro scénář, kdy jsou knihovny rozděleny do optimálního počtu příloh, pro větší části aplikace je nastaveno asynchronní načtení. 49
3.6 Porovnání původního stavu se změnami ze scénáře 3.6. Hodnoty byly získány při typu měření, kdy byly provedeny změny v JS přílohách a uživatel využil mobilní připojení. 50
3.7 Porovnání původního stavu se změnami ze scénáře 3.6. Hodnoty byly získány při měření, kdy uživatel spouští aplikaci poprvé nebo v mezipaměti nejsou uloženy žádné přílohy. 51
A.1 První načtení aplikace 57
A.2 Načtení s mezipamětí 57
A.3 První načtení se simulací mobilního LTE připojení 58
A.4 Načtení se simulací mobilního LTE s využitím mezipaměti 58
A.5 Načtení po změně v klientském JavaScript souboru 58
A.6 Načtení po změně v klientském JavaScript souboru se simulací mobilního LTE připojení 59
A.7 První načtení aplikace 60
A.8 Načtení s mezipamětí 60
A.9 První načtení se simulací mobilního LTE připojení 60
A.10 Načtení se simulací mobilního LTE s využitím mezipaměti 61
A.11 Načtení po změně v klientském JavaScript souboru 61
A.12 Načtení po změně v klientském JavaScript souboru se simulací mobilního LTE připojení 61
A.13 První načtení aplikace 62
A.14 Načtení s mezipamětí 62
A.15 První načtení se simulací mobilního LTE připojení 62
A.16 Načtení se simulací mobilního LTE s využitím mezipaměti 63
A.17 Načtení po změně v klientském JavaScript souboru 63
A.18 Načtení po změně v klientském JavaScript souboru se simulací mobilního LTE připojení 63

A.19 První načtení aplikace 64
A.20 Načtení s mezipamětí 64
A.21 První načtení se simulací mobilního LTE připojení 64
A.22 Načtení se simulací mobilního LTE s využitím mezipaměti 65
A.23 Načtení po změně v klientském JavaScript souboru 65
A.24 Načtení po změně v klientském JavaScript souboru se simulací mobilního LTE připojení 65
A.25 První načtení aplikace 66
A.26 Načtení s mezipamětí 66
A.27 První načtení se simulací mobilního LTE připojení 66
A.28 Načtení se simulací mobilního LTE s využitím mezipaměti 67
A.29 Načtení po změně v klientském JavaScript souboru 67
A.30 Načtení po změně v klientském JavaScript souboru se simulací mobilního LTE připojení 67

Seznam výpisů

1.1 Příklad jednoduché React.js komponenty, která reprezentuje počítadlo s vlastním stavem. 21
2.1 Vstupní body aplikace v konfiguračním souboru webpack.config.js. . . 30
2.2 Nastavení pro výstupní bod aplikace v konfiguračním souboru webpack.config.js. 31
3.1 SplitChunksPlugin – rozdělení na dva výstupní soubory, kdy jeden obsahuje všechny knihovny a balíčky třetích stran, druhý veškerý vlastní kód aplikace. 39
3.2 SplitChunksPlugin – rozdělení JavaScript příloh na jeden soubor obsahující vlastní kód aplikace a jeden soubor pro každou knihovnu či balíček třetí strany. 41
3.3 HTMLWebpackPlugin – nastavení parametrů pro nástroj, který generuje HTML soubor pro část client. 42
3.4 Použití makra v šabloně index.ejs, na toto místo bude ve vygenerovaném HTML souboru vložen JavaScript kód pro nástroj Amplitude. 43
3.5 SplitChunksPlugin – rozdělení JavaScript příloh, kdy je vytvořen soubor pro kód sdílený mezi vstupními body, dále soubor s kódem unikátním pro vstupní bod. Knihovny a balíčky třetích stran jsou rozděleny do souborů podle uvedených kritérií. 44
3.6 SplitChunksPlugin – optimalizovaná verze předchozího scénáře, kdy jsou změněny kritéria pro rozdělení knihoven a balíčků třetích stran do samostatných příloh. 46
3.7 Zjednodušený příklad asynchronního načítání větší části aplikace v React.js s využitím metody lazy. Tento kód je uveden v komponentě Project.tsx 48

Úvod

Single page aplikace jsou tvořeny velkým množstvím JavaScript kódu. Běžně se pro tvorbu těchto aplikací využívají desítky JavaScript frameworků a knihoven. Tyto nástroje jsou nezbytné, protože pomáhají s efektivním využitím potenciálu jazyka JavaScript a jde s nimi udržitelně vyvíjet rychle rostoucí aplikaci.

S přibývajícím počtem využitých frameworků a knihoven se zvětšuje i objem zdrojového kódu, který je potřeba přenést ke každému uživateli předtím, než se aplikace spustí. Na rychlost načítání aplikace má značný vliv také rychlost připojení k internetu, kterou uživatel disponuje. Zejména u mobilního připojení může být rychlost ovlivněna množstvím faktorů.

Při prvním spouštění aplikace na konkrétním zařízení se většinou stahuje největší množství příloh. Tyto přílohy lze uložit do mezipaměti prohlížeče, označované také jako cache, a využít toho při opětovném spouštění aplikace. Z mezipaměti prohlížeče lze příloha použít pouze tehdy, pokud je její verze aktuální. To však komplikuje skutečnost, že se do aplikace může dostat aktualizovaná verze JavaScript příloh i několikrát denně.

Cílem mé práce je najít způsob, jak rozdělit JavaScript přílohy tak, aby provedené změny zasáhly vždy jen nezbytné množství příloh. To povede k efektivnímu využití cache paměti prohlížeče, takže se při provedení byť jen nepatrné změny nebude muset stahovat objemný soubor s veškerým JavaScript kódem, jak je tomu doposud. Dále se v práci snažím zmenšit objem dat, který je nutné přenést k uživateli při prvním spouštění aplikace.

Práce je členěna do třech kapitol. V první z nich je popsán jazyk JavaScript, obecný princip využití single page aplikace a nástroje jako React.js či webpack, které se používají pro jejich tvorbu. Druhá kapitola popisuje, co je a k čemu slouží systém pro správu obsahu, přibližuje zástupce single page aplikace Kentico Kontent, původní nastavení aplikace a jeho nevýhody. Třetí kapitola se zaměřuje na aplikované změny, které jsou rozděleny do scénářů. Pro každý scénář je provedeno měření, pomocí získaných hodnot z měření je vyhodnoceno, jaký dopad měly úpravy na spouštění aplikace. Závěr práce shrnuje a hodnotní výsledky měření a provedené úpravy a změny ve zdrojovém kódu aplikace.

1 JavaScript

Tato kapitola popisuje programovací jazyk JavaScript. JavaScirpt a jeho knihovny lze považovat za aktuálně nejpoužívanější nástroje pro vývoj single page aplikací.

JavaScript je interpretovaný skriptovací jazyk vyšší úrovně, spadající pod specifikaci ECMAScript. Spolu s HTML a CSS je jednou ze základních World Wide Web technologií. Umožňuje interaktivní chování webových stránek a tvoří podstatnou část single page aplikací. Všechny populární internetové prohlížeče dnes obsahují interpretr pro vykonávání JavaScript programů.

Vznik JavaScriptu sahá do roku 1995, kdy byl společností Netscape najat Brendan Eich. Jeho úkolem bylo implementovat Scheme (dialekt Lispu) do webového prohlížeče Netscape Navigator, který byl vydán v předešlém roce. Později byl JavaScript implementován do všech majoritních webových prohlížečů. Webovým aplikacím umožňoval přímou interakci bez potřeby znovu načíst stránku po každé provedené akci [1].

Pojmenováním tohoto jazyka jako JavaScript byl marketingový záměr. Autor jazyka Brendan Eich nejprve používal pojmenování Mocha. Uvnitř společnosti Netscape se také používalo pojmenování LiveScript. Při vydání však z důvodu propagace zvítězil název JavaScript. Jazyk byl od začátku navrhován tak, aby přilákal pozornost zejména Java programátorů, protože v té době byla Java jedním z nejpoužívanějších programovacích jazyků. Slovo "script" bylo v té době populární pro označení programů, které bylo jednoduché implementovat. Jinými slovy název pozicoval JavaScript jako atraktivní alternativu robustní a dobře známé Javě [2].

1.1 ECMAScript

ECMAScript je standardizovaný skriptovací jazyk normovaný mezinárodní neziskovou organizací ECMA International. ECMAScript je založený na technologiích známých pod jménem JavaScript (Netscape) a Jscript (Microsoft). První verze ECMA Standard byla přijata v červnu v roce 1997. V dubnu 1998 byla komisí ISO/IEC JTC 1 schválena norma ISO/IEC 16262 [4] pro ECMAScript. V červnu toho roku byla dále schválena valným shromážděním druhá edice ECMAScript v souladu s normou ISO/IEC 16262 [3].

Ve třetím vydání byl ECMAScript Standard rozšířen o nové součásti jako např. regulární výrazy, vylepšenou manipulaci s řetězci, ošetření výjimek try/catch, přesněji definovaná chybová hlášení atd. Toto vydání bylo valným shromážděním ECMA schváleno v červnu roku 2002 pod aktualizovanou normou ISO/IEC 16262:2002. Po vydání třetí verze se z ECMAScript stal velmi populární programovací jazyk pro

webové stránky, který byl podporován v podstatě všemi webovými prohlížeči. Vývoji čtvrté verze bylo věnováno spoustu práce, nicméně nové součásti nebyly později dokončeny a zveřejněny jako čtvrté vydání. Část z nich se však dostala do pozdějších verzí [3].

Pátá verze ECMAScript byla předložena komisi ISO/IEC JTC 1 k přijetí v rámci zrychleného postupu pod mezinárodní normou ISO/IEC 16262:2011. Tato verze obsahovala nové funkce pro manipulaci s polem, reflexní tvorbu a podporu objektů, podporu formátu JSON a tzv. strict mode, který poskytuje vylepšenou kontrolu chyb a zabezpečení programu. S drobnými úpravami byla pod stejnou normou vydána verze 5.1, která byla přijata valným shromážděním ECMA v červnu 2011 [3].

Vývoj šesté verze začal již v roce 2009 v době, kdy se zároveň pracovalo na dokončení a vydání páté verze. Předcházelo tomu značné úsilí o zlepšení jazyka sahající až do roku 1999. Dá se tedy říct, že dokončení šesté verze znamenalo vyvrcholení dlouholeté snahy o přepracování jazyka. Mezi cíle těchto změn patřilo zejména poskytnutí lepší podpory pro tvorbu velkých aplikací, vytváření knihoven a využití ECMAScript jako cílového jazyka pro kompilování jiných jazyků. Mezi další vylepšení lze zařadit např. změnu v připojování modulů, deklaraci tříd, lexical block scoping, Promise, podporu pro asynchronní programování [3].

ECMAScript 2016 (ES7) byla první edice ECMAScript vydaná pod organizací TC39, to přináší změnu hlavně v pravidelnosti vydávání dalších verzí a otevřenější proces vývoje. Ze zdrojového dokumentu ECMAScript 2015 byl vytvořen dokument, který sloužil jako základ pro další vývoj jazyka, který dále probíhá tak, že členové organizace TC39 přispívají do repozitáře na portálu Github. Během roku vývoje tohoto standardu byly podány stovky pull requestů, které řeší buď opravy různých chyb, nebo přidání nových funkcí. Zároveň byly vyvinuty další nástroje, které usnadňují práci na vývoji nových verzí ECMAScript, jako např. Ecmarkup, Ecmarkdown a Grammarkdown. ES2016 také přidává podporu pro novou metodu do Array.prototype s názvem include [3] .

ECMAScript 2017 přišel s podporou nových asynchronních funkcí a Shared Memory, funkce, která dokáže sdílet data mezi hlavním vláknem aplikace a tzv. Web Workers. Také byly přidány nové funkce Object.entries a Object.values pro jednodušší práci s objekty. Návratová hodnota pro Object.entries tvoří 2D pole se všemi klíči a hodnotami. Object.values vrací pole se všemi hodnotami objektu [3, 5].

ECMAScript 2018 zavedl podporu asynchronní iterace pomocí AsyncIterator a asynchronních generátorů. Dále také funkci Promise.prototype.finally doplňující try/catch konstrukci, rest/spread operátor pro jednodušší manipulaci s polem a objekty, či další vylepšení pro regulární výrazy jako např. podpora zápisu Unicode znaků se speciálním významem [3].

Desátá verze ECMAScript představila nové funkce Array.Flat a Array.flatMap.

První ze zmíněných metod vytvoří nové pole se všemi prvky sub-pole připojené do původního pole do zadané hloubky zanoření. Druhá ze zmíněných pracuje podobně jako spojení mapovací funkce s Array.Flat, nejprve tedy namapuje každý prvek pole a sub-pole a výsledek sloučí do nového pole. Nové funkce String.trimStart a String.trimEnd odstraňují mezery ze začátku a konce řetězce [3, 6].

1.2 Technical Committee 39

TC39 (Technical Committee 39) je skupina dohlížející na vývoj samotného JavaScriptu. Členové skupiny jsou dobrovolníci, většina z nich jsou zaměstnanci velkých společností, pro které je webové prostředí klíčové. Zejména pak vývojáři prohlížečů (Google, Mozilla, Apple) či výrobci zařízení (Samsung, Apple atd.). Organizace má pravidelná setkání na kterých členové diskutují o nově vyvíjených funkcích a o problematice spojené s jejich vývojem.

ECMAScript verze ES6 byla příliš velká a její vydání trvalo téměř 6 let (ES5 – prosinec 2009 vs. ES6 – červen 2015). Protože vydání šesté verze trvalo až příliš dlouho, objevily se dva problémy:

- Nové funkce jejichž vývoj už byl dokončen musely počkat, než bylo připraveno vše pro vydání verze ES6.
- Funkce na jejichž vývoj bylo potřeba více času byly pod nátlakem blížícího se vydání nové verze ECMAScript, zmeškání termínu by vedlo ke zbytečnému odkladu jejich vydání.

Proto se od následující verze ECMAScript 2016 (ES7) přešlo na pravidelný interval vydávání nových verzí. Interval má délku přibližně jednoho roku a obsahuje menší množství nových funkcí [7].

1.2.1 TC39 proces

Každý návrh na novou funkci pro ECMAScript musí projít všemi kroky procesu a být schválen komisí TC39. Proces schvalování má celkem pět úrovní (stage 0 – stage 4).

Stage 0: Strawperson

Volná forma návrhu pro vylepšení ECMAScript. Návrh musí přijít buď od člena TC39, nebo od osoby registrované jako TC39 contributor. Požadavkem je, aby dokument se změnami prošel revizí na setkání TC39, a poté je zařazen do seznamu návrhů pro stage 0 [8].

Stage 1: Proposal

Pro každou novou součást musí být zvolen tzv. šampion, který je za návrh zodpovědný. Tato osoba musí být členem komise TC39. Popíše problém nebo potřebu, kterou by přidání nové součásti vyřešilo, a navrhne obecný tvar řešení. K popisu přidá ilustrační příklady použití, probere klíčové algoritmy, abstrakci, zamyslí se nad možnými problémy s kompatibilitou a komplexností řešení. V této úrovni procesu je prostor pro velké zásahy do původního návrhu nové součásti [8, 9].

Stage 2: Draft

První verze toho, co bude v budoucnu obsahovat implementace. V této úrovni procesu jsou očekávány běžné změny v řešení. V této úrovni musí návrh obsahovat dostatečný popis syntaxe a sémantiky použitím formálního jazyka specifikace ECMAScript. Tento popis by měl být dostatečně obsáhlý, avšak je možné, aby obsahoval komentáře a poznámky o tom, co bude dále do řešení přidáno. Očekávají se dvě verze implementace nové součásti [8].

Stage 3: Candidate

Návrh nové součásti je téměř hotový, další upřesnění bude vyžadovat zpětnou vazbu od editorů. Nyní je zapotřebí revize, kterou provádí členové určení komisí TC39. Dále musí s implementací a jejím popisem souhlasit všichni editoři ECMAScript. Řešení musí být kompletní bez očekávaných změn, musí být podrobně popsána API, syntaxe a sémantika implementace. Nyní jsou povoleny pouze opravy kritických chyb v řešení [8, 9].

Stage 4: Finished

Nyní je návrh připraven na přidání do standardizace ECMAScript. Před zařazením do stage 4 musí být splněn následující seznam:

- Test 262 – implementační sada jednotkových testů pro nejnovější ECMAScript koncepty [10].
- Dvě vyhovující specifikace řešení které prošly jednotkovými testy.
- Uvedené zkušenosti a příklady použití implementace.
- Pull request v repositáři tc39/ecma262 [11] schválen všemi editory ECMAScript.

Poté bude návrh přidán do implementace ECMAScript, hned jak to bude možné. Verze ECMAScript prochází ročním závazným potvrzením jako standard [8].

1.3 JavaScript engine

JavaScript engine je program vykonávající JavaScript kód. První JavaScript engine byl pouhým interpretrem, ale dnešní relevantní enginy využívají Just-in-time kompilaci pro větší efektivitu a výkon kódu. JavaScript enginy jsou vyvíjeny autory webových prohlížečů, avšak většina z nich není s prohlížečem svázána a lze je využívat dle potřeby i v jiných aplikacích.

V8

V8 je open-source Javascript engine vyvíjený společností Google pro vykonávání JavaScript a WebAssembly. Je vyvíjen v programovacím jazyku C++ a používá se v prohlížečích postavených na projektu Chromium a Node.js. V8 obsahuje plnou ECMAScript a WebAssembly implementaci. Díky tomu, že je tvořen jako samostatný program, který nezávisí na webovém prohlížeči, lze jej implementovat do aplikací naprogramovaných v jazyku C++ [12].

Autoři tohoto engine kladli důraz na zvýšení výkonu JavaScriptu v prohlížeči, proto se při uvedení engine V8 odlišoval od ostatních hlavně velkým rozdílem ve výkonu. V8 totiž pouze neinterpretoval JavaScript kód, ale přeložený strojový kód při dalším spuštění také optimalizoval. Později začali využívat Chromium engine (používá V8) i jiné webové prohlížeče, jako např. Opera (verze 15 - 2013), Vivaldi. 15. ledna 2020 se mezi další prohlížeče používající Chromium engine zařadil i Microsoft Edge. Spolu s touhle změnou Microsoft poprvé vydává svůj prohlížeč také na jiný operační systém než Windows, např. macOS či iOS.

ChakraCore

ChakraCore je klíčová část JavaScript enginu jménem Chakra, který byl od konce roku 2015 udržován jako open-source JavaScript engine společnosti Microsoft [13]. Byl používán webovým prohlížečem Microsoft Edge a dalšími Microsoft aplikacemi, které používají HTML/CSS/JS technologie. V lednu 2020 však Microsoft vydal svůj prohlížeč Edge postavený na Chromium engine, který nahradil engine Chakra. ChakraCore podporuje Just-in-time kompilaci JavaScriptu pro architektury x86/x64/ARM, garbage collection a široké spektrum nových součástí JavaScriptu. Dále podporuje JavaScript Runtime (JSRT) APIs, díky tomu lze ChakraCore snadno použít i v jiných aplikacích [14].

1.3.1 Interpretace a kompilace

Interpretace a kompilace jsou dva rozdílné způsoby, kterými lze vykonávat program napsaný v jakémkoliv programovacím jazyku vyšší úrovně.

Kompilátor je program, který převádí kód programovacího jazyka vyšší úrovně na strojový kód nebo na jazyk nižší úrovně. Kód je vždy zkompilován jako celek a samotný proces kompilace je rozdělen do dvou fází na syntézu a analýzu. Během syntézy kompilátor přeloží zdrojový kód jazyka vyšší úrovně na tzv. mezikód. V další fázi je z mezikódu generován strojový kód, který je vykonáván počítačem. Tím je zajištěno, že strojový kód je vždy vygenerován z mezikódu, který není závislý na konkrétním zařízení [15]. Analýza zdrojového kódu kompilátorem je časově náročný proces, avšak vykonávání je ve srovnání s analýzou rychlejší. Chybové hlášky jsou zobrazeny všechny najednou až po ukončení kompilace, což ztěžuje proces ladění programu.

Interpretr na rozdíl od kompilátoru nepracuje se zdrojovým kódem jako s celkem, ale vykonává jej řádek po řádku. Analýza zdrojového kódu interpretrem je proces, který není časově náročný, vykonávání kódu však je časově náročnější, protože interpretr může zdrojový kód zpracovávat více než jednou. Proces interpretace vyžaduje méně paměti než kompilace, to je způsobeno zejména tím, že interpretr vykonává přímo zdrojový kód bez nutnosti převádět ho na mezikód. Chyby jsou zobrazeny jednotlivě, to zjednodušuje proces ladění kódu [16].

Kompilátor i interpretr tedy slouží k vykonávání stejného úkolu, avšak s odlišným přístupem. Oba přístupy mají určité výhody a nevýhody. Interpretované jazyky jsou považovány za multiplatformní, jinými slovy stejný kód bude spustitelný bez ohledu na typ operačního systému. Podmínkou však je, že daný systém musí podporovat interpretr. Kompilované jazyky jsou z hlediska vykonávání rychlejší [17].

1.3.2 Just-in-time kompilace

Nevýhoda použití interpretru přichází, pokud se snažíme stejný kód vykonávat více než jednou. Jako příklad můžeme použít cyklus, kdy je při použití interpretru nutné přeložit stejný kód znovu a znovu při každém průchodu cyklem.

JIT kompilace je kombinací dvou tradičních přístupů překladu zdrojového kódu programu na kód strojový, spojující určité výhody a nevýhody obou přístupů.

Důvodem k využití JIT kompilace bylo odstranění nedokonalostí interpretru, proto se začal ve webových prohlížečích využívat mix kompilátoru a interpretru. Zástupci majoritních prohlížečů používají rozdílné JavaScript enginy, ty však mezi sebou sdílí podobný koncept.

Do enginů byl přidán prvek nazývaný monitor (také se užívá název profiler). Monitor sleduje vykonávání zdrojového kódu a značí si, kolikrát byl daný kód vykonán a jaké používal typy.

Základní kompilace

Při prvním spuštění prochází celý zdrojový kód interpretrem. Pokud je nějaký řádek vykonán více než jednou, monitor si ho označí. Charakter značení každého řádku závisí na počtu jeho vykonání. Pokud se určitá část kódu opakuje častěji, je tato část kódu zkompilována a uložena jako tzv. stub. [18].

Pojmem stub je označována část opakujícího se programu, která je uložena pro pozdější načtení [19].

Tyto uložené části kódu jsou indexovány podle čísla řádku kódu a podle typu proměnné. Pokud monitor zjistí, že je při vykonávání potřeba použít znovu kód který se shoduje s typem proměnné, použije již uloženou kompilovanou verzi. Výše zmíněná schopnost pomáhá ke zrychlení procesu. Základní kompilace však dokáže další důležitou věc pro zrychlení procesu, a to určitou formu optimalizace kódu. Tato optimalizace však nesmí trvat příliš dlouho, aby neblokovala vykonávání programu. Pokud se však určité uložené části programu vykonávají příliš často, je žádoucí ji optimalizovat více [18].

Optimalizace kompilátoru

Části kódu, které se opakují velmi často, jsou monitorem přesunuty pro optimalizaci pomocí kompilátoru. Ten ideálně vytvoří další, ještě rychlejší verzi funkce, která je uložena pro pozdější využití. Optimalizace je provedena na základě určitých předpokladů. Například, pokud se kompilátor domnívá, že objekty vytvořené konkrétním konstruktorem mají stejnou strukturu, tedy jejich vlastnosti mají stejná jména a jsou definovány ve stejném pořadí, může pak provést určité kroky vedoucí k optimalizaci. Optimalizační kompilátor využívá informací zjištěných pomocí monitoru, který sleduje vykonávání zdrojového kódu, a na základě toho tvoří předpoklady.

Kód, který byl optimalizován a zkompilován na základě určitých předpokladů, však musí být zkontrolován před jeho vykonáním. Pokud jsou předpoklady splněny, kód je vykonán. Pokud ne, JIT kompilátor zjistí, že byl kód optimalizován podle špatného předpokladu a je potřeba ho zahodit. Vykonávání se vrací znovu do interpretru nebo do základního kompilátoru, tento proces se nazývá deoptimalizace. Obvykle optimalizační kompilátor zrychluje kód, občas však může dojít k opakované optimalizaci a deoptimalizaci, to má pak za následek děj opačný. Většina prohlížečů však implementuje limit pro cyklus optimalizace/deoptimalizace [18].

1.4 Single page aplikace

Jedná se o webovou aplikaci vybudovanou pomocí technologií HTML, CSS, JavaScript a jejich frameworků. V aplikaci se vše odehrává na jedné stránce – během používání tak nevyžaduje opětovné načtení stránky. Rozhraní single page aplikace reaguje na uživatelské vstupy dynamicky, to může v uživateli vyvolat dojem, jako by používal klasickou desktop aplikaci.

V single page aplikacích nejsou jednotlivé pohledy tvořeny kompletními HTML šablonami. Jsou složené z více menších částí, tzv. komponent, které vytvářejí aktuální pohled. Při úvodním načtení aplikace se stáhnou všechny potřebné přílohy pro vytváření a zobrazení pohledů. Pokud je potřeba aktuální zobrazení změnit, v prohlížeči se z dostupných příloh vygenerují změny, které se připojí k DOM [20].

Pro vytváření SPA aplikací se běžně používají frameworky nebo knihovny, jako např. React.js, které mají na starost dynamické chování. Bez použití těchto nástrojů by při změně aktuálního stavu aplikace musel skript pomocí DOM API najít element závislý na změně a ten přímo upravit. Díky virtuálního DOM a vázání dat tyto nástroje samy detekují změny hodnot a DOM efektivně upraví [21].

Výhody použití Single page aplikace:

- **Dynamické změny** – změna aktuálního pohledu je provedena instantně bez nutnosti opětovného načtení, to je vhodné pro aplikace, které potřebují poskytnout bohaté uživatelské rozhraní.
- **Oddělená prezentační vrstva** – kód ovlivňující podobu uživatelského rozhraní se nachází na straně klienta. Oddělené vrstvy se mohou udržovat a aktualizovat nezávisle na sobě.
- **Úsporná komunikace se serverem** – při prvním načtení aplikace se stáhne veškerý zdrojový kód pro vykreslení uživatelského rozhraní, dále se mezi serverem a klientem posílají pouze data, která mají být v rozhraní zobrazena.
- **Prostředí webového prohlížeče** – alespoň jeden webový prohlížeč má na svém zařízení každý uživatel. Údržba a aktualizace se děje na serveru, který aplikaci hostuje. Tím odpadá potřeba aplikaci do zařízení instalovat.

1.4.1 React.js

React.js je JavaScript knihovna určena pro vytváření uživatelského rozhraní. Je vyvíjena společností Facebook a komunitou samostatných vývojářů. Nejčastěji se knihovna využívá pro tvorbu single page aplikací.

V React.js je kód rozdělen do zapouzdřených komponent, které mohou mít vlastní stav. Uživatelské rozhraní se vytváří skládáním komponent dohromady. Díky toho

je jednoduché vytvářet komplexní uživatelské rozhraní, které se mění podle stavu aplikace. Uvnitř komponent se využívá syntaxe JSX, jde o rozšíření pro JavaScript, které na první pohled připomíná spojení JavaScript a HTML syntaxe.

```
function Counter(props) {
  const [count, setCount] = useState(0);

  return (
    <>
      <p>Counter value: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        {props.buttonValue}
      </button>
    </>
  );
}
```

Výpis 1.1: Přiklad jednoduché React.js komponenty, která reprezentuje počítadlo s vlastním stavem.

Ve výpisu 1.1 je definice komponenty reprezentující jednoduché počítadlo. Komponenta do uživatelského rozhraní vykreslí tlačítko a vypíše aktuální hodnotu počítadla. Hodnota počítadla je uložená ve stavu komponenty pomocí React.hooks `useState` [22]. Výchozí hodnota je nastavená na 0. Při kliknutí na tlačítko metoda `setCount` přičte k aktuálnímu stavu uloženém v konstantě `count` jedničku. Pomocí `props` lze v React.js definovat rozhraní, přes které se do komponenty dostávají data.

Pokud je aktuální stav aplikace změněn, tato změna se musí propagovat do DOM. Document Object Model, zkráceně DOM, reprezentuje uživatelské rozhraní single page aplikace jako ve stromové struktuře uspořádaná data. Propagování změn přímo do DOM je však neefektivní a náročné, protože s přímou změnou se musí znovu načíst stránka s aplikací. React.js pro aktualizaci rozhraní aplikace používá koncept zvaný Virtual DOM, jedná se o virtuální kopii reálného DOM.

Kdykoliv je změněn stav aplikace, vytvoří se nová instance Virtual DOM, která je porovnána s předchozí verzí. V React.js se tento proces nazývá *diffing*. Pouze React ví, která část Virtual DOM byla změněna. V reálném DOM React aktualizuje pouze objekty, které na základě *diffing* procesu klasifikoval jako rozdílné. Využití Virtual DOM konceptu je z hlediska výkonu aplikace daleko efektivnější. Vývojář musí zajistit jen aktualizaci stavu komponenty podle svých potřeb. Výše zmíněný proces provádí sám React a vývojář tento princip nemusí znát [23].

Při vytváření komplexní single page aplikace se spolu s React.js, využívají další nástroje, které usnadňují vývoj. Jedním z nich je Typescript. Jedná se o JavaScript nadstavbu, která nabízí např. statické datové typy, typovou kontrolu, třídy a další. Kromě TypeScript se pro vývoj SPA Kentico Kontent využívá také Redux. Jedná se o kontejner pro definování předvídatelných stavů. To zajišťuje konzistentní chování aplikace a ulehčuje její testování.

V kontextu komponenty uvedené ve výpisu 1.1, kdykoliv uživatel klikne na tlačítko, spustí tím událost `onClick`. Ta pomocí metody `setCount` změní její stav. Následně proběhne proces, kdy se komponenta znovu načte a v rozhraní aplikace se zobrazí aktualizované data.

1.5 webpack

Webpack je nástroj pro zpracovávání souborů, pomáhá usnadnit práci vývojářům webových aplikací. Jde o kombinaci balíčkovače (jako např. Browserify) a spouštěče úloh, podobných jako třeba Grunt nebo Gulp. Webpack je určený primárně pro JavaScript, jeho modularizaci a chytré distribuci pomocí dělení kódu. Při zpracovávání aplikace webpack interně vytvoří graf závislostí, který mapuje každý modul a generuje pak jeden či více balíčků.

1.5.1 Základní koncepty

Pro pochopení toho, jak webpack funguje, je zapotřebí se seznámit se čtyřmi základními koncepty tohoto nástroje.

Vstupní body

Entry point neboli vstupní bod určuje, z kterých modulů webpack vytvoří graf závislostí. webpack potom sám zjistí na kterých dalších knihovnách a modulech závisí vstupní bod. Ve výchozím stavu webpack potřebuje alespoň jeden vstupní bod, nicméně je možné jich připojit i více. Pro každý vstupní bod webpack interně vytváří vlastní graf závislostí. Větší množství vstupních bodů se dá využít např. u multi-page aplikací, přičemž je doporučeno použít pro každý HTML dokument jeden vstupní bod.

Výstupní body

Výstup informuje webpack o tom, jak se má zachovat k vytvořenému balíčku ze vstupního bodu. Ve výchozím nastavení webpack vytvoří jeden výstupní soubor,

který je uložen ve složce dist spolu s dalšími generovanými soubory. Pokud se rozhodneme specifikovat výstup v konfiguračním souboru, minimálním požadavkem je definovat jméno pro výstupní soubor. Pokud potřebujeme vytvořit více než jeden výstupní soubor, je vhodné využít substituci pro zajištění unikátního názvu pro každý výstup.

Loadery

Samotný webpack rozezná pouze syntaxi JavaScript a JSON souborů. Pokud potřebujeme pracovat s jinými typy souborů, na řadu se dostávají tzv. loadery. Ty umožňují převést např. TypeScript na JavaScript a vytvořit validní moduly, se kterými webpack dále pracuje. Webpack ale pomocí loaderů dokáže provádět různé činnosti se soubory rozmanitých typů od JavaScript knihoven, CSS preprocesorů po SVG a spoustu dalších. Dokumentace webpack doporučuje nastavení loaderů provádět v konfiguračním souboru, poskytuje však i další možnosti. Při jednoduchém popisu nastavení, vlastnosti *test* a *use* určují, s jakým typem souborů bude webpack pracovat a jaký loader bude použit pro manipulaci s danými soubory. Loadery jde společně řetězit, přičemž první předává svůj výsledek následujícímu, jako výstup z posledního z nich webpack očekává JavaScript.

Pluginy

Mezitím co se loadery využívají pro transformace určitých typů, pluginy lze využít pro komplexnější úlohy, jako např. optimalizaci balíčků, správu příloh či připojení nastavení prostředí. Samotný webpack funguje na stejném systému, jaký se používá při konfigurací pluginů. Z toho vyplývá fakt, že většina pluginů jde přizpůsobit pomocí nastavení, to se provádí stejně jako u loaderů v konfiguračním souboru. Jelikož lze konkrétní plugin využít opakovaně s vlastní konfigurací pro různé účely, je potřeba při vytvoření nové instance použít klíčové slovo *new*.

2 Systém pro správu obsahu

Content management system, v češtině nazývaný jako systém pro správu obsahu či redakční systém, je softwarová aplikace, která je určena primárně pro vytváření, úpravu a organizaci digitálního obsahu. Pro práci s CMS není potřeba mít technické znalosti v oboru vytváření webových stránek. Účelem takových aplikací je poskytnout intuitivní, uživatelsky přívětivé prostředí a nechat své uživatele se soustředit pouze na práci s obsahem webových stránek. Jednou z klíčových dovedností těchto systémů je možnost publikovat vytvořený obsah ihned po jeho dokončení. Jednotlivé CMS se od sebe odlišují zejména rozsahem řešení, architekturou, stylem doručování obsahu a svojí cílovou skupinou [24].

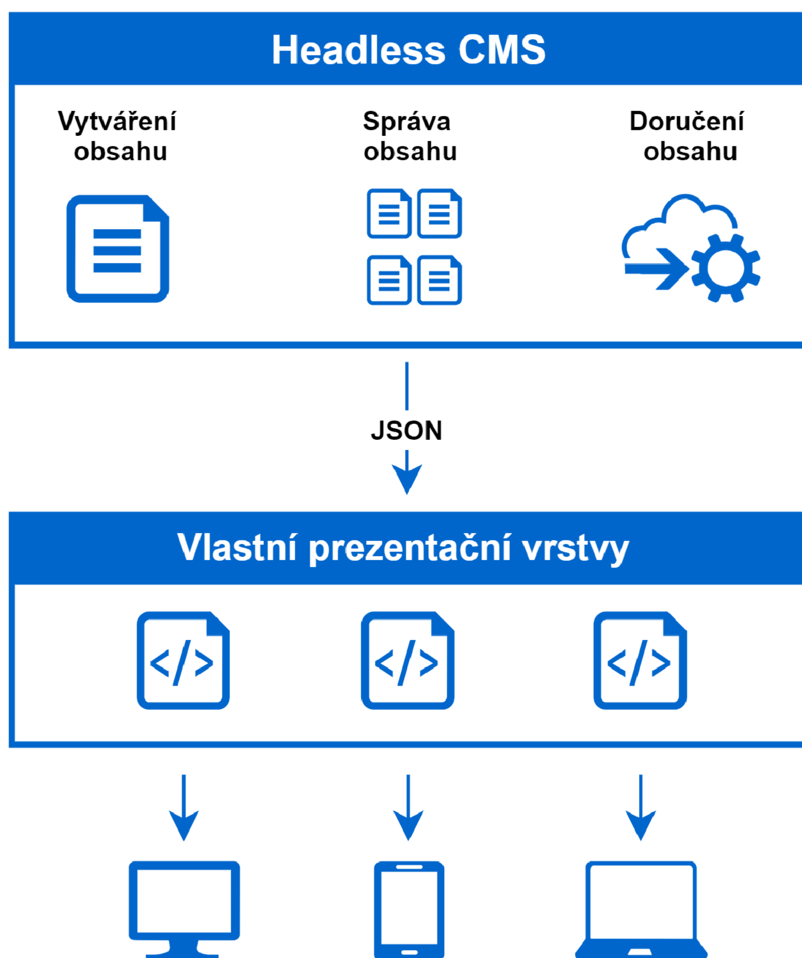
2.1 Headless CMS

Headless CMS je typ řešení redakčního systému, který neposkytuje prezentační vrstvu. Aplikace tedy slouží jako repozitář, ve kterém lze vytvářet a spravovat digitální obsah. Ten je dále distribuován prostřednictvím REST API. Kvůli absenci prezentační vrstvy má uživatel větší volnost v tom, kde a jak budou data zobrazena. Tento princip je zobrazen na obrázku 2.1. Jednou ze stěžejních výhod headless CMS je tedy, že stejný obsah je možné zobrazit na webové stránce, v mobilní aplikaci či v jakémkoli IoT zařízení. Jinými slovy se headless CMS soustředí zejména na uchovávání a distribuci strukturovaného obsahu.

Jednou z metod poskytování headless CMS je tzv. Software as a service, známá pod zkratkou SaaS. Jde o přístup, kdy hostování aplikace zajišťuje společnost, která danou službu vyvíjí, nebo je tato funkce svěřena třetí straně. Uživatelé tohoto typu služeb se tak nemusejí starat o instalaci, aktualizace a zabezpečení softwaru, jelikož je všem poskytována jediná, aktuální verze.

2.2 Kentico Kontent – zástupce headless CMS

Kentico Kontent je webová aplikace vyvíjená ve společnosti Kentico, která byla založena v roce 2004 Petrem Palasem. Hlavním účelem aplikace je poskytnout svým uživatelům komplexní nástroj, ve kterém lze na jednom místě vytvářet, spravovat a publikovat digitální obsah určený pro větší množství kanálů. Použití jediného nástroje pro správu digitálního obsahu zamezuje jeho duplicitě, která může vznikat při decentralizované správě jednotlivých kanálů.



Obr. 2.1: Obecný princip headless CMS.

2.2.1 Uživatelské rozhraní aplikace

Aplikace je členěna do několika celků, první z nich se nazývá *Content & Assets*. V části *Content* se nachází inventář, ve kterém jsou seskupeny položky, které se nazývají *Content Items*. Konkrétní *Content Item* může reprezentovat např. článek blogu nebo položku z digitálního katalogu/e-shopu obsahující vyčerpávající popis produktu včetně multimediálního obsahu. Díky pokročilé možnosti filtrování a vyhledávání lze v inventáři, obsahujícím stovky či tisíce položek, vyhledat konkrétní skupinu nebo také jediný unikátní *Content Item*. Kliknutím na položku v inventáři je uživatel přesměrován do editoru, ve kterém vytváří obsah konkrétní položky, jako na obrázku 2.2.1.

Jednotlivé položky na sebe můžou vzájemně odkazovat nebo do sebe mohou být vnořeny. V části *Assets* se nachází přehled všech příloh, které jsou uloženy v aplikaci. Na tomto místě lze nalézt jak dokumenty, tak vizuální obsah v různých formátech a další.

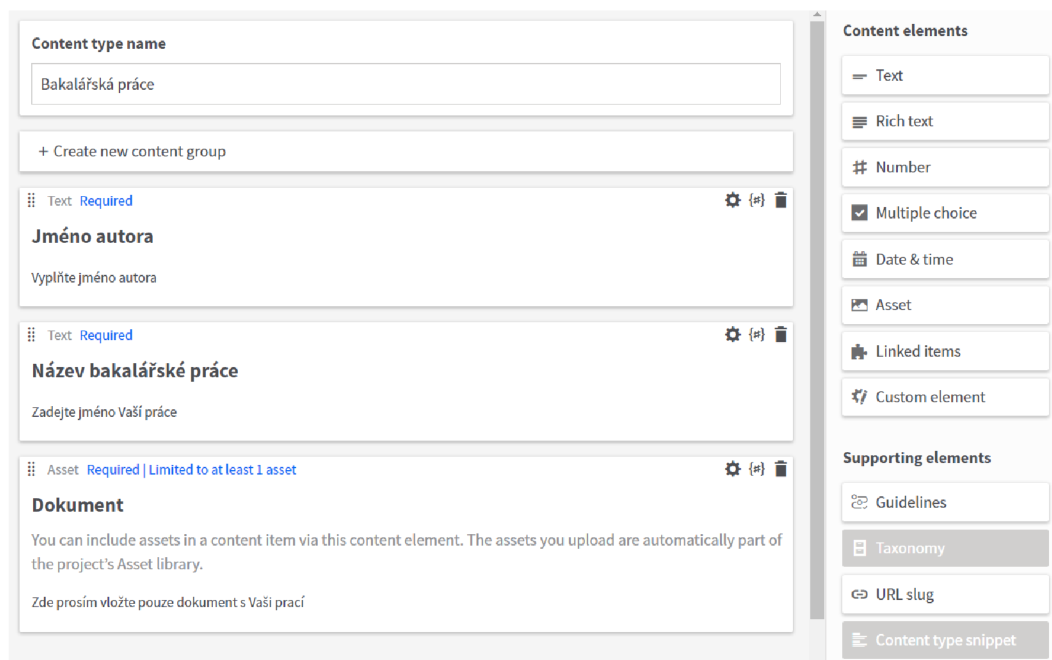
Obr. 2.2: Vytváření obsahu v Content Item editoru.

Druhý celek s názvem *Content models* slouží k modelování šablon a struktur, ze kterých jsou tvořeny jednotlivé *Content Items*. *Content models* se vytvářejí z předem připravených *Content elements*, tak jako na obrázku 2.2.1. Zástupcem těchto elementů může být například textové pole, rich text editor, kontejner pro přílohy a další. Díky tomu lze vytvořit unikátní struktury a záleží jen na uživateli aplikace, jak rozsáhlé jednotlivé modely budou.

Další celek nese název *Project settings*. Zde uživatel, nejčastěji správce nebo manažer projektu, provádí režijní změny, jako například správu jednotlivých uživatelů. Dále také vytváření a přidělování rolí jednotlivým uživatelům – role je skupina pravomocí, které konkrétní uživatel v rámci projektu má. Jednotlivým rolím lze nastavit široké spektrum pravomocí, nejčastěji se jedná o permutace práv pro zobrazení, vytvoření, úpravu nebo smazání jemu přidělených nebo veškerých modelů a položek inventáře. Dále lze uživateli přidělit omezené nebo žádné pravomoce pro nastavení projektu.

Taky lze nastavit tzv. *Workflow steps* – jedná se o plně konfigurovatelný proces, který rozhoduje o tom, v jakém stavu se daná položka z inventáře nachází. Pro zjednodušení si můžeme uvést příklad, kde budou definovány celkem tři stavy procesu - Draft, Revize a Publikováno. Při vytváření obsahu se konkrétní *Content Item* nachází ve stavu Draft, poté přejde do stavu Revize, kde je zkontrolován, a až poté se dostane do stavu Publikováno. Odtud je prostřednictvím Content Management API doručen do jednotlivých kanálů ve formátu JSON.

Samotné možnosti aplikace Kentico Kontent jsou daleko rozsáhlejší, jejich detailní popis je však nad rámec této práce.



Obr. 2.3: Vytváření modelu v Content Model editoru.

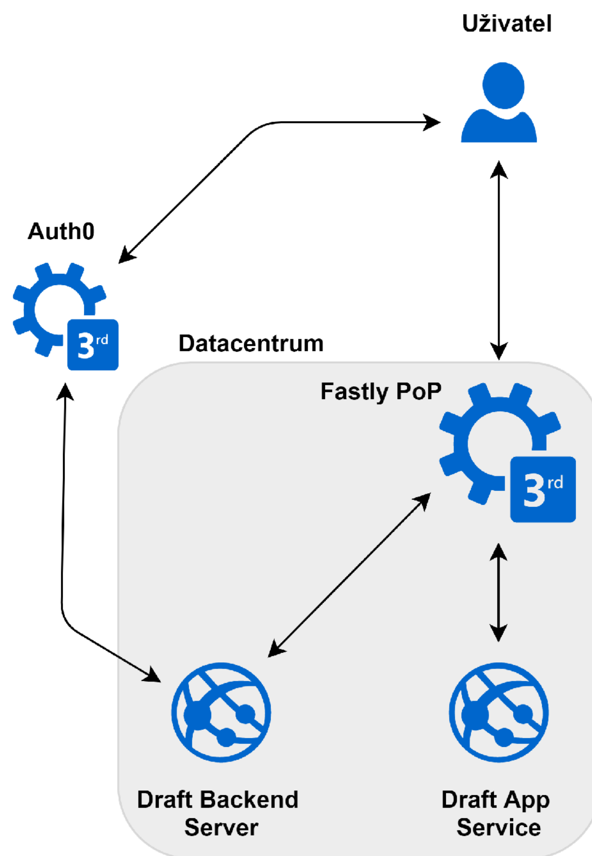
2.3 Infrastruktura aplikace Kentico Kontent

Infrastruktura aplikace Kentico Kontent je rozsáhlá, pro účel této práce se však zaměříme jen na vrstvy přímo související s částí, která se nazývá Draft App Service. Na schématu 2.4 vidíme zjednodušený model komunikace mezi uživatelem (klient) a aplikací(server). Šedě vyznačená část označuje, že tyto služby jsou umístěny v datovém centru, ty jsou strategicky rozmístěny na různých kontinentech. Důvod pro využití datového centra je prostý – přesunout fyzický server, na kterém se nachází kopie služeb, co nejblíže k uživateli.

Draft App Service

Zde se nachází single page aplikace interně pojmenovaná jako Draft Client. Ta je postavena na populární JavaScript knihovně React.js. Kromě této knihovny jsou pro vývoj využívány desítky dalších modulů lišící se jak velikostí, tak svým účelem, např. Typyscript – JavaScript nadstavba nabízející množství výhod potřebných pro udržitelný vývoj či Redux – knihovna pro správu globálního stavu aplikace využívající architektury Flux.

Dále je zde server Express.js, jedná se o Node.js framework, který pracuje s HTTP požadavky přicházejícími skrz Fastly od uživatelů a pro jednotlivá URL má připravenou odpověď v podobě HTML, do kterého jsou připojeny přílohy.



Obr. 2.4: Zjednodušený model infrastruktury aplikace Kentico Kontent související s vrstvou Draft App Service.

Draft Backend Server

Draft Backend Server reprezentuje .NET MVC aplikaci, která se nazývá Draft Backend. Jedná se o jednu z mikroslužeb hostovaných platformou Microsoft Azure. Pro uchovávání dat se zde využívá NoSQL databáze CosmosDB. Mikroslužby jsou samostatné části starající se o určitou část logiky aplikace. Kromě této mikroslužby se v infrastruktuře aplikace Kentico Kontent nachází např. Subscription Service, ta spravuje uživatelské účty a jejich plány, či Notification Service a další. Tyto vrstvy však nejsou na obrázku 2.4 z důvodu zjednodušení. Komunikaci a přenos dat mezi mikroslužbami zajišťuje Azure Service Bus.

Fastly

Fastly je CDN služba třetí strany. Jejím primárním účelem je zefektivnění přenosu požadavků a odpovědí mezi uživatelem a aplikací. To je umožněno díky ukládání statického obsahu do cache paměti Fastly a přesměrováváním API požadavků. Data

jsou uložena na více serverech v tzv. datových centrech, která jsou rozmístěna různě po světe. Využívá se vždy datové centrum nacházející se geograficky nejblíže k uživateli.

V kontextu aplikace Kentico Kontent to pak znamená, že statické soubory jako např. CSS a Javscript z *Draft App Service* jsou uloženy do cache paměti Fastly a požadavky na *Draft Backend Server* jsou přesměrovány.

Auth0

Auth0 je služba třetí strany zajišťující ověření a oprávnění přístupu uživatele do aplikace. Auth0 využívá protokol OAuth2.0, k získání přístupu se využívají přístupové tokeny. Aplikace obdrží token tak, že po úspěšném přihlášení na hostované stránce je uživatel přesměrován na adresu *app.kontent.ai/callback*, kde je přístupový token uložený do místního úložiště prohlížeče. Následuje další přesměrování na domovskou stránku aplikace, zde si aplikace vyzvedne token a uloží si ho do globálního stavu.

Platnost přístupového tokenu po čase vyprší, a proto je nutné ho obnovovat. Díky Auth0 je možné využívat metody single sign-on, jde o jednotné přihlášení do více na sobě nezávislých aplikací.

2.4 Počáteční nastavení aplikace

V této podkapitole je popsáno počáteční nastavení, které je podstatné pro pochopení aspektů prováděných optimalizací.

2.4.1 webpack

Pro sestavení aplikace se využívá nástroj webpack. Nastavení tohoto nástroje probíhá v konfiguračním souboru *webpack.config.js*. Od verze 4 lze spustit sestavení i bez nutnosti konfigurace, nicméně s výchozím nastavením by si tato aplikace zdaleka nevystačila.

Vstupní a výstupní body

V konfiguraci je definováno celkem pět vstupních bodů. Každý vstupní bod reprezentuje TypeScript soubor, do kterého jsou ve stromové struktuře připojeny jednotlivé React komponenty, do nichž se připojují další závislosti. Obsah vstupních bodů není srovnatelně velký, naopak jeden z nich, pojmenovaný jako *client*, je větší než zbylé čtyři dohromady. Ten obsahuje téměř všechny React komponenty a další s nimi související kód, které vytvářejí uživatelské rozhraní aplikace Kentico Kontent. Zbylé

vstupní body zastupují k aplikaci přidružené podstránky sloužící pro vytvoření uživatelského účtu, ověření e-mail adresy, autorizaci přihlášení uživatele či konfiguraci vzorové stránky.

```
const entries = {
  authorizationHandler: './server/views/authorizationHandler/
  scripts/authorizationHandler.ts',
  client: getClientEntry(config),
  sampleSiteConfiguration: './server/views/sampleSiteConfiguration/
  scripts/sampleSiteConfigurationMain.tsx',
  verifyEmailPage: './server/views/verifyEmailPage/scripts/
  verifyEmailPageMain.tsx',
  welcomePage: './server/views/welcomePage/scripts/
  welcomePageMain.tsx',
};
```

Výpis 2.1: Vstupní body aplikace v konfiguračním souboru webpack.config.js.

Z každého vstupního bodu je vytvořen jediný soubor, ten je transpilován z TypeScript na JavaScript. Tento soubor obsahuje veškerý JavaScript kód, jak vlastní, tak knihovny a balíčky třetích stran. V uvedené části kódu 2.2 je definována cesta k adresáři s webpack výstupem.

`Output.filename` může obsahovat přímo jméno výstupního souboru nebo, jako v tomto případě lze využít substituce, kdy o jménu souboru rozhodne webpack. Pokud webpack sestavuje produkční verzi, připojí na konec jména souboru hash řetězec, který identifikuje verzi souboru. Pokud se při novém sestavení aplikace obsah souboru změnil, je vygenerován nový hash. Toto je nezbytná součást nastavení, pokud chceme efektivně využívat cache paměť webového prohlížeče.

`Output.pathinfo` rozhoduje o tom, zda se k výstupu připojí i tzv. source maps. Tyto soubory jsou užitečné zejména pro vývoj, jelikož jsou nositelem informací o původní struktuře před provedením transformace. Debugger v prohlížeči pak dokáže zobrazit rekonstruovaný originál, což je užitečné pro hledání chyb a ladění aplikace. Je doporučeno, aby sestavená verze pro produkci tento typ souboru neobsahovala.

```
output: {
    path: path.join(__dirname, 'public/build'),
    publicPath: '/build/',
    filename: config.buildForSharedEnvironment ?
    '[name]_[hash].js' : '[name].js',
    pathinfo: config.compileSourceMaps,
  },
```

Výpis 2.2: Nastavení pro výstupní bod aplikace v konfiguračním souboru webpack.config.js.

Loadery

Konfigurační soubor obsahuje také nastavení loaderů pro práci s Typescript, SVG, Less a CSS soubory. To, k čemu webpack loadery slouží, bylo vysvětleno v kapitole 1.5.1.

Optimalizace

V konfigurační části pro optimalizaci se v původním nastavení využívá TerserWebpackPlugin [25], ten slouží pro minimalizaci JavaScript souborů. To provádí tak, že ve výstupním souboru, který sestaví webpack, zkracuje názvy např. funkcí a proměnných, odebírá mezery pro formátování zdrojového kódu atp.

Dále zde najdeme OptimizeCSSAssetsWebpackPlugin [26], ten plní velmi podobnou funkci jako předchozí zmíněný plugin, avšak minimalizuje výstupní CSS kód.

Zmíněné optimalizační úpravy mají kladný dopad na výslednou velikost příloh, jelikož jsou z nich vypuštěné zbytečné znaky a mezery. Jejich aplikováním je zmenšení objemu dat, které je nutno ze serveru přenést ke klientovi. Zároveň však velmi ztíží čitelnost ve výstupních souborech, proto jsou tyto změny aplikovány pouze pro produkční verzi, kde tato skutečnost nevytváří překážku.

2.4.2 HTTP komprese

Za účelem zmenšení objemu přenesených dat ze serveru ke klientovi se také využívá komprese HTTP. Jde o metodu, kdy jsou přenášená data před odesláním na straně serveru zkomprimována. Tento typ optimalizace musí podporovat webový prohlížeč. Do požadavku, který odesílá, přidá hlavičku *Accept-encoding*. Ta informuje server, jaké kompresní algoritmy podporuje. Po obdržení odpovědi webový prohlížeč doručená data dekomprimuje. Zda byla přenesená data komprimována lze zjistit například nástrojem Chrome DevTools.

V HTTP odpovědi pro konkrétní přílohu se nachází hlavička *x-content-encoding-over-network*, hodnota v hlavičce informuje o aplikovaném kompresním algoritmu. Pro aplikaci Kentico Kontent se využívá algoritmu *gzip*.

2.4.3 HTTP caching

Caching je metoda, při které se ukládají kopie externích příloh obsahující např. JavaScript kód do mezipaměti, nazývané také jako cache pamět prohlížeče. Pokud je požadovaná příloha uložena v paměti, HTTP požadavek vrátí kopii uloženou v paměti na místo toho aby ji znovu stahoval ze serveru. Tato technika pomáhá jak zmenšit objem přenesených dat mezi serverem a klientem, tak zkrátit čas potřebný k načtení aplikace.

Pro nastavení mechanismu ukládání příloh do mezipaměti prohlížeče je využívána HTTP hlavička *cache-control*. Pomocí jejich vlastností lze nastavit např. kde a jak dlouho mají být přílohy uchovávány. Pro JavaScript přílohy aplikace Kentico Kontent je v hlavičce *cache-control* nastavená hodnota *public*, která povoluje ukládat přílohy do kteréhokoliv typu cache paměti prohlížeče. Vlastnost *max-age* pak definuje maximální dobu, jak dlouho má být uložena příloha považována za aktuální.

K ověření se používá hodnota vlastnosti hlavičky *cache-control* nazývaná *e-tag*, ta má v sobě uložený unikátní řetězec. Po uplynutí doby uvedené v *max-age* je nutné ověřit platnost přílohy, k tomu se využívá vlastnost *If-None-Match*. Na server je odeslán požadavek a proběhne porovnání řetězce přílohy z mezipaměti a přílohy uložené na serveru. Pokud nedošlo ke změně, server vrátí odpověď 304 a příloha je považována za aktuální [27].

Uložená příloha se však může stát nerelevantní daleko dříve. Pokud v příloze dojde ke změně, je nutno ji stáhnout ze serveru znovu a v mezipaměti prohlížeče obnovit její kopii. K ověření se v názvech příloh aplikace Kentico Kontent používá unikátní řetězec, který generuje webpack. Pokud dojde ke změnám, při sestavení aplikace se změní i řetězec v názvu výstupního souboru. Je důležité, aby pro HTML šablonu, do které jsou připojeny přílohy, byla hlavička *cache-control* nastavena na

hodnotu *no-cache*. V opačném případě by ukládání HTML šablony způsobovalo problémy s neaktuální verzí příloh.

2.4.4 Nedostatky počátečního stavu

V následující podkapitole je věnována pozornost nedostatkům původního nastavení, které budou následně optimalizovány.

Neefektivní využití cache paměti prohlížeče

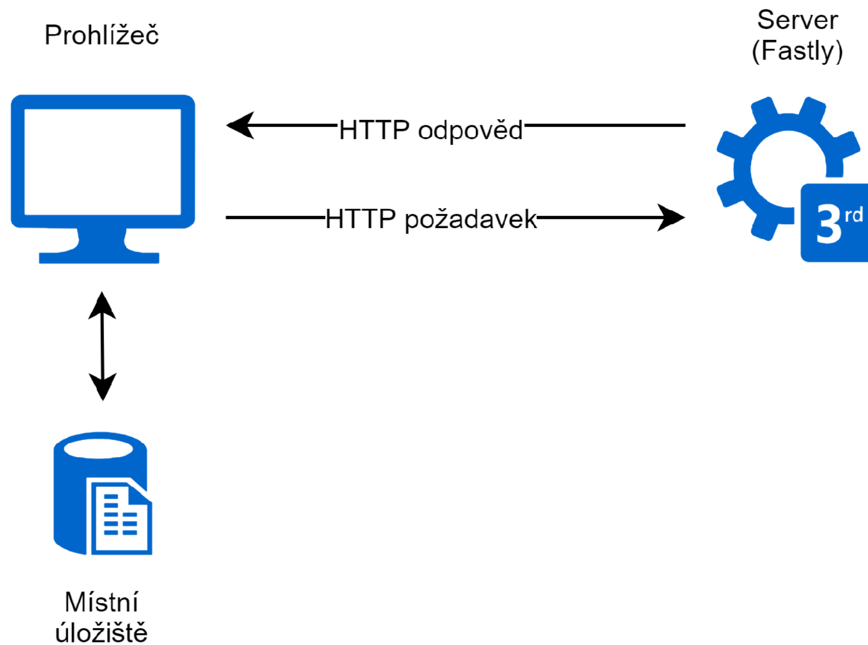
Použít pouze jeden výstupní soubor pro konkrétní část aplikace obsahující veškerý JavaScript kód s sebou přináší určitá omezení a nevýhody. JavaScript kód aplikace Kentico Kontent lze rozdělit do dvou kategorií – kód vytvořený vývojáři této aplikace, dále nazýván jako vlastní kód, a veřejně dostupné knihovny usnadňující vývoj single page aplikací.

Zdrojový kód knihoven po instalaci do aplikace zůstává bez zásahu po celou dobu jejich využívání. Osvědčený postup je nezanášet do těchto knihoven vlastní změny, s velkou pravděpodobností by těmito zásahy byla znemožněna budoucí aktualizace na novější verzi. Proto je manipulace s nimi omezena pouze na změnu verze nebo úplné odebrání z aplikace. Na druhou stranu část kódu vytvořená ve společnosti Kentico podléhá změnám téměř každý den.

Udržovat tak v jediném souboru tyto dva typy zdrojových kódů přináší nevýhodu v podobě neefektivního využívání cache paměti webového prohlížeče. Statické soubory s JavaScript zdrojovými kódy můžou být při prvním načtení aplikace uloženy do paměti prohlížeče a využity při opětovném načtení aplikace, jak lze vidět na obrázku 2.5. To přináší výhody v podobě zkrácení času načítání aplikace a zmenšení objemu přenášených dat ze serveru k uživateli.

O režii uchovávání těchto souborů se stará hlavička HTTP protokolu zvaná *cache-control*. Prostřednictvím *cache-control* lze nastavit, jak se bude prohlížeč chovat k danému typu přílohy. Podobněji byl tento princip popsán v podkapitole 2.4.3. Kromě JavaScript souborů, lze uchovávání příloh v cache využít také pro soubory CSS, obrázky, ikony a další.

Jedno z řešení problému s neefektivním využíváním cache paměti prohlížeče je rozdělení výstupního JavaScript souboru na několik menších celků. Zejména pak oddělení zdrojového kódu, jenž podléhá změnám na denní bázi, od knihoven a balíčků, které mohou zůstat beze změn po dobu několika týdnů.



Obr. 2.5: Komunikace klienta se serverem, kdy je využito místního úložiště, odkud může být načtena část příloh nutných pro spuštění aplikace.

Chybějící asynchronní načítání

Po přihlášení do aplikace je uživatel přesměrován na domovskou stránku. Pokud je veškerý JavaScipt připojen do aplikace v jediném souboru, je potřeba stáhnout a parsovat ho celý. V ideálním případě by měl webový prohlížeč stáhnout jen kód, který je potřeba pro část aplikace, ve které se zrovna uživatel nachází. React.js nabízí možnost asynchronního načítání částí aplikace využitím metody `React.lazy`. Díky ní lze asynchronně načítat JavaScript pro danou část aplikace až v momentu, kdy je do ní uživatel navigován.

S rostoucí komplexitou aplikace začíná být její rozdělení na dynamicky načítané moduly velmi obtížné. Je možné narazit na technické limity vyžadující velké množství změn, které mohou být časově náročné, popř. s sebou přinést další omezení.

3 Měření a aplikované optimalizace

Pro účely měření byl využit nástroj zabudovaný v prohlížeči Google Chrome, který se nazývá Chrome DevTools. Ten poskytuje podrobné informace o přenosu dat, počtu požadavků, dokáže provést simulaci načítání aplikace na různém typu připojení a také s omezeným výkonem procesoru. Pro účely měření bylo vytvořeno testovací prostředí s platným SSL certifikátem, které je srovnatelné s prostředím použitým pro produkční verzi aplikace Kentico Kontent. Dále bylo nutné stanovit výchozí rychlost pro LTE připojení, aby podmínky pro měřené scénáře byly vyrovnané. Pojmem scénář je zde označen soubor změn, které byly provedeny v aplikaci za účelem optimalizace načítání aplikace.

3.1 Měření jednotlivých scénářů

Díky možnosti přidat si do nástroje Chrome DevTools vlastní profil pro simulaci rychlosti mobilního připojení byly stanoveny hodnoty pro LTE připojení na **3,5 Mbps** pro stahování a **5,0 Mbps** pro odesílání. Hodnoty byly získány experimentálním měřením pomocí chytrého telefonu s aplikací Speedtest.

Reálná rychlost mobilního LTE připojení se může výrazně lišit. Faktory ovlivňující rychlost a stabilitu připojení závisí na poskytovateli mobilního připojení, pokrytí oblasti, ve které se uživatel nachází, aktuálního vytížení mobilní sítě atd. Za účelem objektivního zhodnocení výsledků jednotlivých scénářů bylo důležité pro všechna měření stanovit stejné podmínky, na kterých bude jasně viditelný rozdíl postupně aplikovaných optimalizací.

Pro všechny scénáře byly provedeny tyto typy měření:

- **První načtení** – Uživateli se načítá aplikace bez uložených příloh. Stává se tak vždy, když uživatel navštíví aplikaci poprvé nebo pro ni v mezipaměti prohlížeče nejsou uložena data.
- **Načtení s využitím mezipaměti** – Přílohy mohou být uloženy v mezipaměti prohlížeče, to urychlí načtení aplikace a ze serveru je k uživateli přeneseno menší množství dat. Tento typ měření simuluje stav, kdy jsou všechny uložené přílohy aktuální.
- **První načtení, LTE** – Stejný typ měření jako v prvním bodě, ale s využitím simulace LTE připojení.
- **Načtení s využitím mezipaměti, LTE** – Stejný typ měření jako v druhém bodě, ale s využitím simulace LTE připojení.

- **Změna ve vlastních JavaScript přílohách** – Načtení aplikace poté, co v JavaScript kódu proběhla jakákoliv změna. Tento typ měření se snaží simulovat stav, kdy jsou do produkční verze aplikace přidány aktualizace. Ty se zde dostávají buď jednou za dva týdny v pravidelném vydání, nebo pomocí tzv. hotfixu. Obsahem hotfixu jsou nejčastěji opravy chyb či drobné změny, se kterými není třeba čekat do doby pravidelného vydání. Běžně se tak stává, že během zmíněných dvou týdnů je vydáno větší množství hotfixů. Vlastní kód aplikace tak podléhá změnám na denní bázi.
- **Změna ve vlastních JavaScript přílohách, LTE** – Stejně jako v předchozím scénáři, pouze s využitím simulace mobilního LTE připojení.

Pro každý scénář byly zmíněné typy měření provedeny v několika iteracích, z naměřených hodnot byl vypočten medián. Mediány sledovaných hodnot jsou uvedeny v tabulkách pro každý scénář samostatně v podkapitolách s názvem Výsledky měření. Záznam všech získaných hodnot je uveden v příloze A.1.

Při každém měření byly sledovány následující hodnoty:

- **Objem přenesených dat** – kolik dat bylo nutné stáhnout pro načtení a spuštění aplikace, zvlášť je rozdělen objem JavaScript souborů od zbytku
- **DOMContentLoad** – označuje, za jak dlouho je připraven DOM, tedy moment, kdy jsou staženy statické přílohy a nic neblokuje vykonání JavaScript kódu
- **Load** – označuje moment, kdy je stránka plně načtena a je zobrazeno uživatelské rozhraní. Počítají se zde stažené CSS a JS přílohy, obrázky a externí statické přílohy
- **Počet požadavků** – označuje počet vytvořených požadavků, zvlášt je rozdělen potřebný počet JavaScript požadavků od zbytku

3.2 Původní stav aplikace

Name	Protocol	Size ▼	Time	Cache-Control
client_f02e62f07b834fd9e5bd.js	h2	6.0 MB	1.13 s	public,max-age=2592000
player.63006c623ecbc7970f416113465ed960.bare.js	h2	1.9 MB	706 ms	max-age=120, public
frame-modern.bc8731e1.js	h2	62.3 kB	36 ms	max-age=31536000, s-maxage=7200, public
vendor-modern.be979053.js	h2	51.7 kB	43 ms	max-age=31536000, s-maxage=7200, public
gtm.js?id=GTM-P48NWGF>m_auth=k8EqBGxue...	http/2+quic/46	29.5 kB	55 ms	no-cache, no-store, must-revalidate
vendors~sentry-modern.e2185cae.js	h2	23.4 kB	31 ms	max-age=31536000, s-maxage=7200, public
analytics.js	http/2+quic/46	18.5 kB	31 ms	public, max-age=7200
amplitude-5.3.0-min.gz.js	h2	18.3 kB	32 ms	max-age=31536000
bat.js	h2	7.7 kB	51 ms	private,max-age=1800
shim.latest.js	h2	3.3 kB	32 ms	max-age=300, s-maxage=300, public
sentry-modern.a4036dc5.js	h2	1.9 kB	29 ms	max-age=31536000, s-maxage=7200, public

11 / 61 requests | 8.1 MB / 8.7 MB transferred | 8.6 MB / 9.2 MB resources

Obr. 3.1: Chrome DevTools – jednotlivé JavaScript požadavky při původním stavu aplikace.

Na obrázku 3.1 lze vidět výstup z nástroje Chrome DevTools, konkrétně veškeré JavaScript přílohy, které byly staženy při načítání aplikace Kentiko Kontent ve stavu před aplikováním optimalizací. Kromě souboru *client*, který obsahuje knihovny třetích stran a vlastní kód aplikace, můžeme vidět i požadavky na další přílohy. Ty obsahuji kód nutný pro funkci nástrojů jako Amplitude nebo Intercom. Většina z těchto příloh má zanedbatelnou velikost, kromě nástroje Inline manual.

Jde o velmi užitečný nástroj, ve kterém lze vytvářet interaktivní návody přímo uvnitř aplikace, které krok za krokem provedou uživatele při náročnějších úkonech. Pro nové uživatele aplikace je tento nástroj velmi užitečný. Bohužel, příloha se zdrojovým kódem nástroje má velikost 1,9 MB. Jelikož je tato příloha uložena přímo na serveru poskytovatele tohoto nástroje, není možné na ni uplatnit jakoukoliv vlastní optimalizaci.

3.2.1 Výsledky měření

Tab. 3.1: Naměřené hodnoty pro počáteční stav aplikace.

Typ měření	Přeneseno [kB]		DOM ContentLoad [s]	Load [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
První načtení	8100	8700	2,82	3,37	11	62
Načtení s cache	29,9	90,4	1,23	1,55	11	62
První načtení, LTE	8100	8700	19,81	20,20	11	62
Načtení s cache, LTE	32,60	99,40	1,47	1,94	11	63
Změna v JS příloze	6000	6500	2,49	2,79	9	54
Změna v JS, LTE	6000	6500	16,13	16,67	9	54

Jak bylo popsáno výše, na některé přílohy nelze uplatnit optimalizace a momentálně je třeba počítat s jejich velikostí. Se souborem *client*, jehož velikost je 6,0 MB, však můžeme manipulovat i provádět optimalizace. Právě to si kladou za cíl následující kapitoly. V tabulce 3.1 jsou hodnoty získané měřením, popsaném v kapitole 3.1, které budou složit jako výchozí hodnoty pro porovnávání s hodnotami získanými po provedení jednotlivých optimalizací. Všechny naměřené hodnoty lze nalézt v příloze A.1.1.

V přílohách s výsledky si lze u typu měření *Změna ve vlastních JavaScript přílohách* a *Změna ve vlastních JavaScript přílohách, LTE* všimnout velmi dlouhé doby načítání aplikace při prvním pokusu měření. To je způsobeno tím, že po nahrání nové verze aplikace je nutnou přílohy z *Draft App Service* dostat do cache paměti *Fastly*, viz. obrázek 2.4. Právě první uživatel, který navštíví doménu aplikace Kentico Kontent, inicializuje tento proces. Po jeho dokončení každý další uživatel již dostává HTTP odpověď s přílohami z *Fastly* bez zbytečných prodlev.

3.3 Rozdělení knihoven a vlastního kódu do samostatných souborů

Pro rozdělení výstupního JavaScript kódu do samostatných souborů bude v tomto i ve všech následujících scénářích využit nativní webpack nástroj, který se nazývá SplitChunksPlugin, s vydáním čtvrté verze webpack tento nástroj nahrazuje původní CommonChunkPlugin. Díky jeho rozsáhlému rozhraní lze do značné míry ovlivnit způsob, jakým je JavaScript kód rozdělen.

3.3.1 Provedené změny

Jednou ze změn tohoto scénáře bylo aplikovat specifické nastavení pro SplitChunksPlugin, který umožní rozdělit výstupní JavaScript soubor na dva, kdy jeden obsahuje pouze kód knihoven a balíčků třetích stran a druhý vlastní JavaScript kód vytvořený vývojáři aplikace.

```
1  optimization: {
2    splitChunks: {
3      chunks: 'async',
4      automaticNameDelimiter: '-',
5      cacheGroups: {
6        vendors: {
7          test: /[\\/]node_modules[\\/]/,
8          chunks: 'all',
9          name: 'vendors',
10       },
11     },
12   },
13 }
```

Výpis 3.1: SplitChunksPlugin – rozdělení na dva výstupní soubory, kdy jeden obsahuje všechny knihovny a balíčky třetích stran, druhý veškerý vlastní kód aplikace.

Ve výpisu 3.1 můžeme vidět nastavení pro aktuální scénář. Obecné nastavení `chunks: 'async'` zajistí, aby všechen vlastní kód aplikace, který používá statické připojování závislostí, byl vložen do jediného výstupního souboru. Parametr `cacheGroups` však povoluje definovat určité skupiny, pro které budou aplikovány jiná než výchozí pravidla. Na šestém řádku ve výpisu 3.1 je definována skupina `vendors`, regulární výraz v parametru `test` zajišťuje, že tato pravidla budou aplikována na veškerý obsah složky `node_modules`. Parametr `chunks: 'all'` pak zaručí, že dynamicky i

staticky připojené moduly budou spojeny do vlastního výstupního souboru. Parametr `name` nastavuje výchozí jméno souboru, ke kterému může být připojen i hash identifikátor. To však záleží na tom, pro jaké prostředí bude aplikace sestavena.

3.3.2 Výsledky měření

Tab. 3.2: Naměřené hodnoty pro scénář, ve kterém jsou přílohy rozděleny na dva soubory.

Typ měření	Přeneseno [kB]		DOM ContentLoad [s]	Load [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
První načtení	8100	8700	2,48	2,78	12	63
Načtení s cache	30,00	90,50	1,21	1,46	12	63
První načtení, LTE	8100	8700	19,50	19,89	12	63
Načtení s cache, LTE	29,90	90,40	1,36	1,53	12	64
Změna v JS příloze	4100	4600	2,28	3,01	10	57
Změna v JS, LTE	4100	4600	11,39	11,72	10	56

Pokud porovnáme hodnoty naměřené při tomto scénáři a uvedené v tabulce 3.2 s výchozími hodnotami z tabulky 3.1, lze si všimnout menšího objemu přenesených dat a rychlejšího načtení stránky při měření typu změna v JavaScript příloze a změna v JavaScript příloze se simulací LTE připojení.

Díky provedeným změnám ve webpack nastavení zde může být efektivněji využita mezipaměť prohlížeče, ten pak potřebuje aktualizovat jen soubor s vlastním JavaScript kódem aplikace. Příloha s JavaScript balíčky třetích stran zůstala beze změny, a proto je možné použít její předchozí verzi z cache paměti webového prohlížeče. Všechny naměřené hodnoty lze nalézt v příloze A.1.2.

3.4 Samostatný soubor pro každý balíček třetí strany

V předchozím scénáři byly provedeny úpravy pro rozdělení výstupních JavaScript souborů na dva – jeden s knihovnami třetích stran a druhý s vlastním kódem. Další optimalizací směřovanou na efektivnější využití cache paměti prohlížeče by mohlo být vytvoření výstupního souboru pro každou knihovnu třetí strany ze složky *node_modules*. To s sebou přináší výzvu v podobě dynamického připojování výstupních JavaScript souborů do HTML dokumentu.

3.4.1 Provedené změny

V konfiguračním souboru pro webpack byly aplikovány změny uvedené ve výpisu 3.2.

```
1  optimization: {
2    splitChunks: {
3      chunks: 'async',
4      maxInitialRequests: Infinity,
5      minSize: 0,
6      cacheGroups: {
7        vendors: {
8          chunks: 'all',
9          test: /[\\/]node_modules[\\/]/,
10         name(module) {
11           const packageName = module.context.match(
12           /[\\/]node_modules[\\/](.*?)([\\/]|$)/)[1];
13           return `npm_${packageName.replace('@', '')}`;
14         },
15       },
16     },
17   },
18 }
```

Výpis 3.2: SplitChunksPlugin – rozdělení JavaScript příloh na jeden soubor obsahující vlastní kód aplikace a jeden soubor pro každou knihovnu či balíček třetí strany.

Ve výchozím nastavení SplitChunksPlugin vytváří omezený počet výstupních souborů, kdy velikost vytvořeného souboru není před minimalizací větší než 30kB [28]. Parametry `maxiInitialRequests` a `minSize` s hodnotami uvedenými ve výpisu 3.2 je přepsáno výchozí nastavení pro počet souborů a jejich minimální velikost. Dále je pomocí parametru `cacheGroups` definována skupina obdobně jako při předchozím scénáři. Rozdílné je zde však nastavení jména pro výstupní soubory. Funkce

začínající na desátém řádku výpisu zajišťuje, že pro každou knihovnu nebo balíček z adresáře `node_modules` bude vytvořen samostatný výstupní soubor. Podobné nastavení pro webpack je popsáno v článku Davida Gilbertsona zveřejněném na portálu Medium [29].

Dynamické připojení výstupních souborů do HTML dokumentu

Při zmíněném nastavení se počet výstupních souborů mění podle počtu využitých knihoven a balíčků třetích stran. Nastavení statického připojení příloh, při kterém známe vždy přesný počet výstupních souborů, je nadále nedostačující. Ve výpisu 3.3 je příklad použití pluginu HTMLWebpackPlugin. Ten jednoduše vytvoří HTML soubor, do kterého připojí výstupní JavaScript soubory.

```
1  const getPartialContent = (fileName) => fs.readFileSync(
2  path.join(__dirname, 'htmlTemplate/partialContent', fileName));
3
4  new HtmlWebpackPlugin({
5    inject: 'body',
6    template: path.join(__dirname, 'htmlTemplate/index.ejs'),
7    customProps: {
8      amplitude: getPartialContent('amplitude.js'),
9    },
10   filename: 'client.html',
11 }),
```

Výpis 3.3: HTMLWebpackPlugin – nastavení parametrů pro nástroj, který generuje HTML soubor pro část client.

Samotný plugin má velmi dobře konfigurovatelné rozhraní. Parametr `inject` zajistí, že přílohy budou připojeny na konec body elementu. Pomocí parametru `template` je možné specifikovat cestu k výchozí šablonu, ze které plugin sestavuje výstupní HTML soubor. Skrze šablonu pak ve formě HTML lze předat např. hlavičku s meta atributy, nastavení favicony, titulek dokumentu a další.

Do šablony lze vložit i vlastní makro. Toho bylo využito pro připojení zdrojových kódů nutných pro funkci nástrojů sloužících ke sledování a analýze chování uživatelů v aplikaci. Pokud by byly soubory připojeny do hlavičky jako externí přílohy, vytvářely by další požadavky na server, které by mohly blokovat načtení HTML dokumentu. Proto je obsah každého souboru se zdrojovými kódy načten funkcí `getPartialContent` uvedenou ve výpisu 3.3. Při generování výstupního souboru je pak na místo makra vložen JavaScript kód.

```html
<!-- Amplitude -->
<script type="text/javascript"><%= htmlWebpackPlugin.options.
customProps.amplitude %></script>
```

Výpis 3.4: Použití makra v šabloně index.ejs, na toto místo bude ve vygenerovaném HTML souboru vložen JavaScript kód pro nástroj Amplitude.

Ve výpisu 3.4 je uveden zjednodušený příklad připojení zdrojového kódu, který je nutný pro integrování nástroje Amplitude. Díky němu lze sledovat akce, které uživatelé v aplikaci Kentico Kontent provádějí. Na základě získaných poznatků lze lépe pochopit jejich chování nebo měřit, kolik procent uživatelů využívá konkrétní části aplikace. Stejný princip se využívá pro připojení zdrojových kódů pro další nástroje, jako např. Google Tag Manager či Hotjar. Ve výpisech 3.3 a 3.4 je pro jejich zjednodušení uvedeno pouze připojení nástroje Amplitude. Na testovacím prostředí se v hlavičce HTML dokumentu nacházejí všechny tyto nástroje.

3.4.2 Výsledky měření

Tab. 3.3: Naměřené hodnoty pro scénář obsahující jeden soubor s vlastním kódem aplikace a jedním souborem pro každou knihovnu či balíček.

Typ měření	Přeneseno [kB]		DOM ContentLoad [s]	Load [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
První načtení	8200	8700	3,59	4,14	155	204
Načtení s cache	30,4	88,4	1,91	2,29	155	204
První načtení, LTE	8200	8700	20,36	20,55	155	203
Načtení s cache, LTE	31,80	99,80	2,51	3,02	155	204
Změna v JS příloze	4200	4600	2,59	3,01	155	203
Změna v JS, LTE	4100	4600	11,58	12,04	155	202

Při porovnání naměřených hodnot tohoto scénáře v tabulce 3.3 s hodnotami pro počáteční stav z tabulky 3.1 lze usoudit, že rozdělení knihoven a balíčků třetích stran do samostatných výstupních souborů nepřineslo očekávaná zlepšení. Hodnoty *DOMContentLoad* a *Load* jsou větší téměř pro všechny typy měření. To znamená, že velký počet požadavků se projevil negativně na době načítání aplikace, přestože testovací prostředí využívá protokol HTTP/2.

Jediný scénář, při kterém načítání trvalo kratší čas a bylo přeneseno menší množství dat než při původním stavu aplikace, je simulace změny ve vlastních JavaScript

přílohách. Zde pomohlo, jako při předchozím scénáři, oddělení vlastního JavaScript kódu do samostatného souboru. Míra efektivního využití cache paměti prohlížeče tohoto scénáře nepřináší výhody, které by převážily pomalejší načítání aplikace. Rozdělení knihoven a balíčků třetích stran jednotlivě do samostatných souborů se projevilo jako neefektivní způsob optimalizace. Všechny naměřené hodnoty lze nalézt v příloze A.1.3.

3.5 Rozdělení balíčků třetích stran i vlastního kódu do několika samostatných souborů

Vytvoření velkého množství výstupních souborů mělo negativní dopad zejména na rychlost načítání aplikace. Na druhou stranu s velmi malým množstvím objemných JavaScript souborů není možné využívat cache paměť webového prohlížeče efektivně. V tomto scénáři byla změněna kritéria, podle kterých webpack rozděluje JavaScript do výstupních souborů. Snahou je najít kompromis mezi počtem výstupních souborů a jejich velikostí.

3.5.1 Provedené změny

```
1  optimization: {
2    splitChunks: {
3      chunks: 'all',
4      automaticNameDelimiter: '-',
5      minSize: 100000,
6      cacheGroups: {
7        vendors: {
8          test: /[\\/]node_modules[\\/]/,
9          maxSize: 300000,
10         name: () => 'vendors',
11       },
12     },
13   },
14 }
```

Výpis 3.5: SplitChunksPlugin – rozdělení JavaScript příloh, kdy je vytvořen soubor pro kód sdílený mezi vstupními body, dále soubor s kódem unikátním pro vstupní bod. Knihovny a balíčky třetích stran jsou rozděleny do souborů podle uvedených kritérií.

Ve výpisu 3.5 se v obecné části nastavení pluginu objevuje parametr `chunks: 'all'`. Pokud je mezi vstupními body aplikace sdílený JavaScript kód, webpack pro něj vytvoří výstupní soubor. Dále pro každý vstupní bod vytvoří samostatný výstupní soubor, který obsahuje pouze zdrojový kód související s danou částí aplikace. Parametrem `minSize: '100000'` je nastaven spodní limit 100 kB pro velikost každého výstupního souboru před provedením optimalizačních technik, jako je například gzip. Webpack tuto hodnotu nedodržuje striktně, jelikož velikost výstupních souborů ovlivňují i jiné parametry.

V nastavení pro skupinu knihoven a balíčků třetích stran se objevuje další parametr ovlivňující velikost výstupních souborů, `maxSize: 300000`. Pro tuto skupinu tedy platí jak spodní limit definovaný v obecné části, tak horní limit velikosti souboru. Ostatní parametry objevující se ve výpisu 3.5 byly okomentovány v předchozích scénářích.

3.5.2 Výsledky měření

Tab. 3.4: Naměřené hodnoty pro rozdělené knihovny a balíčky třetích stan i vlastní kód aplikace.

Typ měření	Přeneseno [kB]		DOM ContentLoad [s]	Load [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
První načtení	8100	8700	2,37	2,84	39	92
Načtení s cache	30,0	91,1	1,42	1,82	39	92
První načtení, LTE	8100	8700	19,55	19,88	37	86
Načtení s cache, LTE	32,20	99,00	1,76	2,11	41	95
Změna v JS příloze	2400	2900	2,69	3,08	37	85
Změna v JS, LTE	2400	2900	7,80	8,23	39	88

Pokud porovnáme naměřené hodnoty pro aktuální scénář, které jsou uvedeny v tabulce 3.4, s naměřenými hodnotami z tabulek 3.1 a 3.2, lze vidět, že aplikované změny mají kladný dopad zejména na scénáře, kdy se do aplikace dostanou změny a je potřeba ze serveru znovu stáhnout část s vlastním kódem aplikace.

Rozdělením vlastního kódu aplikace do více výstupních souborů podle kritéria, kde všude je potřeba daný kód využívat, vzniká možnost, že se provedené změny se týkají pouze určitého výstupního souboru. V ideálním případě pak stačí ze serveru přenést menší množství dat, díky tomu se zkrátí i doba načítání aplikace. Největšího rozdílu si lze všimnou při simulaci mobilního LTE připojení, kdy oproti původnímu

stavu klesla doba načítání aplikace o polovinu (Typ měření: *Změna v JS, LTE*). Všechny naměřené hodnoty lze nalézt v příloze A.1.4.

3.6 Optimalizace předchozího scénáře a asynchronní načítání větších částí aplikace

V předchozím scénáři přinesly aplikované změny u některých typů měření výrazně lepší výsledky. Míra zkrácení doby načtení aplikace a zmenšení objemu přenesených dat závisela na tom, jaký počet příloh bylo nutné ze serveru znovu stáhnout.

S přidáním asynchronního načítání se přílohy potřebné pro různé části aplikace začnou načítat až v momentu, kdy se do nich uživatel skrz rozhraní aplikace sám naviguje.

3.6.1 Provedené změny

```
optimization: {
  splitChunks: {
    chunks: 'all',
    automaticNameDelimiter: '-',
    minSize: 200000,
    cacheGroups: {
      vendors: {
        test: /[\\/]node_modules[\\/]/,
        priority: -10,
        maxSize: 550000,
        name: () => 'vendors',
      },
    },
  },
}
```

Výpis 3.6: SplitChunksPlugin – optimalizovaná verze předchozího scénáře, kdy jsou změněny kritéria pro rozdělení knihoven a balíčků třetích stran do samostatných příloh.

V uvedeném výpisu 3.6 je velmi podobné nastavení pro SplitChunksPlugin, jako tomu bylo u předchozího scénáře. Změněny byly hodnoty pro parametry `minSize` a `maxSize`. Předchozí hodnoty těchto dvou parametrů, uvedené ve výpisu 3.5, povolovaly, aby webpack vytvořil celkem 28 výstupních souborů pro knihovny a balíčky

třetích stran. Po provedení gzip minimalizace byla velikost většiny souborů menší než 100 kB.

Po nastavení nových limitů pro minimální a maximální velikost webpack vytvořil pro knihovny a balíčky pouze 15 výstupních souborů, z toho po provedení gzip minimalizace jsou pouze 4 soubory menší než 100 kB. Podařilo se tak optimalizovat poměr mezi počtem výstupních souborů a jejich velikostí. Na obrázku 3.2 z nástroje Chrome DevTools lze vidět podrobnosti o jednotlivých výstupních souborech.

Name	Prot...	Size ▼	Time	Cache-Control
vendors-._node_modules_react-c_7f21996e85f2a99a92...	h2	238 kB	914 ms	public,max-age=2592000
vendors-._node_modules_@b_52e4eb18dad4617edf8c.js	h2	178 kB	288 ms	public,max-age=2592000
vendors-._node_modules_react-_031f2a4698116d4f50...	h2	158 kB	857 ms	public,max-age=2592000
vendors-._node_modules_u_592e9e487c95a27b388b.js	h2	140 kB	320 ms	public,max-age=2592000
vendors-._node_modules_d_7577960f5161b56d92ce.js	h2	140 kB	237 ms	public,max-age=2592000
vendors-._node_modules_@_2e27d5bd91ee4686fffd.js	h2	138 kB	241 ms	public,max-age=2592000
vendors-._node_modules_r_2d7975411f49fea7ce1d.js	h2	133 kB	723 ms	public,max-age=2592000
vendors-._node_modules_dn_be3d023ee30f109b62c6.js	h2	132 kB	330 ms	public,max-age=2592000
vendors-._node_modules_c_7e34628dfd71bf921f57.js	h2	126 kB	181 ms	public,max-age=2592000
vendors-._node_modules_p_462fcbd6c93a4deb9818.js	h2	119 kB	704 ms	public,max-age=2592000
vendors-._node_modules_s_02a4196473d464689bc8.js	h2	100.0 kB	319 ms	public,max-age=2592000
vendors-._node_modules_da_5a23589636029efeb9a2.js	h2	98.0 kB	131 ms	public,max-age=2592000
vendors-._node_modules_l_7e141d3d62e3f956106b.js	h2	96.2 kB	326 ms	public,max-age=2592000
vendors-._node_modules_h_6392667d15db10fa2524.js	h2	94.0 kB	988 ms	public,max-age=2592000
vendors-._node_modules_m_84f343504cdfe87748c3.js	h2	74.8 kB	635 ms	public,max-age=2592000

15 / 79 requests | 2.0 MB / 8.2 MB transferred | 2.0 MB / 8.7 MB resources

Obr. 3.2: Chrome DevTools - seznam JavaScript příloh obsahující knihovny a balíčky. Optimalizovaný poměr počtu souborů k jejich velikosti.

Asynchronně načítané části aplikace

V předchozím scénáři popsaném v části 3.5 byly rozděleny části vlastního kódu aplikace do několika souborů. Při prvním načtení aplikace se uživateli stahují všechny přílohy s vlastním kódem aplikace. Pokud se uživatel nachází např. v části aplikace *Home*, stažené přílohy obsahují i JavaScript kód pro části *Content & Assets* a *Content models*. Pro částečné řešení tohoto problému musely být aplikovány změny v připojování rodičovských komponent.

```
const ContentModels = lazy(() => import('../../applications/content
Models/shared/containers/ContentModels'));
<AuthorizedRoute
  appName={AppNames.ContentModels}
  path={routes.contentModels.path}
  render={({ match }: RouteComponentProps<ProjectRouteParams>) => (
    <Suspense fallback={<Loader />}>
      <ContentModels projectId={match.params.projectId} />
    </Suspense>
  )}
/>
```

Výpis 3.7: Zjednodušený příklad asynchronního načítání větší části aplikace v React.js s využitím metody lazy. Tento kód je uveden v komponentě Project.tsx

Ve výpisu 3.7 je uveden zjednodušený příklad použití asynchronního připojení rodičovské komponenty `Content Models`, do které se ve stromové struktuře připojují desítky dalších komponent v kontextu routeru, který se využívá pro navigaci v React aplikacích. Na prvním řádku výpisu se nachází konstanta `ContentModels` s funkcí `lazy`, ta jako parametr dostává anonymní funkci, která vrací dynamické připojení komponenty `ContentModels`.

Asynchronní načítání rodičovské komponenty se děje uvnitř komponenty `Suspense`, ta v uživatelském rozhraní zobrazí stav načítání do doby, než je dokončeno asynchronní stažení potřebných částí.

3.6.2 Výsledky měření

Tab. 3.5: Naměřené hodnoty pro scénář, kdy jsou knihovny rozděleny do optimálního počtu příloh, pro větší části aplikace je nastaveno asynchronní načtení.

Typ měření	Přeneseno [kB]		DOM ContentLoad [s]	Load [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
První načtení	7600	8200	2,25	2,54	27	80
Načtení s cache	31,90	93,90	1,26	1,54	27	81
První načtení, LTE	7600	8200	18,15	18,51	25	75
Načtení s cache, LTE	32,40	90,00	1,44	1,73	27	73
Změna v JS příloze	1700	2200	1,74	2,16	25	72
Změna v JS, LTE	1700	2200	6,16	6,58	25	74

Z tabulky 3.5 lze vyčíst, že provedené změny v tomto scénáři měly kladný dopad na velikost objemu přenesených dat téměř u všech typů měření.

Pokud porovnáme naměřené hodnoty s hodnotami z tabulky 3.1 pro výchozí stav aplikace, zjistíme, že se zkrátila doba potřebná pro načtení aplikace v těch typech měření, které využívají simulaci mobilního LTE připojení. Největší rozdíl je u typu měření, kdy se do aplikace dostane nová verze příloh s vlastním kódem aplikace. To samé platí i pro srovnání hodnot získaných v aktuálním scénáři s hodnotami z tabulek 3.2, 3.3 a 3.4.

Postupně aplikovanými úpravami bylo získáno optimální nastavení pro rozdělování JavaScript kódu do samostatných příloh podle různých kritérií. I v tomto scénáři se však najdou slabá místa, která by šla zdokonalovat. Všechny naměřené hodnoty lze nalézt v příloze A.1.5.

Závěr

Výchozí nastavení aplikace obsahovalo určité formy optimalizace JavaScript a CSS příloh. Jednalo se o zmenšení velikosti příloh, které bylo popsáno v podkapitole 2.4.1. Původní stav aplikace používal jako optimalizační metodu HTTP kompresi, její princip byl popsán v podkapitole 2.4.2. Nedostatky, které původní stav měl, byly popsány v podkapitole 2.4.4. Největší problém spočíval v tom, že při sestavení aplikace vznikl pouze jeden soubor s veškerým JavaScript kódem. To velmi ztížilo efektivní využití mezipaměti prohlížeče, jejíž princip je popsán v podkapitole 2.4.3.

V konfiguračním souboru webpack bylo provedeno velké množství změn. Byly aplikovány různé varianty nastavení pluginu SplitChunksPlugin. Tyto změny jsou detailně o popsány v kapitole 3.

Měřením bylo zjištěno, který z uvedených scénářů byl pro spouštění aplikace nejoptimálnější. Provedená měření potvrdila, že nejlepších výsledků bylo dosaženo při aplikování scénáře se změnami, které jsou popsány v kapitole 3.6. Pro porovnání provedených změn byly z tabulek 3.1 a 3.5 vybrány hodnoty pro typ měření, které simuluje situaci, kdy se do aplikace dostane změna v JavaScript přílohách a uživatel bude využívat mobilní LTE připojení.

Tab. 3.6: Porovnání původního stavu se změnami ze scénáře 3.6. Hodnoty byly získány při typu měření, kdy byly provedeny změny v JS přílohách a uživatel využil mobilní připojení.

Scénář	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
Původní	6000	6500	16,13	16,67	9	54
Optimal.	1700	2200	6,16	6,58	25	74
Porovnání	**-4300**	**-4300**	**-9,97**	**-10,09**	**16**	**20**

Pokud porovnáme hodnoty z tabulky 3.6 dojdeme k výsledku, že při spouštění aplikace po provedení změn v JavaScript přílohách se zmenší objem přenesených dat ze serveru **až o 4300 kB** (o 71,67 % méně) oproti výchozímu nastavení. V případě že by uživatel zároveň využíval mobilní LTE připojení s podobnou rychlostí uvedenou v podkapitole 3.1, aplikace by se spustila **přibližně o 10 vteřin rychleji**. Velikost přenesených dat bude vždy záviset na tom, jaké množství JavaScript příloh bylo s vydáním nové verze nebo hotfixu změněno. Téměř vždy však bude dosaženo znatelně lepších výsledků než s jediným souborem obsahující všechen JavaScript kód. Z pohledu efektivního využití mezipaměti prohlížeče lze aplikované změny považovat za velmi přínosné.

Pro další porovnání jsou využity hodnoty opět z tabulek 3.1 a 3.5. Tentokrát se však zaměříme na typ měření, který simuluje situaci prvního spouštění aplikace, nebo situaci, kdy v mezipaměti prohlížeče nejsou uložený žádné aktuální přílohy.

Tab. 3.7: Porovnání původního stavu se změnami ze scénáře 3.6. Hodnoty byly získány při měření, kdy uživatel spouští aplikaci poprvé nebo v mezipaměti nejsou uloženy žádné přílohy.

Scénář	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
Výchozí	8100	8700	2,82	3,37	11	62
Optimal.	7600	8200	2,25	2,54	27	80
Porovnání	**-500**	**-500**	**-0,57**	**-0,83**	**16**	**18**

Pokud porovnáme hodnoty z tabulky 3.7 dojdeme k závěru, že rozdíl při prvním spouštění nebo spouštění bez uložených příloh před a po provedení optimálních úprav je poměrně malý. Objem přenesených dat se zmenšil **přibližně o 500 kB** (o 5,75 %). Doba nutná pro spouštění aplikace se zkrátila **přibližně o 0,83 vteřin**.

Při rozdělení vlastního kódu aplikace do samostatných částí, které pak lze načíst asynchronně, byla objevena technická omezení. Nástroj webpack s použitím pluginu SplitChunksPlugin totiž nedokázal z výstupního souboru, který obsahuje vlastní kód aplikace sdílený mezi jednotlivými vstupními body, oddělit součásti související s použitím JavaScript knihovny Redux.

Vyřešit tento problém by znamenalo změnit přístup, jakým je knihovna Redux používána. To by obnášelo přepsání značného množství zdrojového kódu způsobem, jaký je uveden v dokumentaci knihovny Redux [30]. To s sebou přináší velké množství práce, která nemohla být zrealizována. Díky této bakalářské práce byl však objeven problém, který bude nutné vyřešit. Jeho řešení totiž umožní rozdělit aplikaci do větších samostatných částí, které půjde načítat asynchronně, což povede k zmenšení objemu dat, které je nutné přenést pro spuštění aplikace.

Hodnoty získané porovnáním v tabulkách 3.6 a 3.7, lze brát jako krajní extrémy. Většina reálných uživatelů, kteří používají aplikaci Kentico Kontent se bude nacházet v situaci, kdy část příloh budou mít uloženou v cache paměti prohlížeče a část se jim při spouštění aplikace bude muset stáhnout znovu. Provedené úpravy, které byly popsány v kapitole 3.6, se již brzy dostanou do produkční verze aplikace Kentico Kontent.

S přihlédnutím k tomuto faktu a k výsledkům které byly díky provedeným úpravám dosaženy, lze považovat poznatky získané při vypracování této práce za přínosné.

Literatura

[1] HAVERBEKE, Marijn, *[2019]. Eloquent JavaScript: a modern introduction to programming.* Third edition. San Francisco: No Starch Press, s. 472. ISBN 1593279507

[2] *You Don't Know JS Yet: Get Started - 2nd Edition: Chapter 1: What Is JavaScript?* [online]. 2020 [cit. 2019-11-16]. Dostupné z: `https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/get-started/ch1.md`

[3] *ECMA International: ECMAScript® 2019 Language Specification* [online]. 2020 [cit. 2020-01-11]. Dostupné z: `https://www.ecma-international.org/ecma-262/10.0/index.html#sec-conformance`

[4] *ISO/IEC 16262:1998* [online]. 2020 [cit. 2020-01-11]. Dostupné z: `https://www.iso.org/standard/29696.html`

[5] *New features of ECMAScript 2017* [online]. 2017 [cit. 2020-02-02]. Dostupné z: `https://lethalbrains.com/new-features-of-ecmascript-2017-c25a9db5f5e0`

[6] *JavaScript: What's new in ECMAScript 2019 (ES2019)/ES10?* [online]. 2019 [cit. 2020-02-02]. Dostupné z: `https://medium.com/@selvaganesh93/javascript-whats-new-in-ecmascript-2019-es2019-es10-35210c6e7f4b`

[7] *2ality – JavaScript and more: Who designs ECMAScript?* [online]. 2015 [cit. 2019-12-03]. Dostupné z: `https://2ality.com/2015/11/tc39-process.html`

[8] *2ality – JavaScript and more: How is ECMAScript designed?* [online]. 2015 [cit. 2019-12-03]. Dostupné z: `https://2ality.com/2015/11/tc39-process.html`

[9] *The TC39 Process, The TC39 Process* [online]. 2015 [cit. 2019-12-01]. Dostupné z: `https://tc39.es/process-document/`

[10] *Test262: ECMAScript Test Suite* [online]. [cit. 2019-12-01]. Dostupné z: `https://github.com/tc39/test262`

[11] *Tc39/ecma262: Repository* [online]. [cit. 2019-12-01]. Dostupné zL: `https://github.com/tc39/ecma262`

[12] *Dev.to: Node.js Under The Hood 4 - Let's Talk About V8* [online]. 2019 [cit. 2019-12-01]. Dostupné z URL: `https://dev.to/khaosdoctor/node-js-under-the-hood-4-let-s-talk-about-v8-1eol`

[13] *Windows Blog: Microsoft Edge's JavaScript engine to go open-source* [online]. 2015 [cit. 2019-12-03]. Dostupné z URL: https://blogs.windows.com/msedgedev/2015/12/05/open-source-chakra-core/#GTVrPh87gyj3osZc.97

[14] *Github: ChakraCore* [online]. [cit. 2019-12-03]. Dostupné z URL: https://github.com/Microsoft/ChakraCore

[15] *Intermediate Code Generation in Compiler Design* [online]. [cit. 2020-01-11]. Dostupné z URL: https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/

[16] *Interpreter Vs Compiler : Difference Between Interpreter and Compiler* [online]. [cit. 2020-01-11]. Dostupné z URL: https://www.programiz.com/article/difference-compiler-interpreter

[17] *Difference Between Compiler and Interpreter* [online]. [cit. 2020-01-11]. Dostupné z URL: https://techdifferences.com/difference-between-compiler-and-interpreter.html

[18] CLARK, Lin. *Mozzila hacks: A crash course in just-in-time (JIT) compilers* [online]. 2017 [cit. 2019-12-03]. Dostupné z URL: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

[19] *WhatIs: stub* [online]. 2005 [cit. 2019-12-03]. Dostupné z URL: https://whatis.techtarget.com/definition/stub

[20] *Chapter 1. What is a single-page application?*, Emmit A. Scott , Jr. SPA Design and Architecture [online]. Manning Publications; 1st edition, 2015, s. 312 [cit. 2020-06-01]. ISBN 9781617292439. Dostupné z: https://livebook.manning.com/book/spa-design-and-architecture/chapter-1/

[21] *SPA Front-End frameworky: Hlavní principy SPA frameworku* [online]. [cit. 2020-06-02]. Dostupné z: https://bonsai-development.cz/clanek/single-page-application-front-end-frameworky-a-ktery-si-vybrat

[22] *React: Using the State Hook* [online]. In: . [cit. 2020-06-03]. Dostupné z: https://reactjs.org/docs/hooks-state.html

[23] *React Virtual DOM Explained in Simple English* [online]. In: . [cit. 2020-06-05]. Dostupné z: https://programmingwithmosh.com/react/react-virtual-dom-explained/

[24] *What Is a Content Management System (CMS)?* In: Kinsta [online]. 14.4.2020 [cit. 2020-04-16]. Dostupné z URL: https://kinsta.com/knowledgebase/content-management-system/

[25] *TerserWebpackPlugin. Webpack: Documentation* [online]. [cit. 2020-05-02]. Dostupné z URL: https://webpack.js.org/plugins/terser-webpack-plugin/

[26] *Optimize CSS Assets Webpack Plugin. Npm.js* [online]. [cit. 2020-05-02]. Dostupné z: https://www.npmjs.com/package/optimize-css-assets-webpack-plugin

[27] *HTTP caching. In: MDN web docs* [online]. [cit. 2020-06-07]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching

[28] *SplitChunksPlugin: Defaults* [online]. [cit. 2020-05-24]. Dostupné z: https://webpack.js.org/plugins/split-chunks-plugin/#defaults

[29] *The 100% correct way to split your chunks with Webpack: Splitting out each npm package.* In: Medium [online]. 13. 9. 2018 [cit. 2020-05-24] Dostupné z: https://medium.com/hackernoon/the-100-correct-way-to-split-your-chunks-with-webpack-f8a9df5b7758

[30] *Code splitting* In: Redux [online]. [cit. 2020-06-07]. Dostupné z: https://redux.js.org/recipes/code-splitting

Seznam symbolů, veličin a zkratek

API Application Programming Interface – rozhraní pro programování aplikace
CSS Cascadian style sheet
CDN Content delivery network – infrastruktura geograficky distribuovaných a spolupracujících serverů
CMS Content management system
DOM Document object model
ES ECMAScript
HTML Hypertext markup language
HTTP Hypertext transfer protocol
IoT Internet of things
JSX JavaScript XML
JSON JavaScript Object Notation
Less Less – CSS preprocesor
LTE Long-Term Evolution – standart pro širokopásmové bezdrátové připojení
MB Megabyte
Mbps Megabits per second
MVC Model View Controller
NoSQL Databáze nevyužívající tabulkový formát
SaaS Software as a service
SPA Single page aplikace
SSL Socket Server Layer
SVG Scalable Vector Graphics
TC39 Technical Committee 39

Seznam příloh

A Přílohy **57**
A.1 Veškeré naměřené hodnoty 57
A.1.1 Výchozí stav aplikace 57
A.1.2 Rozdělení knihoven a vlastního kódu do samostatných souborů 60
A.1.3 Samostatný soubor pro každý balíček třetí strany 62
A.1.4 Rozdělení balíčků třetích stran i vlastního kódu do několika samostatných souborů 64
A.1.5 Optimalizace předchozího scénáře a asynchronní načítání větších částí aplikace 66

A Přílohy

A.1 Veškeré naměřené hodnoty

Tato příloha obsahuje všechny hodnoty, které byly během měření získány. Z níže uvedených hodnoty byly vypočteny mediány, které se nachází v kapitole 3.

A.1.1 Výchozí stav aplikace

Tab. A.1: První načtení aplikace

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	8100	8700	2,89	3,37	11	61
2	8100	8700	2,82	3,27	11	63
3	8100	8700	2,97	3,55	11	63
4	8100	8700	2,74	2,93	11	62
5	8100	8700	2,43	2,63	11	62
Medián	8100	8700	2,82	3,27	11	62

Tab. A.2: Načtení s mezipamětí

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	29,9	88,1	1,54	1,85	11	62
2	29,9	90,4	1,00	1,16	11	67
3	29,5	90,1	1,43	1,59	11	62
4	30,3	90,8	1,21	1,55	11	62
5	29,9	90,7	1,23	1,44	11	64
Medián	29,9	90,4	1,23	1,55	11	62

Tab. A.3: První načtení se simulací mobilního LTE připojení

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	8100	8700	19,90	20,37	11	62
2	8100	8700	19,81	20,20	11	63
3	8100	8700	19,58	20,08	11	62
4	8100	8700	19,68	19,89	11	62
5	8100	8700	19,89	20,23	11	57
Medián	8100	8700	19,81	20,20	11	62

Tab. A.4: Načtení se simulací mobilního LTE s využitím mezipaměti

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	32,20	98,90	1,47	1,94	11	63
2	32,70	99,40	1,31	1,81	11	63
3	32,70	99,10	1,52	1,94	11	64
4	32,60	101,00	1,42	1,94	11	66
5	32,30	100,00	1,50	1,94	11	63
Medián	32,60	99,40	1,47	1,94	11	63

Tab. A.5: Načtení po změně v klientském JavaScript souboru

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	5470	6500	15,73	16,18	10	56
2	6000	6500	2,10	2,37	9	54
3	6000	6500	2,49	2,79	9	54
4	6000	6500	2,21	2,49	10	54
5	6000	6500	2,58	3,05	9	54
Medián	6000	6500	2,49	2,79	9	54

Tab. A.6: Načtení po změně v klientském JavaScript souboru se simulací mobilního LTE připojení

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	6000	6500	27,84	28,17	9	54
2	6000	6500	15,64	16,06	9	54
3	6000	6500	16,13	16,67	9	54
4	6000	6500	16,15	16,72	9	54
5	6000	6500	15,80	16,26	9	54
Medián	6000	6500	16,13	16,67	9	54

A.1.2 Rozdělení knihoven a vlastního kódu do samostatných souborů

Tab. A.7: První načtení aplikace

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	8100	8700	2,44	2,61	10	57
2	8100	8700	2,72	2,97	10	57
3	8100	8700	2,48	2,67	12	63
4	8100	8700	2,42	2,78	12	71
5	8100	8700	2,96	3,14	12	64
Medián	8100	8700	2,48	2,78	12	63

Tab. A.8: Načtení s mezipamětí

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	29,80	91,00	1,16	1,30	12	64
2	30,00	90,50	1,24	1,64	12	63
3	29,90	89,70	1,28	1,49	12	63
4	29,90	90,50	1,21	1,46	12	63
5	30,00	90,20	1,11	1,29	12	64
Medián	29,90	90,50	1,21	1,46	12	63

Tab. A.9: První načtení se simulací mobilního LTE připojení

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	8100	8700	19,47	19,68	12	64
2	8100	8700	19,50	19,89	10	63
3	8100	8700	19,56	20,01	12	63
4	8100	8700	19,56	19,94	12	63
5	8100	8700	19,48	19,88	10	57
Medián	8100	8700	19,50	19,89	12	63

Tab. A.10: Načtení se simulací mobilního LTE s využitím mezipaměti

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	29,50	90,90	1,36	1,53	12	65
2	29,90	90,60	1,50	1,78	12	64
3	30,40	29,50	1,00	1,29	12	64
4	29,90	90,40	1,50	1,80	12	64
5	29,50	90,10	1,02	1,17	12	64
Medián	29,90	90,40	1,36	1,53	12	64

Tab. A.11: Načtení po změně v klientském JavaScript souboru

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	4100	4600	13,47	13,9	10	57
2	4100	4600	2,05	2,40	10	57
3	4100	4600	2,27	2,63	10	55
4	4100	4600	2,71	3,01	10	55
5	4100	4600	2,28	3,56	10	55
Medián	4100	4600	2,28	3,01	10	55

Tab. A.12: Načtení po změně v klientském JavaScript souboru se simulací mobilního LTE připojení

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	4100	4600	33,02	33,50	10	56
2	4100	4600	11,09	11,35	10	56
3	4100	4600	11,39	11,72	10	56
4	4100	4600	11,25	11,64	10	55
5	4100	4600	11,89	12,33	10	55
Medián	4100	4600	11,39	11,72	10	56

A.1.3 Samostatný soubor pro každý balíček třetí strany

Tab. A.13: První načtení aplikace

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	8200	8700	3,59	4,75	155	204
2	8200	8700	4,80	5,24	155	204
3	8200	8700	3,43	3,87	155	203
4	8200	8700	3,79	4,14	155	205
5	8200	8700	3,59	3,74	155	203
Medián	8200	8700	3,59	4,14	155	204

Tab. A.14: Načtení s mezipamětí

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	30,5	89,3	2,49	3,03	155	204
2	29,5	86,5	2,45	2,60	155	204
3	30,4	88,4	1,40	1,79	155	203
4	55,4	113,0	1,90	2,29	157	209
5	29,5	85,7	1,91	2,08	155	203
Medián	30,4	88,4	1,91	2,29	155	204

Tab. A.15: První načtení se simulací mobilního LTE připojení

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	8200	8700	20,84	21,00	155	204
2	8200	8700	20,36	20,55	155	201
3	8200	8700	20,20	20,44	155	203
4	8200	8700	20,94	21,41	155	202
5	8200	8700	19,95	20,15	155	203
Medián	8200	8700	20,36	20,55	155	203

Tab. A.16: Načtení se simulací mobilního LTE s využitím mezipaměti

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	31,80	93,10	2,23	3,02	154	204
2	31,80	98,80	2,30	2,74	155	204
3	31,80	101,00	2,95	3,53	155	204
4	32,30	100,00	2,53	3,03	155	204
5	32,20	99,80	2,51	2,96	165	218
Medián	31,80	99,80	2,51	3,02	155	204

Tab. A.17: Načtení po změně v klientském JavaScript souboru

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	4200	4700	19,17	19,79	156	203
2	4200	4700	2,44	2,94	155	202
3	4200	4700	2,23	2,58	155	203
4	4100	4600	2,59	3,01	155	204
5	4100	4600	2,80	3,29	157	211
Medián	4200	4700	2,59	3,01	155	203

Tab. A.18: Načtení po změně v klientském JavaScript souboru se simulací mobilního LTE připojení

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	4100	4600	32,92	33,79	157	203
2	4110	4600	11,56	12,04	155	202
3	4100	4600	11,53	11,90	155	202
4	4100	4600	11,59	12,03	155	203
5	4100	4600	11,58	12,08	155	202
Medián	4100	4600	11,58	12,04	155	202

A.1.4 Rozdělení balíčků třetích stran i vlastního kódu do několika samostatných souborů

Tab. A.19: První načtení aplikace

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	8100	8700	2,01	2,63	37	86
2	8100	8700	2,37	2,84	37	86
3	8100	8700	2,15	2,67	37	85
4	8100	8700	2,42	2,87	37	86
5	8100	8700	2,61	3,13	37	86
Medián	8100	8700	2,37	2,84	37	86

Tab. A.20: Načtení s mezipamětí

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	29,5	90,4	1,50	1,93	39	92
2	30,1	91,1	1,32	1,80	39	92
3	29,5	90,4	1,42	1,82	39	93
4	30,0	92,8	1,60	2,00	39	93
5	30,4	91,1	1,23	1,67	23	92
Medián	30,0	91,1	1,42	1,82	39	92

Tab. A.21: První načtení se simulací mobilního LTE připojení

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	8100	8700	19,52	20,98	37	84
2	8100	8700	20,17	20,59	37	85
3	8100	8700	19,55	19,85	37	86
4	8100	8700	19,56	19,88	37	86
5	8100	8700	19,50	19,88	37	86
Medián	8100	8700	19,55	19,88	37	86

Tab. A.22: Načtení se simulací mobilního LTE s využitím mezipaměti

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	31,80	107,00	2,09	2,60	39	84
2	32,30	98,90	1,62	2,11	41	85
3	32,50	99,00	1,83	2,29	41	86
4	32,10	98,90	1,75	2,11	41	86
5	32,20	99,40	1,30	1,76	31	86
Medián	32,20	99,00	1,75	2,11	41	86

Tab. A.23: Načtení po změně v klientském JavaScript souboru

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	2400	2900	22,49	23,17	37	85
2	2400	2900	1,72	2,28	37	85
3	2400	2900	2,69	3,08	37	88
4	2400	2900	2,95	3,47	37	87
5	2400	2900	1,81	2,2	37	85
Medián	2400	2900	2,69	3,08	37	85

Tab. A.24: Načtení po změně v klientském JavaScript souboru se simulací mobilního LTE připojení

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	2450	3940	39,18	39,67	39	89
2	2400	2900	7,31	7,69	37	85
3	2400	2900	7,76	8,23	39	88
4	2400	2900	8,11	8,38	39	87
5	2400	2900	7,80	8,11	39	88
Medián	2400	2900	7,80	8,23	39	88

A.1.5 Optimalizace předchozího scénáře a asynchronní načítání větších částí aplikace

Tab. A.25: První načtení aplikace

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	7600	8200	2,12	2,61	25	73
2	7600	8200	2,32	2,46	27	79
3	7600	8200	2,25	2,54	27	80
4	7600	8200	2,06	2,37	27	81
5	7600	8200	2,44	2,78	27	80
Medián	7600	8200	2,25	2,54	27	80

Tab. A.26: Načtení s mezipamětí

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	31,9	95,0	1,44	1,54	27	82
2	31,9	93,9	1,11	1,35	27	81
3	31,9	92,4	1,41	1,58	27	79
4	31,9	94,9	1,09	1,36	27	82
5	32,4	93,9	1,26	1,57	27	81
Medián	31,9	93,9	1,26	1,54	27	81

Tab. A.27: První načtení se simulací mobilního LTE připojení

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	7600	8200	18,32	18,79	25	73
2	7600	8200	18,12	18,43	25	73
3	7600	8200	18,15	18,37	25	74
4	7600	8200	18,11	18,51	25	73
5	7600	8200	18,32	18,79	25	75
Medián	7600	8200	18,15	18,51	25	73

Tab. A.28: Načtení se simulací mobilního LTE s využitím mezipaměti

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	31,90	92,50	1,60	1,73	27	72
2	32,40	89,90	1,30	1,73	25	73
3	32,40	89,60	1,58	1,76	27	73
4	32,40	90,00	1,44	1,82	25	75
5	32,30	95,10	1,03	1,32	27	82
Medián	32,40	90,00	1,44	1,73	27	73

Tab. A.29: Načtení po změně v klientském JavaScript souboru

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	1700	2200	11,88	12,43	26	70
2	1800	2300	1,44	1,88	25	70
3	1800	2300	1,74	2,16	25	74
4	1700	2200	1,80	2,20	25	72
5	1700	2200	1,71	2,15	25	72
Medián	1700	2200	1,74	2,16	25	72

Tab. A.30: Načtení po změně v klientském JavaScript souboru se simulací mobilního LTE připojení

Č. měření	Přeneseno [kB]		DOMContentLoad [s]	Načítání [s]	Požadavky	
-	JS	Celkem	-	-	JS	Celkem
1	1700	2200	31,09	31,61	25	74
2	1700	2200	6,42	8,62	25	74
3	1700	2200	6,14	6,58	25	74
4	1700	2200	5,80	6,08	25	70
5	1700	2200	6,16	6,51	25	72
Medián	1700	2200	6,16	6,58	25	74